# Integrated System Interoperability Testing with Applications to VoIP

Ruibing Hao, David Lee, Rakesh K. Sinha and Nancy Griffeth
Bell Laboratories
Murray Hill, New Jersey

April 14, 2003

**Abstract**

This work has been motivated by the need to test interoperability of systems carrying voice calls over the IP network. The Voice over IP (VoIP) systems must be integrated and interoperate with the existing Public Switched Telephone Network (PSTN) before they are widely adopted. Standards have been developed to address this problem, but unfortunately different standards bodies and commercial consortiums have defined different standards. Furthermore, the prevailing VoIP standard such as H.323 is incomplete, complex, and presents the implementers with "vendors latitudes". As a result, there is no guarantee that the integrated VoIP systems would interoperate properly even if the implementations are all H.323-compliant. Thus interoperability testing has become indispensable.

We want test all the system interoperations by exercising all the required patterns of "interoperating behaviors." On the other hand, test execution in real environment is expensive, and we want to minimize the number of tests while maintaining the coverage.

We present a general method for automatic generation of test cases, which cover all the required system interoperations and contain a minimal number of tests. We study data structures and efficient test generation algorithms, which take time proportional to the total test case size. Finally, we report experimental results on VoIP systems.

Keywords: VoIP, integrated system, interoperability testing, coverage, redundancy.

## 1 Introduction

With the rapid growth of Internet, more and more new protocols are proposed and new products are developed such as Voice over IP(VoIP) and MultiProtocol Label Switching (MPLS). When two or more entities in separate communicating systems are integrated and need to interact with each other to perform a certain task the capability to operate as desired is called interoperability, which is considered an essential aspect of correctness of integrated systems. However, products from different vendors or even from the same vendor often do not interoperate properly. [1] Two main causes of non-interoperation are: ambiguity of protocol Specification, and vendor's proprietary extensions. Interoperability testing is to check the interoperations among integrated system implementations.

In the past, research on communication protocol testing has mainly concentrated on conformance testing that checks the conformance of the implementation of a protocol to its specification (cf. [6, 19, 21] and the bibliographies therein). As the authors of [15] pointed out, although

[1] The difficulty of building interoperable products reflects in the publicity surrounding successful multi-vendor interoperable demos, such as the OIF UNI interoperability demonstration at Supercomm 2001.

conformance testing is regarded as a necessary step on the way to achieve interoperation, direct testing of interoperation is still considered indispensable. In conformance testing, the implementation under test is usually residing in an isolated environment for the tester to execute the test, while in interoperability test the implementations are usually residing in an open environment, and the degree of interoperation between implementations depends not only on the implementations themselves but also on the environment.

The research work on interoperability testing can be roughly classified into two categories: general concepts and experiences of interoperability testing, and systematic generation of interoperability test suites. Most of the early work (beginning of 90s) belongs to the first category [8]. Gadre et al. [7] give a comprehensive discussion of various aspects of interoperability testing. Bonnes et al. [4] present their interoperability testing experiences at IBM. Vermeer et al. [22] describe their experiences with interoperability testing of FTAM protocol that uses a single tester between two Implementations under test that can observe and control the communication between two FTAM entities.

Most of the recent research work in this field is related to interoperability test suite derivation. One approach for interoperability test generation is to apply conformance test generation techniques on composed finite state machines, which are constructed from several components systems via a reachability analysis. A finite set of test cases is then selected to test the interoperability. Rafiq et al. [20] was one of the first papers in this field. It proposes the above test generation procedure and also an interoperability test architecture. Kang and Kim [13, 14, 15] proposed a test generation technique for protocol control portion interoperability testing. However, for actual construction of complete test cases manual steps were required, and it did not show how to avoid state explosion [3, 18, 5]. Furthermore, we may not have complete information on all the interoperating systems. For instance, in VoIP systems, we can model end user behavior and H.323 protocols, but we do not have a specification of the communication system, which includes the IP network that is too complex to model. Viho et al. [23] proposed a formal framework for interoperability testing and also discussed interoperability test generation. They proved that bilateral and global interoperability relations are equivalent in theory and pointed out doing bilateral interoperability testing can avoid the construction of composed specifications during test generation. In practice, we can remove lots of redundancy from the tests generated from the composed specification to improve the efficiency of test execution. However, this is hard to achieve for bilateral interoperability testing.

Our work has been motivated by the need to test interoperability of systems carrying voice calls over the IP network. The Voice over IP (VoIP) systems must be integrated and interoperate with the existing Public Switched Telephone Network (PSTN) before they are widely adopted. We model the system behaviors by extended finite state machines (EFSM), and propose a test generation strategy for interoperability testing. We discuss various criteria that cover the required patterns of interoperations. Based on the experiences of domain experts, a key idea in developing the coverage criteria is that interoperability errors are introduced only when the integrated systems are "interacting" with each other. See also [14, 15]. Based on this observation, our test generation only requires information about part of the integrated communication systems; it is impossible to obtain, model and analyze the whole systems in practice. For instance, for VoIP interoperability testing, we only need information about end users required behaviors and H.323 interfaces. Consequently, it does not require the modeling and construction of the composition of all the integrated systems, including the IP network, and avoids the state explosion problem while checking all the required interoperations.

We design efficient data structures and test generation algorithms, which guarantee desired coverage and contain a minimal number of tests. These algorithms are part of a software tool, ITIS

(Interoperability Testing Intelligent System), developed at Bell Laboratories. With this tool, we have generated interoperability test cases for: End users vs. the rest of the communication system; and End users and H.323 vs. the rest of the communication system. We report experimental results on real VoIP systems.

The remainder of this paper is organized as follows. After introducing a formal model in Section 2, we discuss the interoperability testing of integrated systems and present our interoperability test generation techniques in Section 3. Section 4 contains more details of the test generation algorithms, which incorporate a range of coverage and redundancy criteria. Experimental results on VoIP systems are reported in Section 5.

## 2 A Formal Model

Communicating system behaviors can often be modeled by extended finite state machines, which are finite state machines extended with variables. 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 $P(\vec{x}) = \text{TRUE}$, 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 is an assignment: $\vec{x} := A(\vec{x})$.

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 state, end 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 $s_{init} \in S$ 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., $P_t(\vec{x}) = \text{TRUE}$, 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 $q_t$.

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

Given an EFSM $M = (I, O, S, \vec{x}, T)$, each combination of a state and variable values is called a *configuration*. Given a configuration $[s, \vec{x}]$ and any transition from state $s$: $t = (s, q, a, o, P, A)$, if $P(\vec{x}) = \text{TRUE}$ then upon input $a$ we can execute transition $t$, update variable values by action: $\vec{x} := A(\vec{x})$, 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. Including 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.

An EFSM 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. Taking all the reachable configurations and the transitions between them, we obtain a *reachability graph* of the EFSM. Obviously, a reachability graph is also an FSM [17].

# 3 Interoperability Test Generation

## 3.1 Integrated System Interoperability Test Model

Interoperability testing is rather complex and there are different models for different applications and system implementation environments. A common type of interoperability testing is usually performed on two interconnected implementations from different vendors. That is, vendor $X$ makes implementation $A$ and vendor $Y$ makes implementation $B$, the test operator who performs interoperability testing has complete information of both $A$ and $B$. Pre-testing is often performed on "the underlying service" to make sure it is sufficiently correct so that the test operator can test the interoperability of $A$ and $B$ with a better sense of the cause of error.

In this research, we focus on the interoperability test generation of integrated systems using a different model. In our model, we have an integrated system consisting of system components of which we have complete information and represented by $A$, and system components of which we don't have or choose not to have information, either it is not available or too complex to model, and represented by $B$. We can only access the systems through the interface with $A$. We can apply inputs to $A$ and observe its corresponding output responses and its interoperations with $B$ while they are interoperating. Note that the output responses of $A$ include its local outputs as a system component itself and also its interfaces with $B$, such as sending messages to and receiving messages from $B$. We want to study the interoperability of $A$ with $B$ using this model and we say that we are *testing the interoperability of $A$ with $B$*.

The test architecture based on our interoperability test model is shown in Figure 1. The tester can apply inputs to the integrated system through system components A only at point PCO and observe A's corresponding output responses and its interoperations with B at points PCO and PO.

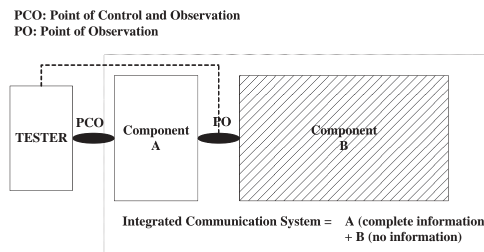


Figure 1: Interoperability Test Architecture of Integrated Systems

We use the integrated VoIP system in Figure 2 to illustrate this model. This VoIP system consists of a pair of phone sets, two $5ESS^{TM}$ telephone switches, a SS7 signaling network, two $MAXTNT^{TM}$ media gateways, one softswitch and an IP core network. To setup a regular two-party phone call between the two phone sets in the figure, the signaling message will need to traverse

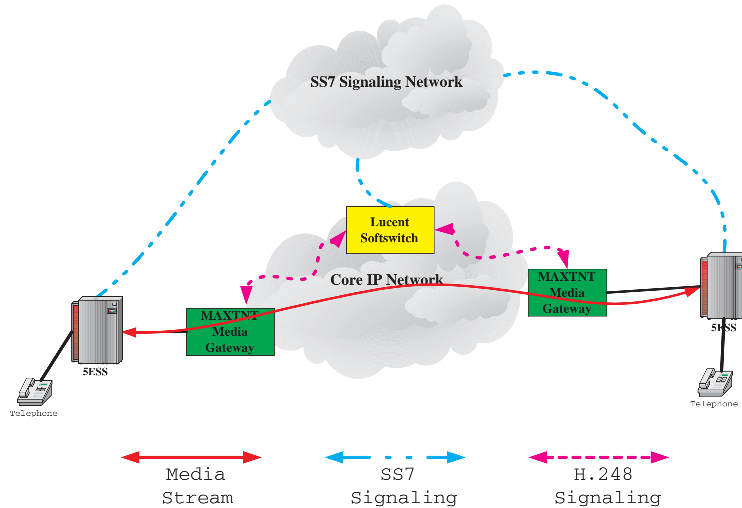


Figure 2: Integrated VoIP System

the two 5ESS and the SS7 network. If the SS7 signaling network decides the call can be setup, the softswitch will be notified to establish a media connection for this phone call. The softswitch will in sequence talk to the two $MAXTNT^{TM}$ media gateways via H.248 protocol to set up a media stream channel for this particular VoIP call. All of these PSTN and VoIP components need to work together to finish an VoIP call. Because of the complexity of an integrated system, usually it is impractical to model the whole system for the purpose of interoperability test generation. So to test the interoperability of components in this VoIP system, for instance, we can take a pair of phone sets as component $A$, and the rest of the system as $B$. Or we can take the two MAXTNT$^{TM}$ media gateways as component $A$, and the rest of the system as $B$. We can apply inputs to $A$ while it is interoperating with $B$, and observe the corresponding outputs. We know the expected outputs. If there is a discrepancy between the expected and the observed outputs, then a fault is detected.

Because of the limited controllability and observability in the test architecture it is a rather intriguing problem for fault identification. In this work, we do not intend to solve the problem of identifying where the faults are in the network. In practice, usually we deploy some traffic sniffers in the network to record the packet exchange which could be used at a later time for a manual analysis and diagnosis.

Interoperability and conformance testing are closely related but different. Given a system specification $A$ and an implementation $A'$, which is a "black-box", we want to apply a test to $A'$ to conclude whether it conforms to the specification $A$, i.e., $A'$ is identical to $A$. Ideally, we can apply all the possible executable sequences as tests to $A'$; if for any test $A'$ and $A$ produce different outputs then $A'$ is faulty, otherwise, they are identical [17]. However, in general there are infinitely many such test sequences. We want to reduce the number of tests to finitely many and minimize the number of tests yet without scarifying the fault coverage, and that leads to the research of checking sequences [17], which check the conformance of the implementations to the corresponding specifications with a polynomial length tests.

For the interoperability testing in our model, we want to test the interoperations between the system components $A$ and $B$; we have complete information and control of $A$ and no information of $B$. By a same rationale as conformance testing, ideally, we want to apply all the possible executable sequences as tests to $A$ to trigger all the possible interoperations between $A$ and $B$ - the interoperations which can be controlled and observed by applying tests to $A$ only. By a same

reason, in general there are infinitely many such tests. We want to reduce the tests to finitely many and minimize

the number of tests yet maintaining the same coverage on interoperations between $A$ and $B$, which are triggered by the tests. For this, we need to explore the redundancy criteria and use efficient test generation algorithms. Obviously, the resulting tests are different than that for the conformance testing.

We use the following three states FSM in Figure 3 as protocol system $A$ to illustrate.

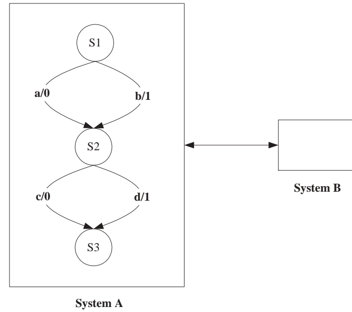


Figure 3: Three States FSM for Protocol System A

For conformance testing, we want to construct a checking sequence from $A$ for testing the conformance of its implementation $A'$. For interoperability testing, we want to test the interoperations of $A$ with another system component $B$. We can apply inputs to $A$ and observe its corresponding output responses while $A$ is interoperating with $B$. For simplicity, suppose that $A$ has status message, i.e., at any moment we know in which state $A$ is in. Assume that the system has reset and we start each test from state S1 in $A$.

For conformance testing of $A$, obviously, the following two test sequences consist of a covering set of tests, i.e., each transition of $A$ is exercised at least once: $ac/00$ and $bd/11$. Since $A$ has a status message, they are also checking sequences, which can test the structural isomorphism of an implementation $A'$ to its specification $A$, and that is all we need for a conformance testing.

However, for interoperability testing, they are not enough; we want to test on every possible input sequence to trigger all the possible interoperations of $A$ with $B$. Obviously, we need 4 tests: $ac/00$, $bd/11$, $ad/01$, and $bc/10$.

For testing interoperability of system component $A$ with $B$, due to the limited controllability and observability, that is, we can only control the inputs to and observe the output responses from $A$, even if we could test all the possible input sequences to $A$, still we couldn't guarantee the interoperability of $A$ with $B$; certain interoperation errors could come from $B$, and that is beyond the control in our model. Again, our goal is to generate a test set on $A$ to trigger the interoperations between $A$ and $B$ as much as possible to reveal the interoperation errors.

Also note that to test interoperability of a complex system, we may have a variety of ways to determine component $A$ and $B$ for their interoperability testing; it depends on the practical coverage needs and the feasibility of testing.

### 3.2 Test Generation Strategy

We want to test interoperability of systems $A$ with $B$, assuming that we have complete information about $A$ and that we can only access $A$. We can model the joint-behavior of A's by an EFSM and construct its reachability graph, which is a directed graph or a transition diagram [17]. The EFSM

that covers the joint-behavior of several system components can be constructed by calculating the Cartesian product of the EFSMs for these components [9]. Throughout the rest of the paper, we will be using graph terminology and working on the reachability graph. For example, we will denote the initial state as the "source node", transitions as "edges", etc.

When we conduct interoperability testing, we are concerned only with those failures that occur when the components of integrated systems are interoperating. So the coverage criteria of our interoperability test generation is to test all the possible interoperations of $A$ with $B$.

Ideally, we want to check every input sequence to $A$, often called scenarios, consequently all its interoperations with $B$, the rest of the system, are tested. Covering all possible execution sequences requires that we cover all the branches and all the possible paths in the EFSM. However, it is impossible; in general, transition diagrams often contain cycles and, therefore, will have infinitely many distinct paths. We need to remove the redundant paths, if possible, and reduce the number of tests to a manageable size.

There are various techniques to remove redundancy. For instance, in the configuration of Figure 2, a key idea in developing the test cases is that interoperability errors will be introduced only when the gateways are actually talking to each other about a call. Thus we can ignore local activities involved in the protocol, such as obtaining information from an end-user. We label as "white" the transitions that can be ignored and label all other transitions as "black." A white transition is purely local and is involved with only one gateway. A black transition involves both gateways. This is illustrated in the call flow for a simple telephone call in Figure 4. The transition "Off-hook A" from *Idle state* to *Dialing state* is white, because it involves only the originating gateway and the end user. The "Dial A B" transition from *Dialing* to *Calling* is black, because the state is changed in both gateways. Similarly, the "Off-hook B" transition from *Calling* to *Connected* is also "black." The test cases that we generate focus only on black edges. The purpose of white edges is to interconnect the sequence of black edges.

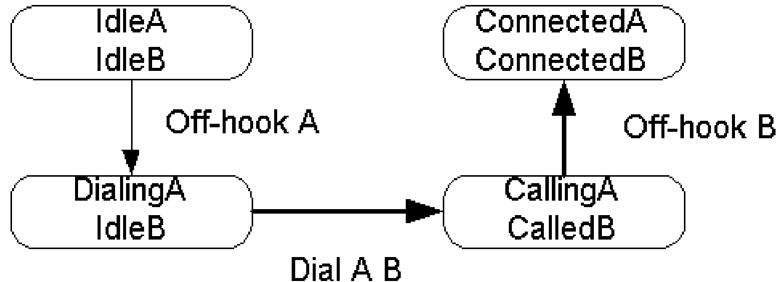


Figure 4: Partial Call Flow of a Telephone Call

Based on our information about $A$, we want to construct a set of non-redundant tests such that all the interoperations of $A$ with $B$ are tested. Our goals are: (1) *Completeness*: all the interoperations between the two systems are tested; and (2) *Non-redundancy*: all the redundant tests are removed to minimize the total number of tests.

We start with an "exhaustive" test set. To trigger all the interoperations between $A$ and $B$, ideally, we want to include all the possible tests, i.e., all the paths from the initial configuration. In general, there are infinitely many such paths. From the practical experiences in testing, exercising a cycle multiple times has a same coverage for interoperability testing as exercising it once. Furthermore, testing a complex cycle (with repeated nodes) is the same as testing each of its simple cycles (without repeated nodes). Therefore, we only need to test each simple cycle once for the

interoperability testing. On the other hand, each path consists of an acyclic path with zero or more simple cycles attaching to it. Therefore, a set of paths that contains all the acyclic paths, on some of which all the simple cycles are attached, has the same coverage as all the paths. Note that in a directed graph, there is a finite number of acyclic paths and a finite number of simple cycles. Consequently, the number of such paths is finite, and they consist of a test set with a complete coverage. It takes three steps to generate such a test set:

**step 1** Generate all possible acyclic paths, i.e. paths without any repeated vertices.

**step 2** Generate all possible simple cycles, i.e. cycles that do not contain any smaller cycles.

**step 3** "Combine" the paths (from step 1) and simple cycles (from step 2) to generate the final set of paths.

First we include all the acyclic paths. Then we generate a set of all the simple cycles $C$. For each acyclic path $P$, find all the simple cycles that remain in $C$ and share a node with $P$. Let these cycles be $C_1, C_2, \ldots, C_k$ and let $v_i$, $1 \leq i \leq k$, be a node common to $C_i$ and $P$ (if there are more than one common nodes, choose one arbitrarily). Then generate a new (cyclic) path by replacing node $v_i$ in $P$ with cycle $C_i$, deleting the $k$ attached simple cycles from $C$. After processing all the acyclic paths, if there are still simple cycles remaining in $C$, we simply discard them; they are not reachable from the initial configuration.

Now we elaborate on steps 1 and 2 using the example FSM shown in Figure 5.

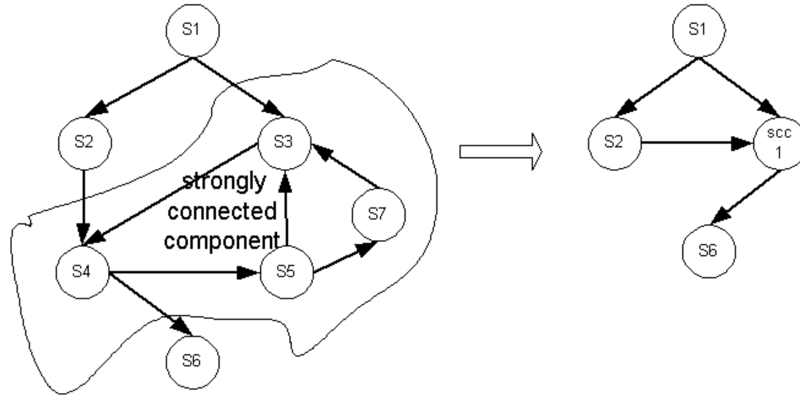


Figure 5: Example Reachability Graph and its Corresponding DAG

To generate all the acyclic paths and simple cycles in the directed graph, we first find all the strongly connected components (SCC) of the directed graph. This has two advantages: we know that any cycle is completely contained inside an SCC. So we can perform step 2 by looking at each SCC in turn and finding all simple cycles within this SCC. The second advantage is that we can "shrink" each SCC into a node and obtain a Directed Acyclic Graph (DAG), i.e., a graph without any cycles [2]. This naturally translates into a two-phase algorithm for step 1. First, generate all acyclic paths on the resulting DAG and then replace each SCC on any given path with a set of acyclic paths within the SCC. Following we will first discuss the data structure and the techniques used to generate acyclic paths and simple cycles within an SCC, and defer the discussion on the test generation algorithms to Section 4.

### 3.2.1 Next-Transition-Tree

We now describe a simple data structure, next-transition-tree, that will be useful for both steps 1 and 2. The data structure can be defined for any graph but in our application, the graph will

always be an SCC.

For any node $v$, next-transition-tree $T(v)$ stores all acyclic paths from $v$ to other vertices in its SCC. The tree has $v$ as its root. The children of $v$ are all the nodes in its SCC that have a direct edge from $v$. In general, the children of any node $u$ are all the nodes in its SCC that have a direct edge from node $u$. However, if a child node has appeared on the path from the root node $v$ to $u$, then it is not included in the tree. Note though that a node may appear multiple times in this tree but not on a tree path.

We do not need a next-transition-tree for each node in the graph. We need it only for a few of the nodes; the details are in Section 4. For ease of presentation, we assume that we maintain a separate next-transition-tree for each node. The actual implementation will have lots of shared pieces among next-transition-trees belonging to the same SCC. It is also possible to maintain this tree implicitly (with some extra state) in the adjacency list of the graph. Figure 6 shows the next-transition-tree for nodes $s3$ and $s4$ in the SCC component of the example FSM.

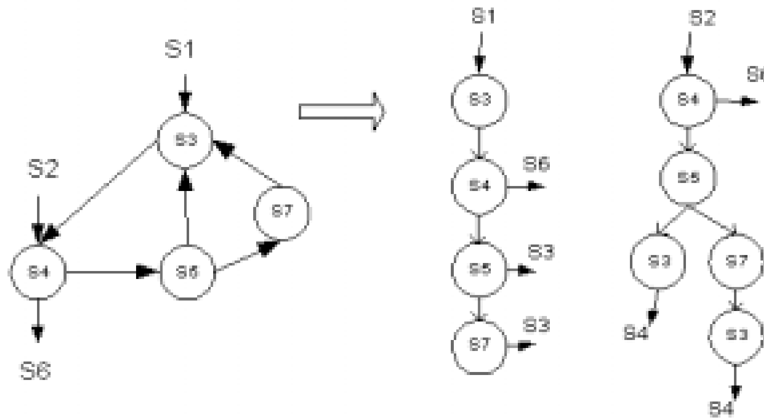


Figure 6: Next-transition-tree

### 3.2.2 Generating All Acyclic Paths and Simple Cycles within an SCC

Let $T(v)$ be the next-transition-tree of node $v$ in an SCC. We can easily obtain all the acyclic paths from $v$ and all the simple cycles containing $v$. It is based on the following lemma, and we omit the straightforward proof.

**Lemma 1** *For a next-transition-tree from node $v$, $T(v)$, each tree path from $v$ to a node $u$ is an acyclic path in the SCC. Conversely, for each acyclic path from $v$ to $u$ in the SCC, there is an identical tree path from $v$ to a node $u$ in $T(v)$. Furthermore, all the tree paths in $T(v)$ are distinct.*

Therefore, we can easily obtain all the acyclic paths from $v$ in the SCC from all the paths from $v$ to nodes of $T(v)$, and they are all distinct.

Since each simple cycle containing $v$ consists of an acyclic paths from node $v$ to a node $u$ and a directed edge from $u$ back to $v$, we can construct all the simple cycles in the SCC as follows. When we construct the tree $T(v)$ and arrive at node $u$, if there is an edge $(u, v)$, we do not include it in the tree but record a simple cycle, which is from $v$ to $u$ along the tree edge, and then from $u$ back to $v$ along the directed edge $(u, v)$.

Figure 7 shows all the acyclic paths from node $s_3$ or $s_4$ and all the simple cycles containing $s_3$ or $s_4$ in the SCC.

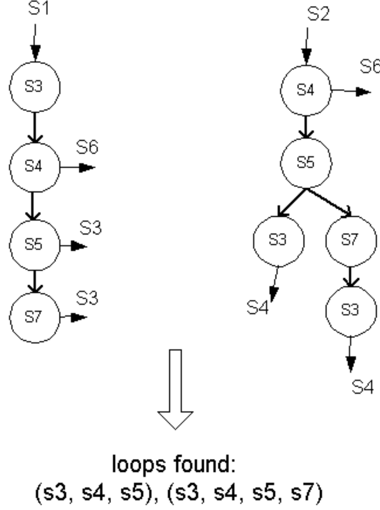


Figure 7: Acyclic Paths and Simple Cycles within an SCC

# 4 Interoperability Test Generation

As mentioned earlier, we first find all the strongly connected components (SCC) of the transition graph. Then we construct a *component graph* where each node represents an SCC and an edge from node $u$ to node $v$ represents an edge (in the original transition graph) from a node belonging to $u$'s SCC to a node belonging to $v$'s SCC. This component graph is a Directed Acyclic Graph (DAG), i.e., a graph without any cycles [2].

We first generate all acyclic paths in the DAG and then replace each SCC on any given path with a set of acyclic paths within the SCC. There are a finite number of distinct paths (tests) in a DAG. However, we do not want to test all of them; Our goal is to generate a minimal number of tests, that satisfies the coverage criterion and does not contain any *redundant* tests.

The tests need to start from the start state, so we only consider paths starting from the source node (node without any incoming transitions). In addition, the following is a commonly accepted *redundancy* criterion in practice.

**(R1)** *Proper prefixes are redundant and can be discarded.*

From the redundancy criterion above, we only consider paths that terminate at a sink node (node without any outgoing transitions). Any path not terminating at a sink node will be a proper prefix of a path that has been extended to a sink node. Combining the above two, we only need to generate paths that start from the source node and terminate at one of the sink nodes.

Figure 8 gives an algorithm for generating all source to sink paths in a DAG.

The algorithm (Figure 8) is bottom-up, starting from one of the sink nodes $t = v_{n-1}$. When processing a node $v_i$, we examine all its outgoing edges $(v_i, w_j)$ where the paths from $w_j$ to sink nodes have been computed. At Line 7, we concatenate edge $(v_i, w_j)$ to each path computed at $w_j$ and collect it at node $v_i$. After processing the source node $s = v_0$, we have obtained all the paths, which are non-redundant and have a complete coverage.

A topological sort takes time proportional to the number of edges. We account for the examination and concatenation of each edge, which is processed only once at Line 7, and the total cost is proportional to the total path lengths:

```
PATHS-IN-DAG
Input. a DAG G with one source s and multiple sinks.
Output. all source–sink paths in G.
Comments. p(v_i) stores all the v_i–sink paths in G.
1  topologically sort nodes in G: s = v_0, v_1, ..., v_{n-1};
2  for i = n - 1, ..., 0
3      if v_i is a sink node then
4          p(v_i) = {Λ}; /* a singleton set of empty path */
5      else
6          let outgoing edges from v_i be: w_1, ..., w_r;
7          p(v_i) = ∪_{j=1}^{r} (v_i, w_j) p(w_j);
8  return p(v_0)
```

Figure 8: Generating All Source–sink Paths in a DAG

**Proposition 1** *The algorithm in Figure 8 constructs all source–sink paths. Its time and space requirement is linear in the output (the total lengths of all the constructed tests).*

As indicated earlier, the coverage criteria for interoperability testing and conformance testing are different. However, if the underlying systems have a reset then the constructed tests from the algorithm in Figure 8 also provide a complete coverage for conformance testing [17]:

**Proposition 2** *For machines with a reset to source $s$, the constructed test sequences by the algorithm in Figure 8 are a checking sequence.*

### 4.1 A General Algorithm for Exhaustive Coverage

Up to now we have discussed test (path) generation for DAGs where nodes can be SCC. For each edge connecting two SCC's on the path, we need to replace it with an edge in the original graph. In general, it will be possible to replace the edge between SCC's with one of the several possible edges in the original graph, and we will obtain a separate path for each choice of the replacement edge. We also need to replace each SCC node with a set of acyclic paths within this SCC. Suppose the incoming and outgoing edges of this SCC node are on nodes $u$ and $v$ (respectively) of this SCC. Then we need to replace this SCC node with all possible acyclic paths from $u$ to $v$ in this SCC. These paths can be generated from next-transition-tree $T(u)$. Figure 9 summarizes the steps in generating all acyclic paths. Because this algorithm generates all the acyclic source–sink paths in the original transition graph, we will call it EXHAUSTIVE-COVERAGE algorithm.

**Remark:** Here we are replacing any SCC with all possible acyclic paths. This provides exhaustive coverage but generates lots of tests. In certain applications, we may not need to cover the SCC as thoroughly. There are two other options commonly used in practice.

*Chinese Postman Tour.* For an SCC, we want to test each edge (transition) at least once and we want to minimize the test sequence length. Such a path is called *Chinese Postman Tour* [1].

*Checking Sequence.* A more thorough coverage is provided by a checking sequence, which guarantees the structural isomorphism of the implementation and specification machines [17]. The length of the test sequence is longer than that of a Chinese Postman Tour.

EXHAUSTIVE-COVERAGE

*Input.* transition graph $G$ of with a designated source node.
*Output.* minimal set of acyclic paths providing exhaustive coverage.

1 find all SCCs of G;
2 for each SCC
3     for each node in this SCC
4         construct next-generation-tree;
5 shrink each SCC of G to obtain a DAG $G'$;
6 apply PATHS-IN-DAG on $G'$ to obtain all acyclic source–sink paths $P'$.
7 for each path $p'$ in $P'$
8     replace nodes and edges in $p'$ with paths and nodes
      in $G$ to obtain the set P of acyclic paths;

Figure 9: Generating All Acyclic Paths for Exhaustive Coverage

Figure 10 gives the algorithm for generating all possible tests based on the EXHAUSTIVE-COVERAGE algorithm.

EXHAUSTIVE-TEST-GENERATION

*Input.* An EFSM of the system.
*Output.* minimal set of tests providing exhaustive coverage.

1 construct an equivalent FSM and its transition diagram G.
2 apply EXHAUSTIVE-COVERAGE on $G$ to obtain acyclic paths P.
3 obtain the set C of all simple cycles.
4 combine the sets P and C (Section 3.2) to obtain the final set of tests.

Figure 10: Generating All Tests for Exhaustive Coverage

Figure 11 shows how this test generation algorithm works on the example FSM.

Note that it suffices to attach the two simple cycles to any of the acyclic paths only once and in an arbitrary way.

### 4.2 Redundancy Criteria

The algorithm outlined in the previous sections (Figure 10) provides exhaustive coverage. However, the number of tests generated is enormous even for systems of moderate size.

As stated in the introduction, a key insight is that that interoperability errors will be introduced only when the gateways are actually talking to each other about a call. We label the transitions as either "white" (local activity) or "black" (involves both gateways). For interoperability test generation, we do not need to cover white transitions. This motivates additional redundancy criteria.

**(R2)** *Remove all-white test sequences.*

**(R3)** *Let $(u, v)$ be the first black edge in the test sequence. Then replace the path from the source node to node $u$ with the shortest path between the source node and node $u$.*

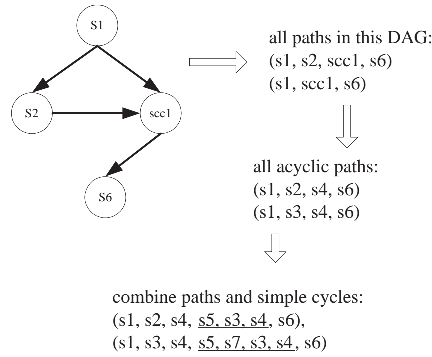


Figure 11: Example Test Generation Using Exhaustive Coverage Algorithm

**(R4)** *Let $(u, v)$ be the last black edge in the test sequence. Then replace the path from node $v$ to the sink node with the shortest path between node $v$ and the sink node.*

These three criteria reflect our belief that white transitions are not relevant for interoperability testing. Their only usage is to connect (relevant) black transitions. For example, $R3$ states that the sequence of transitions before the first black transition is not relevant to us, so we might as well replace that part with the shortest all-white sequence. Redundancy criterion $R4$ is dual of $R3$.

We give an algorithm in Figure 12 for generating all acyclic paths satisfying these criteria.

Complete-Coverage

*Input.* transition graph $G$ with a designated source node.
*Output.* complete set of paths according to redundancy criteria $R1$-$R4$.

1 for each node $v$ that has an outgoing black edge
2 if there is an all-white path from the source node to $v$ then
3 add a super-edge from the source node to $v$;
4 delete all outgoing white edges (not super-edges) from the source node;
5 for each node $v$ that has an incoming black edge
6 if there is an all-white path from $v$ to a sink node then
7 add a super-edge from $v$ to this sink node;
8 delete all incoming white edges (not super-edges) to the sink nodes;
9 generate all acyclic paths in the resulting graph by Exhaustive-Coverage with the modification that no super-edge should follow or precede a white edge or another super-edge;
10 replace each super-edge with the shortest all-white path in the original graph; /* this may introduce some loops */
11 process each path to get rid of loops;

Figure 12: Generating All Acyclic Paths for Complete Coverage (according to $R1$-$R4$)

### 4.2.1 Tests with Complete Coverage

We now discuss algorithms for generating test sequences according to $R1$-$R4$. Such tests provide a *complete* coverage.

The algorithm starts with a graph $G$ and generates another graph $G'$ such that the set of source–sink paths in $G'$ is the same as the set of source–sink paths in $G$ with criteria $R2$-$R4$.

The redundancy criterion $R3$ says that the path before the first black transition should be replaced with the shortest all-white path.

Lines 1-4 considers all possible black transitions that can be the first in any sequence and adds a marker 'super-edge' for the shortest all-white path from the source node. Line 9 makes sure that no 'super-edge' follows or procedes a white edge or another 'super-edge'. The marker 'super-edge' gets replaced in line 10 by the shortest all-white path. This takes care of redundancy criterion $R3$.

Lines 5-8 perform an analogous function for sink nodes and the last black transitions in the sequence, taking care of redundancy criterion $R4$.

Criterion $R2$ is satisfied because line 9 makes sure that there is at least one black transition.

**Proposition 3** *The algorithm in Figure 12 constructs all acyclic paths with a complete coverage.*

Note that to obtain the tests for a complete coverage, we need to attached simple cycles to the generated acyclic paths. We can discard all the simple cycles without any black edges, i.e., without any interoperations, and attach the remaining cycles to the generated acyclic paths in an arbitrary way.

**(R5)** *Remove all-white simple cycles.*

We give the algorithm for generating all possible tests based on the COMPLETE-COVERAGE algorithm and $R5$ in Figure 13.

COMPLETE-TEST-GENERATION

*Input.* An EFSM of the system.
*Output.* minimal set of tests providing complete coverage.
1 construct an equivalent FSM and its transition diagram G.
2 apply COMPLETE-COVERAGE on $G$ to obtain acyclic paths P.
3 obtain the set C of all simple cycles and remove those without any black edges.
4 combine the sets P and C (Section 3.2) to obtain the final set of tests.

Figure 13: Generating All Tests for Complete Coverage

Figure 14 shows how this complete coverage algorithm works on the example FSM.

### 4.3 Additional Redundancy Criteria

For most practical systems, we expect algorithm COMPLETE-TEST-GENERATION (Figure 13) to generate a considerably smaller set of test sequences than the naive algorithm (Figure 10). Unfortunately even such a smaller set may be too large for the test execution on real systems. In this section, we suggest a set of more restrictive criteria for test generation that concentrates only on black transitions.

**(R6)** *Generate acyclic paths that consist only of black edges, except that the prefix from the source node and the suffix to the sink node is allowed to contain white edges.*

**(R7)** *Generate simple cycles consisting only of black edges.*

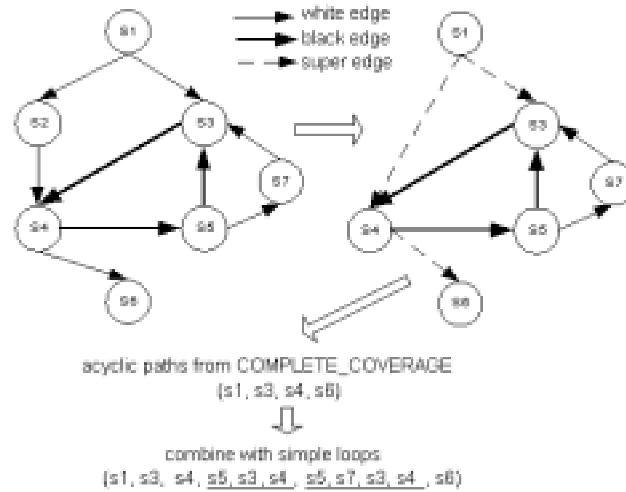


Figure 14: Example Test Generation Using Complete Coverage Algorithm

#### 4.3.1 Tests with Basic Coverage

We now discuss algorithms for generating test sequences according to $R1$-$R7$. Such tests provide a *basic* coverage. Figure 15 shows the BASIC-COVERAGE algorithm for acyclic path generation. The algorithm starts with a graph $G$ and transforms into graph $G'$ such that the set of source–sink paths in $G'$ is the same as the set of source–sink paths in $G$ with criterion $R1$-$R6$.

BASIC-COVERAGE

*Input.* transition graph $G$ of a FSM with a designated source state.
*Output.* test generation according to $R1$-$R6$.

1 for each node $v$ that has an outgoing black edge
2     if there is a path from the source node to $v$ then
3         add a super-edge from the source node to $v$;
4 for each node $v$ that has an incoming black edge
5     if there is a path from $v$ to a sink node then
6         add a super-edge from $v$ to this sink node;
7 delete all white edges from the graph;
8 generate all acyclic paths in the resulting graph by EXHAUSTIVE-COVERAGE with the same modification as in line 9 of COMPLETE-COVERAGE algorithm;
9 replace each super-edge with the shortest path in the original graph;
/* this may introduce some loops */
10 process each path to get rid of all the loops

Figure 15: Generating all Acyclic Paths for Basic Coverage(according to $R1$-$R6$)

The intuition of the algorithm is to delete all white edges except those needed to reach black transitions from the source node (or from the black transitions to the sink nodes). Lines 1-3 maintains a marker 'super-edge' for each source node to black transition path. In line 9, we replace

this marker with the shortest path. Lines 4-6 perform an analogous function for sink nodes.

**Proposition 4** *The algorithm in Figure 15 constructs all acyclic paths with a basic coverage.*

Note that to obtain the tests for a basic coverage, we need to attached simple cycles of black edges only to the generated acyclic paths in an arbitrary way. Following we give the algorithm for generating all possible tests based on the BASIC-COVERAGE algorithm and $R7$ in Figure 16.

BASIC-TEST-GENERATION

*Input.* An EFSM of the system.
*Output.* minimal set of tests providing complete coverage.
1 construct an equivalent FSM and its transition diagram G.
2 apply BASIC-COVERAGE on $G$ to obtain acyclic paths P.
3 obtain the set C of all simple cycles of black edges only.
4 combine the sets P and C (Section 3.2) to obtain the final set of tests.

Figure 16: Generating All Tests for Basic Coverage

**Discussion:** We feel that both algorithms BASIC-TEST-GENERATION and COMPLETE-TEST-GENERATION are interesting in their own rights. If automated test execution is available then COMPLETE-TEST-GENERATION is the preferred algorithm. If on the other hand, test cases have to be executed manually then BASIC-TEST-GENERATION is likely to produce a manageable set of test sequences. Alternatively, one can always start with BASIC-TEST-GENERATION since it captures the most critical interoperability behavior. If the system passes tests generated by BASIC-TEST-GENERATION, one can move to a broader coverage provided through COMPLETE-TEST-GENERATION.

# 5 ITIS: An Interoperability Test Generation Software System

The algorithms described in earlier sections are part of the ITIS (Interoperability Testing Intelligent System) project at Lucent Bell Laboratories. ITIS is a software tool for automated interoperability test generation. It is inexpensive and completely portable. ITIS is written in ANSI C and Tcl/Tk. We first describe its architecture and then report experimental results.

## 5.1 Architecture of ITIS

ITIS has a GUI (Graphic User Interface) for user input and for displaying test sequences. The workflow of ITIS is shown in Figure 17.

The input to ITIS is an Extended Finite State Machine description of the system behavior, of which we have complete information. ITIS will first do reachability analysis to convert the EFSM into an FSM, and then use different algorithms discussed in the above sections to generate test sequences.

## 5.2 Experiments

We have used ITIS to generate the interoperability test cases for the interoperability testing of traditional POTS(Plain Old Telephone Service) feature, authorized phone call feature and the signaling part H.323 of VoIP systems. We report experimental results.

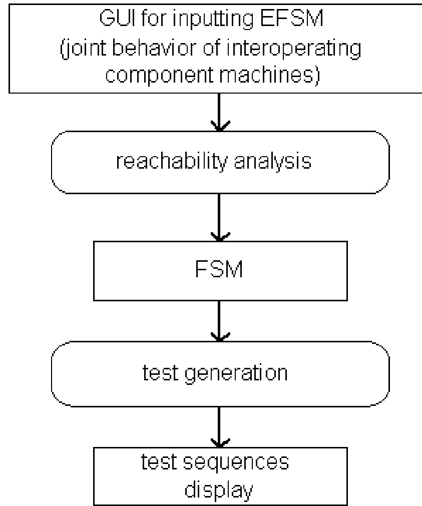


Figure 17: The Work Flow of ITIS for Interoperability Test Generation

#### 5.2.1 VoIP End User Interoperability Testing

VoIP systems need to support the traditional POTS phone calls over IP network. One typical configuration is two end users are still using traditional dumb phone sets to make phone calls while the trunk traffic is carried over IP network.

The EFSM for the POTS feature is shown in Appendix A. This machine has 21 states and 68 transitions, among which 24 are involved with interoperations and are colored black, and the others are local and are colored white. Figure 18 shows the sunny day part of the POTS EFSM. After generating reachability graph from this EFSM, ITIS shrinks each SCC in the reachability graph into a node and obtains a DAG. The DAG contains three SCC nodes, and only one of them contains more than one state. The number of generated test sequences by applying different algorithms is shown in Table 1.

Table 1: Test Cases for End User VoIP Testing

| Algorithms | acyclic paths | simple cycles | final tests |
|---|---|---|---|
| Exhaustive-Test-Generation | 950 | 430 | 950 |
| Complete-Test-Generation | 598 | 116 | 598 |
| Basic-Test-Generation | 14 | 4 | 14 |

The 14 test sequences generated by the algorithm Basic-Test-Generation are included in Appendix B. Each test sequence describes an interoperation scenario between end user A and B. For example, the second test sequence in Appendix B:

```
Off-hook A /DialTone A, Dial A B /AudibleRinging A,Ringing B,
Off-hook B /, On-hook A /, On-hook B /
```

describes the following call scenario: (1) end user A takes the phone off-hook and hears the dialtone; (2) A dials end user B's number, the phone on B side rings and at the same time A can also hear the ringing on B side; (3) B answers the phone but says nothing; (4) A puts the phone on-hook; (5) finally B puts the phone on-hook.

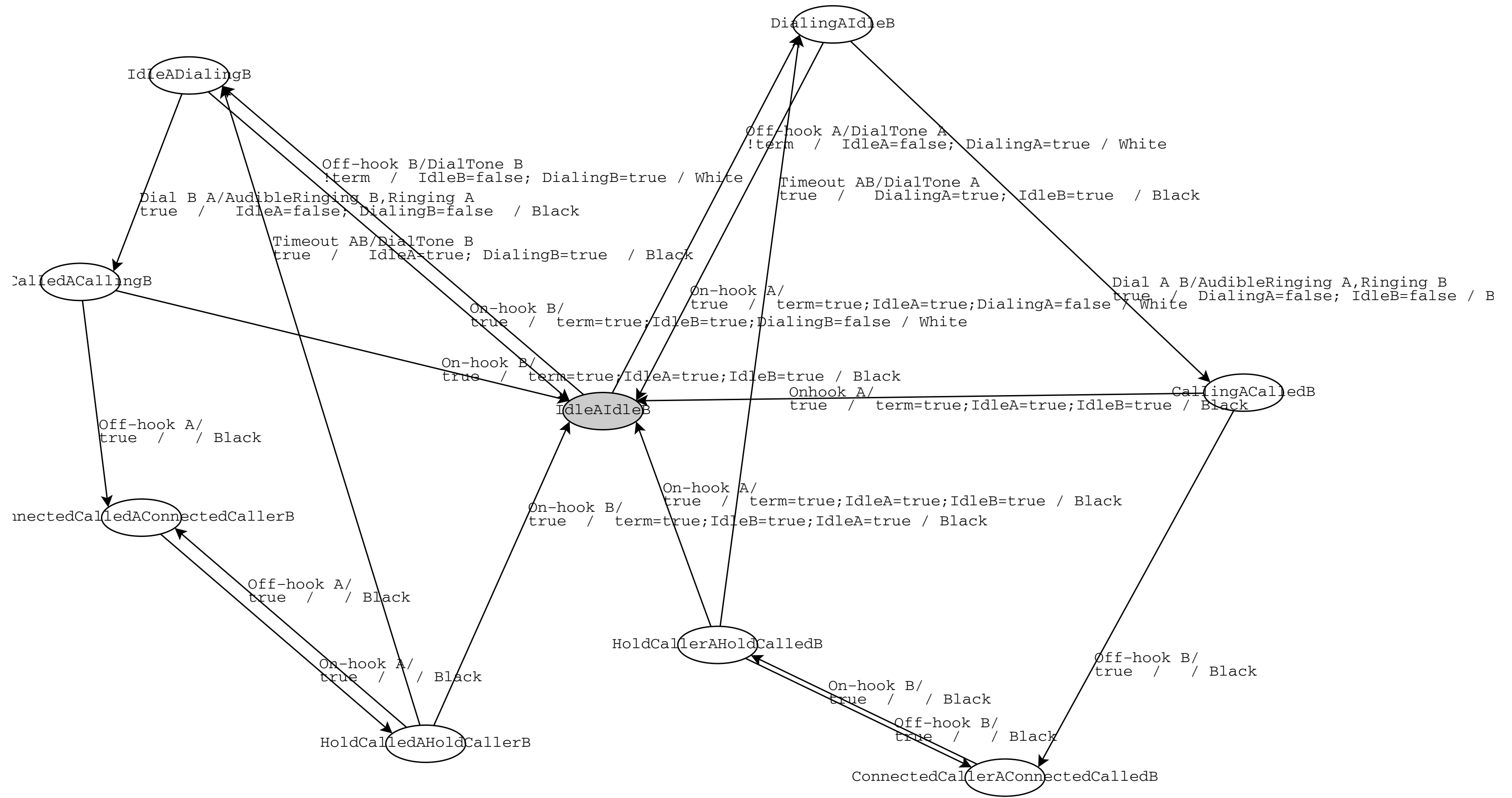


Figure 18: EFSM for End-user VoIP Testing

The coverage of these test sequences is complete with respect to the requirements for the system modeled by the EFSM. The requirements appear in Bellcore's LATA Switching Systems Generic Requirements [16], which includes hundreds of requirements on the behavior of a telephone switching system. These include many, such as trunking requirements and power requirements, that are not relevant to Internet telephony and are not modeled by the EFSM. However, there are 40 requirements that are applicable to Internet telephony. All of these are tested by the 14 test sequences generated by BASIC-TEST-GENERATION.

#### 5.2.2 Interoperability Testing of Authorized Phone Call Feature

We have also conducted interoperability test generation for the authorized phone call feature of VoIP systems. This feature enables or disables the caller to make some particular phone calls, for example, international long distance calls. The number of generated test sequence by applying different algorithms is shown in Table 2.

Table 2: Test Cases for Authorized Phone Call Feature of VoIP Systems

| Algorithms | acyclic paths | simple cycles | final tests |
|---|---|---|---|
| EXHAUSTIVE-TEST-GENERATION | 196 | 108 | 196 |
| COMPLETE-TEST-GENERATION | 56 | 12 | 56 |
| BASIC-TEST-GENERATION | 6 | 2 | 6 |

#### 5.2.3 Interoperability Testing of H.323 Signaling

The H.323 standard is actually both an architecture and a standard for Voice over IP. The H.323 standard is an umbrella standard, specifying the collection of standards to be used by the components of the H.323 architecture [11]. The architecture divides the required functionality among gatekeepers, gateways, multipoint control units (MCU's), and endpoints.

The *endpoints* are the initial creators and ultimate recipients of information streams. They need not comply with any part of the H.323 protocol, but if they do not, they must communicate with the network through a gateway that does comply with the H.323 protocol. An endpoint that complies with H.323 is called an H.323 terminal.

An H.323 *gateway* provides communication between an H.323 network and a non-H.323 device or network. It may also provide communication with a second H.323 network.

An H.323 *gatekeeper* handles address translations (for example, between telephone numbers and IP addresses), and controls access to the network. A gatekeeper controls admission to a zone. Gateways, MCU's, and H.323 terminals must register with a gatekeeper, if one is present in their zone, and get its permission to join a call. Optionally, call signaling may also go through the gatekeeper. Annex G of H.225.0 defines the communication between gatekeepers in different administrative domains [12]. It provides H.323 gatekeepers with the ability to perform address resolution and pricing exchange in a scalable manner that will allow large, international H.323 communication networks to be constructed.

A *multipoint control unit* provides conferencing capabilities in an H.323 network. It consists of a multipoint controller and (optionally) one or more multipoint processors. The multipoint controller handles the signaling that determines who participates in the conference and what information streams they send and receive. The multipoint processors provide centralized processing of

information streams (audio, video, or data) from various parties to a conference. The processing can be mixing, switching, or anything supporting the specific conference connection.

An H.323 terminal or gateway uses the *Registration, Admission, and Status (RAS)* channel (defined in the H.225.0 standard) to register with a gatekeeper, to locate other gateways and terminals, and to request permission to start or join a call. It may also route the call signaling through the gatekeeper. The call signaling protocol is a Q.931 subset, also defined in the H.225.0 standard.

The actual media connections in a call and their types are negotiated directly between gateways and/or endpoints using the H.245 standard. When a call is established via a gatekeeper, the signaling defined in H.245 might be exchanged via gatekeeper(s) as well as call setup signaling. Once the endpoints agree, the transport connections are also set up using the H.245 standard. Transport connections can use any of a number of standards. For audio, the G.711 uncompressed voice standard is required, but not always implemented. Other optional audio standards are G.723.1 and G.729. For video, the H.261 standard is required and H.263 is optional. For data, the T.120 standard is required.

Since there are many choices involved in building an H.323-compliant system, and ITU-T is continuously updating the H.323 protocol, it is in serious doubt whether different H.323-compliant systems would interoperate. To tackle this problem, vendors and industry consortiums are working together on defining interoperation profiles for vendors to follow to promote the interoperability between H.323 equipment from different vendors, these activities include the iNOW initiative spear-headed by leading VoIP vendors such as Lucent and Nokia and the interoperability profile proposed by IMTC (the International Multimedia Telecommunication Consortium, Inc.) [10].

On the other hand, the number of configurations to be tested for the interoperability is enormous. Even for domain experts, it is almost impossible to manually design a set of profiles that cover all the possible interoperations between H.323 devices. So automatic interoperability test generation with a desired coverage is needed.

We modeled the call signaling protocol by an EFSM. This machine has 419 states and 1290 transitions, among which 27 are involved with interoperations and are colored black. The number of generated test sequences by applying different algorithms is shown in Table 3.

Table 3: Test Cases for H.323 Call Signaling

| Algorithms | acyclic paths | simple cycles | final tests |
|---|---|---|---|
| EXHAUSTIVE-TEST-GENERATION | 140,390 | 0 | 140,390 |
| COMPLETE-TEST-GENERATION | 2,204 | 0 | 2,204 |
| BASIC-TEST-GENERATION | 77 | 0 | 77 |

We can see from this table that the COMPLETE-TEST-GENERATION algorithm generates 2204 test scenarios, which is only 1.57% of the original 140,390 test scenarios. This means 98.43% of the 140,390 scenarios generated using EXHAUSTIVE-TEST-GENERATION algorithm are redundant in the sense of only local transition coverage. Although COMPLETE-TEST-GENERATION significantly reduces the number of scenarios to only 2,204, it is still a huge number for manual test execution and only meaningful for automatic test execution. In contrast, the BASIC-TEST-GENERATION algorithm generates only 77 test scenarios which is only 0.05% of the original 140,390 scenarios, yet provides a basic coverage of all the black transitions in the signaling protocol EFSM. This is an affordable set of scenarios for manual test execution to start with. Also note that the generated transition diagram does not contain any loops, which is not uncommon in telephony.

# 6 Conclusion

Heterogeneity is one of the prominent features of networking systems, and interoperability is ubiquitous and has become a major hurdle for system reliability and quality of service. Interoperability testing is indispensable for the integration of reactive systems. Conventional conformance testing techniques do not apply since we want to test the system interfaces but not to check implementations vs. specifications.

We propose several algorithms for automatic generation of interoperability testing cases, incorporating a range of coverage and redundancy criteria. The tests generated are complete and non-redundant. They are applied to VoIP interoperability testing, and provide promising results.

The private sector spends over $2.2 trillion on information technology (IT) annually, and between 50% and 70% of all major projects fail, due to the software quality and interoperability problems, according to a study by the Gartner Group (Gartner Group Homepage). While fully recognized by the industry, integrated system interoperability testing provides a challenge and opportunities for researchers to study the essence of the problem and to invent novel technique for system interoperability testing and for improving the reliability of the integrated systems.

## Acknowledgment

An early version of the paper has appeared in Proc. PSTV-FORTE'2000. We thank the anonymous reviewers for their insightful and constructive comments.

## References

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Trans. on Communication*, 39(11):1604–15, 1991.

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

[3] N. Arakawa and T. Soneoka. A test case generation method for concurrent programs. In R.J. Heijink J. Kroon and E. Brinksma, editors, *Protocol Test Systems, IV*, pages 95–106. Elsevier Science Publisher B. V.(North-Holland), 1992.

[4] G. Bonnes. IBM OSI interoperability verification services. In *IFIP TC6 WG6.1 The 3rd International Workshop on Protocol Test System*, 1990.

[5] R. Castanet and O. Kone. Deriving coordinated testers for interoperability. In O. Rafiq, editor, *Protocol Test Systems, VI(C-19)*, pages 331–345. Elsevier Science Publisher B. V.(North-Holland), 1994.

[6] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on SE*, 4(3):178–187, 1978.

[7] J. Gadre, Rohre C, C. Summers, and S. Symington. A COS study of OSI interoperability. *Computer Standards and Interfaces*, 9(3):217–237, 1990.

[8] R. Hao. *Protocol Conformance and Interoperability Testing based on Formal Methods.* PhD thesis, Tsinghua University, P.R.China, 1997.

[9] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[10] IMTC. Standards-based IP telephony interoperability profile proposal. online URL = `www.imtc.org`, 2001.

[11] ITU-T. H.323 packet-based multimedia communications systems, November 2000.

[12] ITU-T. H.225.0 call signalling protocols and media stream packetization for packet-based multimedia communication systems, March 2001.

[13] S. Kang and M. Kim. Test sequence generation for adaptive interoperability testing. In *Proceeding of Protocol Test Systems, VIII*, pages 187–200, 1995.

[14] S. Kang and M. Kim. Interoperability test suite derivation for symmetric communication protocols. In *Proceeding of FORTE/PSTV'97*, 1997.

[15] S. Kang, J. Shin, and M. Kim. Interoperability test suite derivation for communication protocols. *Computer Networks*, 32(3):347–364, 2000.

[16] LATA. Switching systems generic requirements: Call processing. Technical Report GR-505-CORE, Bellcore, December 1997.

[17] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *The Proceedings of IEEE*, 84(8):1089–1123, August 1996.

[18] G. Luo, G. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on S.E.*, 20(2):149–162, 1994.

[19] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[20] O. Rafiq and R. Castanet. From conformance testing to interoperability testing. In *The 3rd International Workshop on Protocol Test Systems*, 1990.

[21] D. Sindu. Protocol testing: The first ten years, the next ten years. In *Protocol Specification, Testing and Verification*, pages 44–68, 1990.

[22] G. S. Vermeer and H. Blik. Interoperability testing: Basis for the acceptance of communicating systems. In *Protocol Test Systems, VI(C-19).* Elsevier Science Publisher B. V.(North-Holland), 1994.

[23] C. Viho, S. Barbin, and L. Tanguy. Towards a formal framework for interoperability testing. *Proceeding of FORTE'01*, 2001.

# A EFSM for End-user VoIP Testing

```
Initial state: IdleAIdleB
Initial value of variables:
boolean term=false;
boolean IdleA=true;
boolean IdleB=true;
boolean DialingA=false;
boolean DialingB=false;
transitions { 68
// Format
// From-State
//   {Input}/{Output}/{Predicates}/{Actions}/{Color} Next-State
//
 IdleAIdleB
  {Off-hook A}/{DialTone A}/{!term}/{IdleA=false;DialingA=true}/{White} DialingAIdleB
  {Off-hook B}/{DialTone B}/{!term}/{IdleB=false;DialingB=true}/{White} IdleADialingB
 BusyABusyB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} BusyADialingB
  {On-hook B}/{}/{}/{IdleB=true}/{White} BusyAIdleB
  {Timeout A}/{DialTone A}/{}/{DialingA=true}/{White} DialingABusyB
  {On-hook A}/{}/{}/{IdleA=true}/{White} IdleABusyB
 BusyADialingB
  {Dial B A}/{LineBusyTone B}/{}/{DialingB=false}/{Black} BusyABusyB
  {On-hook B}/{}/{}/{DialingB=false;IdleB=true}/{White} BusyAIdleB
  {Timeout A}/{DialTone A}/{}/{DialingA=true}/{White} DialingADialingB
  {On-hook A}/{}/{}/{IdleA=true}/{White} IdleADialingB
 BusyAIdleB
  {Off-hook B}/{DialTone B}/{}/{IdleB=false;DialingB=true}/{White} BusyADialingB
  {Timeout A}/{DialTone A}/{}/{DialingA=true}/{White} DialingAIdleB
  {On-hook A}/{}/{}/{term=true;IdleA=true}/{White} IdleAIdleB
 BusyALimboB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} BusyADialingB
  {On-hook B}/{}/{}/{IdleB=true}/{White} BusyAIdleB
  {Timeout A}/{DialTone A}/{}/{DialingA=true}/{White} DialingALimboB
  {On-hook A}/{}/{}/{IdleA=true}/{White} IdleALimboB
 CalledACallingB
  {Off-hook A}/{}/{}/{}/{Black} ConnectedCalledAConnectedCallerB
  {On-hook B}/{}/{}/{term=true;IdleA=true;IdleB=true}/{Black} IdleAIdleB
  {Timeout AB}/{}/{}/{IdleA=true}/{Black} IdleALimboB
 CallingACalledB
  {Off-hook B}/{}/{}/{}/{Black} ConnectedCallerAConnectedCalledB
  {Onhook A}/{}/{}/{term=true;IdleA=true;IdleB=true}/{Black} IdleAIdleB
  {Timeout AB}/{}/{}/{IdleB=true}/{Black} LimboAIdleB
 ConnectedCalledAConnectedCallerB
  {On-hook A}/{}/{}/{}/{Black} HoldCalledAHoldCallerB
  {On-hook B}/{}/{}/{IdleB=true}/{Black} LimboAIdleB
  {On-hook B}/{}/{}/{}/{Black} HoldCallerAHoldCalledB
  {On-hook A}/{}/{}/{IdleA=true}/{Black} IdleALimboB
 DialingABusyB
  {Dial A B}/{LineBusyTone A}/{}/{DialingA=false}/{Black} BusyABusyB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} DialingADialingB
```

```
  {On-hook B}/{}/{}/{IdleB=true}/{White} DialingAIdleB
  {On-hook A}/{}/{}/{DialingA=false;IdleA=true}/{White} IdleABusyB
 DialingADialingB
  {Dial A B}/{LineBusyTone A}/{}/{DialingA=false}/{Black} BusyADialingB
  {Dial B A}/{LineBusyTone B}/{}/{DialingB=false}/{Black} DialingABusyB
  {On-hook B}/{}/{}/{DialingB=false;IdleB=true}/{White} DialingAIdleB
  {On-hook A}/{}/{}/{IdleA=true; DialingA=false}/{White} IdleADialingB
 DialingAIdleB
  {Dial A B}/{AudibleRinging A,Ringing B}/{}/{DialingA=false;IdleB=false}/{Black}
                                                                  CallingACalledB
  {Off-hook B}/{DialTone B}/{}/{IdleB=false; DialingB=true}/{White} DialingADialingB
  {On-hook A}/{}/{}/{term=true;IdleA=true;DialingA=false}/{White} IdleAIdleB
 DialingALimboB
  {Dial A B}/{LineBusyTone A}/{}/{DialingA=false}/{Black} BusyALimboB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} DialingADialingB
  {On-hook B}/{}/{}/{IdleB=true}/{White} DialingAIdleB
  {On-hook A}/{}/{}/{IdleA=true;DialingA=false}/{White} IdleALimboB
 HoldCalledAHoldCallerB
  {Off-hook A}/{}/{}/{}/{Black} ConnectedCalledAConnectedCallerB
  {Timeout AB}/{DialTone B}/{}/{ IdleA=true; DialingB=true }/{Black} IdleADialingB
  {On-hook B}/{}/{}/{term=true;IdleB=true;IdleA=true}/{Black} IdleAIdleB
 HoldCallerAHoldCalledB
  {Off-hook B}/{}/{}/{}/{Black} ConnectedCallerAConnectedCalledB
  {Timeout AB}/{DialTone A}/{}/{DialingA=true;IdleB=true}/{Black} DialingAIdleB
  {On-hook A}/{}/{}/{term=true;IdleA=true;IdleB=true}/{Black} IdleAIdleB
 IdleABusyB
  {Off-hook A}/{DialTone A}/{}/{IdleA=false;DialingA=true}/{White} DialingABusyB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} IdleADialingB
  {On-hook B}/{}/{}/{term=true;IdleB=true}/{White} IdleAIdleB
 IdleADialingB
  {Dial B A}/{AudibleRinging B,Ringing A}/{}/{IdleA=false;DialingB=false}/{Black}
                                                                  CalledACallingB
  {Off-hook A}/{DialTone A}/{}/{IdleA=false; DialingA=true}/{White} DialingADialingB
  {On-hook B}/{}/{}/{term=true;IdleB=true;DialingB=false}/{White} IdleAIdleB
 IdleALimboB
  {Off-hook A}/{DialTone A}/{}/{IdleA=false;DialingA=true}/{White} DialingALimboB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} IdleADialingB
  {On-hook B}/{}/{}/{term=true;IdleB=true}/{White} IdleAIdleB
 LimboABusyB
  {Timeout A}/{DialTone A}/{}/{DialingA=true}/{White} DialingABusyB
  {On-hook A}/{}/{}/{IdleA=true}/{White} IdleABusyB
  {Timeout B}/{DialTone B}/{}/{DialingB=true}/{White} LimboADialingB
  {On-hook B}/{}/{}/{IdleB=true}/{White} LimboAIdleB
 LimboADialingB
  {Timeout A}/{DialTone A}/{}/{DialingA=true}/{White} DialingADialingB
  {On-hook A}/{}/{}/{IdleA=true}/{White} IdleADialingB
  {Dial B A}/{LineBusyTone B}/{}/{DialingB=false}/{Black} LimboABusyB
  {On-hook B}/{}/{}/{DialingB=false;IdleB=true}/{White} LimboAIdleB
 LimboAIdleB
  {Timeout A}/{DialTone A}/{}/{DialingA = true}/{White} DialingAIdleB
  {On-hook A}/{}/{}/{term=true;IdleA=true}/{Black} IdleAIdleB
  {Off-hook B}/{DialTone B}/{}/{IdleB=false;DialingB=true}/{White} LimboADialingB
}
```

## B Test Sequences Generated by BASIC-COVERAGE

1. Off-hook A /DialTone A, Dial A B /AudibleRinging A,Ringing B, Onhook A /
2. Off-hook A /DialTone A, Dial A B /AudibleRinging A,Ringing B, Off-hook B /, On-hook A /, On-hook B /
3. Off-hook A /DialTone A, Off-hook B /DialTone B, Dial A B /LineBusyTone A, On-hook A /, On-hook B /
4. Off-hook A /DialTone A, Off-hook B /DialTone B, Dial B A /LineBusyTone B, Dial A B /LineBusy-Tone A, On-hook A /, On-hook B /
5. Off-hook A /DialTone A, Off-hook B /DialTone B, Dial B A /LineBusyTone B, On-hook A /, On-hook B /
6. Off-hook A /DialTone A, Dial A B /AudibleRinging A,Ringing B, Off-hook B /, On-hook B /, On-hook A /
7. Off-hook A /DialTone A, Off-hook B /DialTone B, Dial A B /LineBusyTone A, Dial B A /LineBusy-Tone B, On-hook A /, On-hook B /
8. Off-hook B /DialTone B, Dial B A /AudibleRinging B,Ringing A, Timeout AB /, On-hook B /
9. Off-hook B /DialTone B, Dial B A /AudibleRinging B,Ringing A, On-hook B /
10. Off-hook B /DialTone B, Dial B A /AudibleRinging B,Ringing A, Timeout AB /, Off-hook A /Dial-Tone A, Dial A B /LineBusyTone A, On-hook A /, On-hook B /
11. Off-hook B /DialTone B, Dial B A /AudibleRinging B,Ringing A, Off-hook A /, On-hook B /, On-hook A /
12. Off-hook B /DialTone B, Dial B A /AudibleRinging B,Ringing A, Off-hook A /, On-hook A /, On-hook B /
13. Off-hook A /DialTone A, Dial A B /AudibleRinging A,Ringing B, Timeout AB /, On-hook A /
14. Off-hook A /DialTone A, Dial A B /AudibleRinging A,Ringing B, Timeout AB /, Off-hook B /Dial-Tone B, Dial B A /LineBusyTone B, On-hook A /, On-hook B /