Testing Self-Similar Networks

Constantinos Djouvas[1], Nancy D. Griffeth[1], Nancy A. Lynch[2]

[1] Lehman College, The City University of New York
[2] Massachusetts Institute of Technology

Abstract. Network testing presents different challenges from software testing. One challenge is that only a small number of networks, at best, can actually be tested, even when the goal is to test a class of networks. One solution is to select a representative network, which will display any faults present in any network of the class. This paper introduces the use of "self-similarity" to select such a network.

1 Introduction

When a vendor tests the network equipment it provides, the goal is to verify that the equipment works in an entire range of network topologies and configurations. Network tests performed by network users may also require verifying correctness of a class of networks. For example, ISP networks change continuously. Even small organizations add new hosts regularly. Anyone may add or swap in new network equipment as new technologies or higher bandwidths become available. The remaining equipment must continue working as expected.

This problem motivates the question of how to choose a network for testing, when the real goal is to verify that an entire class of networks works. The central goal of this work is to find a single representative of a class of networks, whose correctness implies the correctness of the class. This paper investigates the use of a subnetwork that is common to all of the networks in the class and whose behavior looks like the behavior of any of the networks. When a subnetwork has this property, we call the networks "self-similar" because each is similar to a substructure of itself.

Perhaps the best-known example of this is the use of proxies in a network. A Web server behind a proxy looks like a Web server to a client; similarly, a proxy and client together look like a client to the Web server.

2 Related Work

Protocol conformance testing solves the network testing problem by verifying that the implementation of each network device conforms to the required protocol standards. If the protocol standards guarantee that the network has the required properties, then protocol conformance testing shows that the network has the required properties. An excellent review of protocol conformance testing appears in [7].

This approach presupposes a validated formal model of each protocol and proofs that the models have the required properties. In practice, Internet standards have rarely been formalized and the job of developing formal proofs has barely begun. Some standards, such as BGP, have actually been shown to have serious problems [4]. Others, such as DHCP, work correctly with high probability, but will behave incorrectly on rare occasions [2]. However, networks may still have many desirable properties, and methods of verifying these are required.

A different approach to network testing is to extend protocol conformance testing to "network interoperability testing," as in [3, 5]. This approach treats the network as a black box, whose external behavior is known but whose internal behavior cannot be observed. The test methodology requires coverage of all possible sequences of visible actions.

However, the problem of determining what network to test has not yet been addressed. The central contribution of the paper is the method for choosing the network to be tested, by finding a common substructure of all the networks that behaves like each of the networks. This is a reasonable approach, since most networks are built from components that behave like the network as a whole.

Section 6 presents a case study, containing two proofs of the self-similarity of learning bridges.

3 The I/O Automaton Model

To analyze the properties of networks, we use the I/O automata model [8], which models network components as automata and their interactions as shared actions of the automata.

The model provides the formal framework for saying that one network behaves like another. Automaton A is said to *implement* automaton B if all externally visible behaviors of A are also externally visible behaviors of B.

An important technique for proving that one automaton implements another is *simulation*. One automaton is said to simulate another if there is a *simulation relation* (defined in Section 6) relating the states of the first to the states of the second.

A self-similar automaton is an automaton A that can be replicated and connected to itself via a channal to form a new automaton that implements the original automaton A. Another important concept is self-similar properties, which are properties of an automaton that are preserved by such a composition.

We review the definition of I/O Automata briefly here. For details, see [8].

Definition 1. *An I/O automaton consists of the following components:*

- sig(A)*, a signature, consisting of three disjoint sets of actions: the input actions* in(A)*, output actions* out(A)*, and internal actions* int(A)*. Output and internal actions are* locally controlled*; input actions are controlled by an automaton's environment. The set of all actions in the signature is denoted* acts(A)*.*
- states(A)*, a nonempty, possibly infinite set of states.*

- start(A), *a nonempty subset of* states(A), *called the* start states.
- trans(A), *a state-transition relation, contained in* $states(A) \times acts(sig(A)) \times states(A)$. *We require that for each state* s *and input action* π, *there is a transition* (s, π, s').
- tasks(A), *a* task partition, *which is an equivalence relation on the locally controlled actions of* A *and which has at most countably many equivalence classes.*

An *execution* of an I/O automaton is a sequence $s_0, \pi_1, s_1, ..., s_{n-1}, \pi_n, s_n$ where s_0 is a start state and (s_{i-1}, π_i, s_i) is a transition for each $i \geq 1$. An execution can be finite or infinite. The set of executions of I/O automaton A is denoted as *execs(A)*. We define *traces(A)* as the set of all sequences $\pi_1, \pi_2, ..., \pi_n, ...$ obtained by removing the states from a sequence in *execs(A)*. Traces capture the notion of externally visible behavior.

A *trace property* of an automaton A is a property that holds for all traces of A.

The composition operation allows the construction of complex I/O automata by combining primitive I/O automata. To compose automata, we consider actions with the same signature in different automata to be the same action, and when any component performs an action π, it forces all the components having the same action to perform it. Thus for composition to work, automata must be *compatible*.

Definition 2. *A countable collection* $\{S_i\}_{i \in I}$ *is* compatible *if for all* $i, j \in I, i \neq j$, *all of the following hold:*

1. $int(S_i) \bigcap acts(S_j) = \phi$
2. $out(S_i) \bigcap out(S_j) = \phi$
3. *No action is contained in infinitely many sets* $acts(S_i)$

Definition 3. *Given a compatible collection* $\{A_i\}_{i \in I}$ *of automata, the* composition $A = \Pi_{i \in I} A_i$ *is formed by the following rules:*

- sig(A) *is defined by:*
  - $out(A) = \bigcup_{i \in I} out(A_i)$
  - $int(A) = \bigcup_{i \in I} int(A_i)$
  - $in(A) = \bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i)$
- $states(A) = \Pi_{i \in I} states(A_i)$.
- $start(A) = \Pi_{i \in I} start(A_i)$.
- trans(A) *is the set of triples* (s, π, s') *such that for all* $i \in I$, *if* $\pi \in acts(A_i)$ *then* $(s_i, \pi, s_i') \in trans(A_i)$
- $tasks(A) = \bigcup_{i \in I}(A_i)$

We denote a finite composition of automata $A_1, ..., A_n$ by $A_1 \parallel ... \parallel A_n$.

After composing I/O Automata, we may want to hide actions used for communication between components, turning them into internal actions of the composed automaton. The operation $ActHide_{\Phi}(A)$ for $\Phi \subset out(A)$ is defined as the automaton obtained from A by changing each output action in Φ to an internal action.

4 Self-Similarity

The problem that motivates this paper is the problem of finding a representative network to test instead of testing all members of a class. If there is a subnetwork N that looks like the entire network, then the smallest such subnetwork is an obvious candidate. This is because we can test N by itself to determine the properties of the entire network.

Defining Self-Similarity Because we are interested in networks, we consider only automata with output actions named *send* and input actions named *receive*. These automata are parameterized by the number of interfaces they have on the network. Each *send* action is associated with one of the interfaces, and sends the message out the interface. Each *receive* action is also associated with an interface and receives a message arriving on the interface.

An automaton with n interfaces has a signature containing at least the following actions:

$send(m : Message, i : Int)$, where $1 \leq i \leq n$
$receive(m : Message, i : Int)$, where $1 \leq i \leq n$

$Message$ is the set of possible messages over the interface.

To combine automata, we use a channel automaton $Channel(a, b)_{i,j}$, as described in [8], which joins interface i of automaton a to interface j of automaton b. When there are only two automata in the composition, we write $Channel_{i,j}$ This automaton has input actions $send(m, i)_a$ and $send(m, j)_b$ and output actions $receive(m, i)_a$ and $receive(m, j)_b$. We assume a reliable, FIFO channel automaton, guaranteeing that messages are delivered reliably, in-order, and with no duplication.

Suppose that an automaton N is parameterized by the number of interfaces n. Then we say that $N(n)$ is self-similar if

$ActHide_{\Phi}(traces(N(n)) \parallel Channel_{i,j} \parallel traces(N(n))) \subseteq traces(N(2n-2))$,
where $\Phi = \{send(m, i)_a, send(m, j)_b, receive(m, i)_a, receive(m, j)_b\}$.

In other words, the externally visible actions of the composition of $N(n)$ with itself, using a channel connecting interfaces i and j, looks like a single automaton $N(2n-2)$, ignoring actions on the interfaces connecting the automata.

We also define self-similarity for properties of networks, since it may be easier to establish self-similarity of interesting properties than for entire automata. We say that a trace property T is *self-similar* if the network $N(n) \parallel Channel_{i,j} \parallel N(n)$ has property T whenever network $N(n)$ has property T. Thus test results concerning a self-similar property of a network $N(n)$ can be generalized to apply to larger networks.

Using Self-Similarity in Testing By the definition of self-similarity, correct behavior of a self-similar network N implies correct behavior of a larger network

composed of multiple instances of N. Perhaps more important, if there are bugs in the larger network, they will also be found in N.

There are two approaches that allow us to take advantage of self-similarity to reduce the size of the network under test. First, we can define a self-similar model of the network that has the properties of interest in the test effort . Second, we can test directly whether the properties of interest are self-similar. The case study of learning bridges in Section 6 follows the first approach. We apply the second approach in a longer version of this paper at [1].

Self-Similar Models This approach requires a generalized model M of the network that is self-similar. If the specification holds for M and if we establish by testing that N implements M, we can use the test results as if N itself were self-similar. The following theorem is the basis of this claim.

Theorem 1 *If $M(n)$ is self-similar and if*

$$traces(N(n)) \subseteq traces(M(n)) \subseteq traces(S)$$

then

$$ActHide_{\Phi}(traces(N(n)) \parallel Channel_{i,j} \parallel traces(N(n))) \subseteq traces(S).$$

This theorem says that given a network $N(n)$ and a self-similar model $M(n)$, where $M(n)$ implements S and $N(n)$ implements $M(n)$, we can conclude that two composed instances of network $N(n)$ implements S. By induction, we can compose any number of instances of $N(n)$ and still conform to S.

Proof. Follows immediately from the definitions.

Self-Similar Properties If self-similar trace properties S and T both hold for a network N, then clearly so does the trace property $S \wedge T$. This can be used to show that if a complex network requires that a number of properties $T_1, ..., T_n$ be true, it is necessary to prove only that each property is self-similar, rather than trying to prove all of them at once.

In general, we won't be able to show self-similarity of every network property that we are interested in. However, we may be able to show self-similarity of a significant subset, so that testing of those properties can be carried out on a smaller network.

5 Learning Bridges

A learning bridge incorporates a forwarding algorithm and a spanning tree algorithm. In this section we give a brief description of each algorithm as it is relevant to this paper.

Learning Bridge Algorithm Learning bridges interconnect separate IEEE 802 LAN segments into a single bridged LAN.

A learning bridge relays and filters frames "intelligently" between the separate LAN segments [6]. Initially, the bridge forwards every frame that arrives at a port out every other port. Also when a frame arrives at a port, it "learns" the relationship between the source address and the port. It records this relationship in a $filteringdatabase$. Once the address-to-port relationship has been learned, any frame sent to that address will be sent out the corresponding port.

Spanning tree algorithm The Spanning Tree Algorithm converts an arbitrary topology to a tree. This eliminates cycles from the network so that frames won't be forwarded forever. We assume that the following important property is enforced by the Spanning Tree Algorithm, as required by the standard:

> The Spanning Tree Algorithm creates a single spanning tree for any bridged LAN topology.

Thus, there is a unique path between any two hosts and cycles are eliminated.

6 Self-Similarity of Learning Bridges

This section presents a proof that learning bridges are self-similar. The proof is based on a generalized model of learning bridges. The self-similarity property allows a tester to use Theorem 5.1 to justify testing only a single learning bridge to verify an entire network[3].

Learning bridge operation can be described briefly as "send incoming frames out all ports until the correct port is known; then send out the correct port only."

A network of bridges that conform exactly to this requirement is not self-similar. Consider the following example:

> Bridges A and B are connected to each other, with A preceding B in a path from S (source) to D (destination). Suppose that the filtering database in A does not contain an entry for D, while the filtering database of B does contain an entry for D. Then if a message initiated is from S to D, A will forward this message to every active port but B will forward it only to the correct port.
>
> Now suppose we compose A and B to one bridge AB. If our model included the requirement mentioned above, on external observer would expect the trace of AB to have only one outgoing message having as destination D. But this will not happen. Instead the message will forwarded to all ports that have been inherited by A and to a single port inherited by B, the same one as the B would have forwarded the message to.

[3] Note that we address only the forwarding of messages in this paper, not the construction of the spanning tree.

So we define a generalized model, which requires only that the bridge copies each message to the "correct port", and perhaps other ports as well. By "correct port" P, we mean that P is the port through which the destination is reachable. The learning bridge implements this by using a *filtering database* to record the source address of each arriving message along with the port at which it arrived. All subsequent messages sent to that address will be copied to the corresponding port (and possibly other ports). If no message has been received from the destination address, the *filtering database* does not have an entry for the address, and the bridge forwards the message to all ports.

The Generalized Model Each bridge has four actions: input action *receive*, output action *send*, and internal actions *copyIn*, *copyOut*, and *delete*. It has a filtering database, an input and output buffer for each port, and an array of queues corresponding to each pair of input port and output port. The array entry $queue[i, j]$ is a queue of messages that arrived at port i and destined to be sent out port j.

The *receive* action adds received messages to the input buffer for the arrival port and updates the *filtering database*. The *send* action sends the first message in a port's output buffer to the channel connected to the port. The *copyIn* action copies a message from an input buffer to the end of all the internal queues for the input port; *copyOut* copies a message from one internal queue to an output buffer. Finally, the *delete* action can delete a random message m from an internal queue, if the correct port is known at the time of the delete and the queue doesn't correspond to the correct port for the message[4].

automaton $bridge(n : Int)_i$

signature

input
$receive(m, inPort)_i$
output
$send(m, outPort)_i$
internal
$copyIn(m, inPort)$
$copyOut(m, inPort, outPort)_i$
$delete(m, inPort, outPort)_i$

states

inbuf, an array of input buffers, indexed by {1, ..., n }, one for each port
outbuf, an array of output buffers (FIFO queues) indexed by {1, ..., n}, one for each port, initially all *empty*.

[4] The *delete* action is one of many ways to model the possibility that a bridge is allowed to forward a message out a port other than the correct one. It works by nondeterministically removing messages from queues that don't lead to the correct port.

queue, an array of FIFO queues indexed by {1, ..., n} × {1, ..., n}
one for each pair of ports, initially all *empty*.
filterDB, a mapping of message destinations to ports of $bridge_i$ indexed by {1, ..., n}, initially all *nil*.

transitions

$receive(m, inPort)_i$
effect
add m to *inbuf(inPort)*
set $filterDB(m.src) := inPort$

$send(m, outPort)_i$
precondition
m first element on *outbuf(outPort)*
effect
remove first element from *outbuf(outPort)*

$copyIn(m, inPort)$
precondition
m is the first element on $inbuf[inPort]$
effect
add m to $queue[inPort, i]$ for all $i \neq inPort$
remove m from $inbuf[inPort]$

$copyOut(m, inPort, outPort)_i$
precondition
m first element on *queue[inPort,outPort]*
effect
add m to *outbuf[outPort]*
remove m from $queue[inPort, outPort]$

$delete(m, inPort, outPort)_i$
precondition
m is in the queue $queue[inPort, outPort] \wedge$
$filteringdb[dest(m)] \neq nil \wedge filteringdb[dest(m)] \neq outPort$
effect
remove m from $queue[inPort, outPort]$

We assume that there are a finite number of active ports in any bridge and that the spanning tree algorithm determines which ports are active.

Composition of Bridges In this section we describe the composition of two learning bridges. Remember that we assume that the Spanning Tree Protocol has been run to completion by all the bridges in the network and that there are no failures. Because of this, there is only one active path between any two bridges.

Let $bridge_1$ and $bridge_2$ be two learning bridges running the IOA defined above. We use the convention that port i is a port of $bridge_1$ and j is a port of $bridge_2$. Without loss of generality, we assume that port i_0 of $bridge_1$ is connected with the port j_0 of $bridge_2$ through $Channel_{i_0,j_0}$. Because of the Spanning Tree Protocol, these are the only active ports connecting $bridge_1$ and $bridge_2$.

Let $bridge_c$ be the result of renaming ports of $bridge_2$ to $n+1, ..., 2n$ to avoid conflict with port numbers of $bridge_1$, then composing $bridge_1$ and $bridge_2$ with a connecting channel, and finally hiding the *send* and *receive* actions on the channel between them:

$bridge_c = ActHide_{\Phi}(bridge_1 \parallel Channel_{i_0,j_0} \parallel bridge_2)$ and
$\Phi = \{send(m, i_0)_1, receive(m, i_0)_1, send(m, j_0)_2, receive(m, j_0)_2\}$.

The goal is to show that $bridge_c$ is essentially the same as a single bridge, which we will call $bridge_p$, running the learning bridge IOA. $bridge_p$ must have the same number of ports as $bridge_1$ and $bridge_2$ together, minus the two connected ports. Thus if $bridge_1$ and $bridge_2$ each have n active ports, $bridge_p$ has $2n - 2$ active ports.

Port i of $bridge_p$ with $1 \leq i \leq n$, is connected to the same channel as the corresponding port i of $bridge_1$. Similarly port j of $bridge_p$, with $n+1 \leq j \leq 2n$, is connected to the same channel as the corresponding port j of $bridge_2$.

Finally, the input and output actions of $bridge_p$ are renamed so that the actions on port i, $1 \leq i \leq$n, are $receive(m, i)_1$ and $send(m, i)_1$ (instead of $receive(m, i)_p$ and $send(m, i)_p$); similarly, actions on port j, $n+1 \leq j \leq 2n$, are $receive(m, j)_2$ and $send(m, j)_2$.

Simulating a bridge with a composition of bridges We use an important theorem about IOA to show the equivalence of $bridge_c$ to $bridge_p$. The theorem says that if there is a simulation relation (defined below) from an IOA A to an IOA B, then $traces(A) \subseteq traces(B)$.

Definition 4. *A* simulation relation *from an IOA A to an IOA B is a relation $R \subseteq states(A) \times states(B)$. Define $f : states(A) \rightarrow \mathcal{P}(states(B))$ by $f(s) = \{t | (s, t) \in R\}$. To be a simulation relation, R must satisfy the following conditions:*

1. *If $s \in start(A)$, then $f(s) \cap start(B) \neq \phi$ (start condition).*
2. *If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and $(s, \pi, s') \in trans(A)$, then there is an execution fragment α of B starting in state u and ending in some state $u' \in f(s')$ such that $trace(\alpha) = trace(\pi)$ (step condition).*

We define a relation from $bridge_c$ to $bridge_p$ and prove that it is a simulation relation. This gives us the desired result.

Theorem 1. *The learning bridge automaton* bridge(n) *is self-similar.*

Proof. Let s be a state of $bridge_c$ and t be a state of $bridge_p$. We use dot notation to denote a state variable in a bridge, e.g., $s.filterDB_1$ is the value of the filtering database of $bridge_1$ in state s of $bridge_c$.

The pair (s, t) belongs to the relation R if:

Condition 1 *The filtering database of $bridge_p$ contains the same entries as the union of the filtering databases of the two component bridges of $bridge_c$, excluding the entries for the internal ports:*

$$t.filterDB = s.filterDB_1 \cup s.filterDB_2 - \{\langle addr, port\rangle | port \in \{i_0, j_0\}\}$$

Condition 2 *The output buffer for each port of $bridge_p$ contains the same messages as the output buffer of the corresponding port of $bridge_c$:*

$t.outbuf[i] = s.outbuf[i]_m$ *for* $i \in ports_1 \bigcup ports_2 - \{i_0, j_0\}$, *and the value* $m \in \{1, 2\}$ *depends on the value of* i.

There are no buffers in $bridge_p$ corresponding to i_0 and j_0. These buffers in $bridge_c$ may contain any messages consistent with the next condition.

Condition 3 *The input buffer for each port of $bridge_p$ contains the same messages as the input buffer of the corresponding port of $bridge_c$:*

$t.inbuf[i] = s.inbuf[i]_m$ *for* $i \in ports_1 \bigcup ports_2 - \{i_0, j_0\}$ *and the value of* $m \in \{1, 2\}$ *depends on the value of* i.

Condition 4 *The internal array of message queues t.queue corresponds to the combined arrays $s.queue_1$ and $s.queue_2$ as follows:*

- $t.queue[i, i'] = s.queue[i, i']_1$ *if* $i, i' \in ports_1, i, i' \neq i_0$
- $t.queue[j, j'] = s.queue[j, j']_2$ *if* $j, j' \in ports_2, j, j' \neq j_0$
- $t.queue[i, j]$ *is a concatenation of the following queues for* $i \in ports_1, j \in ports_2$, *with* $i \neq i_0, j \neq j_0$*:*

  $s.queue[j_0, j]_2$,$s.outbuf[j_0]_2$,$s.queue_{j_0,i_0}$,$s.inbuf[i_0]_1$,$s.queue[i, i_0]_1$
- $t.queue[j, i]$ *is defined symmetrically for* $i \in ports_1, j \in ports_2$, *with* $i \neq i_0, j \neq j_0$*:*

To show that the above relation is a simulation relation, we must prove two conditions, the start condition and the step condition. The former is trivial because all the states of both bridges are initially empty. The latter condition requires the proof that the states of $bridge_p$ and $bridge_c$ correspond after each action. First we prove state correspondence for the filtering databases.

State Invariant 1 *In all reachable states of the composed IOA, the filtering database of $bridge_c$ corresponds to the filtering database of $bridge_p$ as defined by the simulation relation.*

The proof is by induction of the length of an execution. The result is clear if a message is forwarded out only ports of the bridge at which it arrived. It's less obvious when a frame arrives at one bridge and is forwarded out the second bridge. In this case, the filtering databases of both $bridge_1$ and $bridge_p$ are updated on receipt of the message with the relationship between the arrival port and the source address. Later, the filtering database of $bridge_2$ is updated to show the path to the source goes through $bridge_1$. Since the simulation relation refers only to the entry in $bridge_1$ and ignores the entry in $bridge_2$, itis preserved in this case (as well as all others).

To show that input buffers, output buffers, and internal queues correspond after each action, we consider all actions π. The following table summarizes all the possible actions of $bridge_c$, the corresponding execution fragment of $bridge_p$ and the trace, which is the same for both bridges.

	Action of $Bridge_c$	Execution fragment of $Bridge_p$	Trace
1	$receive(m,i)_1, i \neq i_0$	$receive(m,i)_1$	$receive(m,i)_1$
2	$receive(m,j)_2, j \neq j_0$	$receive(m,j)_2$	$receive(m,j)_2$
3	$receive(m,i_0)_1$	λ	λ
4	$receive(m,j_0)_2$	λ	λ
5	$send(m,i)_1, i \neq i_0$	$send(m,i)_1$	$send(m,i)_1$
6	$send(m,j)_2, j \neq j_0$	$send(m,j)_2$	$send(m,j)_2$
7	$send(m,i_0)_1$	λ	λ
8	$send(m,j_0)_2$	λ	λ
9	$delete(m,i,i')_1, i' \neq i_0$	$delete(m,i,i')_p$	λ
10	$delete(m,j,j')_2, j' \neq j_0$	$delete(m,j,j')_p$	λ
11	$delete(m,i,i_0)_1$	Sequence $delete(m,i,j)_p$ for $j \in ports_2, j \neq j_0$	λ
12	$delete(m,j,j_0)_2$	Sequence $delete(m,j,i)_p$ for $i \in ports_1, i \neq i_0$	λ
13	$copyIn(m,i)_1, i \neq i_0$	$copyIn(m,i)_p$	λ
14	$copyIn(m,j)_2, j \neq j_0$	$copyIn(m,j)_p$	λ
15	$copyIn(m,i_0)_1$	λ	λ
16	$copyIn(m,j_0)_2$	λ	λ
17	$copyOut(m,i,i')_1, i' \neq i_0$	$copyOut(m,i,i')_p$	λ
18	$copyOut(m,j,j')_2, j' \neq j_0$	$copyOut(m,j,j')_p$	λ
19	$copyOut(m,i,i_0)_1$	λ	λ
20	$copyOut(m,j,j_0)_2$	λ	λ

Table 1: Correspondence between actions of $Bridge_c$ and $Bridge_p$

A simple case analysis establishes the result.

7 Conclusions

In this paper, we have shown that the self-similarity of network devices and their properties provides a powerful tool for reducing the size of a network testing effort. All networks in a class of self-similar networks can be tested by testing the smallest self-similar subnetwork. This reduces to one the number of networks to be tested while minimizing the size of the network.

A case study of the self-similarity of learning bridges illustrates one approach to using self-similarity in network testing. This approach uses a self-similar network model that captures the behaviors that the network must implement. A longer version of this paper [1] shows how to define required properties of learning bridges and prove self-similarity.

Additional work is needed to identify other self-similar networks and important self-similar properties of networks. Another line of investigation is to determine how to evaluate the coverage of a set of tests for a network and to develop ways to measure the level of confidence we have that a network works, given a test suite for the network.

References

1. Constantinos Djouvas, Nancy Griffeth, and Nancy Lynch. Using self-similarity for efficient network testing, September 2005.
2. Ralph Droms. RFC 2131: Dynamic host configuration protocol, March 1997.
3. Nancy Griffeth, Ruibing Hao, David Lee, and Rakesh Sinha. Integrated system interoperability testing with applications to voip. In *Proceedings of FORTE/PSTV 2000*, Pisa, Italy, October 2000.
4. Timothy G. Griffin and Gordon T. Wilfong. An analysis of BGP convergence properties. In *Proceedings of SIGCOMM*, pages 277–288, Cambridge, MA, August 1999.
5. Ruibing Hao, David Lee, Rakesh K. Sinha, and Nancy Griffeth. Integrated system interoperability testing with applications to voip. *IEEE/ACM Trans. Netw.*, 12(5):823–836, 2004.
6. IEEE standard for local and metropolitan area networks: Media access control (MAC) bridges, June 2004.
7. D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
8. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., March 1996.