A Formal Model of Compositional Network Architecture

September 21, 2023

1 Introduction

This document is a companion to *The Real Internet Architecture: Past, Present, and Future Evolution.* It presents a declarative formal model of compositional network architecture, written in the Alloy language (see [2] and `alloytools.org`). Alloy is a very well-designed modeling language—a seamless blend of first-order logic, relational algebra, object orientation, and temporal logic. The Alloy Analyzer incorporates state-of-the-art technology for logical solvers. To learn about the language and its tools, please refer to Alloy documentation and tutorials.

The purposes of the formal model of compositional network architecture include:

- to ensure that compositional network architecture has a sound formal foundation;
- to support a variety of validation, verification, implementation, and optimization activities;
- to help network designers "future-proof" their designs by understanding the true range of possible network behaviors and network architectures;
- to relate compositional network architecture to other, less general and more computationally efficient, formal models of networking.

The architecture of individual networks, as presented in Chapter 2, is modeled in §2 on basic networks, §3 on the traffic model, and §4 on group communication. The operators for composing networks, as presented in Chapters 3 and 4, are modeled in §5. Simplifying assumptions, introduced when needed, are summarized in §6. Finally, §7 discusses the limitations of the current model and how it might be extended.

The subject of network architecture is large, the formal model is correspondingly complex, and the amount of detail is necessarily limited. To maintain focus on what *can* be accomplished with a limited model, we introduce three guiding principles that have been followed. The underlying motivation for these

principles is to make the architecture as general as possible, without arbitrary and unnecessary constraints. The principles are needed because they expose many truly subtle issues, which could easily be missed without guidance. The principles are also summarized in §6.

Most people think of formal models as something used for *verification*—in particular, something used to verify that an implementation is correct. But even before verification comes *validation*, which is the partly-formal, partly-informal process of comparing a formal model to the real world it is intended to describe. Is it accurate and precise? Is it comprehensible, i.e., does it mean what we think it means? Is it general or extensible? Is it useful? The formal model has been systematically and thoroughly validated, and this document describes some validation methods and the very extensive Alloy code devoted to validation.

This is a living document. Both the Alloy model and this document are works in progress, and will be updated as we learn more and do more.

Eventually there will be a companion document on the reference implementation of compositional network architecture, which is a program written in Lucid [3]. The Lucid language compiles to P4 [1], which in turn compiles to programmable packet-processing hardware. We use Lucid and P4 to show that compositional network architecture is inherently efficient. Nevertheless, hardware constraints are not friendly to generality, so the Lucid implementation is necessarily less general than the Alloy model.

2 Basic networks

2.1 Machines and distributed systems

The file `dist.als` (dated 1 October 2022 or later) formalizes simple relationships among distributed systems and machines, mainly (i) all distributed systems have a scope that is a set of machines; (ii) on each machine in the scope of a distributed system, there is at least one module of the system; (iii) a network is a distributed system with an administrative authority that is a legal person, and a state that is a network state. None of this is very useful so far, so it is included only for completeness.

2.2 Components of a network

2.2.1 Network state

The remainder of §2 refers to the file `net.als` (dated 3 March 2023 or later). In the text we refer to line numbers, and `net.num` is a non-executable version of `net.als` with these line numbers.

The file begins with declarations of the types (basic types and object signatures) used (lines 4-72). We will refer back to these declarations when necessary.

The most important thing in this file is the declaration of a `NetworkState` object, in lines 81-184. This definition has two parts. In the first part, up to

line 95, fields of the object are declared. The second part consists of Boolean expressions, which must evaluate to true for any instance of the object.

In terms of real-world meaning, these Boolean expressions fall into two categories. Some are *constraints*, expressing the necessary relationships and correlations among different parts of the network state. The actual constraints exist in the real world, so the real world cannot violate them, and our formalization of them is simply ensuring that our formal objects cannot violate them either. For example, if a `Link` is a component of a network, then at least one of its endpoints is a member of the network (lines 103-104). There is also one constraint placed outside the object declaration: on lines 78-79, the fact states that each instance of NetworkState has a unique `networkName`. This must be true in the real world of networks compliant with compositional network architecture, or some things will not work right.

The other Boolean expressions are in the category of *derivations*, and they match up with *derived* object fields (see comments on the declarations of these fields). The contents of a derived field are computed from other, non-derived, fields, using the Boolean expression as the computational rule. So non-derived fields describe a constrained real world, while derived fields represent consequences of the non-derived fields. Derived fields are used as shorthands in other expressions.

2.2.2 Members and links

A network has a set of `members` (line 84), each of which belongs to a primitive type `Name`. This declaration represents a major simplifying assumption in the model. In compositional network architecture, there are no constraints on how members of a network are named—a member can have no name, one name, or many names, and none of them need be unique. In this model, every member of a network must have a name that is unique within the network. (Eventually we will show that it can have non-unique group names as well.) Furthermore, the model makes no distinction between the member itself (which is a module on a machine) and the unique name of the member. The consequences of this assumption are significant, and will be discussed in several subsequent sections.

There are two member subsets, `users` and `infras` (user and infrastructure members, respectively). Their declaration on line 85 includes the constraint that they are disjoint, and the constraint on lines 98-99 says that each member is either a user or infrastructure member. Members, users, infras, and other fields of a network state discussed in this subsection are all illustrated in Figure 1.

A network `Link` is a complex object, defined on lines 42-59. All links in the formal model are unidirectional, because that is necessary for computation of reachability. So a link has a single `sndr` (sender member) and a `sndrIdent`, which is its local identifier at the sender member. Links can be group links, but we will not discuss any aspect of group communication until §4. For now, as a temporary simplifying assumption, think of a link as having a `rcvrs` field which is an individual receiver member, and a field `rcvrIdents` that maps the receiver to the link's local identifier at the receiver.

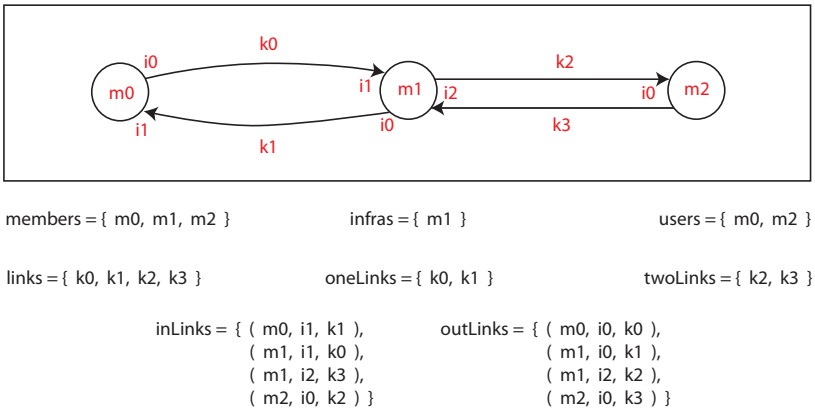

Figure 1: Members and links of a small network.

We model two-way point-to-point links with the help of the `mode` field, which can have values `OneWay` and `TwoWay` (along with other values discussed in §4). There are disjoint link sets `oneLinks` and `twoLinks`. The derivations on lines 100-102 say that these sets contain the links with mode `OneWay` and `TwoWay`, respectively. Furthermore, the constraint on line 107 says that for each link in twoLinks, there is another link in twoLinks that is its reverse. The predicate `ReverseLink` is defined in lines 60-63; in a pair of reverse links, the sender of one is the receiver of the other, and vice-versa. In other words, the real-world phenomenon of a two-way link is modeled with a matching pair of one-way links.

Lines 108-125 express constraints on link identifiers. At each member, each link of which it is an endpoint must have a different link identifier. There is an exception for a reverse pair in twoLinks, because at each endpoint, the two links share the same link identifier. This does not cause confusion because one is an incoming link and one is an outgoing link.

The `outLinks` and `inLinks` fields (each a relation) encode the same information as the network's set of links. They are derived from the links because they present the information in a way that is more convenient for many modeling purposes.

2.3 Validation and network topology

The concept of validation was introduced in §1. Even though validation is fundamentally informal, automated tools can still be a significant help. This is especially true with a declarative (constraint-based) language such as Alloy, because we can inadvertently write such strong constraints that they can never be satisfied (formally, the model is inconsistent). Equally likely, we can write such strong constraints that the real-world examples we care about do not satisfy its constraints, and therefore cannot be instances of the model. To prevent such

problems, it is important to use the Alloy Analyzer to make sure that important real-world networks *are* instances of the model.

Chapter 2 of the book has diagrams of common network topologies, and we use these for validation. Specifically, there is a section of `net.als` (lines 186-253) defining predicates for a number of network topologies. For each topology, if a network satisfies its rules, then the NetworkState object satisfies the corresponding predicate. To validate the model using these predicates, we formulate "tests" that are predicates formed from various combinations of the topology predicates, and ask the Alloy Analyzer to find instances of them, by executing the commands in lines 292-334. If the test predicate is inconsistent (has no instance), there is usually a modeling bug somewhere. If the predicate has an instance, we check it to make sure it looks right.

Of the simple, regular topologies defined here, the spanning tree is the most complex, and it is difficult to model without some additional fields in the network state. So instead of a predicate there is a declaration of a special kind of NetworkState object with extra fields (lines 234-253), and the constraints we would otherwise write in a predicate are written as part of the object signature.

A network can include external links, which are links with endpoints both inside the network and outside the network. The topological predicate `No_external-links` distinguishes networks without these.

§6 in Chapter 2 says that topological properties can represent facts or assumptions about the user machines served by a network, and can also represent design decisions about the arrangement of infrastructure machines. The topological properties provided here are general, each applying to many specific networks. Other topological properties, used for example in design, might be very specific—they might even name specific members on individual machines.

2.4 Packets

In the current formal model, there is a minimal formalization of sessions. They appear primarily in session identifiers (`SessionIdents`), which are atomic objects (line 14).

Packets are represented only by their network headers (which of course are not unique identifiers of packets). As a simplifying assumption, network footers are not modeled. A network header is a `NetHdr` object, declared in lines 28-32 with the fields defined in Chapter 2. Currently the `protocol` field has only general information, but the `Protocol` signature can be expanded whenever needed.

Two other model pieces accompany the declaration of a NetHdr. There is a fact (lines 33-37) telling us that NetHdrs are "records," meaning that there is only one NetHdr object with a given set of field values. This prevents spurious problems with headers that are not equal (they are different objects), but are the same from a networking viewpoint (they have the same field values). There is also a convenient predicate `ReverseNetHdr` (lines 38-40) for checking that two headers reverse each other.

2.5 Network behavior

The forwarding behavior of a network is described by three tables, the `acquireTable, forwardTable`, and `transmitTable`, declared in lines 90-93. Note that the acquireTable and forwardTable are defined as separate objects (lines 69-72) rather than simple relations. This is due to Alloy only, and is not semantically important. In the internals of Alloy, the transmitTable is a set of four-tuples. If acquireTables and forwardTables were defined in the same way, they would be sets of five-tuples, which must be handled so inefficiently in Alloy that they really cannot be used.[1] Also note that there are some options in the tables that will not be explained until §5.

As you will see, the model distinguishes between *sending* and *transmitting* a packet, and between *receiving* and *acquiring* a packet. A packet is *sent* once, and either dropped on the way or *received* once.[2] In contrast, a packet being forwarded through a network is *acquired* by every forwarder on its path, and *transmitted* by every forwarder on its path.

The semantics of the tables is described operationally, as a flowchart, in Figure 2. In the figure, the attributes of each packet include its header and fields of metadata attached to it (or removed from it) as it flows through a network member. So a packet enters the forwardTable with an *inLink* and a *netHdr*. If it is forwarded, the forwardTable replaces its *inLink* with the identifier of an outgoing link *outLink*. The transmitTable removes the outLink metadata, leaving only the netHdr.

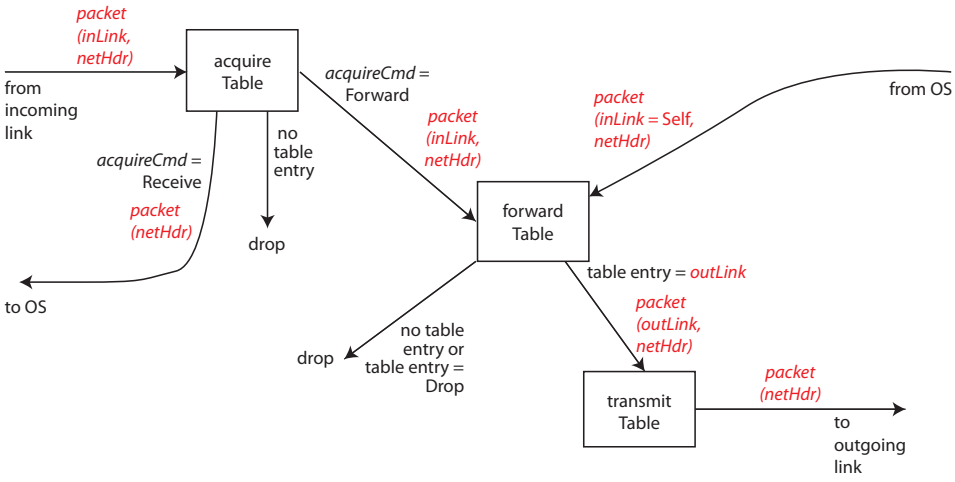

Figure 2: Operation of a network member with no network composition.

[1] The reason lies deep in the bowels of logic solvers, so it is not easily explained, and not confined to Alloy.

[2] If a protocol like TCP requests retransmission of a packet, then the retransmission, being sent at a different time, is a different packet.

Each table operates as a little packet processor inside the network member, which is controlled by the data in the table. If a packet is acquired from an incoming link, its inLink and netHdr are matched in the acquireTable. If there is no match, the packet is dropped. If there is a match, the table yields a command that steers the packet either toward the operating system or the forwardTable. The forwardTable processor matches a packet's netHdr and inLink in the forwardTable, yielding the identifier of an outgoing link outLink or the command `Drop`. If there is an outLink, then the packet goes to the transmitTable. If the yield is Drop or there is no match in the table, then the packet is dropped.

In this version the transmitTable has no important information, and its processor only removes metadata from packets before they leave the network member. There will be an expanded version of this table in §5, in which the transmitTable has greater operational importance.

You may have noticed that if these tables were instantiated for real networks, they would be extremely large. This is not a problem because the tables are merely being described for formal purposes, and will never be fully instantiated. Even the tables for forwarding, which must be implemented in some form, are not implemented like this. For example, in real forwarding tables, header patterns with name prefixes and wildcards substitute for explicit enumeration of all possible headers.

2.6 Reachability

2.6.1 The meaning of the model

This section concerns the meaning or semantics of `net.als`, which is captured by its derived `reachable` relation. Formal methods are subtle, and it will be necessary to explain the precise significance of this relation in several steps.

Our Alloy model leverages the conventional separation in networking between the "control plane" and the "data plane." For each network, the formal representation of its state contains primarily its current topology and the current state of its tables—data structures computed by the control plane and interpreted by the data plane. The meaning we assign to the network state is based on three fundamental assumptions:

- The model contains nothing but static network states, i.e. snapshots of what the current state of a network might be.
- We assume that all of the network components are functioning. If in the real world a link has failed, then we assume the link simply does not exist in the current network state. The same is true for network members.
- The formal semantics assumes that for each network member, there is an implemented "data plane" that *will* perform the packet processing implied or instructed by its tables. The `reachable` relation summarizes, based on a current network state, how the network would behave over a period of observation in which the network state does not change. Given full

information about a situation, the model tells us exactly what the network will do. If there is uncertainty in what the model tells us, it is because we have given it less-than-complete information.

There are three reasons for this approach: simplicity, simplicity, and simplicity. Any of these assumptions can be removed, and formal methods exist to reason about models without the assumption, but the result may become too complex for comprehension or efficient formal reasoning. Put another way, if you want a formal model with fewer simplifying assumptions, you should write one. Put yet another way, the subject matter of "all possible network architectures" is very broad. With narrower subject matter, more detailed or dynamic models become useful, as mentioned in §7.

2.6.2 The reachable relation

The behavior of a network in transmitting and forwarding packets is captured by its `reachable` relation, which is derived from other fields of a network object. Given all the assumptions made in the last subsection, the only remaining design freedom is in how much information to take into account.

Because the amount of information in `net.als` has been minimized deliberately, there is actually only one choice to make. Do we take into account the distinction between user members and infrastructure members? For example, we could include forwarding by infrastructure members, but not by user members. On grounds of generality, we do not take this distinction into account.

In subsequent Alloy files there will be more state information to use, and other reachability relations defined for more specific purposes. Our purpose now is to define the most basic version of reachability for individual networks. The principle is that all subsequent relations about individual networks, using more information, should be subsets of this basic relation. Note that the basic reachable relation includes group communication—for more on this topic, see §4. The basic relation has this declaration:

```
95    reachable: NetHdr -> members -> members          -- derived
```

For a tuple (h, m0, m1) in this relation,[3] the meaning is as follows:

> If a member m0 transmits a packet with header h (on any outgoing link), then that packet will be acquired by m1 (on some incoming link).

With this semantics, there is uncertainty about whether m0 will transmit such a packet, but if it does, it is certain that the packet will be delivered to m1 and acquired by it.

The definition of reachable begins with an auxiliary relation `oneStep`, declared and defined as follows:

[3]The comma notation for tuples is the easiest to use in text, but it is not Alloy notation (see below).

```
 94    oneStep: NetHdr -> links -> links                -- derived

170       oneStep = {  h: NetHdr, disj k, k": links |
171          some m: members, kin, kout: LinkIdent |
172             (m -> kin -> k) in inLinks
173          && (m -> kout -> k") in outLinks
174          && (kin -> h -> Forward) in (m.acquireTables).arows
175          && (kin -> h -> kout) in (m.forwardTables).frows  }
```

This relation describes the forwarding behavior of individual network members. For a member to forward a packet, it must have an incoming link and an outgoing link (lines 171-173). Note that `(m -> kin -> k)` is a three-tuple of individuals, which is written in Alloy as the Cartesion product (operator: single arrow) of three singleton sets. To forward a packet with header h, the acquireTable must say to forward it, and the forwardTable must map it from an incoming link to an outgoing link. For example, let's imagine a header h with src = A and dst = D. In the network illustrated in Figure 3, these are the table contents relevant to header h:

```
acquireTable                    forwardTable
B:  1 -> h -> Forward           B:  1 -> h -> 0
C:  1 -> h -> Forward           C:  1 -> h -> 0
```

so the tuples in `oneStep` beginning with h are { (h, k1, k2), (h, k2, k3) }. Note that oneStep need not use the transmitTable, as the transmitTable is giving information about how to transmit on a link, rather than whether to transmit on a link.

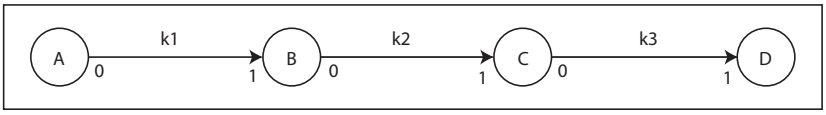

Figure 3: A simple network.

Here is how reachable uses oneStep, in a draft version:

```
reachableDraft1 = {  h: NetHdr, m, m": members |
   some k, k": links |
      m in k.sndr && m" in k".rcvrs
   && (k -> k") in ^(h.oneStep)                }
```

By joining a particular header with oneStep (on the last line), we get a binary relation on links. The transitive closure (carat) of this binary relation gives all forwarding paths for that header. Applying this definition to our particular header h, we get the following results:

```
^(h.oneStep)                    h.reachableDraft
k1 -> k2                        A -> C
k1 -> k3                        A -> D
k2 -> k3                        B -> D
```

This is an odd relation—it says that the packet reaches C and D from A, but not B from A. It also says that the packet reaches D from B, but not C from B. The problem is that the relation includes only paths with at least two links and one forwarding step. Here is a better definition, allowing paths with only one link:

```
       reachableDraft2 = {  h: NetHdr, m, m": members |
          some k, k": links |
             m in k.sndr && m" in k".rcvrs
          && (  k = k" || (k -> k") in ^(h.oneStep)  )  }

h.reachable
A -> B
A -> C
A -> D
B -> C
B -> D
C -> D
```

This definition represents the paths in a more sensible way, but it still has a problem: it is *too* inclusive. With this definition, whenever there is a link from member m0 to member m1, (h, m0, m1) is in the relation *for all headers h*. This holds because the relation does not take into account any information about whether m0 will transmit any such packet, nor any information about whether m1 will acquire the packet.

Fortunately, some information is available in the acquireTables and forwardTables. By using acquireTables, the definition can exclude packets that will not be acquired. By using forwardTables, the definition can also exclude packets that will not be forwarded and then transmitted. Both of these cases are shown as drop actions in Figure 2. So the final definition is:

```
176       reachable = {  h: NetHdr, m, m": Name |
177          some k, k": links, kin, kout: LinkIdent |
178             (m -> kout -> k) in outLinks
179          && (m" -> kin -> k") in inLinks
180          && (h -> kout) in
181                (LinkIdent + NoInLink).((m.forwardTables).frows)
182          && (kin -> h) in ((m".acquireTables).arows).AcquireCmd
183          && (  k = k" || (k -> k") in ^(h.oneStep)  )         }
```

2.7 Validation of network behavior

The reachable relation describes network behavior. In `net.als` there is a section with predicates about (properties of) network behavior (lines 255-270). `No_routing_loops` is self-explanatory. If a network state satisfies `Fully_reachable`, then every member can reach every other member. §4 will explain that packet replication for forwarding to broadcast or multicast groups appears as multiple entries in a forwardTable for the same header. So the predicate `Only_unicast-forwarding` is true of a network state in which there is no replication.

These predicates are used to make tests for validation, just as the topology predicates are (lines 336-429). First we check that the predicates can be violated (lines 340-350). This is important because, if they cannot be violated, then the model has too many built-in constraints, and does not represent the full range of possible network behaviors. Then we test that the predicates can be satisfied (lines 352-377), using topology properties to make sure the Alloy Analyzer produces interesting instances.

Another powerful validation technique is the use of theorems we call "sanity checks." If you can think of a property that all instances should have, just because that is how the model works, you should write it down as a theorem, then check that the theorem holds. If it does not, you have learned something important about your model—usually that it does not quite say what you mean!

Often sanity-check theorems are simple, even trivial. The sanity checks in this section are not so simple, although there are some trivial ones in the final section of `net.als`. As sanity checks, our two theorems (lines 379-429) fill in some of the table entries for hub-and-spoke networks and one-way rings. Each theorem then states a reachability property that should be implied by the table entries.

§6.3 of Chapter 2 says that reachability properties are the most prominent network requirements. For particular networks, reachability properties can be as general or specific as needed. Either way, they are formalized as properties of the network's reachable relation. Often blocking properties are distinguished from reachability properties, but these are really just reachability properties stated in the negative—such as "members of type X must not be able to reach members of type Y," meaning that no packet header will allow a member of one type to reach a member of another type.

2.8 Network equivalence

The final property of network behavior in `net.als` is network equivalence (lines 272-280). Two networks are considered equivalent if they have the same reachable relation. We will see the significance of network equivalence in §5.

Like other behavioral properties, network equivalence can be used to further validate the `reachable` relation. The predicates in lines 435-443 find two networks that are equivalent but have different links (`Test1`), and two networks that are equivalent but have different members (`Test2`). Like all other equivalence relations, network equivalence should be reflexive, symmetric, and

transitive. The file ends (lines 445-466) with sanity checks that this is so.

3 The traffic model

3.1 Traffic model as requirement

The traffic model of a network is a description of the packets that network members are expected to send and receive, where "receive" is meant in the active sense of accepting and passing to some module on the machine, as opposed to the passive sense of receiving on a wire or other communication channel. The traffic model of a network can be interpreted as a requirement on the network. If the traffic model were quantitative (which the Alloy model is not), such as a bandwidth, it could be a load requirement.

The traffic model is found in the file `traffic.als` (dated 3 March 2023 or later). For modularity in the Alloy model, it is an optional extension of `net.als`. The file begins with the signature of a `NetworkStateWithTraffic`, which is a NetworkState object with additional fields. The heart of the traffic model consists of two relations, a `sendTable` and a `receiveTable`, declared as follows:

```
12    sendTable: members -> NetHdr -> SessionIdent,

13    receiveTable:
14       members -> NetHdr -> (Primitive + ExternalLink),
```

where `Primitive` is a constant, and `ExternalLink` will be explained in §5. If a tuple (m, h, s) is in a network's `sendTable`, then member m is expected to send packets with header h. If a tuple (m, h, k) is in a network's `receiveTable`, then member m is expected to receive packets with header h.

We are dividing the Alloy model into files so it will be more manageable, but in this case the modularity is a little forced. Each table entry defined above has a third column or tuple entry, neither of which is necessary for its purpose as a requirement. They are, in fact, needed primarily for network composition, as will be explained in §5. In the meantime, Figure 4 uses the new tables to add slightly more detail to the packet processing. In particular, a packet can arrive at the network member, from within its own machine, with either a full NetHdr or just a SessionIdent. If it has just a session identifier, the information in the sendTable is used to attach a full network header.

3.2 Traffic properties

Lines 53-124 define various properties of traffic, useful for constructing particular traffic models for validation and verification. For example, a traffic model is "fully active" if there are sessions between all pairs of members (with various constraints for the various versions of this predicate). The remaining properties have comments to explain what they mean. Because there are no explicit

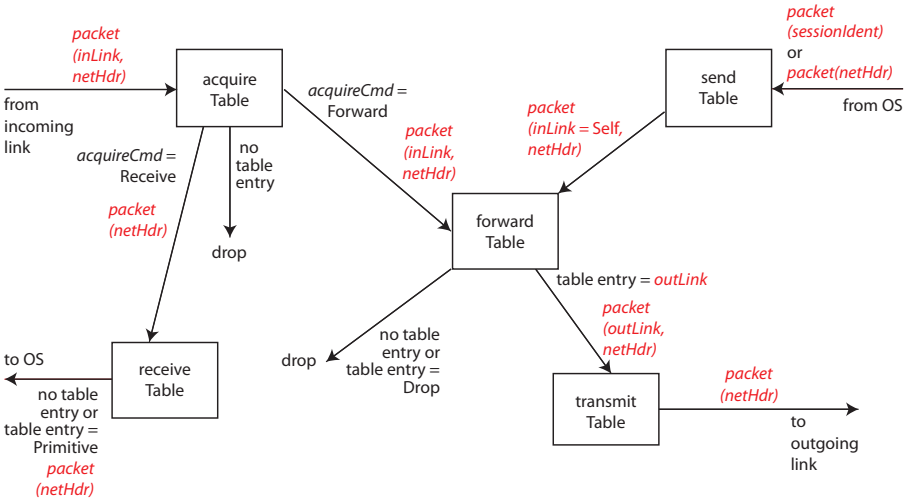

Figure 4: Operation of a network member with no network composition, but with a sendTable and receiveTable.

sessions in the model so far, sessions are represented by their corresponding headers in the traffic model.

The traffic properties are used for validation. Lines 194-268 instantiate the properties in various combinations and in various network topologies, to show that they are consistent.

3.3 Effective reachability

The traffic model introduces new information, which can be used to define a narrower form of reachability called "effective" reachability. The relation `effectivelyReachable` has the same type as reachable, but a tuple appears in it only if supported by the traffic model.

Effective reachability is defined with the help of two derived relations:

```
15    effectiveSend: members -> NetHdr -> Link,        -- derived
16    effectiveReceive: members -> NetHdr -> Link,     -- derived
```

Let's consider the difference between sendTable and effectiveSend. The sendTable may indicate that a member m sends packets with a header h, but if m's forwardTable does not forward these packets, they will never leave m. However, if a tuple (m, h, k) is in effectiveSend, we know that m will forward sent packets with header h, onto outgoing link k. Similarly, a member m's receiveTable may indicate that it receives packets with header h, but this will not help unless m accepts them first. But if a tuple (m, h, k) is in effectiveReceive, we know that m will accept receivable packets with header h, from incoming link k. With the

help of these derived relations, the definition of `effectivelyReachable` (lines 43-50) is a straightforward variation on the definition of reachable in `net.als`.

The effectivelyReachable relation adds endpoint information to forwarding information. According to the principle in §2.6.2, because effectivelyReachable is based on more information than reachable, it should be contained in reachable. The `Effective_reachability_inclusion_theorem` (lines 178-182) validates that this sanity check holds. Another consequence of focusing on endpoint information is that reachable includes middleboxes on the path between sender and receiver, while effectivelyReachable excludes them. This difference is illustrated by the predicate `Effective_reach_excludes_middlebox` (lines 184-192), which constructs a network in which m0 sends packets received by m2. The packets go through a middlebox m1 on the way; this can be seen in reachable but not in effectivelyReachable.

Like reachable, effectivelyReachable can be the basis of an equivalence relation on networks. This is validated for effectivelyReachable in lines 334-357 of `traffic.als`.

3.4 Properties of network behavior

Whenever there is a formal requirement for a system, the primary verification task is to verify that the system satisfies the requirement. For example, the predicate `Network_satisfies_communication_demands` (lines 130-136) interprets a network's send and receive tables as a communication requirement, and compares the network's behavior (as captured by its effectivelyReachable relation) to them. If the communication required by the send and receive tables is provided by effectivelyReachable, then the network is verified correct with respect to that requirement.

Other behavioral properties defined in this section are `Only_authentic-traffic_delivered` (received packets must have authentic source fields in their headers) and `Effective_reachability_is_symmetric`. The latter holds if and only if all the effective sessions are two-way. For validation purposes, these properties are instantiated in various ways in lines 271-318.

4 Group communication

4.1 Group basics

So far the model includes only point-to-point sessions and links. This section will explain how anycast and allcast groups are added to the formal model (recall from Chapter 5 that *allcast* groups formalize both broadcast and multicast groups). Group communication is found in the file `group.als` (dated 9 March 2023 or later). For modularity in the Alloy model, it is an optional extension of `net.als` and `traffic.als`.

Our first concern is naming. Group names (found in `groups`) cannot also be member names (line 14). For each group there is a set of senders and a set of

receivers, as determined by the relations `groupSenders` and `groupReceivers` (lines 16-17). Although senders and receivers are not necessarily members of the network (see §5.1), they cannot be groups. One difference between anycast groups and allcast groups is that the senders of an anycast group cannot also be receivers (line 26).

In the book, group names are assigned to network members so that packets can be delivered to them by means of their group names as well as their unique names. Thus, a receiver in an anycast group has the anycast name as well as its unique name, and a receiver in an allcast group has the allcast name as well as its unique name.

We now introduce the first guiding principle of the formal model, stated informally as "group communication = point-to-point communication." It means that, to the greatest extent possible, the capabilities and generality of group communication channels should be the same as those of point-to-point communication channels. The purpose of this principle is to guard against unnecessary exceptions and special cases. It also has the advantage of simplifying the formal model overall.

As the book emphasizes, there are two kinds of communication channels: sessions and links. These two will be discussed in separate sections.

4.2 Group sessions

A two-way point-to-point session consists of packets with two different senders. Allcast sessions can include packets from more than two senders, but the destination field of all packets is the group name. For example, DHCP sessions are allcast sessions with potentially several senders.

Allcast service (broadcast or multicast) can be implemented by forwarding, as follows. A forwarding table is nondeterministic when there is more than one row (tuple) with a given incoming link and NetHdr, mapping to different outgoing links. The semantics of nondeterminism is packet replication: the network member makes as many copies of the packet as there are outgoing links, and sends one copy to each link. Packet replication makes it possible for a packet sent once to be received at multiple network members. No extension of `net.als` or `traffic.als` is needed for packet replication, as it is already included in all previous definitions and constraints.

The model of allcast service is validated in several ways. The predicate `Fully_allCast_active` (lines 93-105) is satisfied by a network with an allcast group including all members as both senders and receivers. There is a single allcast session—it has only one session identifier—in which every network member is a sender. It is instantiated by `GroupTrafficTest1`, and by the trivial sanity check `AllCastTrafficTheorem` (lines 214-229). `GroupBehaviorTest1` through `...Test4` (lines 242-279) show how allcast service can be implemented by forwarding in various network topologies.

Anycast service is used to create point-to-point sessions. There can be true anycast sessions, which are point-to-point sessions in which the destination of all messages in one direction is an anycast group name. Obviously, for these sessions

to work, messages must be forwarded with session affinity. More commonly, anycast delivery (without session affinity) is used for the setup message only of a dynamic session. Then the remaining messages of the session have unique names as destinations, as explained in Chapter 2.

Anycast is usually implemented with forwarding. The semantic effect of anycast forwarding is that some network members' forwarding tables change over time, so that setup messages destined for the group are spread fairly over the group members. Anycast semantics cannot be represented directly in our formal model, for the simple reason that there are no state changes. Anycast semantics can be represented indirectly, however, by constraining forwarding tables enough to include all possible choices among the anycast receivers, and to exclude all impossible choices. Then any reachable relation based on these tables will represent what *can* happen with the anycast group, but not necessarily what *will* happen in any particular case.

Anycast service is validated with the predicate `Fully_anyCast_active` (lines 107-122). The exact meaning of this predicate is explained in §5.1.3, where it is used to illustrate a solution to an important problem. In this file it is instantiated by `GroupTrafficTest2` (lines 231-236), by `GroupBehaviorTest6`, which shows that anycast service can be implemented by forwarding, and by `GroupBehaviorTest7`, which shows that anycast service can be implemented by links (lines 290-302).

4.3 Group links

Although it has not been discussed so far, Link objects are general enough to include links that provide group communication channels (lines 23-24 and 42-48 of `net.als`):

```
sig Link {
   sndr: Name,
   sndrIdent: LinkIdent,
   rcvrs: some Name,
   rcvrIdents: rcvrs -> one LinkIdent,
   mode: OneWay + TwoWay + AllCast + AnyCast,
   group: lone Name
}
```

A link with mode `AllCast` or `Anycast` must have a corresponding `group` name. Group links usually have multiple receivers (the qualifier `some` means one or more), and the relation `rcvrIdents` maps each receiver to its local identifier for the link.

For computation of reachability, all links must be directional. This means that a "link" represented in the model is actually a link plus direction, which implies that each link can only have a single sender. Figure 5 shows what this means for the allcast links implementing an allcast group.

As the figure shows, A is a group sender only, C and D are group receivers only, and B is both a sender and receiver. Overall the group has two senders,

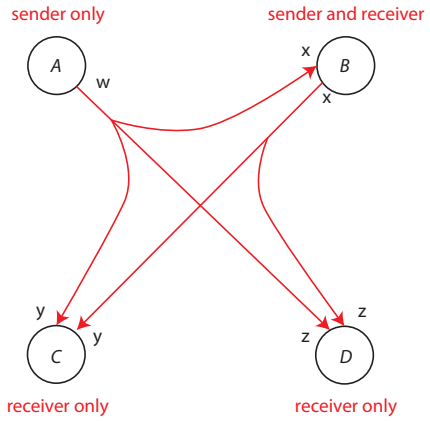

Figure 5: Allcast group links fully implementing an allcast group with multiple senders. Letters w through z represent link identifiers.

so two allcast links are needed to implement it fully with links. Each link has a single sender and a receiver for each group receiver except the sender (if it is also a receiver); the semantics of an allcast link is that a packet transmitted by the sender is replicated and delivered to *all* link receivers. At each member, all links implementing the same group share the same link identifier.[4]

Anycast links are topologically the same as allcast links. The semantics of an anycast link is that a packet transmitted by the sender is delivered to *one* of the receivers. No extension of `net.als` or `traffic.als` is needed for group links, as they are already included in all previous definitions and constraints. The only problem with group links is the same as with group sessions: because it takes a dynamic model to give a real anycast semantics, including fair choice from the receiver set, when a network path contains anycast links its semantics is weakened from what *will* happen to what *can* happen.

In `group.als`, tests and a sanity check validate network topologies with group links (lines 158-208). `GroupBehaviorTest5` and `GroupBehaviorTest7` (lines 281-302) show that both allcast and anycast services can be implemented with group links alone.

From the model, the capabilities and generality of group communication channels (at least virtual ones) seem to be the same as those of point-to-point communication channels, satisfying the principle that "group communication = point-to-point communication." The problem with anycast semantics is a limitation of the current model, not an anomalous characteristic of anycast sessions or links.

[4]Because there can be more than one point-to-point link from one member to another, to maintain the principle that group communication = point-to-point communication, it must be possible to have two disjoint sets of links implementing the same group. In this case, the two link sets are distinguished by having different link identifiers at each member.

5 Network composition

The three composition operators on networks—bridging, layering, and subduction—are all defined in `compos.als` (dated 10 March 2023 or later). As before, line numbers refer to `compos.num`.

5.1 Bridging

5.1.1 Bridging basics

A `Bridging` object (lines 20-67) declares that networks in the set `nets` are bridged together. This could also be derived by analysis of the bridging links in a set of networks, so it is not important by itself. The point is to provide a place in the model to store the semantics of bridging within a particular set of networks. This semantics is found in the relation `reachableWithBridging`. Its definition is similar to the definition of reachable, but it takes the tables of all the networks into account.

To get an unambiguous model of bridging, it is necessary to extend the uniqueness of member names: now all the member names in a set of bridged networks must be unique (lines 27-28). For the same reason of preventing ambiguity, the same Link object cannot be used for two different purposes in the bridged set (lines 50-55). In keeping with the guiding principle of §4, bridging links can be point-to-point or group links. However, there are several constraints on group names and membership, which will be explained in See §5.1.3.

In lines 69-84, Bridging objects are extended to use traffic models. In lines 493-695 bridging is validated with various tests and sanity checks.

5.1.2 Guiding principle: Reachability is a structured partial order

This second guiding principle was foreshadowed in §3.3, when we made the point that the effectivelyReachable relation is contained in the reachable relation.

The semantics of network composition operators are captured in various reachability relations. All reachability relations have the same type, so set inclusion is a partial order on reachability relations. The guiding principle "Reachability is a structured partial order" refers to inclusion in two ways. First, it says that we should know exactly where each reachability relation fits into the partial order. Second, it says that we should explain any additional properties of the partial order in terms of the aspects of network architecture that produce them. The inclusion ordering with isolated networks and bridging is shown in Figure 6.

To get a fair comparison between reachableWithBridging and reachability of isolated networks, for any set of networks `nets`, we compare reachableWithBridging in a Bridging object with nets to the union of reachable relations in all the networks of nets. In the figure, all arrows indicate proper inclusion, meaning that for some nets, the upper relation is larger than the lower relation.

The additional property, shown by annotation in Figure 6, is that we can use architectural information to predict when a reachability relation for isolated

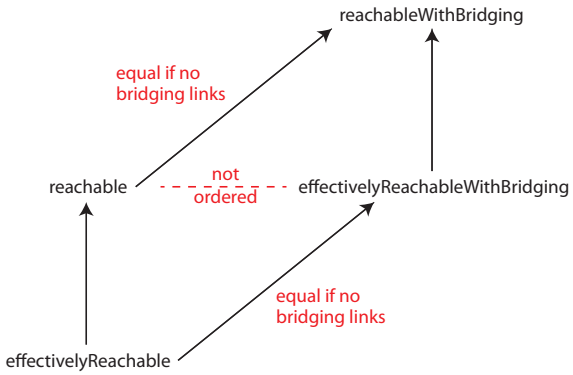

Figure 6: Inclusion is a partial order on reachability relations. An upward arrow denotes that the lower relation is included in the upper relation.

networks is equal to the corresponding relation for bridged networks. This occurs whenever the set of networks nets has no bridging links.

All of these properties are checked in lines 621-695. Containment is a sanity check (theorem). Proper inclusion is shown by instantiating a predicate that the upper relation is larger than the lower. A "vestigial" theorem predicts when the upper and lower relations will be equal.

Finally, there is no ordering between reachable and `effectivelyReachableWithBridging`. Compared to effectivelyReachable, which is included in both, reachable extends it logically (by not considering the traffic model) and effectivelyReachableWithBridging extends it structurally (by including bridging links). Neither extension need be contained in the other, as a test confirms.

5.1.3 The Bridging Closure theorem and the flat Internet

The Bridging Closure theorem, stated and verified in lines 597-619, says that any set of bridged networks is equivalent (in the exact sense defined in §2.8) to a single network. In mathematical terms, the set of all networks is closed under bridging.

This result is significant for Internet architecture. Throughout Chapter 4 of the book, we refer to "the flat Internet" as a single network, even though it is a bridging of many networks. The Bridging Closure theorem is the technical justification for this shorthand.

There is a very significant challenge in defining the formal semantics of the flat Internet, which is the reuse of private IP names and allcast group names (for example, the broadcast group name 255.255.255.255) in different IP edge networks. The reuse of private IP names violates the bridging constraint that all member names in a set of bridged networks must be unique. We will now explain how this challenge is overcome with the use of group names.

Firstly, group names are part of a network's name space, so they must be properly constrained. Within a Bridging object, group names and member names cannot overlap (lines 47-49). A declaration of a group's name, senders, and receivers must appear in every network hosting a member of a group (lines 43-46). The constraints do not say, however, that group names must be globally unique. Rather, there are nonlocal group names, which must be unique across the set of networks and may have members in multiple networks, and local group names, which need not be unique across the set of networks, but whose members must be completely confined to one network (lines 29-42). Consequently, the IP broadcast group name 255.255.255.255 can be a local group of every network in the flat Internet.

To deal with private IP names, we must assume that every member of every network in the flat Internet has a globally unique name. If the network—in practice—uses private IP names for some members, then the unique names of these members must be fictitious, in the sense that they exist in the model but not in real life. In the network, the private IP names of these members are anycast names also assigned to them (along with their fictitious unique names). Each anycast group has one receiver (the member with that anycast name), and has every other member of the network as a potential sender. The anycast groups are local, so the same anycast names can be reused in other networks. For completeness, there should be a constraint that fictitious names do not appear in IP headers, so that members with private anycast names are always referred to in IP headers by their anycast names.

When IP networks are modeled in this way, it is possible (theoretically, because of scale) to have a complete reachable relation for the flat Internet. Note that a reachable relation contains unique member names only, so there is no semantic ambiguity. In edge networks with private anycast names, anycast service is typically implemented by forwarding within the local network.

5.2 Layering

5.2.1 Layering basics

A `Layering` object (lines 94-203) declares that a set of bridged overlay networks is layered on a set of bridged underlay networks. The `implementations` field of the Layering object maps implemented overlay links to the session identifiers of the underlay sessions implementing them. The `attachments` field of the Layering object maps overlay members to the underlay networks and members to which they are attached. This is enough information to describe completely[5] how layering should work.

Note that the underlays are not described in the Layering object as a Bridging object, but as a BridgingWithTraffic object. This is because the traffic model has been recruited to serve a new purpose. Now, in addition to their original purpose, sendTables and receiveTables store some of the network data required to make layering work.

[5]Not counting layering involving group communication, with is explained below.

Unlike bridging, layering does not change the semantics of any network. All it does is remove the necessity of providing physical implementations of network links, by implementing them in one or more underlay networks. So there is no new reachability relation for layering, and no equivalence relation associated with layering.

Note that names in the overlay networks can overlap with names in the underlay networks it is layered on. In fact, it is common for overlay members to be attached to underlay members, each of which has the same name as the member attached to it. In these instances of layering, there is no need for a directory mapping overlay names to underlay names.

We will show how layering operates with the help of Figure 7, which is the same as Figure 4, except that the network of Figure 4 is now identified as network B. Also, the figure contains one table from an overlay network A, and two tables from an underlay network C. This figure defines packet processing at a machine with members of networks A, B, and C (and possibly others). It includes both composition by layering and composition by subduction, although subduction will not be discussed until §5.3.

For layering, we must consider two aspects of packet-processing behavior. First, what happens when a packet is transmitted on a virtual link of B? The transmitTable of B will contain an `ExternalSession`, which is a record containing an external underlay network (here C) and the identifier of the implementing session. The session identifier and packet are passed to the sendTable of network C for further handling.

When sendTable C gets a packet from the machine's operating system, it can be in the form of a packet with no network header, plus a session identifier. When sendTable C gets a packet from transmitTable B instead, it comes in the same form. Note that this is *exactly the same packet* that entered transmitTable B. It has a network header for network B, but to the tables of network C, this is just part of the payload.

Second, what happens when a packet is received in a session of B that implements a virtual link? The receiveTable will contain an `ExternalLink`, which is a record containing an external overlay network (here A) and the link identifier of the incoming virtual link on which it is being acquired. The incoming link identifier and packet are passed to the acquireTable of A for further handling.

When acquireTable A gets a packet from a physical link, it is in the form of a packet with a header for network A, along with a link identifier of the incoming link. When acquireTable A gets a packet from receiveTable B instead, it comes in the same form. Note that this is *almost the same packet* that entered receiveTable B. Processing in the receiveTable B component has removed its outer header for network B. The formal model guarantees that inside its payload, there is a header for network A.

To round out this explanation, Figure 8 shows the Layering fields and table entries related to the implementation of a link. This link just happens to be a bridging link implemented by a session spanning bridged underlay networks, which allows the maximum number of networks to be involved.

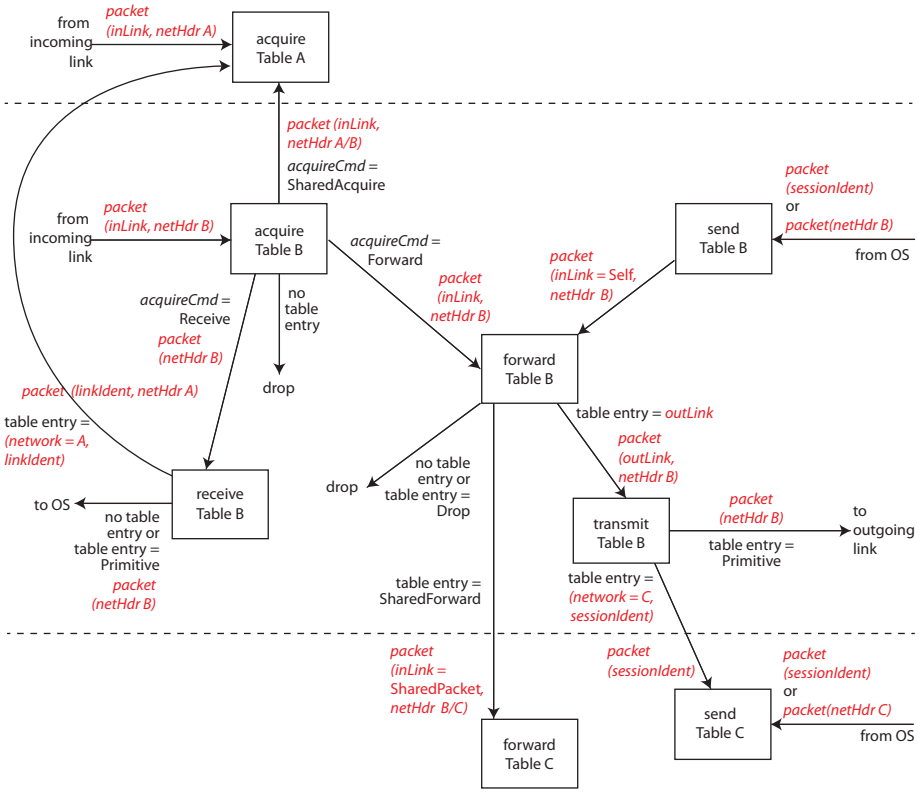

Figure 7: Operation of a networked machine with members of three networks composed by layering and/or subduction.

5.2.2 Guiding principle: Links = sessions

The third, and most important, guiding principle of the model is summarized informally as "links = sessions." It means that the capabilities and generality of links should be exactly the same as those of sessions. Because layering requires a correspondence between links and sessions, without this principle, layering might be constrained unnecessarily. In other words, this is the principle that allows networks to be composed like Lego blocks.

Although this guiding principle is fully satisfied by the model, you probably have not noticed it. There are two reasons for this. One is that the formalization of sessions is lightweight, consisting only of their unique session identifiers. The other is that every detail of the formal model has been designed, from the beginning, with this principle as a goal.

Perhaps the best example of this principle combines it with the first principle, resulting in the idea that group links should have the same generality as group

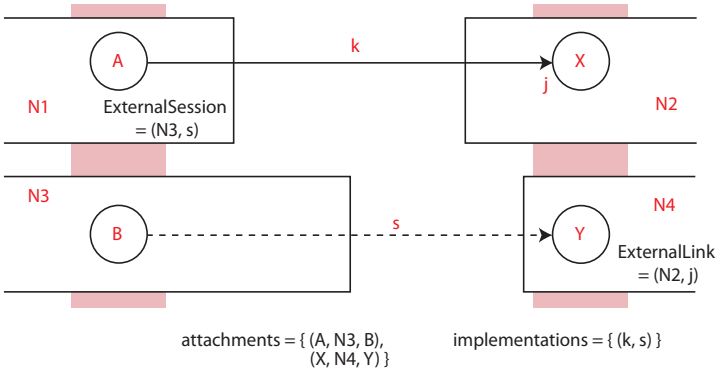

Figure 8: Implementation of a bridging link. The implementing session can also be contained within a single network.

sessions. In particular, it should be possible, with layering, to implement group links with group sessions. This is possible, as evidenced by `LayeringTest14` (lines 887-899) and `LayeringTest15` (lines 901-912).

5.2.3 Satisfaction property and validation of layering

There is really only one interesting property related to layering. `Underlay-satisfies_layering_demands` (lines 213-225) says that reachability in the underlay networks is sufficient to support all the sessions implementing overlay links. Without this property, constructing the right table entries at session endpoints does not do much good.

Layering is validated with many tests (lines 697-912). Every test includes the property Underlay_satisfies_layering_demands, because without it, the layering is useless. The tests are highly varied, making use of properties from all the other Alloy files. They test implementation of bridging links, and also implementation of links with sessions over bridging links. They test layering in a three-level hierarchy. They test that self-links can be implemented by self-sessions. They test implementation of group links. They also test many different ways in which different layering operators can operate on the same networks, despite the uniqueness constraints.

5.2.4 Constraints on layering

The purpose of a Layering object is to store the state necessary to implement some collection of links by means of layering. To enable this state to be modular, the formal model is designed to allow multiple Layering objects to apply to the same networks. This way, each purpose for layering can be formalized separately, and the effect of these Layering objects is cumulative.

Recall from §5.1.3 that, in a Bridging object, group names can be used locally or nonlocally. In a Layering object, only links of nonlocal groups can be implemented (lines 143-148). This leads to a good example of the need for multiple Layering objects: what if, in the flat Internet, multiple edge networks have groups with the same names (broadcast or private IP names), and it is necessary to implement group links in these networks? The answer is multiple Layering objects—in each object, the group links of only one edge network are implemented, so in that object the groups are local.

The price we pay for this convenience is the risk that some Layering objects might conflict with each other in some way. Consequently, there are global constraints on Layering objects (lines 227-263) to ensure that they can co-exist safely.

First, `Layering_is_hierarchical` (lines 227-232) says that the dependency of overlay networks on their underlay networks must be a hierarchy, i.e., can be pictured as a directed acyclic graph. A network cannot depend on itself!

Second, there must be a one-to-one relationship between implemented link and implementing session. So `Implementation_is_unique` (lines 234-239) says that only one Layering object can implement a particular link. Its companion constraint, `Implementation_user_is_unique`, says that a session can implement only one overlay link.

The trickiest constraints concern attachments, because it is possible for a network to have multiple members on the same machine. Consider what would happen if an underlay member could be the attachment of more than one member of the same overlay network. If the member were playing the role of underlay session receiver, as Y is in Figure 8, it would not know which member of N2 to deliver a session packet to. From examples, this kind of attachment does not seem to be necessary, so it is prohibited by the constraint `Underlay_member_attaches_unique_overlay_member` (lines 247-254).

Also consider what would happen if an overlay member could be attached to more than one member of the same underlay network. If the member were playing the role of overlay link sender, as as A is in Figure 8, it would not know which member of N3 to pass sent packets to. This kind of attachment does not seem to be necessary, either, so it is prohibited by the constraint `Overlay_member_attaches_to_unique_underlay_member` (lines 256-263).

In all cases, members can be attached to different members of different networks, or be the attachment of different members of different networks.

Although the formal model is designed for generality, it is possible to take its generality too far. For example, because the location of members on machines is not made explicit in this formal model, it is possible for two members of the same network on the same machine to have a network link between them, and even for that link to be virtual, thus implemented by other networks! Why is this inadvisable? It would be taking a building block (communication between modules on the same machine) and something built on top of it (networked communication) and then re-building the building block on top of both.

5.3 Subduction

Before reading this section, it might be a good idea to refresh your memory by reading about subduction in Chapter 4.

5.3.1 Shared members and links

In the first version of compositional network architecture [5], it seemed that it would sometimes be necessary to layer networks on themselves. This was bad news for rigorous reasoning, because so many properties rely on inductive reasoning about hierarchical relations (relations pictured as directed acyclic graphs). Fortunately, it is now clear that the right way to model these perplexing situations is with subduction. Subduction is an extension of layering, and preserves its hierarchical properties.

Recall that when one network is layered on another, their namespaces often overlap—more specifically, an overlay member often has the same name as the underlay member it is attached to. Recall also that networks bridged together must have disjoint name spaces.

In a Layering object that is extended to be a `Subduction` object, there are `sharedLinks` and `sharedMembers`, each of which is part of an overlay network *and* an underlay network. In a pair of shared members, the overlay member must be attached to the underlay member, and both must have the same name.

Subduction is also like bridging in the sense that the overlay networks and the underlay networks can both be bridged sets of networks. However, subduction adds something new, which we know because the overlay networks and underlay networks *cannot* be bridged with each other—precisely because they do not have disjoint name spaces! Subduction makes it possible to get the semantics of bridging a network together with a network layered on it.

When visualized (see examples in Chapters 1 and 4), a shared link always looks like two separate links sharing a common endpoint. One of these links is a peer-to-peer link within the underlay networks, while the other link is a bridging link between an overlay network and an underlay network. At the common endpoint, the underlay network member sees a single link that is normal in every way; if the link is two-way, it can both send and receive packets on the link. On the other end of the shared link, it has endpoints in two different network members on the same machine. The semantics of subduction allows these two members, which are shared members, to coordinate their use of the shared link. These shared members must have the same names because, from the perspective of the shared link, there is only one endpoint. For simplicity of implementation, a shared link has the same link identifier at both of its shared members.

There is one major violation of the principle that group communication = point-to-point communication: a shared link cannot be a group link. Given that a shared "point-to-point" link already has more than two endpoints in reality, this constraint should come as no surprise.

All the new fields in a Subduction object are derived (lines 296-299). There

are many constraints on shared links and shared members (lines 301-340). There are also constraints on the layering itself (lines 341-347). Lines 348-375 constrain the acquire and forward tables of shared members so that they produce legal packet processing.

5.3.2 Packet processing

Packet processing for subduction is also shown in Figure 7, although it was not pointed out before.

When a packet arrives at a physical inLink of network B (playing the role of an underlay member), if the link is shared, the table entry in B's acquireTable may have the command `SharedAcquire`. This means that the packet should be processed as if it arrived at the overlay endpoint of the shared link. It is passed to the acquireTable of network A (playing the role of overlay member), where it is processed normally.

The red notation on this arrow in the figure says "netHdr A/B," meaning that it has a network header suitable for both A and B. Just as bridged networks are required to have the same header design, so are networks composed by subduction.

This explains how a subduction packet ascends from the level of the underlays to the level of the overlays in a session view. How does it descend again? Figure 7 illustrates this with network B playing the role of overlay and network C playing the role of underlay.

If a packet arrives at the forwardTable of a shared member in B, it can simply be forwarded onto any outgoing link, shared or otherwise. However, B can also delegate its forwarding to C. To do this, it must have the command `SharedForward` in place of an outgoing link in its forwardTable. In this case, the packet is given the inLink annotation `SharedPacket`, and passed to the forwardTable of the shared member in C. As in the ascending case, it has a header suitable for both networks.

In the forwardTable of C, the packet must match an entry with its header and `SharedPacket` in the place of an inLink identifier. Because this entry is coordinating with B, it cannot forward the packet onto any outgoing link. It must forward the packet onto a *shared* outgoing link, because this is something B could have done.

Because subduction is an extension of layering, all the uniqueness properties of layering apply. There is also one new global constraint on subduction, which is that a shared member cannot be shared among three or more networks (lines 430-435). This is prohibited because it would make subduction semantics dependent on more than two levels, which seems difficult to define and probably ill-advised.

5.3.3 Properties and validation

Figure 9 is a straightforward extension of Figure 6. As with bridging, there is an "effective" version of subduction that makes use of the traffic model of the

overlays.

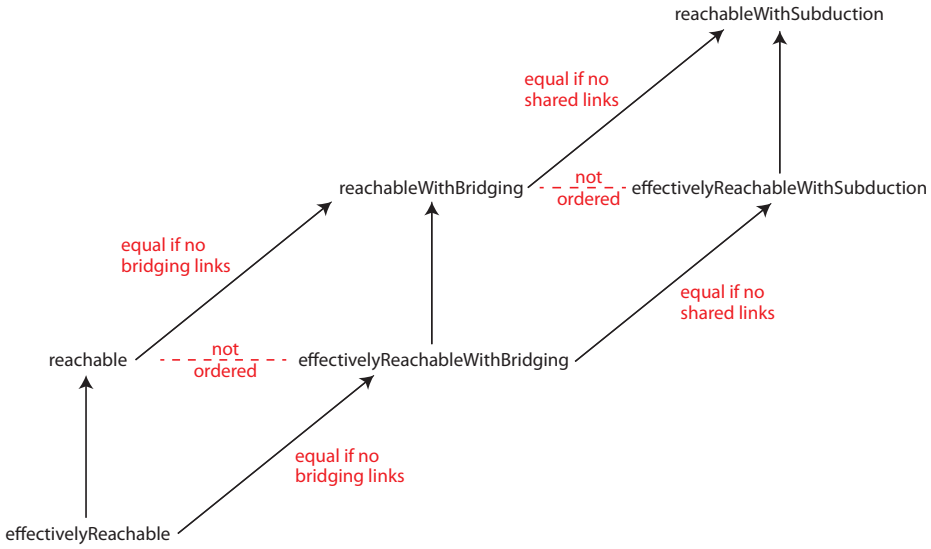

Figure 9: The full graph of reachability relations, partially ordered by inclusion.

For a fair comparison, `reachabilityWithSubduction` is compared to the union of reachableWithBridging in its overlay and underlay components. In the absence of shared links, subduction adds nothing to reachability. As before, all the properties in the figure are validated with tests or verified with theorems (lines 1018-1146).

6 Summaries

6.1 Simplifying assumptions

6.1.1 Unique member names

In compositional network architecture, there are no absolute rules about member names. However, by convention in the formal model, every member of a network has a unique name in that network, and unique names are conflated with members (§2.2.2). Group names are distinguished from unique names, and are unconstrained.

As explained in §5.1.1, for networks to be composed by bridging, the uniqueness of each member name must extend across the bridged set. As explained in §5.3.1, for networks to be composed by subduction, each pair of shared members must have the same unique name.

We have shown, in §5.1.3, that these restrictions are no impediment to modeling the flat Internet with its use of private IP names. The same modeling

techniques can be used to cover NDN networks, in which members have many anycast names but no unique names of their own. NDN members can be given fictitious unique names, which never appear in packet headers.

6.1.2 No network footers

Network headers on packets are modeled, but network footers are not (§2.4).

6.1.3 Only states are modeled

The model can be used to describe only the static state of a network or set of composed networks. Although reachable relations represent how static networks would behave over time, they do not represent behavior of networks in which there is any intermediate state change (§2.6.1).

6.1.4 No failures

Component failures are not modeled. If a component has failed, it simply does not exist in the modeled network state (§2.6.1).

6.2 Guiding principles of the formal model

6.2.1 Group communication = point-to-point communication

This principle (introduced in §4.1) says that the capabilities and generality of group communication channels should be the same as those of point-to-point communication channels, or as close as possible. The purpose of this principle is to guard against unnecessary exceptions and special cases. It also has the advantage of simplifying the formal model overall.

6.2.2 Reachability is a structured partial order

This principle is explained in §5.1.2. The fruits of the principle are shown in Figure 9. The principle, when applied to the domain of network architecture, tells us which properties should hold for which pairs of relations. The uniform patterns observable in the figure protect us from accidental omissions, and also reveal the real semantic distinctions among the relations.

6.2.3 Links = sessions

The capabilities and generality of links should be exactly the same as those of sessions, to facilitate layering. It is the similarity of links and sessions that allows networks to be composed like Lego blocks, so its importance cannot be overstated.

7 Limitations and potential extensions

7.1 Arbitrary compositions of networks

Although the model gives the semantics of particular bridging, layering, and subduction operations, it does not give a semantics for arbitrarily nested composition operations. To do this, it would be necessary to define a generic `Composition` object that can be specialized as a Bridging, Layering or Subduction object, and can be used—with suitable constraints—as a field in any `Composition` object.

This seems feasible, and it might be worthwhile, especially if it yields new insights. The reason it has not been done so far can be found in the many verification examples in Chapter 6. In these examples, the emphasis is modularity—on getting meaningful results by verification over the smallest possible scope, which is usually one individual network. In other words, a global reachability relation might be obtainable, but it seems unlikely that anything useful could be said about it, apart from the behavior of individual networks or clusters of networks within it.

7.2 Composition operators on sessions

In addition to the three composition operators on networks, compositional network architecture features two other composition operators that work on a smaller scale: protocol embedding and compound sessions. The formal model should definitely be extended to include these operators, which will require additional detail about sessions, session protocols, and session properties.

7.3 Path properties

There are many interesting properties of the paths that packets take through networks. Do all packets of a flow take the same path? Do all packets of a flow pass through specific middleboxes? If so, in which order? Here a "flow" can be any specified aggregate of packets.

Currently, the model provides no convenient way to represent path properties. Presumably all or most of the information is already available in the network tables, but the model seems to need convenient derived objects for formalizing the properties themselves.

7.4 Modeling of state transitions

The current formal model represents static network states only. However, the capabilities of Alloy 6 make it easy to convert it into a temporal model with mutable states and state transitions. This will be convenient for exploring aspects of networking requiring more detail of this kind.

We have already done this once to study session-location mobility (see Chapter 5), which is one of the two patterns for providing mobility service in networks.

In [4] we modeled the state transitions involved when endpoints of a point-to-point session move during the session, and verified that session connectivity is maintained even if both endpoints move simultaneously.

7.5 Alloy scopes and verification

As all Alloy users understand, when "verification" is mentioned in this document, it does not mean true verification, but only verification within a bounded universe (scope) of atomic objects.

For many application domains bounded verification seems completely adequate, but it is less so for networks. Because of the numbers of members, links, and other important objects in network models, some of our network analyses (including more advanced work not reported here) are limited to scopes of disappointing size. This is a situation we are keen to improve, making use of whatever opportunities appear in the future.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communications Review*, 44(3), July 2014.

[2] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006, 2012.

[3] J. Sonchack, D. Loehr, J. Rexford, and D. Walker. Lucid: A language for control in the data plane. In *Proceedings of SIGCOMM*. ACM, 2021.

[4] P. Zave and J. Rexford. Compositional network mobility. In E. Cohen and A. Rybalchenko, editors, *Proceedings of the 5th Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 68–87. Springer LNCS 8164, 2014.

[5] P. Zave and J. Rexford. The compositional architecture of the Internet. *Communications of the ACM*, 62(3):78–87, March 2019.