A Formal Model of Addressing for Interoperating Networks

Pamela Zave

AT&T Laboratories—Research, Florham Park, NJ 07932, USA
pamela@research.att.com

Abstract. Designing network address spaces for interoperation among domains is a challenging task. A formal model in Alloy is used to clarify the problems and explore solutions. Basic connectivity requirements are proposed, and two different sets of constraints are shown to satisfy them.
Keywords: networks, network design, network requirements, Alloy

1 Introduction

Universal connectivity is an important goal of networking. Today the world-wide network is divided into a vast and diverse collection of administrative domains. These domains include topologically distinct networks such as cellular networks, WiFi networks, and private IP subnetworks. They also include overlay networks such as virtual private networks and protocol-specific voice-over-IP networks.

To achieve the goal of universal connectivity, administrative domains must interoperate. By definition, an administrative domain controls its own address space. Yet interoperation requires that a client attached to one domain be able to produce and use an address identifying a client attached to another domain.

This paper concerns the problem of designing address spaces and interoperation mechanisms that satisfy basic connectivity requirements. This is more difficult than it sounds at first hearing. Addresses[1] can be non-unique, syntactically constrained, scarce, transient, and used for many purposes at many levels of abstraction. There is no established notion of "good addressing" [4].

Equally important, interoperation is an inherently confusing subject. This work was motivated by my experience designing an overlay voice-over-IP network [1]. Our team had seemingly endless discussions about interoperation, which never led to any clarity or comfort with the subject. I built the formal model described here in the hope of dispelling that confusion. Unfortunately, now that the confusion is gone, it is impossible to recreate what was so confusing. Fortunately, now that the model exists, no one need experience that particular confusion again.

Another difficulty is that very little is known about the user-level requirements on connection networks. Networking has always been and still is, to an

[1] The identifiers used in networking are known as as *names, addresses,* and *locators*, among other things. This paper uses the term *addresses*, because it is most general and fits well with an emphasis on routing.

overwhelming extent, a bottom-up engineering activity. Researchers are just beginning to look at the global properties they might try to satisfy with better network designs, which makes any contribution in this area especially timely.

The formal model is written in the Alloy language, which offers a powerful combination of relational and predicate logic [8]. The language is also attractive because of the Alloy Analyzer [7], which was used extensively in this study.

The model shown here is both simplified and abbreviated. The full model is available on the Web [3].

2 Connections

The model is concerned with networks that form persistent connections between *agents*. Agents represent either hardware devices, particularly I/O devices, or software systems. Most of the concepts in this section are illustrated in Figure 1.

The set of agents is partitioned into *clients*, which are the users of networking, and *servers*, which are part of the network infrastructure. The signatures of these object types in Alloy are:

```
abstract sig Agent { attachments: set Domain }
sig Server extends Agent { }
sig Client extends Agent { knownAt: Address -> Domain }
```

Each agent has a set *attachments* of domains to which it is attached so it can use their facilities. If an *address, domain* pair appears in the *knownAt* field of a client, then *address* is published as a way of reaching the client from *domain*. An extra fact (Alloy constraint) says that each such pair can be published by at most one client.

Addresses are primitive objects. Each *domain* has an address *space* (set of usable addresses) and a *map* from its address space to agents:

```
sig Domain { space: set Address, map: space -> Agent }
```

An additional fact says that if an agent is in the range of a domain's map, it is attached to the domain. At this stage of modeling there is no relationship between *knownAt* and *map*, because *map* is in the network infrastructure and *knownAt* is in the user environment.

The persistent connections created by domains are called *hops*. A hop has fields containing the *domain* that created it, its initiating agent *initiator*, its accepting agent *acceptor*, and its *source* and *target* addresses. If a field declaration in an object does not have an explicit set or relation marking, the value of the field is always a single object.

```
sig Hop { domain: Domain,
          initiator, acceptor: Agent, source, target: Address }
```

Additional facts say that both agents are attached to hop's domain, and that both addresses are in the *space* of the hop's domain. In Figure 1, the arrow

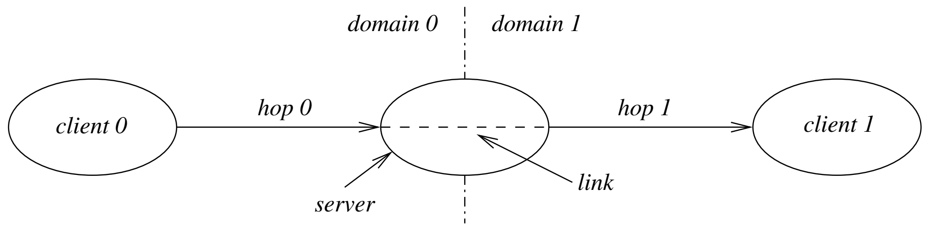

Fig. 1. A connection through two domains. The server is attached to both domains.

representing *hop 0* shows that its initiator is *client 0* and its acceptor is the *server*.

Most important of all, a fact says that the hop domain's *map* relation maps the hop's target address to the hop's acceptor agent. A domain's map models routing in the domain.

Servers can form multi-hop connections by creating internal *links* between hops they are participating in. Like a hop, a link is expected to transmit data more or less transparently.[2] A *link* has fields representing the server creating it and the two distinct hops *oneHop* and *anotherHop* it connects:

```
sig Link { server: Server, oneHop, anotherHop: Hop }
```

Additional facts say that the server of a link is a participant (initiator or acceptor) in both of its hops. Also, a hop belongs to at most one link in each server. This ensures that each connection between clients will be a linear chain of alternating hops and links, without forks or joins.

Because hops and links work together to form connections, it is convenient to have a direct representation of their closure. This is contained in two fields of a *connections* object. A pair of hops is in the binary relation *atomConnected* if and only if they are linked together. The binary relation *connected* on hops is the transitive closure of *atomConnected*.

```
one sig Connections { atomConnected, connected: Hop -> Hop }
```

The modifier *one* indicates that there is exactly one *connections* object.

This basic model is extremely simple for something as complex as networking, but it is sufficient to study many questions, particularly those related to routing and addressing. It allows servers to be gateways between domains, as in Figure 1, or to link hops within domains.

For simplicity, it completely eliminates the temporal dimension of network protocols. It ignores the possibility that an agent might refuse a connection, or be unable to accept one because it is busy. It also ignores multipoint connections, because these are formed using point-to-point connections as building blocks, and are not directly relevant to routing.

[2] Part of the function of a server might be to filter or transform the data in some way, in which case it will be less than fully transparent.

An instance of this model is a snapshot of network state. Thus agent mobility is not represented directly, but is reflected by changes in agents' attachments and domains' maps.

3 Interoperation

In this model, interoperation between domains is viewed as a *feature* that can be added to networks, along with many other types of feature not discussed in this paper. Any feature is installed in a domain and has some set of servers that implement it:

```
abstract sig Feature { domain: Domain, servers: set Server }
```

An additional fact says that every feature has at least one server, and every server implements exactly one feature.

A server of an interoperation feature is a gateway from its *domain* to a second domain called its *toDomain*. When the server accepts a hop in its *domain*, it initiates a corresponding hop in its *toDomain*, and links the two hops together. The source and target addresses of the initiated hop are obtained by applying an interoperation translation relation *interTrans* to the source and target addresses of the accepted hop.

If a client wishes to connect to a client attached to a different domain, it must have a target address it can use in its own domain to request the connection. Conversely, a client must have an address in every domain from which it is reachable. The presence of *interTrans* reflects the fact that a client's addresses in foreign domains may look very different from its native addresses in domains to which it is attached. These differences can arise because of syntactic restrictions, overlapping native address spaces, and historical factors.

The relationships among address spaces are illustrated by interoperation of the PSTN and two Internet overlay networks for telecommunications, the ones defined by SIP [10] and BoxOS [1].

The PSTN was the first domain to be designed. Its address space allows only digit strings of limited length, so a typical native address is `12223334444`.

SIP was the second domain to be designed (history is often important because newer domains usually bear the burden of interoperating with older domains). The SIP address space is based on URI syntax. A typical native address is `sip:alice@host1`. In SIP *all* addresses have the prefix `sip`, so a foreign PSTN address has the form `sip:12223334444?user=phone`.

BoxOS was the third domain to be designed. Its address space is also based on URI syntax, and a typical native address is `boxos:bob@host2`. A foreign PSTN address has the form `pstn:12223334444`. A foreign SIP address has the form `sip:alice@host1`.

Note that native addresses of a domain can be *contained* in the address space of another domain, as SIP addresses are contained in the BoxOS address space, or *encoded* in the address space of another domain, as PSTN addresses are encoded

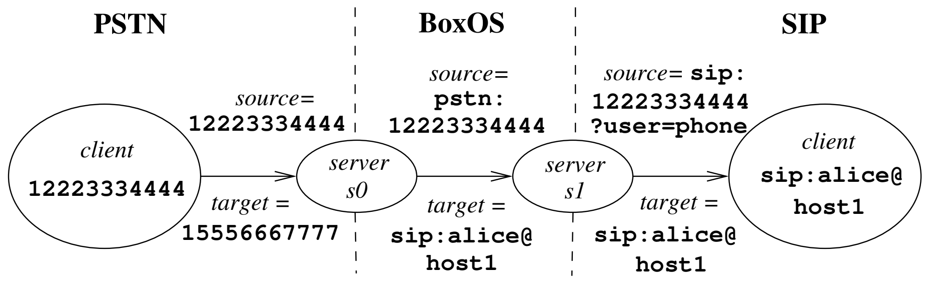

Fig. 2. Interoperation of two domains through a third domain.

in the BoxOS address space. This distinction determines whether translation is an identity or not.

Figure 2 shows how the PSTN and SIP domains would interoperate if they were attached to each other only through a BoxOS domain. The three source addresses are addresses of the initiating client in three different domains. The target addresses are addresses of the accepting client in different domains. Note that server *s1* must know how to translate the address of a PSTN client, even though the PSTN is neither its *domain* nor its *toDomain*.

The PSTN address space has no encoding of SIP addresses. There are two ways that a PSTN client can request a connection to a SIP client. The first possibility is to dedicate a PSTN address to each reachable SIP client, so that interoperation translation of addresses is one-to-one. This is the method illustrated by Figure 2, where PSTN `15556667777` corresponds to `sip:alice@host1` in both BoxOS and SIP.

A more common method is to dedicate a single PSTN address to an interoperation server. The interoperation server prompts the user for a foreign address; because it has full use of the voice channel, it can use speech recognition or digit codes to get the alphabetic characters and punctuation of a URI. The server then translates its target PSTN address to the entered foreign address, so it is performing a one-to-many translation. This situation is discussed further in Section 5.3.

An example of a many-to-one translator is a dynamic, single-address Network Address Translator (NAT). This is an interoperation server that translates many private, unregistered IP addresses to a single public, registered IP address representing the entire subnetwork served by the NAT (see also Section 6).

The signature of an interoperation feature is as follows. The constraints within the signature apply separately to each interoperation feature.

```
sig InteropFeature extends Feature { toDomain: Domain,
   exported, imported, remote, local: set Address,
   interTrans: exported some -> some imported } {
   domain != toDomain
   exported in domain.space   && remote in exported
   imported in toDomain.space && local  in imported
   remote.interTrans = local                          }
```

Four address sets play a role in interoperation. The sets *exported* and *imported* are in the address spaces of the *domain* and *toDomain*, respectively. They are the true domain and range of the feature's *interTrans* relation. This is indicated by the declaration of *interTrans*, whose keywords say that each element of *exported* corresponds to *some* element of *imported*, and each element of *imported* corresponds to *some* element of *exported*,

The subset *remote* of *exported* contains those addresses that trigger the feature because they point to agents in domains other than the *domain* of the feature. The set *local* is the relational image of *remote* under *interTrans*. Note that "local" is a relative term; for example, in Figure 2, server *s0* translates remote `15556667777` to `sip:alice@host1`, which is more local to BoxOS than to the PSTN, but only truly local to SIP.

An address is defined as *foreign* in a domain if it triggers some interoperation feature in that domain. An address is defined as *native* in a domain if it maps to a client in that domain. An unused address is neither foreign nor native.

A fact in the model says that if an address is foreign in a domain, it maps to some agent in that domain, and every agent it maps to is a server of an interoperation feature triggered by it. The constraint allows a domain to have more than one interoperation feature triggered by the same address.

The primary function of an interoperation server is described by this fact:

```
fact { all f: InteropFeature, g: Agent, h1: Hop |
   g in f.servers && h1.acceptor = g &&
   h1.domain = f.domain && h1.target in f.remote
=> (some l: Link, h2: Hop |
      l.agent = g && l.oneHop = h1 && l.anotherHop = h2 &&
      h2.domain = f.toDomain && h2.initiator = g &&
      h2.target in (h1.target).(f.interTrans) &&
      (h1.source in f.exported =>
         h2.source in (h1.source).(f.interTrans)) )        }
```

Note that the relational composition operator (dot) is also used for field selection in Alloy, because a field in a signature is really a function from objects of the signature type to field values. Note also that there is no distinction between an individual and a singleton set, nor between a set and a unary relation, so individuals can participate in relational composition.

The fact determines what an interoperation server must do if it is the acceptor of a triggering hop, meaning a hop in its feature's domain and with a target address in its feature's *remote* set. The server must initiate a corresponding hop in its *toDomain*, and link the two hops together. Note that if the source of the triggering hop is not in *exported*, it cannot be translated by *interTrans*, and the source of the initiated hop is unconstrained.

4 Requirements on interoperation

Now we come to the most interesting question: What requirements should interoperation satisfy? As mentioned in Section 1, this territory is largely unexplored.

One complication in formulating requirements is that a connection network can be modified by a wide variety of features, as mentioned in Section 3. Because the purpose of many features is to alter network behavior in ways that are observable by users (and presumably serve the needs of users), it seems almost impossible to find properties that should be satisfied regardless of which features are present.

For one example, many addresses used to request connections represent, not particular clients, but more abstract concepts [12]. An abstract address might represent a *group* of interchangeable clients, or it might represent a *person* who might be located near, and thus able to use, different devices (clients) at different times. A request for a connection to an abstract address is routed to a feature server that chooses a target client appropriate to the time or other circumstances, and does whatever else is necessary to redirect the request to that target. Thus features that support abstract addresses can make routing nondeterministic.

To understand interoperation, it seems necessary to isolate it from the effects of features that might interact with it, such as those supporting abstract addresses. Its requirements can then be based on the assumption that an address should point to at most one client.

In the absence of a classification of other network features that would tell us which ones can interact with interoperation, we simply eliminate them all with a fact stating that all features are interoperation features.

Other constraints on the model (not shown here) say that an interoperation server cannot do anything but perform its primary function as described in Section 3. A hop with an unused target address can be routed to an interoperation server, but the server cannot link it to any other hop.

As an incidental result of these restrictions, in any connection between two clients, one client is the initiator of its hop and the other client is the acceptor of its hop. This incidental result is employed to facilitate formalization of interoperation requirements.

The most obvious requirement is that an *address, domain* pair published as a way of reaching a client always reaches that client. The formalization of the *reachability requirement* says that if a client is requesting a connection to an address in a domain in another client's *knownAt* set, then the first client is connected to the second client through that request. As explained above, we can assume that the second client is the acceptor of its hop:

```
assert Reachability { all c: Connections,
   g1, g2: Client, h: Hop, a: Address, d: Domain |
      g1 = h.initiator && d = h.domain && a = h.target &&
      (a->d) in g2.knownAt
   => (some h2: Hop | g2 = h2.acceptor && (h->h2) in c.connected) }
```

The second requirement concerns the *returnability* of connections. It is desirable that a client accepting a connection should be able to take the source address it has received, request a second connection to it, and get a connection to the same client that initiated the first connection. Many telecommunication features for automatic callback rely on an assumption of returnability. Naturally, real callback features operate in a temporal context, so the second connection exists at a later time than the first connection.

The formalization of the *returnability requirement* postulates a connection between two clients, and identifies a return-request hop *h3* with the necessary relation to a hop *h2* from which it is derived. It then asserts that a complete return connection exists.

```
assert Returnability { all c: Connections,
   g1, g2: Client, h1, h2, h3: Hop |
   h1.initiator = g1 && h2.acceptor = g2 &&
   (h1->h2) in c.connected &&
   h3.initiator = g2 &&
   h3.domain = h2.domain && h3.target = h2.source
=> (some h4: Hop | h4.acceptor = g1 && (h3->h4) in c.connected) }
```

The third requirement considered in this paper is motivated by the fact that many real address spaces overlap. The *non-uniqueness requirement* means that an address for a client need not be globally unique. Formally, the requirement is satisfied if the following predicate can be instantiated in a model that satisfies the other requirements.

```
pred NonUniqueness (g1, g2: Client, d1, d2: Domain, a: Address) {
   (a->d1) in g1.knownAt && (a->d2) in g2.knownAt              }
```

5 Satisfying the requirements

5.1 Methods of reasoning

Satisfying the requirements entails adding constraints to the model, checking that the model with the additional constraints is still consistent and allows the expected useful instances, and proving that the model with the additional constraints satisfies the requirements.

The Alloy Analyzer finds instances of predicates, for example the non-uniqueness predicate above. Such instances show that a model is consistent, and that it does, indeed, allow the expected configurations and behavior.

The Alloy Analyzer also searches for counterexamples of assertions. Although the search is limited to instances of a bounded size, within those limits it is exhaustive. Every theorem and lemma was checked in this way by the Alloy Analyzer, and no counterexamples to them were found. With respect to the thoroughness of the search, there are two cases.

If an assertion refers to no recursive concepts, then the searchable instance set is satisfactorily large. A typical search space would allow up to 3 domains, 6 features, 10 agents, 6 addresses, 4 hops, and 3 links, which is large enough to include all conceivable counterexamples with three domains. For these assertions, Alloy analysis is more convincing than a manual proof (see Section 7).

If an assertion includes recursive concepts, on the other hand, the search bounds must be smaller. Also, Alloy analysis of this model has a fundamental limitation associated with recursive concepts (see Section 7). For these assertions analysis is less convincing, and is supplemented by manual inductive proofs.

5.2 Satisfying the requirements with generic constraints

This section shows how to satisfy the requirements with a set of general-purpose constraints. While the constraints are plausible, they may be too stringent in some circumstances. Section 5.3 show a special case might be handled with looser constraints.

In Section 2, *connections* was introduced as a signature for a unique object containing only derived fields. A previously unmentioned field of *connections* is a ternary relation *reachedBy*, defined so that a *client, address, domain* triple is present if and only if the address in the domain can reach the client, either directly or through interoperation:

```
one sig Connections {... reachedBy: Client -> Address -> Domain } {
   ...
   all g: Client, a: Address, d: Domain | (a->d) in g.reachedBy iff
   ( g in a.(d.map) ||
     some f: InteropFeature |
        f.domain = d && a in f.remote &&
        some ( (a.(f.interTrans) -> f.toDomain) & g.reachedBy )
   )                                                                 }
```

An address in a domain reaches the client directly if the address maps to the client in the domain. An address in a domain reaches the client indirectly if it triggers an interoperation feature in the domain, and if the feature can map it to an *address, domain* pair that reaches the client. An arrow is a Cartesian product operator, so *(a.(f.interTrans) → f.toDomain)* is the set of all pairs that can be produced by *f* from *a*. The last expression intersects this with *g.reachedBy*, and evaluates to *true* if the intersection is nonempty.[3]

The general-purpose strategy for guaranteeing reachability is straightforward. First, constraints ensure that if an *address, domain* pair can be used to reach a client, then routing to that pair is deterministic, and *always* reaches the client. The constraints for deterministic routing are:

```
fact Constraint1 { all a: Address, d: Domain |
   some ( a.(d.map) & Client ) => one a.(d.map) }
fact Constraint2 {
```

[3] The actual Alloy code for *reachedBy* has an additional constraint to guarantee that the value is a least fixed point.

```
   all f: InteropFeature, a: Address | lone a.(f.interTrans) }
fact Constraint3 { all a: Address, d: Domain |
   lone f: InteropFeature | f.domain = d && a in f.remote }
```

Constraint 1 says that if the agents that a domain maps an address to, intersected with the set of all clients, is nonempty (*some*), then the domain maps that address to exactly one agent. In other words, if an address maps to a client in a domain, it maps only to that client in that domain. Constraint 2 says that the address translation performed by an interoperation feature is a partial function (the quantifier *lone* means one or zero). Constraint 3 says that an address triggers at most one interoperation feature in a domain.

The second part of the strategy is to constrain a client's *reachedBy* set to contain its *knownAt* set:

```
fact Constraint4 {
   all c: Connections, g: Client | g.knownAt in g.(c.reachedBy) }
```

This relates its published addresses to network routing.

Constraints 1 through 3 are easy to apply, because they constrain individual domains. Constraint 4 is not localized, because of the recursive definition of *reachedBy*. This is not surprising, as reachability demands a routing path from any domain in which in which a client has a known address to a domain where the client is directly accessible.

The *reachedBy* set of a client is often larger than its *knownAt* set. For one example, a mobile device might be attached temporarily to a domain where its address is not published. For another example, a domain might provide connections among other domains without having any clients of its own, in which case there is no need to publish any of its addresses. When resource allocation changes, interoperation routes can change without any change observable to clients.

Returnability is much more difficult to satisfy. It depends on every previous constraint except Constraint 4. In addition, to begin with the obvious, the return address of a hop is its source address, so we need constraints to guarantee that the information in the source address is accurate and complete:

```
fact Constraint5 { all h: Hop | h.initiator in Client =>
   h.source in ((h.domain).map).(h.initiator) }
fact Constraint6 {
   all f: InteropFeature | f.domain.space in f.exported }
```

Constraint 5 says that if a hop is initiated by a client, its source must be an address of the client in the domain. Constraint 6 says that every interoperation feature's *exported* set must include the address space of its domain. This prevents the loss of source information during interoperation.

The core constraints for returnability require that each interoperation feature have a partner feature that provides its return path. The constraints are obvious, while the definition of an adequate partner feature is not:

```
pred PartnerTo (f1, f2: InteropFeature) {
   f1.domain = f2.toDomain && f1.toDomain = f2.domain &&
   (f1.imported - f1.local) in f2.remote                  }
fact Constraint7 { all f1: InteropFeature |
   some f2: InteropFeature | PartnerTo(f1,f2)             }
fact Constraint8 { all f1, f2: InteropFeature |
   PartnerTo(f1,f2) => (f1.interTrans).(f2.interTrans) in iden }
```

Constraints 7 and 8 say that each interoperation feature has a partner, and that the *interTrans* relations of partners invert each other.

Figure 3 provides the intuition to understand the definition of partnership, and how it supports returnability. This figure describes a network in which the domains could be pictured in a horizontal line, with each domain interoperating only with the domains on its immediate left and right.

The figure shows the address spaces of two neighboring domains, except for unused addresses. Each address space is divided into native addresses of the domain, addresses that encode native addresses of domains to the left (*Lnative*), and addresses that encode native addresses of domains to the right (*Rnative*). These two domains have interoperation features *f1* and *f2* that are partners of each other.

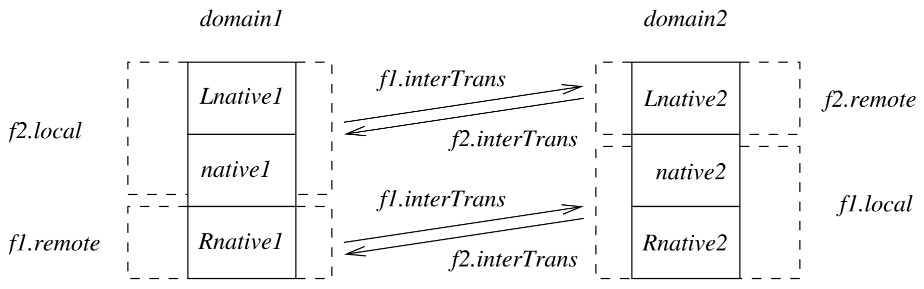

Fig. 3. Partner interoperation features.

If a hop in *domain1* is routed to a server of *f1*, its target will be in *Rnative1*. Its source will be in *Lnative1* or *native1*. Its source cannot be in *Rnative1* because if it were, *the connection path would have passed through the native domain of the target on its way to domain1*. Because routing is deterministic, the path would have ended in the native domain of the target.

Because of Constraint 8, the partition of the address space of *domain1* corresponds to a partition of the address space of *domain2*. The hop target translates to an address in *f1.local*. The hop source translates to an address in *f1.imported - f1.local*. If *f2* satisfies *(f1.imported - f1.local) in f2.remote* then any hop in *domain2* targeting the translated source address will trigger *f2*, and will be linked by *f2* to a continuing hop in *domain1*.

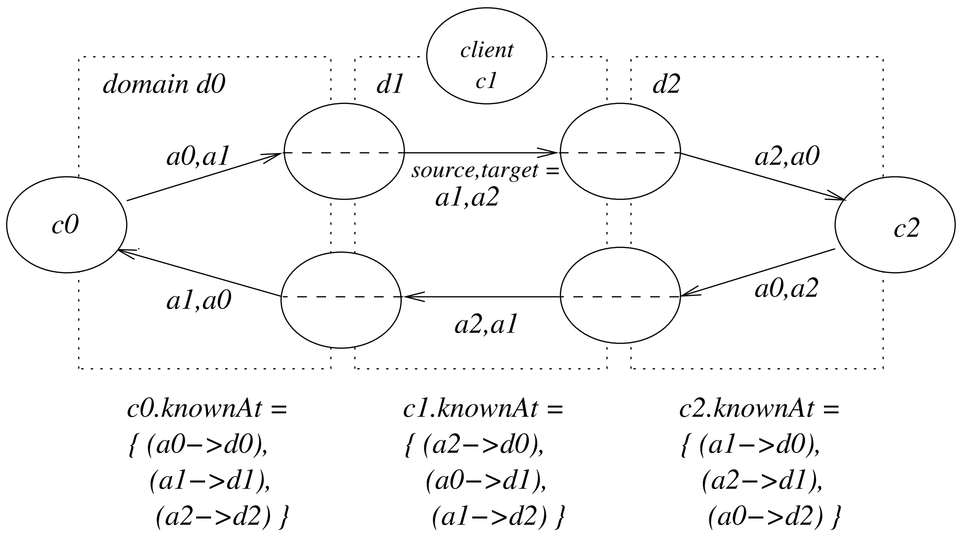

Fig. 4. The requirements do not depend on global uniqueness of addresses.

The actual proof of returnability encompasses all connection topologies, including rings and grids. Its essence is a generalization of the above argument, which can be summarized as follows: From any *address, domain* pair reaching a client, there is a unique path (sequence of domains) to the client. If an address *source* is the source of a hop in a *domain* then the path of the connection from its originating client to the hop is the reverse of the unique path from *source, domain* to the client. If the hop's *target* triggers a feature *f* in *domain*, and if *source* is in *f.remote*, then the unique path to the client of *target, domain* retraces at least one step of the path routing so far. This contradicts the assumption of deterministic routing, so *source* cannot be in *f.remote*.

The model instance in Figure 4, generated by the analyzer with all constraints in force, shows that the non-uniqueness requirement is satisfied. Each of the three clients is attached to one domain and has address *a0* in that domain. Yet each client is known in every domain. In the figure, a hop is labeled with a *source, target* pair of addresses. Either path could be the return path of the other.

Because returnability does not require Constraint 4, it can be satisfied even when reachability is not. A real-world example of this is a dual-mode cellphone[4] at a WiFi hotspot, placing voice-over-IP calls. If the WiFi domain has no cooperative agreement with the device's home cellular domain, then it does not inform the home cellular domain of the device's presence at the WiFi hotspot. In this case there is no forwarding from its known address in the home cellular domain to its temporary WiFi address, and the device is not reachable. At the same time, if the constraints for returnability are satisfied, the device's outgoing calls will be returnable at the temporary address until it leaves the WiFi hotspot.

[4] A dual-mode cellphone is a WiFi device as well as a cellphone.

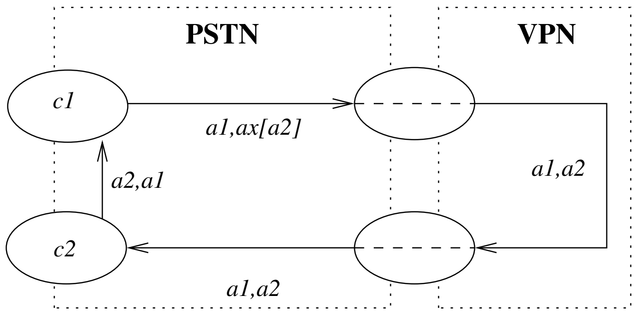

Fig. 5. Interoperation of a VPN with the PSTN.

5.3 A special case

Figure 5 shows a Virtual Private Network (VPN) interoperating with the PSTN. The picture is not geographically accurate, as the two clients are on opposite coasts of the U.S. The VPN belongs to a corporation which provides it so employees can make long-distance business calls at low cost. PSTN hops to and from the interoperation servers are local, while VPN hops are long-distance.

In the figure, client *c1* at PSTN address *a1* is using access address *ax* to reach the VPN gateway, then entering address *a2* using touch tones on the voice channel. The gateway translates *ax* to *a2* and makes the long-distance connection to client *c2* at PSTN address *a2*. The figure also shows the connection when *c2* (not an employee of the VPN owner) returns the call to *c1*.

We wish to know if this network satisfies the requirements, and the results in the previous section are not suitable. As in an example in Section 3, address translation from the PSTN to the VPN is one-to-many, violating Constraint 1.

We can create a formal model that is closer to the truth by extending the PSTN address space to include pairs of numbers, where the first number is dialed, and the optional second number is entered through touch tones. Then interoperation translation from the PSTN to the VPN projects a pair of numbers *ax[an]* onto its second number *an*, and is a function. But now Constraint 8 is violated, because interoperation translation applied to number pairs is not invertible.

Fortunately, there is additional information that can be brought to bear: number pairs are never used as source addresses. Because the basic Alloy model is already available, it is quick work to try a version in which the *targetOnly* addresses of a domain are never used as source addresses in the domain, and Constraint 8 does not apply to them.

This version does not satisfy returnability, and examination of a counterexample shows why. The interoperation feature *f1* from the PSTN to the VPN translates both *a2* and the *targetOnly* address *ax[a2]* to *a2*. The address *a2* must be in the *f2.remote* set of its partner *f2*, but it is not, because *ax[a2]* is in *f1.remote*, and therefore *a2* is in *f1.local*. The solution is to remove the influence of the *targetOnly* address *ax[a2]* from the computation of the partnership constraint on *f2.remote*.

With the definition of partnership modified appropriately, analysis shows quickly and convincingly that returnability is satisfied for this network. The *reachedBy* set of client *c2* consists of *(a2→PSTN)*, *(a2→VPN)*, and *(ax[a2]→PSTN)*.

6 Related work

The most prominent address-related networking problem is understanding reachability at the level of Internet routing, which is "staggeringly complex" [6]. At this level of abstraction, routing information is distributed dynamically by the policy-driven Border Gateway Protocol (BGP) and other local protocols. A path from one point to another includes multiple hops within the same domain. There are packet filters to block packets, and packet transformers to modify them.

While it does not seem feasible to capture all of this in an Alloy model, important aspects of it do seem approachable. For example, Feamster and Balakrishnan study routing only in steady BGP states, taking the position that important requirements can be violated by steady states as well as transient states [6].

The model presented here is valid for packet routing in the sense that only connection requests are really manipulated, and a connection request is equivalent to a packet. In [11] reachability is defined directly for packet routing: for any pair of points, there is a reachable set containing the packets that can travel from the first to the second point. It would be interesting to see how an Alloy model based on this definition compares to the present one. The goal of [11] is polynomial computation of reachable sets in a stable configuration of a specific network, taking into account routing information, packet filters, and packet transformations.

It seems clear that logic-based modeling and analysis has something to contribute to these efforts. The proofs in [6] are informal, yet my experience suggests that network routing has subtleties that only the precision of a completely formal model is likely to expose. The algorithm in [11] might be made even more useful if it were possible to explore invariant relationships among various architectural constraints. Certainly Feamster argues for continuing, broad-spectrum research on correctness and verification of network routing [5].

It is well-known that NATs cause problems in the Internet by allowing addresses that are not globally unique. Currently the worst problems are dynamic and protocol-specific [2]: How does a NAT know that a protocol is finished using an address, so that it can re-use the address while maintaining a one-to-one interoperation translation? How does a NAT find and apply interoperation translation to addresses embedded in the payloads of packets? Nevertheless, general principles of addressing and interoperation should be a sanity check on all specific proposals.

7 Evaluation and future work

As a modeling language, Alloy is very pleasant to use. The combination of relational logic and predicate logic is a powerful one. Although Alloy is first-order, quantification over objects with relation-valued fields provides many of the benefits of a second-order language. Typing is strong but avoids unnecessary distinctions. The syntax is highly streamlined, with a few operators applied uniformly in many contexts to do many jobs. The extended quantifier vocabulary *no, lone, one, some* and *all* provides major shortcuts in writing logical and relational expressions. The complete model is 190 lines of Alloy code.

Analysis by model enumeration is often exhilarating and illuminating, and equally often tedious and frustrating. The Alloy team is working on features and capabilities in the Analyzer that will reduce tedium and frustration. Many of them are already installed, but because they are not yet documented, they are not really available to most users.

Recursion in the definition of *reachedBy* corresponds, in a network, to extending a connection path with additional hops. If the routing data contains a closed loop, the effect on routing will be a path that is extended indefinitely (until terminated by some external mechanism).

Alloy analysis of this model cannot reveal a problem of this kind. Any analysis will impose a maximum length on paths. If a model instance contained a request for a connection to an address that caused such a problem, the connection could not be completed according to the model's constraints within the maximum path length. Thus there cannot be a model instance containing such a request, and its absence tells us nothing about whether there is a routing problem of this kind.

Despite this limitation, the research was undertaken in an analyze-first-prove-last style, which worked fairly well. Alloy's push-button analysis was extremely helpful in building intuition, encouraging experimentation, and finding errors at all levels. It was also a huge reassurance that a continually evolving model was improving as well as changing. The entire experience causes me to trust bounded model enumeration more than manual proof, when enumeration is known to be reasonably comprehensive or at least representative.

All the discoveries were made using analysis alone—proofs served only to confirm and explain. Considering the most difficult result in this paper, which is the satisfaction of returnability, this had good and bad sides. On the good side, experimentation gave me a hypothesis and the confidence to try to prove it. On the bad side, I did not really understand why the hypothesis worked until I proved it, although I believed that I understood it before.

Of course, this is not a controlled experiment. It is likely that a practitioner of a prove-first-analyze-last style would find that proof attempts detect most errors. And a person who was able to find an optimal interleaving of the two techniques, particularly with the help of an automated proof checker, would have the best results of all. Above all, it is important to remember that all of these tools are aids to thought, and none of them is a substitute for it.

Experience suggests that Alloy models with many more object types and facts could be written and read easily. As model enumeration became less convincing, proof could take a larger role. This is fortunate, because there is an unlimited supply of unanswered questions about networking. Section 6 hints at the richness of packet routing at the resource level. At the level of features and services, [12] shows how to manage interactions among features that manipulate abstract addresses. The results are limited, however, by the assumption that every address is globally unique. It would be valuable to have a single model combining interoperation features, which remove the limitation, with abstract-address features.

Other areas of networking in which addresses have semantics and are manipulated include directory lookup (including DNS and a growing number of protocol-specific Internet name spaces), security (including uses of self-authenticating names and trusted domains), and mobility [9]. Experience shows that almost any two functions that translate addresses can interact, so the likelihood of address-related problems in these areas is high.

References

1. G. W. Bond, E. Cheung, K. H. Purdy, P. Zave, and J. C. Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology*, 4(1):83–123, February 2004.
2. R. Bush and K. Moore. NATs are evil—Well, maybe just bad for you. `https:// rip.psg.com/ ~randy/ 040226.apnic-nats.pdf`, 2004.
3. The Distributed Feature Composition (DFC) Web site. `http:// www.research.att.com/projects/dfc`.
4. P. Faltstrom and G. Huston. A survey of Internet identities. Internet Architecture Board, draft-iab-identities-00.txt, 2004.
5. N. Feamster. Practical verification techniques for wide-area routing. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks*, 2003.
6. N. Feamster and H. Balakrishnan. Towards a logic for wide-area internet routing. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, 2003.
7. D. Jackson. Automating first-order relational logic. In *Proceedings of the Eighth ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 130–139. ACM, 2000.
8. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the Ninth ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 62–73. ACM, 2001.
9. G.-C. Roman, G. P. Picco, and A. L. Murphy. Software engineering for mobility: A roadmap. In *Proceedings of the Twenty-second International Conference on Software Engineering*, pages 241–258. IEEE Computer Society, June 2000.
10. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments 3261, 2002.
11. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. Technical report, AT&T Research, 2004.
12. P. Zave. Address translation in telecommunication features. *ACM Transactions on Software Engineering and Methodology*, 13(1):1–36, January 2004.