Testing Self-Similar Networks

Constantinos Djouvas [1,2]

Computer Science
The Graduate Center, The City University of New York
New York, NY, USA

Nancy D. Griffeth [3]

Mathematics and Computer Science
Lehman College, The City University of New York
Bronx, NY, USA

Nancy A. Lynch [4]

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Boston, MA, USA

Abstract

A hard problem in network testing is verifying the correctness of a class of networks, as well as the actual networks under test. In practice, at most a few networks (sometimes only one) are actually tested. Thus an important question is how to select one or more networks that are sufficiently representative to apply the results to a class of networks. We present a model-based technique for selecting a representative network. The central theorem establishes that the representative network displays any faults present in any network of the class. This paper introduces the concept of "self-similarity," which is used to select the network, and presents the results of an experiment in testing one class of networks.

Key words: Testing, verification, model-checking, I/O automata, parameterized processes.

[1] This work supported in part by DARPA/AFOSR MURI Award F49620-02-1-0325, MURI AFOSR Award SA2796PO 1-0000243658, NSF Award CCR-0121277, NSF Award NeTS-0435130, USAF,AFRL Award FA9550-04-1-012, DARPA Air Force (STTR) contract FA9550-04-C-0084, and Cisco URP Award.

[2] Email:cdjouvas@gc.cuny.edu
[3] Email:ndgriffeth@lehman.cuny.edu
[4] Email:lynch@csail.mit.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

When a vendor tests its own network equipment, the goal is to verify that the equipment works for a range of network topologies and configurations. Network users may also need to verify 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 observation motivates the problem of how to choose networks for testing, when the real goal is to verify that a 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, the class is called "self-similar". A tester can also use a weaker condition, self-similiarity with respect to a property, to establish that the network conforms to a single requirement imposing that property. In the latter case, it is necessary only to state the property and prove that if a network conforms to it, any composition consisting of multiple copies of the network also conforms to it.

Internet protocols are designed in such a way that many properties of Internet protocols are self-similar. Proxies are a well-known example of self-similarity.. 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. Switching and routing algorithms are designed to hide the structure of the networks they support, so that the behavior of a single switch or router can look like the behavior of a larger network. DHCP failover servers are designed to look like a single, highly-reliable server.

In this paper, we address how to reduce the size and complexity of the network under test without reducing the test coverage. The central contribution of the paper is a method for choosing the network to be tested, by finding a common substructure of all the networks that behaves like each of the networks. Definitions and the basic theorem are presented in sections 3 and 4. In sections 5 and 6, we describe a case study, in which we modeled the forwarding function of learning bridges and proved self-similarity. In section 7, we describe a experiment on network testing, in which three tests were run, each consisting of a different learning bridge configuration.

2 Related Work

The general question is how to identify a small test that will verify correctness an entire class of networks. Protocol conformance testing solves the problem by verifying that the implementation of a single network device conforms to

the required protocol standards. Then, assuming that the protocol standards guarantee that the network has the required properties, protocol conformance testing shows that a network consisting of any number of interconnected devices has the required properties. An excellent review of protocol conformance testing appears in [**?**].

However, conformance testing 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 been shown to have serious problems [**?**]. Others, such as DHCP, work correctly with high probability, but behave incorrectly on rare occasions [**?**]. Nonetheless, these protocols have desirable properties, and it is important to be able to verify desired properties for specific implementations.

A different approach to network testing is to extend protocol conformance testing to "network interoperability testing," as in [**?**,**?**]. 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 developing a formal model of the network's external behavior to generate tests that cover all possible sequences of visible actions. As noted above, models of networks and protocols are rarely available and time-consuming to develop.

Descriptions of industrial network testing based on actual practice appear in [**?**,**?**]. Buchanan[**?**] presents *ad hoc* and common-sense approaches to testing networks. While these techniques are valuable, it is hard to analyze and optimize them. Griffeth[**?**] presents a case study of interoperability testing in an industrial lab. A study of time required in each stage of testing for four test projects (one Voice over IP, two Data Center, and one Network Management) at the Lucent Next Generation Networking Interoperability Lab (NGN) shows that the overwhelming majority of time is spent on test network setup[**?**]. Figure 1 summarizes the results from this study along with the results of the current experiment. The hypothesis of this paper is that testing only one configuration will result in significant savings in time since only one network needs to be set up.

A similar problem, that of verifying a parameterized collection of processes, has been addressed in model-checking. Wolper and Lovinfesse[**?**] and Kurshan and McMillan[**?**] have shown how to apply induction to verify a parameterized collection of processes. Their results apply to collections of identical processes, which is not strictly required in this paper. Also, they require bisimulation of the processes; the present result requires only containment. They also require the tester to devise an invariant. This is not necessary for this work. In the simplest case, the tester must identify only that a requirement impose a self-similar property. Other work on reducing the complexity of model-checking

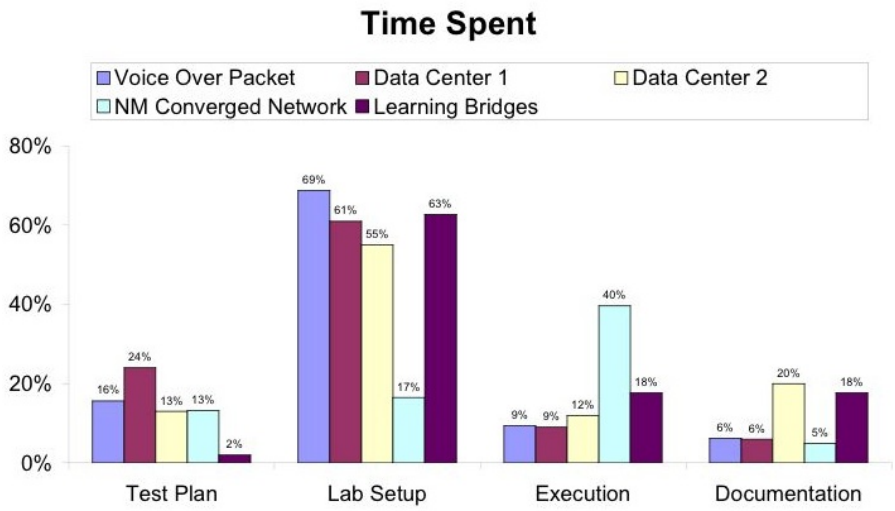

Fig. 1. Time Required in Stages of Network Testing. The shaded bars show the results of the NGN study. The only test in which test lab setup did not take most of the time was a test of network configuration tools, i.e., network setup. The cross-hatched bars show the results of the experiment reported in this paper.

3 The I/O Automata Model

To analyze network properties, we use the I/O automata modeling framework [**?**], which models network components as automata and their interactions as shared actions of the automata. The model provides a formal basis 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*. Automaton A is said to simulate B if there is a *simulation relation* (defined in Section 6) relating the states of A to those of B. A *self-similar* automaton A is one that can be replicated and connected to itself via a channel to form a new automaton that implements the original automaton A. Another important concept is that of a *self-similar property*, which is a property of an automaton that is preserved by such a composition.

We review the definition of I/O Automata briefly; for details, see [**?**].

Definition 3.1 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 the 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 having at most countably many equivalence classes.

An *execution* of I/O automaton A 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 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 and internal actions 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 treat actions with the same signature in different automata as the same action, and when any component performs an action π, it forces all the components having the same action to perform it. To compose automata, they must be *compatible*:

Definition 3.2 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) $out(S_i) \bigcap out(S_j) = \phi$, and (3) No action is contained in infinitely many sets $acts(S_i)$.

Definition 3.3 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: $out(A) = \bigcup_{i \in I} out(A_i)$, $int(A) = \bigcup_{i \in I} int(A_i)$, and $in(A) = \bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i)$.
- $states(A) = \Pi_{i \in I} states(A_i)$.
- $start(A) = \Pi_{i \in I} start(A_i)$.
- *trans(A)* is the set of triples (s, π, s') such that for all $i \in I$, if $\pi \in acts(A_i)$ then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise, $s_i = s'_i$.
- $tasks(A) = \bigcup_{i \in I}(A_i)$.

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, making them internal actions of the composition. Thus, $ActHide_{\Phi}(A)$, for $\Phi \subseteq out(A)$, is defined as the automaton obtained from A by reclassifying each action in Φ as internal.

4 Self-Similarity

The problem that motivates this paper is that of finding a representative network to test instead of testing all members of a class. If there is a small

network N that "looks like" all larger networks in the class, then the smallest such network is an obvious candidate. This is because we can test N by itself to determine properties of the entire class.

Defining Self-Similarity.

Because we are interested in networks, we consider only automata with *send* output actions and *receive* input actions. These automata are parameterized by the number of ports (interfaces) they have to the network. Each *send* action is associated with a port, and sends the message out using the port. Each *receive* action is also associated with a port and receives a message arriving on the port.

$Message$ is the set of possible messages over a port. An automaton with n ports 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 $1 \leq i \leq n$.

To combine automata, we use a channel automaton $Channel(A, B)_{i,j}$, as described in [**?**]. It joins port i of automaton A to port j of automaton B. (When only two automata are being composed, we write just $Channel_{i,j}$.) 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 that messages are delivered reliably, in-order, and with no duplication.

Suppose that an automaton N is parameterized by the number n of ports. Then we say that $N(n)$ is self-similar if
$traces(ActHide_{\Phi}(N(n) \parallel Channel_{i,j} \parallel N(n))) \subseteq traces(N(2n-2))$, 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 ports i and j, looks like a single automaton $N(2n-2)$, ignoring actions on the ports 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 importantly, 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. A set of axioms for learning bridges and proof that a composition of two automata obeying the axioms is presented in a longer version of this paper [**?**].

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 4.1 *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 $N(n)$ implement 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 fact can be used in showing that a complex network satisfies a conjunction of properties $T_1 \wedge T_2 \wedge \ldots \wedge T_n$: in showing this, one can prove that each individual property T_i is self-similar, rather than considering the properties together.

Not every property we are interested in testing will turn out to be self-similar. However, we believe that many will be; for these, testing can be carried out using small networks.

5 Learning Bridges

A learning bridge interconnects separate IEEE 802 LAN segments into a single bridged LAN. It relays and filters frames "intelligently" between the separate LAN segments [**?**].

A learning bridge incorporates a forwarding algorithm and a spanning tree algorithm. The forwarding algorithm initially forwards every frame that arrives at a port out every other port. Also, when a frame arrives at a port, the forwarding algorithm "learns" the relationship between the source address and the port. It records this relationship in a *filtering database*. Once the forwarding algorithm learns the address-to-port relationship, it forwards any frame sent to that address on the corresponding port.

The spanning tree algorithm converts an arbitrary topology to a tree. This eliminates cycles from the network so that frames will not 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 our 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 [5].

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:

Example 6.1 Learning Bridge. 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 forwards this message to every active port but B forwards it to only the correct port.

Compose A and B into one bridge AB. The requirement above means that an external observer would expect the trace of AB to have only one outgoing message with destination D. But this does not happen. Instead the message is forwarded to all ports that are inherited from A and to a single port inherited from B—the same one that B forwards the message to.

[5] 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 to 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 five 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 (input port, output port) pair. The array entry $queue[i, j]$ is a queue of messages that have arrived at port i and are 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 an arbitrary 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[6]. 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.

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*.
  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

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

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 $queue[inPort, outPort]$

Composition of Bridges:

Now we describe the composition of two learning bridges. We assume that the spanning tree algorithm 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 code given 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 port j_0 of $bridge_2$ through $Channel_{i_0,j_0}$. Because of the spanning tree algorithm, 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
$\Phi = \{send(m, i_0)_1, receive(m, i_0)_1, send(m, j_0)_2, receive(m, j_0)_2\}$.

Our 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 6.2 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:

(i) (Start condition:) If $s \in start(A)$, then $f(s) \cap start(B) \neq \phi$ (start condition).

(ii) (Step condition:) 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 $trace(\alpha) = trace(\pi)$.

Below, we define a relation R from $bridge_c$ to $bridge_p$ and prove that R is a simulation relation. This gives us the desired result:

Theorem 6.3 *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:

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

(ii) $t.outbuf[i] = s.outbuf[i]_m$ 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.

(iii) $t.inbuf[i] = s.inbuf[i]_m$ 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.

(iv) 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 $i \neq i_0, j \neq j_0$:

These conditions mean that:

(i) 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.

(ii) The output buffer for each port of $bridge_p$ contains the same messages as the output buffer of the corresponding port of $bridge_c$. 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 other conditions.

(iii) The input buffer for each port of $bridge_p$ contains the same messages as the input buffer of the corresponding port of $bridge_c$.

(iv) Entries in the internal array of queues are the same in $bridge_p$ as $bridge_c$ if the entry connects an input port to an output port of the same component bridge; otherwise, they are a concatenation involving the channel queue and the buffers for ports i_0 and j_0.

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

Definition 6.4 State Invariant: 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 only on ports of the bridge at which it arrived. It is 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 $bridge_2$, it is 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. Table 1 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 Experiment

We performed three tests on learning bridges with the goal of quantifying the impact of self-similarity in reducing test time. The first test used a single bridge, the second two connected bridges, and the third used three connected bridges. Our hypothesis was that doing only the first test would reduce the test time by at least a factor of 2 over testing three connected bridges, since only one configuration need be tested rather than three.

Test setup in this case is much simpler than most network test setup, so that time savings should be under-stated. In our tests, we used three Cisco Catalyst 2950 switches, each with four hosts connected to it, all on a single vlan (vlan1). We used 300 seconds (the default) for the expiration time of an entry in the `mac-addr-table`, which is the internal table on Cisco switches containing the learned MAC addresses. Thus entries that are not used for 5 minutes will be removed from the table.

The hosts were configured with network addresses in the 192.168.0.0/24 network. Four hosts were connected to each switch. The network was not connected to a router, so that only traffic from the LAN was visible. In each test, one of tne hosts executed a script to ping each other connected host 5 times. In addition, the `pinger` tried to ping various non-existent hosts 5 times each. After attempting to ping all hosts in the list, the `pinger` slept for 600 seconds, allowing the `mac-addr-table` entries to expire, and then repeated the pings.

For the ping, the `pinger` used the parameters `-f -c 5 -p <pattern>` .

- **-f**: Flood ping with 0 interval: send packets as fast as the host supplies them.
- **-c 5**: Packet count is 5.
- **-p** <*pattern* >: Fill the packet with the given hexadecimal pattern

The flood option was used to stress the switch as much as possible, assuming that errors are more likely when the switch is stressed. The pattern was varied in each ping to pick up potential data-dependent issues on the network.

The network traces were captured with the command `tcpdump -s0`, to capture the entire frame. For analysis, we used tcpdump with options `-exxtts0`, meaning:

- **-e**: Print the link-level header with each frame. This is required to evaluate the switch behavior, since it is a link-layer device.
- **-s0**: Capture all octets in the frame, for use in evaluating unexpected behavior.
- **-tt** Print an unformatted timestamp with each frame, to disambiguate which messages match.
- **-xx**: Print each frame, including its link level header, in hex.

Correct bridge behavior would require that hosts capture the following messages:

- **Broadcast**: All ARP request messages broadcast by any host must appear in the traces for all hosts. In other words, if an ARP request message appears in the trace for the source host, it must also appear in the trace for each other host. For connected hosts, the number of ARP requests was one or two, although the number could correctly be higher (on other tests, we have seen as much as three on larger LANs). For hosts that were not available, six messages were broadcast.
- **Unicast**: For each unicast message appearing in the trace for a source host, the trace at the destination host must contain the same message (ARP reply message, echo request message, or echo reply message).
- **Received messages**: Each message received must match a message that was sent.

Subsequent analysis of the network traces generated by tcpdump found all of the required messages.

The tests were set up and run by a single member of the project research staff, a recent graduate of the computer science program at Lehman College. Since the first tests run were the single switch tests, followed by the two switch tests, and finally the three switch tests, it is possible that learning from earlier tests reduced the time required for setting up the later tests. Because of time constraints, we actually used only one host as the `pinger` instead of rotating through the hosts; this affected the total execution time, which is predictable since we used scripts. It would have been multiplied by the number of hosts in each test (4 for the one switch case, 8 for the two switch case, and 12 for the three switch case). We assume that the effect on setup time would have been minor, on the order of a few minutes for copying the scripts to the other hosts.

Table 2 shows the distributions of time observed for running the tests. The short time required for test planning can be attributed to the simple nature of the test. We observe that after setting up for the first test suite, on the single learning bridge configuration, the time required for setting up the lab for later test suites was greatly reduced.

	One bridge	Two bridges	Three bridges
Test Planning	1 hour	-	-
Test Lab Setup	12.5 hours	1.08 hours	.92 hours
Test Execution	2.33 hours (9.3 hours)	2.33 hours (18.6 hours)	2.33 hours (27.9 hours)
Test Documentation	3 hours	2 hours	2 hours
Total	18.83 hours	5.41 hours	5.25 hours

Table 2. Times required for stages of testing for 1, 2, and 3 bridges. Presumed test execution times for using each of the hosts as a `pinger`, instead of only one, are shown in parentheses.

It took approximately 1.6 times as long to run three tests as it did to run the first, instead of 2 times as long. One reason for this was that, because the networks are self-similar, the test setup is also almost the same; thus the experience gained setting up one configuration reduces the time required to set up the next configuration. Another reason was that the configuration tasks themselves were not difficult. Creating test execution scripts and verifying that the network configuration was correct was the most difficult part of the setup.

We note that in practice, rather than testing until a desired level of confidence is reached, testers actually test until they run out of time. This phenomenon affected this test as well. Thus it is likely that the primary contribution of using self-similarity in testing will be to help testers select better tests and to improve the level of confidence in the results of testing.

A secondary goal of this experiment was to identify useful tools that might be built to use test models, especially self-similar models, to support more cost-effective testing. Difficult problems observed in the testing were evaluating test results (i.e., correct or not) and verifying correctness of the test lab setup. A model that supports determining whether a network trace is valid would be useful for evaluating test results. Better network management tools would help verify correctness of the test lab setup.

8 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 [**?**] shows how to define required properties of learning bridges and prove self-similarity. The latter approach will be necessary when a model of the network protocol is not available.

Additional work is needed to identify other self-similar networks and important self-similar properties of networks. Also, it will be useful to investigate the use of models for evaluating the results of a network test. 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.

Acknowledgments. We are grateful to Pearl Abotsi for her excellent work running the tests.

References

[1] Robert W. Buchanan. *The Art of Testing Network Systems*. Wiley, 1996.

[2] Constantinos Djouvas, Nancy Griffeth, and Nancy Lynch. Using self-similarity for efficient network testing. http://comet.lehman.cuny.edu/griffeth/Papers/selfsimlong.pdf, September 2005.

[3] Ralph Droms. RFC 2131: Dynamic host configuration protocol, March 1997.

[4] 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.

[5] Nancy Griffeth and Frederick Stevenson. An approach to best-in-class interoperability testing. *Journal of the International Test and Evaluation Association*, 23(3):68–82, October 2002.

[6] Timothy G. Griffin and Gordon T. Wilfong. An analysis of BGP convergence properties. In *Proceedings of SIGCOMM*, pages 277–288, Cambridge, MA, August 1999.

[7] Ruibing Hao, David Lee, Rakesh K. Sinha, and Nancy Griffeth. Integrated system interoperability testing with applications to voip. *IEEE/ACM Transactions on Networking*, 12(5):823–836, 2004.

[8] IEEE standard for local and metropolitan area networks: Media access control (MAC) bridges. Standard 802.1D-2004, June 2004.

[9] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 239–247, New York, NY, USA, 1989. ACM Press.

[10] 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.

[11] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., March 1996.

[12] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1989.