University of New Mexico
UNM Digital Repository

Electrical & Computer Engineering Technical Reports

Engineering Publications

3-20-2008

Towards Formalizing Network Architectural Descriptions

Joud Khoury

Chaouki Abdallah

Gregory Heileman

Follow this and additional works at: https://digitalrepository.unm.edu/ece_rpts

Recommended Citation

Khoury, Joud; Chaouki Abdallah; and Gregory Heileman. "Towards Formalizing Network Architectural Descriptions." (2008). https://digitalrepository.unm.edu/ece_rpts/26

This Article is brought to you for free and open access by the Engineering Publications at UNM Digital Repository. It has been accepted for inclusion in Electrical & Computer Engineering Technical Reports by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING
UNIVERSITY OF NEW MEXICO

Towards Formalizing Network Architectural Descriptions

Joud Khoury Chaouki Abdallah Gregory Heileman [1,2]

UNM Technical Report: EECE-TR-08-02

Report Date: February, 2008

[1]Joud Khoury, Chaouki Abdallah, and Gregory Heileman are with the Department of Electrical and Computer Engineering, of the University of New Mexico, Albuquerque NM 87131, {*jkhoury, cabdallah, heileman*}@*ece.unm.edu*.

[2]The work presented in this report is partially funded by the National Science Foundation NSF under the Future Internet Design (FIND) Grant CNS-0626380.

Abstract

Despite the rich literature on network architecture and communication system design, the current practice of describing architectures remains informal and idiosyncratic. Such practice has evolved based on idiomatic terminology and hence, it is failing to provide a formal framework for representing and for reasoning about network architectures. This state of affairs has led to the overloading of architectural terms, and to the emergence of a large body of network architecture proposals with no clear indication of the their cross similarities, their compatibility points, their unique properties, and their architectural performance and soundness. Formalizing network architectural descriptions is therefore a timely contribution, and this paper presents a first step in that direction. The paper builds upon architectural style modeling concepts from the software engineering field, and applies them to the network architecture space. Our approach is presented through a case study detailing a formal model for a common class of network architectures. The model uses a simple declarative language based on relations and first-order logic.

1 Introduction

Despite the rich literature on network architecture and communication system design, the current practice of describing architectures remains informal and idiosyncratic. This was caused by the evolution of a semantically rich terminology that has been adopted by network architects over time. The terminology, despite being informal, reveals a lot of architectural information and has so far enabled efficient communication between architects. This scenario is very similar to the evolution of software architecture modeling in the context of software engineering [18]. This state of affairs has however, led to the overloading of architectural terms, and to the emergence of a large body of network architecture proposals with no clear understanding of their cross similarities, compatibility points, their unique properties, and architectural performance and soundness.

Several models for communication systems have been recently proposed, some of which are focused on particular communication aspects such as binding [22, 23] or routing [11]. Others [13, 20] are more general, and concern themselves with multiple communication aspects such as forwarding, naming, addressing. It is important to notice however, that the formal modeling and representation of network architectures is fundamentally different from that of communication systems. In fact, while the communication structure is necessary for defining and representing a network architecture, it is not sufficient. In addition to the communication structure, information and computation structures are building blocks that need to be properly understood within modern network architectures. Communication systems tend to share the same set of elements and are generally concerned with switching properties of networks and their associated communication and control primitives. On the other hand, network architectural descriptions are concerned with high-level architectural abstractions, their interactions, their structural and behavioral properties, and the constraints and invariants that define each architecture.

Towards formalizing network architectural descriptions, we utilize concepts relevant to architectural *style* modeling. An architectural style [1] is a family of network architectures that share a common representation vocabulary. Hence, while architectural *instance*s specializing a particular style may vary in their particulars, their overall structure remains the same and obey the general style constraints. There are significant advantages associated with architectural style design. Those include a better overall system understandability by defining a precise common design vocabulary, the availability of design re-use among all instances of a class, architectural interoperability, and specialized analysis of a class of architectures by constraining the design space [15]. This paper presents a design methodology for formally describing and reasoning about network architectures and architectural styles. The methodology is demonstrated by detailing a formal model for the FARA [8] family of network architectures. Our work provides a framework for network architects to formally group various architectures into a set of styles based on their common structural and behavioral characteristics, enabling researchers to better represent, analyze, reason about, and infer their important properties.

The rest of the paper is organized as follows: Section 2 presents the necessary background related to architectural styles and to the language *Alloy*. Alloy is a simple declarative language based on relations and first-order predicate logic and is the language that we shall use throughout the discussion for formal modeling and verification. Section 3 details our approach through a case study of the FARA [8] class of network architectures. Section 4 then discusses the related work. Finally, we present a discussion of our approach and our current and future work towards formalizing network architectural descriptions in section 5, before concluding in sections 6.

2 Background

2.1 Architectural Styles: What and Why?

Software architectures are usually viewed as a set of interconnected elements that define the structure of a system. The elements are mainly components (computational and storage elements) and connectors (interactions among

[1] Architecture *style* (or *pattern*) is a term commonly used in the software engineering field [18].

the components) [2]. An architectural *style* represents a family of architectures that share a common structural organization. Despite the different representations of a style [3, 5, 6, 18], it is typically composed of component/connector types, and a collection of constraints on how the types are combined. Associated with a style are a design vocabulary, an underlying computational model, and invariants [18].

Styles may be treated as stand-alone structures and may be related through inheritance, or composition. *Inheritance*, an extremely attractive property for describing architectural styles is the ability of a sub-style to extend one or more super-styles inheriting their structural properties, vocabulary, and constraints/invariants. *Composition* is another form relating multiple styles. The composed style is an aggregation of the vocabulary, structure, and constraints of the its constituent styles. Generally, the composed style introduces a new structure to relate the constituent styles together.

The advantages of *modeling* architectural styles are several. First, given the abstraction level of an architectural style, it is generally hard to verify properties pertaining to the style or even to implement the style itself. A compact model then allows the verification of a style's structural and behavioral properties over constrained instance sets without having to actually implement the style. This is an important step when applied prior to the actual instantiation of a complete architecture from the style. In other words, a formal model helps the transition from abstract style design to actual instantiations. Additionally, claims of compatible network architectures, whether those pertaining to general architectures, or to scoped architectures (such as naming, addressing, or routing) may then be logically verified. Finally, a formal model helps to classify the literature into related styles and architectures, and to succinctly illuminate the relations between them, whatever forms those may end up taking.

Modeling the structural properties of software architectural styles has generally been associated with the component/connector abstractions, and has utilized architectural description languages (ADLs) [3, 5, 15, 6] for formal description. We believe that traditional component/connector abstractions associated with style modeling do not provide sufficient abstractions for network architects to work with [3]. Therefore, we simply borrow the notion of "architectural style" without constraining ourselves to the component, connector, port, and role abstractions. Additionally, we choose to use the Alloy modeling language [12] rather than ADLs based on Alloy's simplicity, its expressive power and ability to describe structural and behavioral aspects of an architectural style, and its ability to model desired specification properties that fit our needs (invariants, inheritance, and composition). Despite Alloy's scalability concerns, we have fond it useful to formally describe network architectures/styles because of the presumably small scope of abstractions involved in describing network architectural styles.

2.2 Alloy

Architectural design revolves around exploring the right abstractions, which are simple ideas expressed in some primitive form. Designing those abstractions requires a formal specification language that is intuitive, expressive, and at the same time avoids the intricacies of coding. Alloy [12] is one such language that we use to write our formalization of the FARA style [8] (to be detailed shortly). Alloy is a declarative language based on relations and first-order predicate logic. A brief overview of Alloy's logic, language, and analysis follows. A complete reference is located elsewhere [12].

The Logic - At the core of Alloy is a relational logic that combines relational algebra with first-order predicate logic. Structures are composed of *atoms* and *relations*. Atoms represent typed, immutable structures that are uninterpreted and can be related through relations. A relation is a set of tuples each being an atom and can have arbitrary arity. Relations are combined with *operators* to form *expressions*. Some of the most common operators in Alloy are tabulated in Table I.
Constraints are formed of expressions and logical operators. Quantified constraints take the form $Q\ x : e|F$, where F is a constraint over x, e is an expression bounding x, and Q is a quantifier that can take values **all**

[2]For example, in a client-server architecture description, one might model the client and server elements as components and an RPC communication protocol between them as a connector.

[3]The component/connector abstractions might be sufficient when modeling communication systems, as may be deduced from the axiomatic model in [13].

Set operators	Relational operators	Logical operators
$+$ for union	$\rightarrow$ for product	! for negation
$-$ for difference	. for join	&& for conjunction
& for intersection	˜ for transpose	|| for disjunction
in for subset	ˆ for transitive closure	$\Rightarrow$ for implication
$=$ for equality	* for reflexive-transitive closure	, for alternative
		$\Leftrightarrow$ for bi-implication

Table I: Operators in Alloy.

(universal), **some** (existential), **no** (no values), and **lone** (at most one value). For example, **no** $x : e|F$ is true when no x in e satisfies F. When *Let* is used as in *Let* $a = b|F$, every occurrence of a in F is replaced by b.
Declarations in Alloy take the form *relation-name* **:** *expression*, where expression is the bounding expression for the declared relation. For example, $r : Am \rightarrow nB$, where m and n are multiplicities, is a declaration saying that relation r is constrained to map each element of set A to n elements of set B, and each element of set B to m elements of set A.

The Language - In addition to the logic, Alloy provides some language constructs to help organize a *model*. A model in Alloy may consist of signatures (**sig**), facts (**fact**), functions (**fun**), predicates (**pred**), and assertions (**assert**).
Signature: A signature, declared with **sig**, introduces a basic type along with a collection of fields, their types and restrictions over their values. A signature can **extend** another signature inheriting its fields and constraints. An **abstract** signature has no elements except those belonging to its extensions. For example, if we write:

```
abstract sig A {                        abstract sig B {}
 f: set B                               sig A1 extends A {}
}{--constraints go here}                sig A2 extends A {}

one sig C{} --'one' means sig constrained to one element
```

we have declared three elements A, $A1$, and $A2$. Since $A1$ and $A2$ extend A, it follows that A **in** $A1 + A2$. Additionally, because A is abstract, it follows that $A = A1 + A2$ and $A1$ and $A2$ are disjoint sets that partition A. A declares a field f of type B. This is saying that for each element A, $A.f$ is a set of type B i.e. the relation f is mapping from elements in A to elements in B.
Facts, Predicates, Functions, and Assertions: A **fact** is simply a constraint that is assumed always to hold, and hence needs not be explicitly invoked. Facts usually describe global model constraints. The facts and the signature constraints thus constitute a complete set of structural constraints over the model.
A function, declared with **fun**, is a named reusable expression that can be invoked within the model. A function takes zero or more arguments and returns either a true/false or a relational value.
A predicate, declared with **pred**, is a named reusable constraint that can be invoked. A predicate takes zero or more arguments.
An assertion, declared with **assert**, is a named constraint that is intended to follow from the model's facts. Assertions take no arguments and are usually checked by the Alloy Analyzer as discussed next.

The Analysis - The Alloy Analyzer (AA) [1] is an automated tool for analyzing models written in Alloy. Two kinds of analysis are enabled by AA, based on *commands*. The first is *simulation* (using **run** command) whereby the validity of a predicate or function is verified by showing a snapshot of the system for which the predicate is valid. The second analysis technique is *checking* (using **check** command), whereby an assertion is tested and AA tries to find a counterexample. This requires a finite *scope*, bounding the number of atom instances within the universe, within which AA looks for solutions. Given the undecidability of predicate logic, a finite scope is necessary to bound the space within which AA searches. Finding an instance to a predicate or a counterexample to an assertion guarantees the consistency of the constraint. However, failure to find such instance simply makes it inconsistent *within the scope*. The intuition is that subtle design bugs are likely to be detected even in small scopes.

Listing 1

```
abstract sig AID{}

abstract sig Entity{
 associations: Entity->Time,
 state: associations->one AID,
}{
 no (this & associations.univ)
 all t:Time, aid:AID |
        lone (state.aid).t
 #state = #associations
}
abstract sig RIString {}
```

Listing 2

```
abstract sig FD{}
abstract sig Packet{
  dstFD: FD,
  replyFD: FD
}
abstract sig DPacket extends Packet{
  srcAID: AID,
  dstAID: AID
}
abstract sig SPacket extends Packet{
  ri: RIString
}
```

3 Case Study

To motivate the usefulness of formal architectural modeling, and the expressiveness of the Alloy language, we represent the FARA[8] family of network architectures (or the FARA architectural style) using a formal model. Briefly, FARA [8] is an abstract network model in which the current Internet architecture is generalized and re-modeled to enable clean separation of endpoint names from network addresses. Modeling FARA is an illustrative exercise in *architectural abstraction*, whereby a basis set of structural and behavioral components, assumptions, and constraints (invariants) that pertain to a desired class of architectures are extracted at the first stage of design to describe the general architectural model. Instantiations of the general model may then specialize it, obeying the general design assumptions and invariants. The authors of FARA had to implement a prototype of a FARA instantiation, M-FARA [8], in order to validate FARA's usefulness, and self-consistency. One of the goals of this section is to show how a formal model can be expressive and efficient in validating architectural design decisions, hoping to replace "validation through implementation" by "validation through formal modeling". Aside from providing a conceptual framework for reasoning about a class of architectures, a formal model of an architectural style (such as FARA) transcends into a formal framework over which essential architectural design decisions can be modeled and verified.

3.1 FARA model

We hereby lay out a formal description of FARA's basic structural and behavioral components (static and dynamic properties) along with the constraints attached to components and to the overall architectural style. The description accounts for dynamic behavior by explicitly including logical time steps to model evolution over time [4].

Structural aspects

A formal definition of the *entity* and the *association* is given in Listing 1. An **Entity** is an abstract element that can have multiple concurrent **associations**. An association is a relation between two entities over time. Each entity maintains local immutable **state** per association, the association ID (**AID**). A particular association has exactly one AID, and AIDs are reusable over time. Several constraints are attached to the entity definition: the first constraint eliminates associations that connect an entity to itself for simplicity. The second constraint is one of FARA's key assumptions, and it states that no two associations of an entity can have the same AID at any given

[4]Note however that analyzing the static properties of the architecture, simply requires dealing with a snapshot of the system at some timestep t, i.e., constraining the analysis scope of the *Time* signature to 1 instance.

time. The third structural consistency constraint forces each association to have state. An entity does not define a universal name since FARA does not require a global namespace [5].

Listing 2 defines the Forwarding Directive (FD) and the packet abstractions. The **FD** encapsulates enough topological information to allow the substrate to deliver a packet to its intended destination. A generic packet, **Packet**, says nothing about the identity of the entities, and must indicate a destination forwarding directive (**dstFD**) that will be used by the communication substrate (to be defined shortly) to deliver the packet to a destination entity. A packet might also include a reply FD (**replyFD**) which the destination entity utilizes on the reverse path. FARA distinguishes between a packet that belongs to an association, a **DPacket**, and a setup packet, **SPacket**, that bootstraps an association. DPacket must specify the association state at both ends of an association, **srcAID** and **dstAID**, allowing the destination entity to correctly demultiplex the packet to its association. SPacket includes a rendezvous information string, **ri** of type **RIString**, and does not include association state since the association is being bootstrapped.

Listing 3 defines the communication substrate component, **CommSubstrate**, representing a single global medium (the underlying operating systems and network) that is able to deliver packets on behalf of associations. The substrate assumes a basic connectionless delivery, **delivery**, without making any assumptions about the delivery function itself. A particular FARA instance, as we shall see later, will provide the respective addressing, routing and forwarding mechanisms required for successful packet delivery. Supplied with an FD, the substrate delivers a packet all the way to its destination entity. The point-to-point assumption in FARA is modeled as part of the CommSubstrate constraints specifying that an FD can lead to a single entity at any time. So far, the model defines entities and associations independently of the mechanisms employed by the substrate for packet delivery. This acknowledges FARA's "red line" logical separation, whereby entities and associations operate above the line while the communication substrate operates below the line. Additionally, as a key assumption of FARA, no global address space is defined, with the intent of supporting a multitude of forwarding mechanisms.

Global style constraints, or simply invariants, are specified in Listing 4. The first consistency invariant constrains association to be symmetric. Hence, entity A has an association with entity B if and only if the latter has an association with entity A. The second constraint eliminates dangling association states.

Having formally described the style, we may now proceed to validate some of its properties, specified as predicates and checked through the AA. For example, to check whether an entity might have overlapping state for distinct associations at some time, we define and run the predicate in Listing 5. AA does not find any instance of overlapping state within the simulated scope (7 Entity, Packet, FD, etc.; 15 AID; and 20 Time instances). This guarantees the correctness of the above claim only within the specified finite scope, and not in general. However, if inconsistent models can indeed be found, it is likely to find those within the specified scope.

Functional aspects

This section shows how functional aspects are formally specified at a high level of abstraction, leaving the details for architectural instances to specify.
The first function specified in FARA deals with the creation of associations. To model the system's dynamic be-

[5] Our approach to modeling an association as part of the entity's signature versus modeling it as a separate semantic element renders the dynamic constraints simpler and clearer.

Listing 3

```
abstract one sig CommSubstrate{
 delivery: FD-> Entity -> Time
}{
 all t:Time | delivery.t in
         FD -> one Entity
}
```

Listing 4

```
fact Invariants{
 all t:Time | associations.t
         =  ~(associations.t)
 Time.(Entity.(Entity.state))
         = AID
}
```

```
pred showOverlapState {
all t:Time |
 some disj e1,e2,e3:Entity
 |let w12=getAssociation[e1,e2,t],
  w13=getAssociation[e1,e3,t]
   |e1.w12=e1.w13 and some w12
}
run showOverlapState for 7
      but 15 AID, 20 Time
```

Listing 5

```
--Returns the entity AIDs on both
--sides of the association
fun getAssociation
[fst,snd:Entity,t:Time]:Entity->AID
{
  fst ->  t.(snd.(fst.state)) +
  snd ->  t.(fst.(snd.state))
}
```

```
pred init[t:Time]{
  no associations.t
}
pred establishAssociation
[t1,t2:Time, fst,snd: Entity]{
 --Preconditions
 ---association does not exist
 let aset = {fst->snd+snd->fst}
   | no (aset & associations.t1)
 --Postconditions
 --no association change
 let aset = {fst->snd+snd->fst} |
  {
  noAssociationStateChange[t1,t2]
  associations.t2 =
         associations.t1 + aset
  }
}
```

Listing 6

```
pred teardownAssociation
[t1,t2:Time,  fst,snd: Entity]{
 --association exists
 let aset={fst->snd+snd->fst}|
  some (aset & associations.t1)
 --remove it
 let aset={fst->snd+snd->fst}|
  associations.t2 =
     associations.t1 - aset
}
--associations @t1 valid @t2
pred noAssociationStateChange
[t1,t2: Time] {
 all e1,e2:Entity |
  getAssociation[e1,e2,t1]
   in getAssociation[e1,e2,t2]
}
```

havior as a response to establishing and tearing down associations, we use Alloy traces to capture state transitions over time. Initially, at time t_0, there are no associations. As presented in Listing 6, we consider two events that may change the system's state, the establishment or the tearing down of an association. The time instants **t1** and **t2** describe the state of the system before and after an operation is performed, respectively.

Given the possible state transitions of the system, we can form those into an execution trace by modeling the latter as a fact (Listing 7). Assertions may then be checked against the trace. An invalid assertion will demonstrate a trace showing how the assertion was violated. The Alloy analyzer may be used to show some execution trace of

Listing 7

```
fact Traces {
 init [TO/first[]]
 all t:Time-TO/last[] |
  let t' = TO/next[t] |
   some disj e1,e2:Entity|
    establishAssociation[t,t',e1,e2]
     or teardownAssociation[t,t',e1,e2]
}
```

```
assert showSomeState{
 no e:Entity |
  #e.associations >=1
}
check showSomeState for 4
 but 7 AID, 7 Time,
  0 RIString, 0 Packet
```

the system. For example, running the **showSomeState** assertion using AA, we obtain a counterexample showing a sample trace which, when projected over time, clearly demonstrates the state change resulting from creating or tearing down associations.

M-FARA: an Instantiation

M-FARA [8] is an instantiation of FARA that specifies its own addressing, forwarding, and FD management mechanisms. M-FARA is not a complete architecture, but it is specific enough to explore two points in the FARA design space: 1) location/identity separation, and 2) mobility. This section models M-FARA, particularly its addressing and forwarding mechanisms, using Alloy to demonstrate style specialization.
First, a new module for M-FARA is created importing the FARA module just defined. Several new addressing and topological abstractions are introduced by the M-FARA module, as shown in Listing 8. M-FARA assumes multiple addressing realms, **Domain**s, each having a **space** of unique addresses. A **subFD** represents a set of addresses that determine a local path within a domain. A domain has a static address space, **space**, and a dynamic forwarding mechanism, **forwarding**. The latter delivers a packet that is destined to some subFD to the entity that is bound to the respective subFD. Moreover, the topology assumed in M-FARA consists of a two-level domain

Listing 8

```
sig subFD{}
abstract sig Domain {
 space: set subFD,
 forwarding:space->Entity->Time
}{
 --point2point forwarding
 all t:Time | forwarding.t in
     subFD -> lone Entity
}
--*No global address space*--
one sig MF_CommSubstrate
extends CommSubstrate{
   domains: set Domain,
}
```

```
one sig Core extends Domain {}

sig PrivDomain extends Domain{
  upspace: some subFD,
  downspace: set subFD
}{
  upspace in space
  downspace in space
  no (upspace & downspace)
  -- up forwarding is implicit
  no ((forwarding.Time).Entity)
          & upspace
}
```

hierarchy with a single distinguished central "**Core**" domain to which the private domains, **PrivDomain**s, connect (Listing 8). The extended communication substrate, **MF_CommSubstrate**, may thus be viewed as the set of all domains including the core. Part of a private domain's space, **upspace**, is used to reach the "core" domain. Similarly, part of the "core" domain's space, **downspace**, is used by the core to reach the private domains. In this model, it is implicitly assumed that the forwarding function of every domain delivers subFDs belonging to upspace to the core. On the other hand, forwarding from the core down to the domain is explicitly specified in the domain's forwarding function (hence subFDs belonging to downspace originate at the "core").
Listing 9 defines the complete end-to-end FD in M-FARA, **MF_FD**. It consists of a tuple $(FDup, FDdown)$ which the substrate can use to forward a packet from the source up to the "core" (**up**), and then from the "core" down to the destination entity (**down**). Regarding the entity abstraction, **MF_Entity**, M-FARA extends the entity definition with the local subFD to which the entity is bound, **fddown** and on which it is reachable. M-FARA does not specify whether an entity may be multi-homed (simultaneously bound to multiple domains) or not and our model does not restrict that either.

Some general structural constraints apply to the model and are expressed in Listing 10. No dangling subFDs or domains are allowed. Additionally, a subFD can belong to a single domain's address space. Finally, the forwarding operation is local to the domain i.e. an entry in the domain's forwarding table means that the entity is bound to the domain.

Modeling mobility in M-FARA is another interesting exercise, which we do not address in this paper. This task requires extending the FARA dynamical behavior, which so far includes establishing and tearing down associations, with a new mobility operation.

Listing 9

```
sig MF_FD extends FD {
  up: lone subFD,
  down: one subFD
}
```

```
sig MF_Entity extends Entity{
  --canonical route
  fddown: subFD -> Time,
}
```

Abstract style properties

We have so far modeled an architectural style, FARA, and a particular instantiation of the style, M-FARA. The FARA style advertises a global theme of separating the entity from the communication substrate, and a set of style goals and properties. Despite the fact that the style leaves much of the functional details unspecified (such as addressing and forwarding mechanisms in our example), it is still essential for the style architect to model *super-properties*. *A super-property is a property of the style that is expressed in terms of abstract unspecified functionality*. In other words, the architect needs to confirm that any instantiation of the style that specifies the missing functionality will do that in such a way that the super-properties are respected. In object-oriented programming, such design methodology is known as polymorphism. This section demonstrates a process for modeling style super-properties and checking those against the instantiation, by referring back to the FARA style and the M-FARA instantiation models.

As a first step, the style model includes the super-properties as facts, predicates, or assertions expressed in terms of unspecified functionality. The snippet in Listing 11 augments the previous FARA model with two new invariants (super-properties), expressed in Alloy as facts. The first fact is a “below the line” property. It states that delivery, which we have previously defined as part of the **CommSubstrate** in FARA, must be supported by the substrate’s addressing and forwarding mechanisms. In other words, if the substrate is able to deliver a message to an entity based on some destination FD, then the substrate’s forwarding mechanism must be able to deliver to that entity, hence satisfying **isDeliverable**. Again, note that isDeliverable is left unspecified by the style (in step 1), and is to be implemented by an instantiating architecture based on the forwarding mechanisms employed. The second fact is an end-to-end property (“above the line”) stating that an association exists and is valid only if packets are able to flow over the association from source to destination. In other words, there must exist some FD that satisfies **ise2eDeliverable**.

As a second step, the style instantiation extends the style model implementing the unspecified functionality. Super-properties are then enforced and checked against the instantiation to verify that the desired style goals are satisfied by all instantiations. To illustrate this step, the M-FARA model is augmented with the Alloy snippet in Listing 12, overriding the abstract functionality, isDeliverable and ise2eDeliverable [6].In M-FARA, isDeliverable or deliverability implies that: 1) some packet may be forwarded from the “core” down to destination’s domain i.e. the $FDdown$ part of the destination FD should belong to the **downspace** of the entity’s current domain, and 2) the domain’s forwarding function delivers to the entity given $FDdown$. End-to-end deliverability, in turn, requires two valid paths: one from the source entity’s domain up to the core, and another from the “core” down to the

[6] In Alloy, the super-properties have to be replicated to the M-FARA model since Alloy does not directly support inheritance of a style or “module”.

Listing 10

```
fact Invariants{
  --no dangling subFDs
  Domain.space = subFD
  --no Dangling Domains
  MF_CommSubstrate.domains
       = Domain
  --space is private
  all sf: subFD | lone space.sf
```

```
  --Forwarding is local to a domain
    all t:Time, d:Domain |
     let fwd = d.forwarding.t
      | all sfd:subFD, e:MF_Entity
       | {sfd ->e in fwd
          =>  sfd->t in e.fddown}
}
```

Listing 11

```
--Step1: super-property 1
fact {
 all t:Time |
  let delv=CommSubstrate.delivery.t
   | all fd:FD, e:Entity
    | {fd->e in delv =>
        this/isDeliverable[fd,e,t]}
}
--super-property 2
fact {
 all t:Time, e:Entity |
  let ea = e.associations.t
   |some fd:FD |
    this/ise2eDeliverable[e,ea,fd,t]
}
--*To be specified by Instance
pred isDeliverable
[fd:FD, e:Entity, t:Time]{}

pred ise2eDeliverable
[src,dst:Entity,dstfd:FD,t:Time] {}
```

Listing 12

```
--Step 2
--Replicate facts from FARA
...
--*overriden function
pred isDeliverable
[dst:FD,e:Entity,t:Time]{
 let d_sfd=dst.down,
  d_dom = (getDomain[d_sfd])
   |d_sfd in d_dom.downspace and
    (d_sfd->e in
       d_dom.forwarding.t) }

--*overriden function
pred ise2eDeliverable
[src,dst:Entity,dstfd:FD,t:Time]{
 some dstfd.up and
  dstfd.up in
  (getEntityAttachments[src,t].univ)
                         .upspace
this/isDeliverable [dstfd,dst,t]
}
```

destination entity.
In the same fashion that facts about the style were replicated in the instantiation above, assertions and predicates may also be replicated. It is straightforward to add assertions that verify the facts introduced above. For example, assertions dealing with mobility may easily be implemented.

Composition

Having already demonstrated inheritance and polymorphism in style modeling, we proceed to define and briefly overview (due to lack of space) *composition* as a means for composing separately defined modules or styles and checking for their compatibility. Let $S_i|_{i=1}^{n}$, $n > 1$ be two or more styles, and let P_i, $i = 1..n$, be the global consistency constraints defined by S_i. The new composed style is denoted by $S = C(S_1,..,S_n)$ and contains the merged constraint set $\bigcup_{i=1}^{n} P_i$. S_is are compatible styles iff the new consistency constraint $P = \&\&_{i=1}^{n} P_i$ is satisfied by S. As an example of composition, assume that a global-hierarchical addressing style, GHAR, is defined in which address spaces or domains are composed hierarchically (for example through customer-provider or peering relationships) with a distinguished core. The FARA style may then be composed with GHAR into a new style, say FARA-GH. An entity in FARA-GH extends the FARA entity and defines a global address field that is inherently hierarchical. Interestingly, the new FARA-GH architecture resembles the NIRA [21] routing architecture with the added conceptual clarity and design space partitioning.

4 Related Work

There are two broad areas of related work. The first is concerned with network architecture and communication system modeling, while the second deals with software system modeling.

Regarding network architecture modeling, the Internet architecture has been thoroughly studied over the past decade. The design principles of the DARPA Internet are clearly outlined in Clark's seminal paper [7]. The paper highlights the connection between the intended goals of the DARPA Internet and design decisions that govern its

current operation. The paper was intended to illuminate the Internet's design principles rather than to formally model the Internet architecture. The same applies for other architectural design papers [9, 10].
A methodology for designing and assessing evolvable network architectures based on *invariants* (or *fixed points*) is proposed in [4]. The authors highlight a useful point which calls for considering invariants at an early design phase. However, they do not provide a complete design methodology or formal framework for reasoning about network architectures. Our formalization model inherently accounts for invariants as a part of the complete architectural description, and hence provides the architect with a clearer formal framework to work with invariants.

As to communication system modeling, we identify the several relevant proposals that we believe are complementary to our work. However, our work is again concerned with modeling general architectural descriptions rather than switching properties of networks. Karsten et al. [13] have proposed a general axiomatic basis to consistently model communication primitives such as forwarding, naming, and addressing for better expressing architectural invariants and formally proving properties about node reachability within any communication system.
Another relevant work is that proposed by Zave [22, 23]. In [23], the author utilizes the Alloy modeling language to formally model identifier binding schemes which enables informed architectural design decisions for better supporting networking services. A less general abstraction of the domain and the requirements on binding composition to satisfy inter-operation was modeled by in [22].
The proposal MDCM [20] attempts to describe a wide variety of multi-domain, multi-layer communication systems through a unified model.

Regarding the modeling of software architectures, a lot of work has focused on formally describing those using Architecture Description Languages (ADL) [3, 5, 15, 6]. Some of the common ADLs are the Acme ADL with the underlying first-order logic [15], extended WRIGHT [5], process ADL with the underlying process algebra [6], and π-ADL with the underlying π-calculus [17]. The Acme model in [15] utilizes Alloy and is a very relevant work to ours. Style inheritance and composition as well as verification of structural properties and compatibility checking are concepts demonstrated by the authors; however, their current model falls short of capturing the behavioral aspects of the architectural style. Alternatively, the model in [6] explicitly involves topology specification (i.e. component/connector instances and their interconnections) as part of the architectural style description, which we believe is not an efficient approach considering the level of abstraction at hand.
Finally, Alloy has been utilized within several modeling case studies that as described on the Alloy website [1]. We mention here some of those that pertain to networking and that were useful for this work. Khurshid [14] has used Alloy for modeling and correcting the architecture of the Intentional Naming System (INS). Jackson [1] has used it to model the Chord peer-to-peer lookup protocol. Some recent work by Narain [16] utilizes Alloy's model finding techniques to find network configurations that satisfy a set of input requirements expressed with predicate logic.

5 Discussion, Future Work, and Conclusion

As previously stated, we have refrained from using the component/connector/interface abstractions for modeling network architectural styles. By surveying the network architecture literature, we have noticed that architects have different approaches to modeling abstractions. It is our belief that constraining them to component/connector/interface abstractions limits the expressiveness of the model and hence the innovation. It is additionally hard to anticipate whether and what modeling abstractions for networks will emerge in the future. The language we have utilized, Alloy, is generic and flexible enough to allow the architect to represent whatever abstractions she finds suitable. Despite the scalability concerns associated with constrained instances in Alloy, which does not represent a major limitation to us considering the high level of abstraction being modeled (and hence the presumable small instance sets required), the problem is currently being addressed in the literature (such as in [19]).

While this work has presented a first step towards formalizing network architectures and architectural styles, several research challenges remain to be solved and we address those as part of our current and future research.

First, there needs to be a consensus regarding the most imminent styles that span the network architecture design space. Modern and future network architectures, as has been recently acknowledged [2], are being equipped with more intelligence, generally introducing information and computation structures that are manifested through increased in-network processing and storage. Extracting a complete, and disjoint set of network architectural styles may potentially frame the architectural problem and provide a formal framework for classifying, relating, and reasoning about architectures. Towards this end, we believe that a taxonomy of network architectures is a timely and essential contribution and represents a significant part of our current work.

6 Conclusions

To conclude, this paper has presented a methodology towards formally describing and modeling network architectures and architectural styles. Style inheritance, polymorphism, and composition were demonstrated on the FARA class of network architectures using the Alloy modeling language. Our work helps network architects and researchers, whereby architects are able to formally represent and group various architectural patterns into styles, while researchers are provided with a means to better understand, analyze, and reason about network architectures.

References

[1] The alloy analyzer. http://alloy.mit.edu/.

[2] NSF nets find initiative. http://www.nets-find.net/.

[3] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.

[4] Bengt Ahlgren, Marcus Brunner, Lars Eggert, Robert Hancock, and Stefan Schmid. Invariants: a new design methodology for network architectures. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 65–70, New York, NY, USA, 2004. ACM Press.

[5] Robert Allen and David Garlan. A case study in architectural modelling: The aegis system. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 6, Washington, DC, USA, 1996. IEEE Computer Society.

[6] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.

[7] D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 106–114, New York, NY, USA, 1988. ACM Press.

[8] David Clark, Robert Braden, Aaron Falk, and Venkata Pingali. Fara: reorganizing the addressing architecture. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 313–321, New York, NY, USA, 2003. ACM Press.

[9] David D. Clark, Karen Sollins, John Wroclawski, and Ted Faber. Addressing reality: an architectural response to real-world demands on the evolving internet. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 247–257, New York, NY, USA, 2003. ACM Press.

[10] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: defining tomorrow's internet. *IEEE/ACM Trans. Netw.*, 13(3):462–475, 2005.

[11] Timothy G. Griffin and Joäo Luís Sobrinho. Metarouting. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 1–12, New York, NY, USA, 2005. ACM Press.

[12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[13] Martin Karsten, S. Keshav, Sanjiva Prasad, and Mirza Beg. An axiomatic basis for communication. In *Proceedings of SIGCOMM 2007*, pages 217–228, New York, NY, USA, 2007. ACM Press.

[14] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *ASE*, pages 13–22, 2000.

[15] Jung Soo Kim and David Garlan. Analyzing architectural styles with alloy. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM Press.

[16] Sanjai Narain. Network configuration management via model finding. In *LISA'05: Proceedings of the 19th conference on Large Installation System Administration Conference*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[17] Flavio Oquendo. π-method: a model-driven formal method for architecture-centric software engineering. *SIGSOFT Softw. Eng. Notes*, 31(3):1–13, 2006.

[18] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[19] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. pages 632–647. 2007.

[20] Y. Wang, J. Touch, and J. Silvester. A unified model for end point resolution and domain conversion for multi-hop, multi-layer communication. Technical Report ISI-TR-590, USC/ISI, June 2004.

[21] Xiaowei Yang. Nira: a new internet routing architecture. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 301–312, New York, NY, USA, 2003. ACM Press.

[22] Pamela Zave. A formal model of addressing for interoperating networks. In *FM*, pages 318–333, 2005.

[23] Pamela Zave. Compositional binding in network domains. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 2006.