# Using Self-Similarity to Increase Network Testing Effectiveness

Constantinos Djouvas, Nancy D. Griffeth, Nancy A. Lynch

June 1, 2005

## 1 Introduction

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. For example, when vendors test their network equipment, they are trying to verify that the equipment works in an entire range of network topologies and configurations.

Networks vary in other contexts as well. An ISP network changes continuously. Even small organizations add new hosts regularly. They also add or swap in new network equipment as new technologies or higher bandwidths become available, as for example adding new wireless access points. 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 (Figure 1).

## 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. The network can then been assumed to have the required properties, as long the protocol standards have been shown to have them. An excellent review of protocol conformance testing appears in [9].

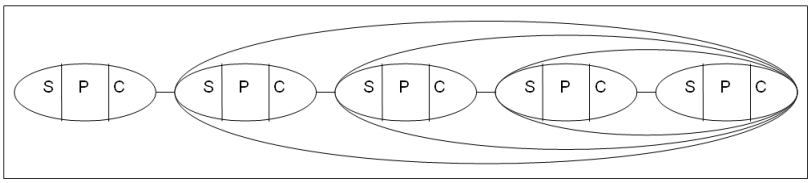


Figure 1: Self-Similarity Example

In practice, however, this requires a validated formal model of each protocol and proofs that the models provide the required network properties. 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 [6]. Others, such as DHCP, work correctly with high probability, but will behave incorrectly on rare occasions [4]. Also, as examination of network configuration manuals and how-tos reveals, most protocols - even if they have been proven correct - require careful configuration to operate correctly (see, for example, [2]).

This situation requires a different approach to verifying that TCP/IP networks behave correctly. One such approach is to extend protocol conformance testing to "network interoperability testing," as in [5, 7]. This approach does not assume that the model of the network has any particular relationship to the actual states of the network. Instead, the network is 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 end-user actions.

However, the problem of determining what network to test has not been addressed. In this paper, we suggest an initial approach to this. The approach requires modeling the network at some level of abstraction. We use the I/O automata model [10], because it provides the ability to model at a variety of levels of abstraction; because the composition operation provides a good reflection of the interaction of components in a network; and because of the variety of proof techniques available.

This paper proposes an approach to reducing the number of networks to be tested and to minimizing the size, by using the concept that networks are usually designed to hide substructures. Hence, most networks are built from components that behave like the network as a whole. We show how this knowledge can be used to justify testing a single bridge to determine its forwarding behavior, instead of having to construct an entire network of bridges. In general, the substructure to be tested will be larger than a single device.

The testing problems raised by the paucity of models for Internet protocols are being addressed in work reported elsewhere [3]. The basic concept in that work is to develop the formal model from the network itself, and ask whether the model has the required properties, instead of developing the model from the standard and asking whether the network implements it.

Section 3 introduces learning bridges and their functions. Section 4 con-

tains the necessary definitions for the I/O Automata model. Section 5 defines self-similarity of I/O automata and presents some preliminary results for self-similarity. Section 6 contains two proofs of self-similarity of learning bridges. Section 7 presents the conclusions and some directions for future work.

# 3 Learning Bridges

A learning bridge incorporates two important algorithms, the learning bridge algorithm and the spanning tree algorithm. In this section we give a brief description of both algorithms.

## 3.1 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 [8]. Before the bridge learns the location of a MAC address, it forwards every packet sent to that address out every port except the one it arrived on. Once a learning bridge has learned the location, it forwards the frame out only one port, which is the port leading to the destination address. This reduces traffic in parts of the bridged LAN that do not lie in the path between the source and destination.

The forwarding algorithm maintains a database, called the *filtering database*, that records the port through which each LAN destination address can be reached from the bridge. Each time the bridge receives a packet, the forwarding algorithm updates the filtering database entry for the source address to reflect the port at which it arrived.

Entries in the filtering database age. If no frame arrives having a given MAC Address as a source for a configurable period of time, the entry for the MAC address is removed from the database. Usually the ageing time is long (e.g. Ciscos default value is 5 minutes), since during normal operation, information in the database must change only after a physical relocation of stations (see [1], page 13-8).

## 3.2 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. Since this is a crucial property for correctness of a learning bridge, we assume the following properties required by the standard are enforced by the Spanning Tree Algorithm:

- 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.

- If there is a failure, the algorithm automatically reconfigures the spanning tree topology.
- In the case of network expansion, the algorithm automatically adjusts the network eliminating the possibility of cycles.

# 4 The I/O Automaton Model

We use the I/O Automata model [10] to model the network under test. This model represents system components and their interaction with each other. For our purposes, the important operations on automata are composition (of interacting components) and hiding (making actions invisible to the environment of an automaton).

The I/O Automaton Model also provides techniques for proving properties of the protocols. An important technique for proving correctness of distributed algorithms is *simulation*. The idea is that an algorithm that simulates a second algorithm can be considered to implement the second algorithm. A *simulation relation* relates the states of the simulating automaton to the simulated automaton.

Properties of I/O Automata are often stated as *trace properties, which are properties of externally visible sequences of actions of the I/O automata.*

## 4.1 Definitions

*An I/O automaton A consists of the following five components:*

- *sig(A), a signature, consisting of three disjoint sets of actions: the input actions in(sig(A)), output actions out(sig(A)), and internal actions int(sig(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(sig(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* $\pi$, *there is a transition* $(s, \pi, s')$.
- *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* $(s_{i-1}, \pi_i, s_i)$ *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).*

$A$ 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 $\pi$, it forces all the components having the same action to perform it. Thus for composition to work, automata must be *compatible*, that is, 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 $acts(S_i)$

Given a compatible collection $\{A_i\}_{i \in I}$ of automata, the composition $A = \Pi_{i \in I} A_i$ is defined by:

- $sig(A)$ is defined by:
  - $out(sig(A)) = \bigcup_{i \in I} out(sig(A_i))$
  - $int(sig(A)) = \bigcup_{i \in I} int(sig(A_i))$
  - $in(sig(A)) = \bigcup_{i \in I} in(sig(A_i)) - \bigcup_{i \in I} out(sig(A_i))$
- $states(A) = \Pi_{i \in I} states(A_i)$.
- $start(A) = \Pi_{i \in I} start(A_i)$.
- *trans(A)* is the set of triples $(s, \pi, s')$ 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 $\Phi$ to an internal action.

# 5 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.

We say that if a subnetwork $N$ looks like the entire network, then the network is *self-similar*.

## 5.1 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 [10], 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 $receive(m, j)_b$. In this paper, we assume a reliable, FIFO channel automaton, guaranteeing that messages are delivered reliably, in-order, and with no duplication, but a variety of possible channels can be modelled. This is a reasonable assumption, since the channels we are modeling are Category 5 or higher cables and these rarely fail.

Suppose that $n$ denotes the number of interfaces of an automaton. Then we say an automaton $N(n)$ 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 $N(2n - 2)$.

We can generalize self-similarity in the obvious way to a set of channels and networks having a more complex topology, but the above definition provides a very basic view of self-similarity and suffices for the results of this paper.

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.

## 5.2 Using Self-Similarity in Testing

By the definition of self-similarity, correct behavior of a self-similar network $N$ implies correct behavior of any larger network composed of multiple instances of $N$. Also, bugs in $N$ implies that there are bugs in the larger network.

However, we have observed above that we may not be testing a network $N$ that is self-similar. We describe two approaches that still allow us to take advantage of self-similarity to reduce the size of the network under test. First, we may be able to define a generalized model of $N$ that is self-similar and still close enough to $N$ to conform to the specification of $N$. Second, we can define self-similar ¡em¿properties¡/em¿ and test for their presence in the smaller network.

### 5.2.1 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.* The theorem follows easily from the properties of self-similarity and composition. $traces(N(n)) \subseteq traces(M(n))$ implies that
$ActHide_{\Phi}(traces(N(n)) \parallel Channel_{i,j} \parallel traces(N(n)))$
$\subseteq ActHide_{\Phi}(traces(M(n)) \parallel Channel_{i,j} \parallel traces(M(n)))$ by the composition theorem [10]. Since $M$ is self-similar, Theorem 5.1 follows. □

### 5.2.2 Self-Similar Properties

As noted above, test results concerning a self-similar property of a network $N(n)$ can be generalized to apply to larger networks.

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 show only that each property is self-similar, rather than trying to show that all are self-similar. We hope to use this fact to use self-similarity of properties of TCP to show the overall self-similarity of TCP. This may provide some insight into the observed performance behavior of TCP.

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.

# 6 Self-Similarity of learning bridges

This section contains two proofs that learning bridges are self-similar. The self-similarity property allows a tester to use Theorem 5.1 to justify testing a single learning bridge to verify the necessary properties for an entire network built from a collection of identical learning bridges [1].

We first prove the result for bridges satisfying axioms that capture the essential properties of a learning bridge. This proof appears in Section 6.1

Section 6.2 contains a model of a learning bridge as an I/O automaton $M$ and a proof that the composition of two such bridges implements a single bridge. This proof is much more complicated than the property-based proof, but is of interest because it contains the construction of the actual composed bridge.

## 6.1 Proof Using Self-Similar Properties

Given a set $M$ of messages and an integer $n$, we define a set of actions $Acts = \{send(m,i), receive(m,i) | m \in M \wedge 1 \leq i \leq n\}$ and a mapping $dest$ from messages $m \in M$ to ports $i$ with $1 \leq i \leq n$.

**Bridge Axioms**

For any trace of a bridge IOA, over actions in $Acts$, there is a function $f$ from *send* actions to *receive* actions in the trace, satisfying the following axioms:

1. If $f(send(m,i)) = receive(m,j)$ then $receive(m,j)$ precedes $send(m,i)$ in the trace.
2. For each $receive(m,i)$ in the trace, if $i \neq dest(m)$, then there is an action $send(m, dest(m))$ in the trace with $f(send(m, dest(m))) = receive(m,i)$.
3. $f$ doesn't map different sends from the same port to the same receive.
4. If $f(send(m,i)) = receive(m,j)$ then $i \neq j$.
5. If $receive(m,i) = f(send(m,j))$ precedes $receive(m',i) = f(send(m',j))$ in a trace then $send(m,j)$ precedes $send(m',j)$ in the trace.

**Theorem 1.** *Suppose that we have a failure-free network of bridges, connected using universal reliable FIFO channels in a tree configuration. Let $dest_i(m)$ be the destination function for $bridge_i$ and let it satisfy the condition that the unique path from $bridge_i$ to the host that is the destination of $m$ goes out port $dest_i(m)$.*

*Then the composition of any pair of bridges, together with the channel (if any) connecting them, in this network obeys the above axioms.*

*Proof:* Let $f_1$ be the mapping from *send* to *receive* actions in $bridge_1$ and $f_2$ be the mapping from *send* to *receive* actions in $bridge_2$ and let $f_C$ be the mapping from *receive* to *send* actions in the channel. Let $dest_1$ be the function mapping

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

messages to destination ports of $bridge_1$ and let $dest_2$ be the function mapping messages to destination ports of $bridge_2$.

If the bridges are not connected to each other in the tree, then $f = f_1 \cup f_2$ satisfies the axioms.

If they are connected to each other, then let $i_0$ be the port connecting $bridge_1$ to the channel and let $j_0$ be the port connecting $bridge_2$ to the channel. Let $\Phi = \{send(m, i_0)_1, receive(m, j_0)_2, send(m, j_0)_2, receive(m, i_0)_1$ and consider any trace of the composition $ActHide_\Phi(bridge_1 \parallel Channel(1, 2) \parallel bridge_2)$.

Note that the mapping $f_C$ from the *receive* actions of the $Channel_{1,2}$ to the *send* actions of the channel, as defined in [10] is injective (one-to-one) and surjective (onto).

For purposes of this proof, we assume that all messages $m$ are distinguishable from each other, for example by a sequence number. Thus we can say that $f_C(send(m, i_0)_1) = receive(m, j_0)_2$ and $f_C(send(m, j_0)_2) = receive(m, i_0)_1$.

We claim that one of the following must hold for the destination functions $dest_1$ and $dest_2$. For all possible messages $m$, either $dest_1(m) = i_0$ or $dest_2(m) = j_0$, but not both. This follows because the topology is a tree. Thus we can define a function $dest(m)$ for the composed automata as follows:

1. If $dest_1(m) = i_0$ then $dest(m) = dest_2(m)$.
2. If $dest_2(m) = j_0$ then $dest(m) = dest_1(m)$.

Define a mapping $f$ from external *send* actions of the composition to *receive* actions as follows:
If $f_1(send(m, i)_1) = receive(m, i')_1$ where $i' \neq i_0$, then $f(send(m, i)_1) = f_1(send(m, i)_1)$.
If $f_1(send(m, i)_1) = receive(m, i_0)_1$, then $f(send(m, i)_1) = f_2(f_C(receive(m, i_0)_1)) = f_2(send(m, j_0)_2)$.
If $f_2(send(m, j)_2) = receive(m, j')_2$ where $j' \neq j_0$, then $f(send(m, j)_2) = f_2(send(m, j)_2)$.
If $f_2(send(m, j)_2) = receive(m, j_0)_2$, then $f(send(m, j)_1) = f_1(f_C(receive(m, j_0)_2)) = f_1(send(m, j_0)_1)$.
We claim that $f$ satisfies the axioms above.

Axiom 1 Since each function has the property that its image precedes its argument, this follows.

Axiom 2 Let $receive(m, i)_1$ be a *receive* action in a trace of the composite bridge. If $i = dest_1(m)$ there is nothing to prove. Suppose that $receive(m, i)_1$ is an action in the trace with $i \neq dest_1(m)$. Then, by axiom 2, there is an action $send(m, dest(m))_1$ with $f_1(send(m, dest(m))_1) = receive(m, i)_1$. If $dest(m) \neq i_0$, then $f(send(m, dest(m))_1) = f_1(send(m, dest(m))_1)$ and we are done. If $dest(m) = i_0$, then $m$ proceeds through $Channel(1, 2)$ to $bridge_2$, arriving at port $j_0$. Since $dest_1(m) = i_0$, $dest_2(m) \neq j_0$, so that there is an action $send(m, j)_2$ with $f_2(send(m, j)_2) = receive(m, j_0)_2$, and $f(send(m, j)_2) = f_1(f_C(f_2(send(m, j)_2))) = f_1(f_C(receive(m, j_0)_2)) = f_1(send(m, i_0)_1)$, which by the above is $receive(m, i)$.

Axiom 3 Suppose that *send* actions $\pi_1$ and $\pi_2$ from the same port are mapped to the same *receive* action. If that action is on a port of the same bridge as the send port, this contradicts the assumption that the axiom holds for the component bridges. Hence $\pi_1$ and $\pi_2$ correspond to two *receive* actions $\pi_1'$ and $\pi_2'$ that arrived at the port joining to the other bridge. Because the channel connecting the bridges never duplicates messages, there are two *send* actions $\rho_1$ and $\rho_2$ that happened at the other end of the channel. And by axiom 3, these must be mapped to distinct *receive* actions. By contradiction, axiom 3 holds for the composed bridges.

Axiom 4 If $f(send(m,i)_k) = receive(m,j)_k$ then $f_k(send(m,i)_k) = receive(m,j)_k$ for $k \in \{1,2\}$, i.e., if $f$ maps the send to a receive on the same bridge, then since Axiom 4 holds for $f_1$ and $f_2$ and $f$ is equal to one or the other of these, Axiom 4 holds also for $f$. If $f$ maps the send to a receive on a different bridge, obviously the send and receive involve different ports.

Axiom 5 The only cases we need to prove is if $f(send(m,i)_1) = f_2(f_C(receive(m,i_0)_1)) = f_2(send(m,j_0)_2) = receive(m,j)_2$ and $f(send(m,j)_2 = f_1(f_C(receive(m,j_0)_2)) = f_1(send(m,i_0)_1) = receive(m,i)_1$.

Suppose $receive(m,i)_1)$ precedes $receive(m',i)_1$, then $send(m,i_0)_1$ precedes $send(m',i_0)_1$, because the component automata keep the messages in order by Axiom 5. Also, the channel keeps messages in order, so $receive(m,j_0)_2$ precedes $receive(m',j_0)_2$. By the hypothesis that the component automata obey Axiom 5, this implies that $send(m,j)_2$ precedes $send(m',j)_2$.

□

## 6.2 Proof Using a Generalized, Self-Similar Model

The purpose of a learning bridge in a LAN is to deliver messages directly from the source to the destination, while avoiding collisions and duplicated messages. To avoid duplicating messages, the bridge maintains a *filtering database* that maps each destination to the bridge port leading to the destination. Once the bridge has the necessary information in its filtering database, it forwards a message only through the port that leads to the destination.

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.

So we define a generalized model in which it is required that the bridge copies each message to the "correct port", perhaps along with other ports. By "correct port" $P$, we mean that $P$ is the port that received the last message with source equal to the destination of the current message. To implement this, each time a message is received, the learning bridge algorithm records the source address with the port at which the message arrived in the *filtering database*. Subsequent messages sent to that address will be copied to the port. If no message has been received from the destination address, the *filtering database* will not have an entry for the address, and the bridge forwards the message to all ports. This generalized model, models precisely the forwarding behavior of connected learning bridges.

### 6.2.1 The Generalized Model

In this section, we model the generalized bridge as an I/O automaton. Each bridge has a number of ports and it has four actions: input action *receive*, output action *send*, and internal actions *copy* and *delete*. It also maintains a filtering database, an input and output buffer for each port and a tracking array to keep track of where messages have been copied to. The *receive* action adds messages received from other bridges to the designated input buffer of the port at which the message arrives and also updates the *filtering database*. The *send* action sends the first message in the output buffer of a port to the channel that the port is connected to. The *copy* action is responsible for copying messages from input buffers to output buffers. It copies the first message $m$ of an input buffer to an output buffer, without duplicating messages, i.e. it copies $m$ to each output buffer at most one time. It uses the tracking array to avoid duplicate copies. Finally the *delete* action deletes the first message $m$ from an input buffer. This action is enabled either when $m$ has been copied to at least the "correct" output buffer or $m$ has been copied to all output buffers. An output buffer is "correct" for a message $m$ if its port, based on the *filtering database*, leads to $m$'s destination.

**automaton** $bridge_i$**:**

**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*.
*table*, 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 $table[inPort, i]$ for all $i \neq inPort$
remove $m$ from $inbuf[inPort]$

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

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

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.

### 6.2.2 Composition of Bridges

In this section we describe the composition of two bridges. This composition will be used to prove that the learning bridges are self-similar. To do so, in both sections 7 and 8, we are guided by the assumption that the Spanning Tree Protocol has been run to completion by all the bridges in the network and that there are no failures. As a result, there is only one active path between any two bridges. We use the convention that port $i$ is a port of $bridge_1$ and $j$ is a port of $bridge_2$.

To show that learning bridges are self-similar we first compose two primitive bridges. Let $bridge_1$ and $bridge_2$ be two bridges running the IOA defined in Section 6.2. Without loss of generality, we assume that port $i_0$ of $bridge_1$ is connected with the port $j_0$ of $bridge_2$ through $Channel_{i_0,j_0}$ and that these are the only active ports connecting $bridge_1$ and $bridge_2$.

Let $bridge_c$ be the result of composing $bridge_1$ and $bridge_2$ and hiding the $send(m, i_0)_1$ action of $bridge_1$ and the $receive(m, j_0)_2$ action of $bridge_2$, i.e.,

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

The goal is to show that $bridge_c$ is essentially the same as a single bridge, which we will call $bridge_p$, running the IOA defined in Section 6.2. Our goal requires that $bridge_p$ is able to handle the same number of connections as $bridge_1$ and $bridge_2$ can handle together. The number of ports of $bridge_p$ is two fewer than the number of ports of $bridge_1$ and $bridge_2$ combined, since two ports are used for the "internal" communication. Thus if $bridge_1$ and $bridge_2$ have n active ports, $bridge_p$ has 2n-2 active ports. In other words, the active ports of $bridge_p$ correspond to the active ports of $bridge_1$ and $bridge_2$, minus the two "internal" ports. We also define $bridge_p$ so that port $i \in ports_p$, where $1 \leq i \leq n$, is connected to the same channel as the corresponding port $i \in ports_1 - i_0$ of $bridge_1$. Similarly port $j \in ports_p$ , where $n \leq j \leq 2n$, is connected to the same channel as the corresponding port $j \in ports_2 - j_0$. 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 $send(m, i)_1$; similarly, actions on port $j$, $n \leq j \leq 2n$, are $receive(m, j - n)_2$ and $send(m, j - n)_2$.

### 6.2.3 Simulating a bridge with a composition of bridges

We use an important theorem about IOA to show the equivalence of the composition $bridge_c$ to the single bridge $bridge_p$ (Figure 2). The theorem says that if there is a simulation relation (defined below) from an IOA $A$ to an IOA $B$, then $traces(A) \subseteq traces(B)$. In this section, we define a relation between the two bridges.

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:

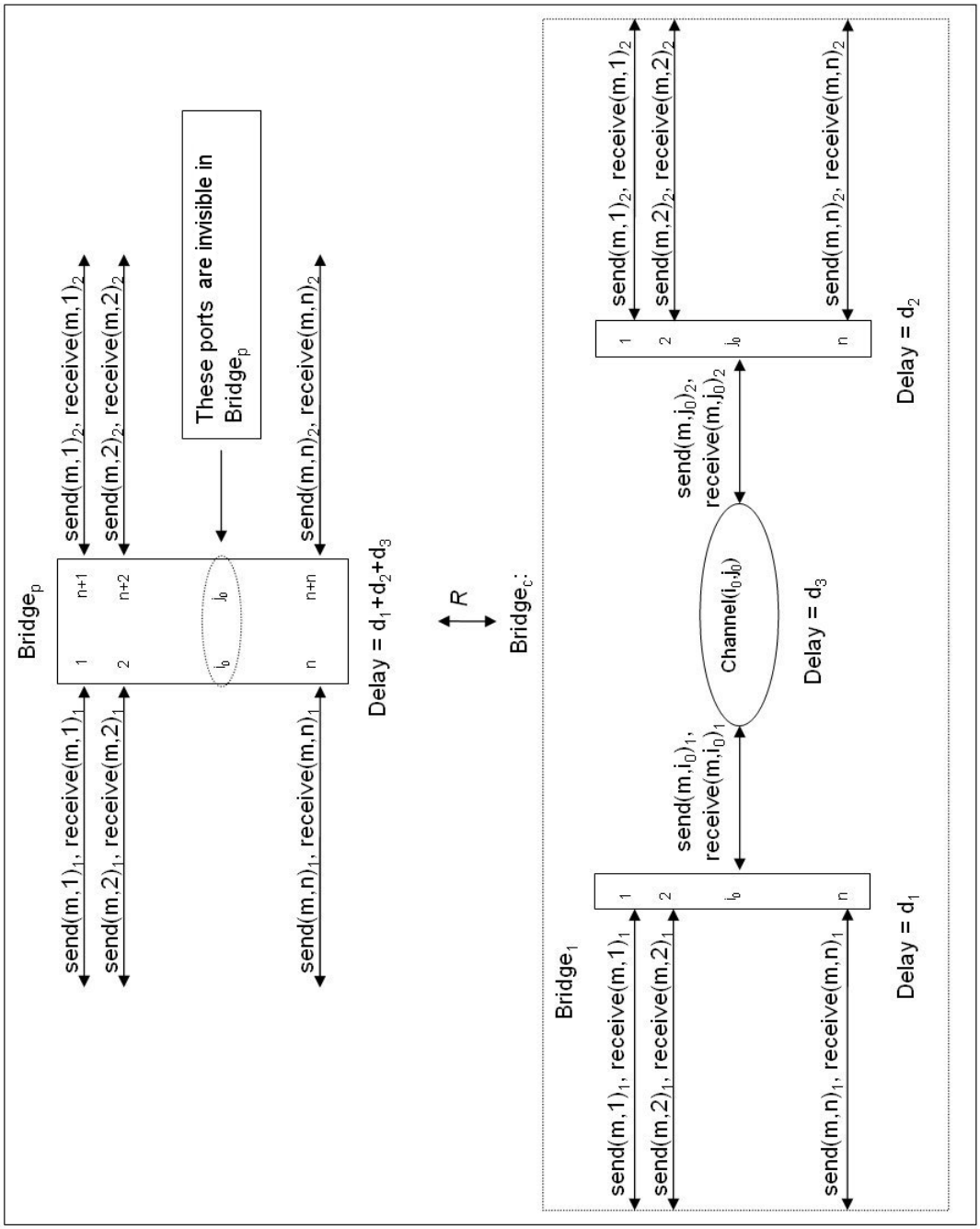


Figure 2: Composition and Simulation Relation

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 $\alpha$ of $B$ starting in state $u$ and ending in some state $u' \in f(s')$ such that $trace(\alpha) = trace(\pi)$ (step condition).

Now we define a relation from $bridge_c$ to $bridge_p$. In sction 8.1 we will prove that it is a simulation relation. The pair $(s, t)$ with $s \in states(bridge_c)$ and $t \in states(bridge_p)$ belongs to the relation $R$, provided that the following conditions hold:

**Condition 8 1.** *The filtering database of* $t.bridge_p$[2] *contains the same entries as the union of the filtering databases of* $s.bridge_1$ *and* $s.bridge_2$*, excluding the entries for the "internal" ports:*

$$t.bridge_p.filterDB = s.bridge_1.filterDB \cup s.bridge_2.filterDB \\ -\{\langle addr, port\rangle | port \in \{s.bridge_1.i_0, s.bridge_2.j_0\}\}$$

**Condition 8 2.** *The output buffer for each port of* $t.bridge_p$ *contains the same messages as the output buffer of the corresponding port of* $s.bridge_c$*:*

$$t.bridge_p.outbuf[i] = s.bridge_c.outbuf[i] \text{ for } i \in ports_1 \bigcup ports_2 - \{i_0, j_0\}.$$

*(Note that* $t.bridge_p$ *does not contain any buffers corresponding to* $i_0$ *and* $j_0$*. These buffers in* $s.bridge_c$ *may contain any messages consistent with the next condition.)*

**Condition 8 3.** *The input buffer for each port of* $t.bridge_p$ *contains the same messages as the input buffer of the corresponding port of* $s.bridge_c$*:*

$$t.bridge_p.inbuf[i] = s.bridge_c.inbuf[i] \text{ for } i \in ports_1 \bigcup ports_2 - \{i_0, j_0\}.$$

**Condition 8 4.** *The internal table of message queues* $table_p$ *corresponds to the combined tables* $table_1$ *and* $table_2$ *as follows:*

- $table[i, i']_p = table[i, i']_1$ *if* $i, i' \in ports_1, i, i' \neq i_0$
- $table[j, j']_p = table[j, j']_2$ *if* $j, j' \in ports_2, j, j' \neq j_0$
- $table[i, j]_p$ *is the concatenation of the following queues for* $i \in ports_1, j \in ports_2$*, with* $i \neq i_0, j \neq j_0$*:*
  - $table[j_0, j]_2$
  - $outbuf[j_0]_2$
  - $queue_{j_0, i_0}$

[2]We use the dot notation to denote the value of a given variable in a state as well as to denote a given bridge in the composition.

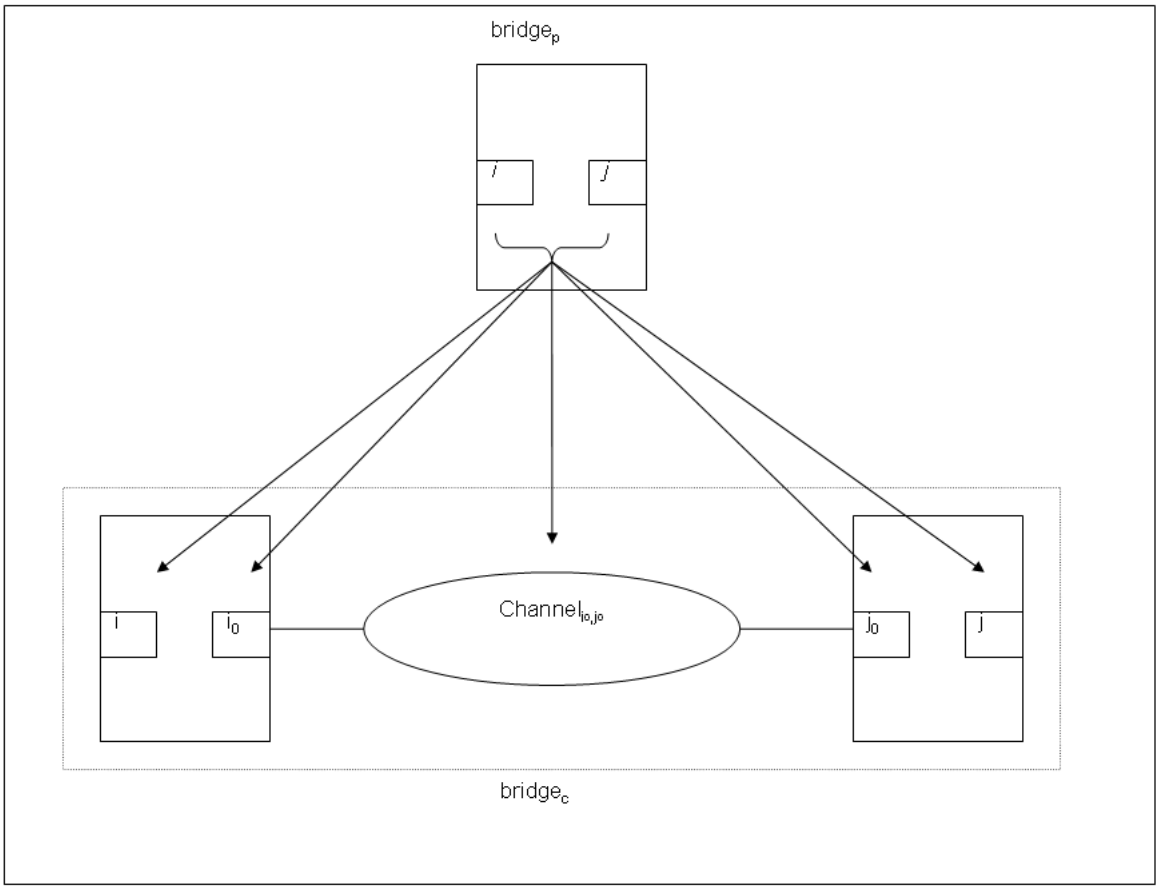


Figure 3: Buffers Correspondence

- $inbuf[i_0]_1$
- $table[i, i_0]_1$

- *The relationship between* $table[j, i]_p$ *and* $table_1$ *and* $table_2$ *is defined symmetrically for* $i \in ports_1, j \in ports_2$*, with* $i \neq i_0, j \neq j_0$*:*

#### 6.2.4 Self-Similarity of the generalized bridge model

Self-similarity requires that $traces(M \parallel M) \subseteq traces(M)$, i.e., $traces(bridge_c) \subseteq traces(bridge_p)$. Based on the definition of 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.

**Invariant 8 1.** *At any time during the execution, the filtering database of* $bridge_c$ *corresponds to the state of the filtering database of* $bridge_p$*.*

*Proof.* Invariant 8.1 can be proved using induction on the number of steps.

*Base*: At the beginning everything is empty so the state of the filtering database of $bridge_c$ corresponds to the state of the filtering database of $bridge_p$.

*Inductive Hypothesis*: Suppose that the invariant holds after $t$ steps.

*Inductive Step*: Suppose that a message from source $s$ is received at the beginning of step $t + 1$. (If nothing is received, no changes will be made, so the filtering databases remain in corresponding states.) Assume also that the message is received at a port of $bridge_1$ (the case for a message received at $bridge_2$ is symmetric).

There are four different cases:

- Only the filtering database inherited from $bridge_1$ has an entry for the source.
- Only the filtering inherited from $bridge_2$ has an entry for the source.
- Both have an entry.
- Neither has an entry.

First we should note that entries in the filtering databases never change (again once they are non-null), given our assumption that there are no failures in the network and that the active ports form a tree. This is the case because there is unique path between every pair of ports. As a result a message from each source to each destination always follows the same unique path.

In the first case, $bridge_c$ already has an entry in the filtering database (for $bridge_1$), and so the state is unchanged. $Bridge_p$ also has an entry already (based on the inductive hypothesis), and so changes nothing to its filtering database, so the databases remain in corresponding state.

For the second case, the $bridge_c$ entry is in the filtering database for $bridge_2$, but the message arrives at a port inherited from $bridge_1$. Again, because the network is a tree and $bridge_2$ has an entry but not $bridge_1$, the message must have come to $bridge_1$ from $bridge_2$. Thus the $bridge_1$ port at which the message arrives is $i_0$ (the port connected to $bridge_2$) and so the filtering database of $bridge_1$ will point to $bridge_2$. In $bridge_p$ the "internal" entries, i.e. entries pointing from $bridge_1$ to $bridge_2$ and vice versa, are omitted. Thus the correspondence between $bridge_p$ and $bridge_c$ is unchanged.

In case 3, both bridges have an entry in their filtering databases. Since entries never change, the filtering databases remain in corresponding states.

In case 4, neither database has an entry (hence the message cannot have arrived on the port $i_0$ connected to $bridge_2$). $Bridge_p$ will set the entry for the source in its filtering database to the arrival port. Similarly, $bridge_c$ will set the entry in the filtering database for $bridge_1$ to the arrival port. Thus the databases maintain their correspondence. □

We must also prove that the states of $bridge_p$ and $bridge_c$ correspond after each action in spite of changes to input and output buffers. In order to do that we must consider all the cases based on all actions $\pi$ that can be taken. The following table (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 (note that the action of $bridge_p$ in this table have been remained to reflect the actions renaming provided in section 7.1).

| | 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 | $receive(m,i_0)_1$ | $\lambda$ | $\lambda$ |
| 4 | $receive(m,j_0)_2$ | $\lambda$ | $\lambda$ |
| 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 | $send(m,i_0)_1$ | $\lambda$ | $\lambda$ |
| 8 | $send(m,j_0)_2$ | $\lambda$ | $\lambda$ |
| 9 | $delete(m,i,i')_1, i' \neq i_0$ | $delete(m,i,i')_p$ | $\lambda$ |
| 10 | $delete(m,j,j')_2, j' \neq j_0$ | $delete(m,j,j')_p$ | $\lambda$ |
| 11 | $delete(m,i,i_0)_1$ | Sequence $delete(m,i,j)_p$ for $j \in ports_2, j \neq j_0$ | $\lambda$ |
| 12 | $delete(m,j,j_0)_2$ | Sequence $delete(m,j,i)_p$ for $i \in ports_1, i \neq i_0$ | $\lambda$ |
| 13 | $copyIn(m,i)_1, i \neq i_0$ | $copyIn(m,i)_p$ | $\lambda$ |
| 14 | $copyIn(m,j)_2, j \neq j_0$ | $copyIn(m,j)_p$ | $\lambda$ |
| 15 | $copyIn(m,i_0)_1$ | $\lambda$ | $\lambda$ |
| 16 | $copyIn(m,j_0)_2$ | $\lambda$ | $\lambda$ |
| 17 | $copyOut(m,i,i')_1, i' \neq i_0$ | $copyOut(m,i,i')_p$ | $\lambda$ |
| 18 | $copyOut(m,j,j')_2, j' \neq j_0$ | $copyOut(m,j,j')_p$ | $\lambda$ |
| 19 | $copyOut(m,i,i_0)_1$ | $\lambda$ | $\lambda$ |
| 20 | $copyOut(m,j,j_0)_2$ | $\lambda$ | $\lambda$ |

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

**Proof** : We need to show that if $bridge_c$ is in state $s$ and $bridge_p$ is in state $t$ and if $t \in f(s)$ then for any action $\pi$ of $bridge_c$, with $(s,\pi,s') \in transitions(bridge_c)$, the corresponding execution fragment of $bridge_p$, given by Table 1, takes $bridge_p$ to a state $t'$ such that $t' \in f(s')$.

Line 1 $\pi = receive(m,i)_1, i \neq i_0$. This action adds a message to $inbuf[i]_1$ in $bridge_1$ of $bridge_c$. The corresponding action (also $receive(m,i)_1$) in

$bridge_p$ adds a message to $inbuf[i]_p$. Thus the messages in the input buffers remain the same, so $t' \in f(s')$.

Line 2 $\pi = receive(m, j)_2, j \neq j_0$. The reasoning is the same as for Line 1.

Line 3 $\pi = receive(m, i_0)_1$. This removes a message $m$ from the head of $queue_{i_0,j_0}$ and adds it to the end of $inbuf[i_0]_1$. This could affect the relationship between $table[filterDB(m.src), i]_p$ and the various queues in the composition, since the concatenation involves $inbuf[i_0]_1$ and $queue(i_0, j_0)$. However, for all $i \in ports_1$, this leaves unchanged the value of the concatenation

$$table[i_0, i]_1 \widehat{\ }( \ ) inbuf[i_0]_1 \widehat{\ }( \ ) queue_{j_0,i_0} \widehat{\ }( \ ) outbuf[j_0]_2 \widehat{\ }( \ ) table[filterDB(m.src)_2, j_0]_1$$

it follows for all $t \in f(s)$ that $t \in f(s')$. This justifies the choice of $\lambda$, which doesn't change the state $t$, as the corresponding execution fragment in $bridge_p$.

Line 4 $\pi = receive(m, j_0)_2$. The reasoning is the symmetric to line 3.

Line 5 $\pi = send(m, i)_1, i \neq i_0$. This action removes the message $m$ from the queue $outbuf[i]_1$ in $bridge_1$ of $bridge_c$. The corresponding action in $bridge_p$ is also $send(m, i)_1$, which also removes $m$ from $outbuf[i]_p$. Thus the state $t' \in f(s')$.

Line 6 $\pi = send(m, j)_2, j \neq j_0$.The reasoning is symmetric to Line 5.

Line 7 $\pi = send(m, i_0)_1$. This action removes a message $m$ from $outbuf[i_0]_1$ and adds it to $queue_{i_0,j_0}$ in the composition bridge. As with the action in Line 3, this leaves the concatenation of queues between $bridge_1$ and $bridge_2$ unchanged, so that if $s'$ is the resulting state and $t \in f(s)$ then $t \in f(s')$.

Line 8 $\pi = send(m, j_0)_2$. The reasoning is symmetric to Line 7.

Line 9 $\pi = delete(m, i, i')_1, i' \neq i_0$. This action removes $m$ from $queue[i, i']_1$. If $i \neq i_0$, this entry is related by the simulation relation to the corresponding entry of $queue[i, i']$ of $bridge_p$ and so the corresponding execution fragment is just $delete(m, i, i')_p$, which removes the same entry from $queue[i, i']_p$.

Line 10 $\pi = delete(m, j, j')_2, j' \neq j_0$. The reasoning is the same as in Line 9.

Line 11 $\pi = delete(m, i, i_0)_1$. This removes the message $m$ from $table[i, i_0]$. This queue is part of every concatenation of queues of the form:

$$table[j, j_0]_2 \widehat{\ }( \ ) inbuf[j_0]_2 \widehat{\ }( \ ) queue_{i_0,j_0} \widehat{\ }( \ ) outbuf[i_0]_1 \widehat{\ }( \ ) table[i, i_0]_1$$

In other words, the corresponding queues in $bridge_p$ are the queues $table[i, j]_p$ for every $j \in ports_2$ with $j \neq j_0$. Thus any sequence of deletes $\alpha$ that removes $m$ from all queues $table[i, j]_p$ with $j \in ports_2$ and $j \neq j_0$ will leave each of these queues in correspondence with the above concatenation, i.e., if $(s, delete(m, i, i_0)_1, s')$ is a transition of $bridge_c$, then $(t, \alpha, t')$ is a ¡what's the term?¿ of $bridge_p$ and $t' \in f(s')$.

Line 12 $\pi = delete(m, j, j_0)_2$. The argument is symmetric to the argument for Line 11.

Line 13 $\pi = copyIn(m, i)_1, i \neq i_0$. The result of this action is to put $m$ in all queues $table[i, i']_1, i' \neq i$. The action $copyIn(m, i)_p$ does the same to the corresponding queues $table[i, i']_p$ for $i' \in ports_1$ and $i' \neq i_0$. The case of $i' = i_0$ is more interesting. Adding $m$ to the queue $table[i, i_0]_1$ adds $m$ to all concatenations of the form

$$table[j, j_0]_2 \widehat{(\ )} inbuf[j_0]_2 \widehat{(\ )} queue_{i_0,j_0} \widehat{(\ )} outbuf[i_0]_1 \widehat{(\ )} table[i, i_0]_1$$

where $j \in ports_2, j \neq j_0$. But also, the action $copyIn(m, i)_p$ adds $m$ to all queues $table[i, j]_p$ with $j \in ports_2, j \neq j_0$. So the simulation relation is preserved by these actions.

Line 14 $\pi = copyIn(m, j)_2, j \neq j_0$. The argument is symmetric to Line 13.

Line 15 $\pi = copyIn(m, i_0)_1$. This removes $m$ from $inbuf[i_0]_1$ in $bridge_1$ and adds it to $table[i_0, j]_1$ for all $j \neq i_0$. This doesn't change the value of any of the concatenations, and so using the corresponding execution fragment $\alpha$ preserves the simulation relation.

Line 16 $\pi = copyIn(m, j_0)_2$. The argument is symmetric to Line 15.

Line 17 $\pi = copyOut(m, i, i')_1, i' \neq i_0$. This action removes $m$ from $table[i, i']_1$ and adds it to $outbuf[i']_1$. The same effects are achieved in $bridge_p$ by $copyOut(m, i, i')_p$.

Line 18 $\pi = copyOut(m, j, j')_2, j' \neq j_0$. The argument is the same as in Line 17.

Line 19 $\pi = copyOut(m, i, i_0)_1$. This action removes $m$ from $table[i, i_0]_1$ and adds it to $outbuf[i_0]_1$. This has no effect on the value of any of the relevant concatenations, and so the empty execution fragment in $bridge_p$ preserves the simulation relation.

Line 20 $\pi = copyOut(m, j, j_0)_1$. The argument is symmetric to Line 19.

Based on the above we can conclude that $traces(bridge_p) \subseteq traces(bridge_c)$

# 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 two approaches to using self-similarity in network testing. One is to identify self-similar

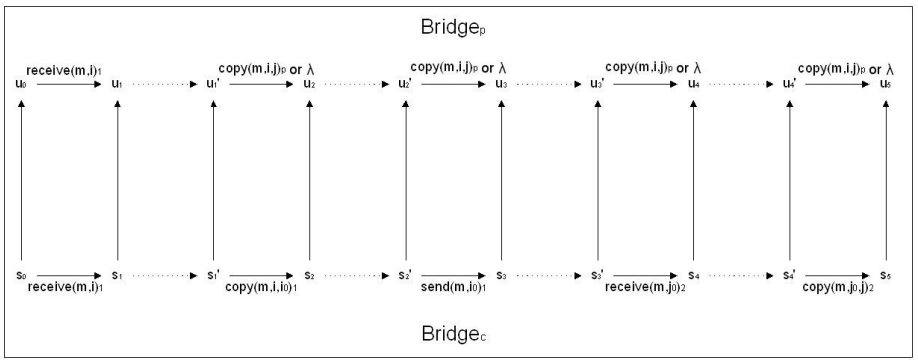


Figure 4: Execution

properties to be tested. The second is identify self-similar network models that the network should implement, in the formal sense, if it has been built correctly.

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] Catalyst 2950 and catalyst 2955 switch software configuration guide, 12.1(22)ea5, 1992-2005.

[2] Cisco ios ip configuration guide, release 12.2, 1992-2005.

[3] Constantinos Djouvas and Nancy Griffeth. Experimental method for testing networks. In *Proceedings of SERP'05 - The 2005 International Conference on Software Engineering Research and Practice*, June 2005.

[4] Ralph Droms. Rfc 2131: Dynamic host configuration protocol, March 1997.

[5] 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.

[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 Trans. Netw.*, 12(5):823–836, 2004.

[8] Ieee standard for local and metropolitan area networks: Media access control (mac) bridges, June 2004.

[9] 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.

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