A Theory of Networking and its Contributions to Software Engineering

Pamela Zave

Abstract This paper presents a compositional theory of networking as an example of a useful and realistic domain theory. First, the paper uses networking to illustrate all the parts of a domain theory, including a re-usable domain description with intrinsic state and behavior, software interfaces and specifications, requirements, proof obligations, and theorems. Next, the theory is extended with composition of network domains, which is directly relevant to solving today's most critical networking problems. Finally, the paper proposes ways in which the theory can contribute directly to the design and development of network software.

1 Introduction

Currently the biggest obstacle to doing research in software engineering that translates to industrial practice is the size and complexity of real-world software systems. They are built and maintained by large, international groups of people. Research groups are small, and cannot keep up.

In [18], it is argued that one of the most promising avenues of software-engineering research is the development of re-usable domain models. In this context a *domain* is the subject matter of a software system. A *domain model* is a formal description of everything relevant that is known or assumed about the domain. To make it re-usable, it must be general enough to describe all real-world instances of a family of domains, for development of a family of software systems. The model includes the *interface* to the software system, consisting of shared phenomena controlled by the domain and input to the system, as well as those controlled by the system and output to the domain.

Two additional types of description accompany a domain model: *requirements* state the properties that the domain should have when the software system is built

Pamela Zave
AT&T Labs—Research, Bedminster, New Jersey, USA, e-mail: pamela@research.att.com

and installed, while *specifications* state how the software system should behave at the system/domain interface. The desired relationship, among the descriptions for a specific development project, is that the domain model and the specification together should guarantee that the requirements are satisfied [11, 21].

A domain model is elevated to a *domain theory* when it is enhanced with theorems [20]. Theorems capture re-usable knowledge about how the system can solve the problems posed by the requirements, and assist formal reasoning to show how the effects of the system outputs propagate as intended throughout the domain.

Domain theories meeting this standard of completeness are few and far between. However, based on the many successes of model-based software engineering, it seems reasonable to expect that the benefits of a domain theory that really captures the essence of a complex domain would include the following.

- A domain theory records essential concepts and facts in an abstract, comprehensible, and re-usable way.
- Having a formalization of the domain is a prerequisite for the use of any logic-based tool. Logic-based tools can perform syntactic analysis, verification, constraint satisfaction, optimization, code generation, code synthesis, and automated testing.
- A domain theory can facilitate recognition of recurring patterns, design principles, and structured trade-off spaces.
- A domain theory can provide modularity and compositional reasoning.
- A re-usable theory invites the development of domain-specific languages for customizing generic descriptions. These generic descriptions might cover domains, requirements, or specifications.
- Once there is a useful domain theory, it can be improved continuously with respect to coverage, theorems, and the power and efficiency of the tools that operate on it.

The scarcity of domain theories is due (at least partially) to the difficulty of working theory-building into the schedule, budget, and process of software-development projects. The purpose of this paper is to encourage researchers to build domain theories, as the technical challenges and potential benefits equal or exceed the challenges and benefits of other approaches to research. The paper illustrates concretely major parts of a domain theory for networking, hopefully enough to show the nature of such work. To provide some context, Section 2 gives a brief overview of the current networking scene.

Section 3 sketches out the major parts of a theory of a network domain. The domain description includes generic network state and generic algorithms for how messages are processed by network elements in accordance with domain state. In networking terminology, this is the *data plane*. Most network software is part of the *control plane*, which initializes and maintains the network state so that it controls the data plane properly. Section 3 also discusses some of the requirements, specifications, and theorems used to develop software for network control planes.

The domain theory is designed to be compositional, as networks today can only be described usefully as layered compositions of many networks, each potentially

serving a different purpose at a different level of abstraction. Section 4 introduces composition of networks, including uses of composition, examples from cloud computing, and how the theory in Section 3 must be extended to cover composition of networks.

Finally, in Section 5, we propose a number of ways that the theory of networks can contribute to the design and development of network software. This section focuses on composition, because composition is the aspect of networking that has been the most neglected—in fact, altogether unacknowledged—in practice. The paper concludes (Section 6) with lessons about research in building theories of software domains.

The domain theory in this paper has been called the *geomorphic view of networking* [22]. It was inspired by John Day, who showed the existence of patterns that appear in network architectures at many levels for many different purposes [5]. Although the exposition in this paper is informal, many aspects of the theory have been formalized in Alloy [10]. Its presentation is organized according to typical parts of a domain theory, as enumerated in [20]. Although this seems to be the best way to convey what a domain theory is like, it tends to obscure the reasons why this is a *good* domain theory. A much more purpose-driven explanation of the basic ideas can be found in [19].

2 Networking today

The original Internet architecture was intended to empower users and encourage innovation [3], and it has succeeded beyond most people's wildest dreams. As a result of this success, the Internet has outgrown its original architecture, and does not meet current needs in many areas. The networking community has recognized serious deficiencies concerning security, reliability, mobility, and quality of service. It is proving difficult to achieve the desired convergence of data, telephone, and broadcast networks, and difficult to balance the needs of all of the Internet's stakeholders [4, 6, 8, 14].

At the same time that external requirements have been expanding, growth and competition have intensified the need for better resource management. Network providers must use elastic resource allocation, rather than simply over-building their networks, as was the previous practice. These pressures have led to widespread use of cloud computing.

The original or classic Internet architecture [3] has five layers, as shown in Figure 1. It is similar in spirit to the OSI reference model [9], which has seven fixed layers. Figure 1 would lead us to expect that a typical Internet packet would have four headers, for example Ethernet, IP, TCP, and HTTP. Each layer requires a message header to carry out its function. Each layer is indispensable because it has a distinct and necessary function.

Because each of the changes mentioned above has made demands that the original Internet architecture cannot satisfy, today new requirements are satisfied by

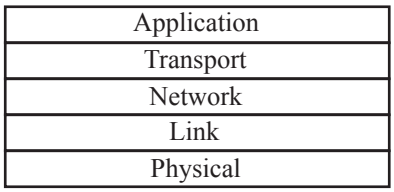

Fig. 1 The classic Internet architecture, with exactly five layers.

adding many *ad hoc* intermediate layers of virtual networking, as described by Spatscheck [15]. To illustrate these *ad hoc* layers, Figure 2 shows the twelve headers in a typical packet transmitted across the AT&T backbone—obviously a lot of new things are going on! The problem with these *ad hoc* layers is that each is typically designed and understood in isolation. The overall network behavior when new layers are run amongst old layers is neither understood nor predictable.

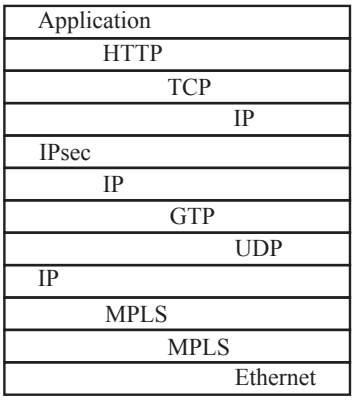

Fig. 2 Headers in a packet in the AT&T backbone, suggesting the presence of approximately seven networks composed hierarchically.

The first principle of the geomorphic view of networking is that networks as we know them are layered compositions of modular networks. Each network-as-a-module is a microcosm of networking, with all the basic parts and functions native to networking. A network architecture can have as many or as few of them as needed.[1] So the same basic network function can occur several times, at several different levels, for several different purposes. The ultimate goal of our theory of networking is to reason rigorously and compositionally about the properties of networks.

[1] The name "geomorphic view" comes from the varied arrangements of layers in the earth's crust. Layers vary in number and composition from place to place, and they can abut and overlap in interesting ways.

3 A theory of a network as a software domain

3.1 Domain objects

A domain has objects. Figure 3 shows the most important objects in a particular network, *i.e.,* an instance of the domain theory.

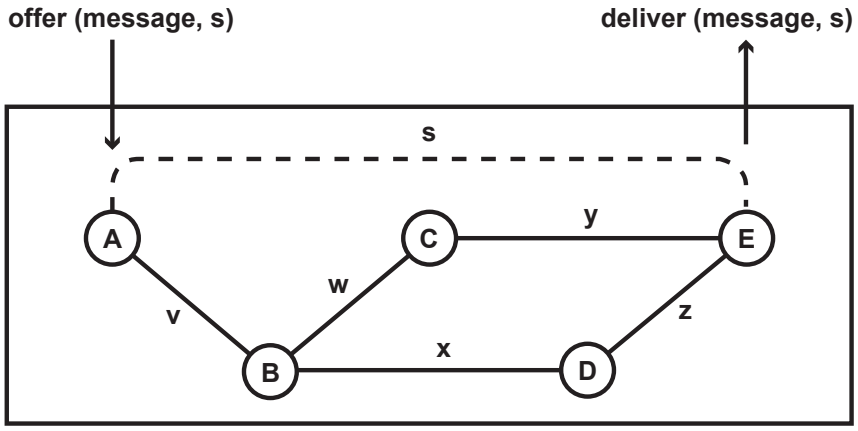

Fig. 3 In the boxed network, members are named with capital letters. Links are solid lines, while sessions are dashed lines; each link and session is named with a small letter. *Offer* is an interface operation to be invoked by a client, to give a message to the network, while *deliver* is an interface operation invoked by the network, to give a message to a client.

The *members* of the network are software or hardware modules executing in computers. Each member of the network has a unique *name* drawn from the network's namespace.

Members of the network are connected by unidirectional point-to-point *links*.[2] A link is a communication channel through which its sending member can send messages to its receiving member. Physical implementations of links include wires, optical fibers, and radio channels.

In some networks, all the members are pairwise connected to each other by links. More commonly there are fewer links, although enough to ensure that there are paths (concatenations of one or more links) connecting each pair of members in each direction. In Figure 3, if each solid line actually represents a link in each direction, then the network members are fully connected in this way.

Now the problem is to deliver messages to their destinations over paths of links. The universal solution to this problem is:

[2] The theory also encompasses other kinds of link such as broadcast links. Non-point-to-point links and some other structures are omitted here for simplicity.

- The name of the destination of the message is added to the message, in a data structure called the *header*.
- When a message arrives at a member, and the destination of the message is not the name of that member, the member *forwards* the message toward its destination on another link.

A *session* is like a link in being a unidirectional point-to-point communication channel. Unlike a link, it does not have an atomic implementation in the domain. Rather, a session is implemented as an identifier for a group of messages that have the same sender and receiver, are considered to be related to each other, and are treated as such by their sender and receiver.

Unlike domains that existed before computers, networks are built for the purpose of allowing computers to communicate. To achieve this purpose, a network must have a client interface. The client interface is built into the operating system of each computer having a member of the network. When some entity on a computer wishes to transport a message to an entity on another computer, it invokes *offer* at the client interface to its network member, and the network member *sends* the message through its network. When a member *receives* a message destined for it through its network, it invokes *deliver* at its client interface, so the operating system can pass the message to the proper client entity.

More generally, a network provides one or more *communication services* to its users. The properties of a communication service are associated with sessions. For example, a network might provide the service of FIFO message delivery. This does not mean that *all* messages are delivered in the order in which they are sent, an unobservable and unimplementable global property. Rather, it means that all the messages of a particular session are delivered in the order in which they are sent. This is why the client interface in Figure 3 shows that each message is offered and delivered within the context of a particular session.

3.2 Domain state and behavior

A network has state, most of which is distributed over its members so that the members have quick access to what they need. Each member has known behavior, standardized across the network, that performs network functions as controlled by its state. As mentioned in Section 1, the known behavior is called the *data plane* because it performs all the operations on the data (messages) transmitted through the network. The software that maintains the network state is called the *control plane* because the network state controls the data plane. Most software development for network domains is development of the control plane (see Section 3.3).

Not surprisingly, each member has state recording all the sessions and links of which it is an endpoint. This state includes the names of the members at the far ends of these communication channels.

Network functions require that each message sent through the network has a *header*, minimally containing the source name, destination name, and identifier of the session in which it is sent.

To implement forwarding as described in Section 3.1, the state of a network must include *routes*, which tell the members where to forward messages so that they reach their destinations. The routes used by a member are typically formalized as a relation or table with three columns of types *header, link, link*. If a tuple *(msgHead, inLink, outLink)* is in the *routes* relation at a member, then a message with header *msgHead* received by the member on its link *inLink* must be forwarded by sending it on *outLink*. For example, consider how to get messages from A to E in Figure 3. Simplifying headers to destination names only, *routes* of B may contain the tuple *(E, v, x)* and *routes* of D may contain the tuple *(E, x, z)*, or B may have *(E, v, w)* while C has *(E, w, y)*.

The known behavior of a network is conventionally described in two parts, a *forwarding protocol* and one or more *session protocols*. The forwarding protocol is executed by each member. Its behavior always includes at least the following aspects:

- There is a set of conventions about how digital messages are represented on the links.
- Members send messages produced by the session protocol. When members receive messages, they pass them to the session protocol.
- When a member M sends a message with header h, it finds a tuple *(P_h, self, k)* in its *routes* relation, where P_h is a pattern that matches H, and *self* is a distinguished pseudo-link. M sends the message on outgoing link k.
- When a member M receives a message with header h on link $k1$, provided that the destination name in the header is not M, it finds a tuple *(P_h, k1, k2)* and forwards the message on link $k2$.
- When a member M receives a message with header destination M, the message is passed to the session protocol.

A forwarding protocol can be extended to enhance security and monitor traffic, among other network functions.

The exact behavior of a session protocol depends on the communication services that it is supposed to provide. At a minimum, it does the following:

- It accepts an offered message, encapsulates it in a larger message containing the header, and passes it to the forwarding protocol.
- It gets a message received by the forwarding protocol, decapsulates it by stripping off the header, and delivers it to the operating system of its computer.

Session protocols can also offer much more. For example, consider the best-known session protocol, which is TCP.

As we shall see in Section 3.3, every single part of a network is dynamic. When links are changing or failing and whatever algorithm maintains the routing state is not keeping up, some messages will inevitably be lost. Despite the unreliability of forwarding, TCP provides the communication service of a reliable, FIFO,

duplicate-free byte stream. TCP actions are performed by the implementations of the session protocol in each of the two members at the session endpoints. These members achieve their goals through acknowledgments, detection of lost messages, retransmission, and reconstruction of a properly ordered byte stream.

To return to the comparison between Figures 1 and 2, if layers are viewed as having distinct and indispensable functions as in Figures 1, then an IP header goes with the Network layer and a TCP header goes with the Transport layer. If a layer is a microcosm of networking, on the other hand, each layered network contains both a forwarding protocol (such as IP) and a session protocol (such as TCP). Allowing for the facts that in some networks some parts are vestigial, and that a packet alone does not tell us everything about how protocols are being used, Figure 2 shows evidence of approximately seven networks layered on top of one another.

3.3 Software development for networks

Sections 3.1 and 3.2 have presented the objects, state, and known behavior of a network. Most software development for networks is development of the control plane. The control plane receives customer and service-provider requirements through APIs, and real-time monitoring information from the network members. Its output is the initial configuration of the network state, followed by real-time updates throughout the life of the network. This section uses software for ordinary routing as an example of the control plane.

3.3.1 Routing requirements

The primary purpose of ordinary routing is to make it possible for messages from any network member to reach any other network member. It must satisfy this requirement with whatever links are currently available, or at least satisfy it as well as possible. In addition, routing may be expected to satisfy other requirements such as these examples:

- For security, as an exception to general reachability, member A is not reachable from member B.
- There is a minimum bandwidth for certain traffic.
- Messages are delivered within a certain latency bound, with probability P.
- Averaged over 10-minute periods, no link is more than 80% utilized.
- There are no loops in message forwarding.
- All messages in a session travel on the same path.

Routing must be dynamic because the state of a network is always changing, even when the requirements are not changing. Members and links can disappear as a result of failure or retired resources, and can appear as new or re-instated resources.

The load on the network, *i.e.,* the number and characteristics of the messages being sent, can also change radically over time.

These observations show how the domain model presented so far is incomplete. A complete domain model should include resource failures and restarts, and how they affect the domain state. It should also include the performance attributes of links, such as bandwidth and latency. Characterizations of the network load, for example traffic or utilization measurements at specific points in the network, are also indispensable.

3.3.2 Specifications of routing

Specifications state how the software system should behave at the system/domain interface. At this interface, routing software must receive as inputs information about the status of links and members. The software must also receive information about the current load and/or resource utilization. Once the software has computed changes to the current *routes*, it must output these changes to the domain. More specifically, every new tuple *(headerPattern, inLink, outLink)* must be installed in the member *forwarder* to which it belongs. Note that in many networks, there are specialized members called *routers* that do all the forwarding. The other members are the sources and destinations of ordinary messages, and have only *self* tuples (see Section 3.2) for forwarding.

The exact content of a routing specification depends very much on the exact nature of the system/domain interface. In networking today, there are two major variations on this interface.

In older networks, the routing algorithm is distributed. There is a piece of the routing algorithm running in the computer of every router (see Figure 4), and there is a local system/domain interface between the router and the local routing agent. At this interface, the router passes all its information to the routing agent. The local routing agents communicate through their own private messages, so that each has a sufficient view of its region of the network. Each agent of the routing algorithm is responsible for the forwarding state in its own router.

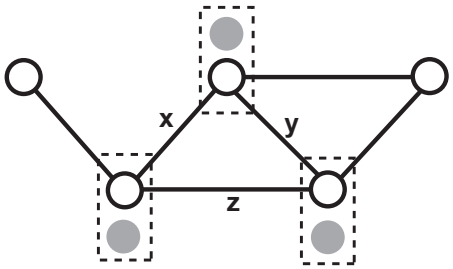

Fig. 4 A distributed routing algorithm. The dashed boxes are computers on which the network members are routers. The gray dots represent the software modules of the routing algorithm, called local routing agents. Through the routers, they communicate with each other by sending messages on links *x, y,* and *z*.

More recently, the concept of Software-Defined Networking (SDN) has become popular [13]. With SDN, the system/domain interface at the network routers is defined by the OpenFlow standard. OpenFlow allows a router to send informational messages and queries to a separate, centralized controller (see Figure 5), and to receive new *routes* tuples from the centralized controller.

3.3.3 Theorems about routing

Not surprisingly, many theorems of graph theory are relevant to network routing. One use of graph theory is "network verification," in which snapshots of the *routes* tables in each router are analyzed for desirable general properties such as reachability, security blocking, and routing through middleboxes [1, 12, 17].

The most advanced theory for network routing can be found in [7] and subsequent papers. The contribution of the "metarouting" work is easiest to explain in the context of distributed routing algorithms, as in Figure 4.

All distributed routing software uses some basic distributed algorithm based on *advertisements*. To give a trivial example based on Figure 3, let us assume that the link labels are not letters but rather numbers representing link lengths. C advertises to all its neighbor routers that it has a path to E of length y, while D advertises to all its neighbor routers that it has a path to E of length z. Their neighbor B now knows that it has two paths to E, of lengths $w+y$ and $x+z$ respectively. B chooses whichever of these paths is shorter, and advertises to its other neighbors (not shown) that it has a path to E of the shorter length.

To generalize these basic ideas, metarouting separates the distributed advertisement algorithm (there are two important ones) from the *algebra* being used to evaluate paths. A path algebra defines a path metric, an operator to combine path metrics, and a preference ordering. In the preceding example, path lengths are combined by addition, and the shortest length is preferred. If the requirements concern minimum bandwidth of a path, on the other hand, path metrics are combined by taking their minimum, and the largest minimum bandwidth is preferred.

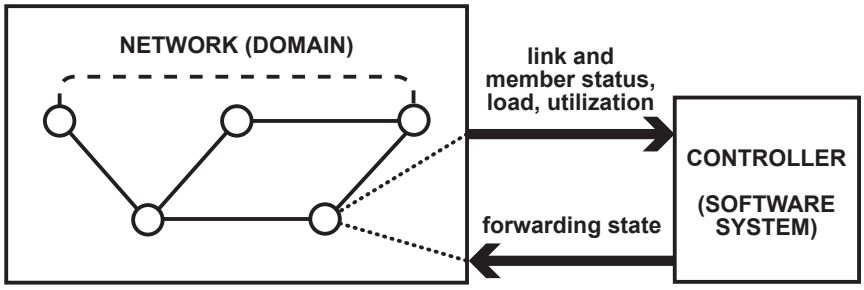

Fig. 5 A centralized routing algorithm. Each OpenFlow-enabled router communicates with a centralized software controller that implements the routing algorithm.

Supported by a theory built on these generalizations, network operators can use a domain-specific language to specify their routing requirements in terms of one or more composable algebras [16]. Tools can then check the requirements for consistency and generate a complete routing algorithm automatically. There are metarouting implementations for both distributed and centralized network control.

4 Composition of networks

4.1 Definition of composition

The fundamental mechanism through which all networks compose is shown in Figure 6. When a member of an "overlay" network uses the services of an "underlay" network, its computer must also host a member of the underlay network. The overlay and underlay members communicate through the operating system of the computer. In the terminology of our network theory, the member of the overlay network is *attached* to a member of the underlay network, and the underlay member is the *location* of the overlay member.

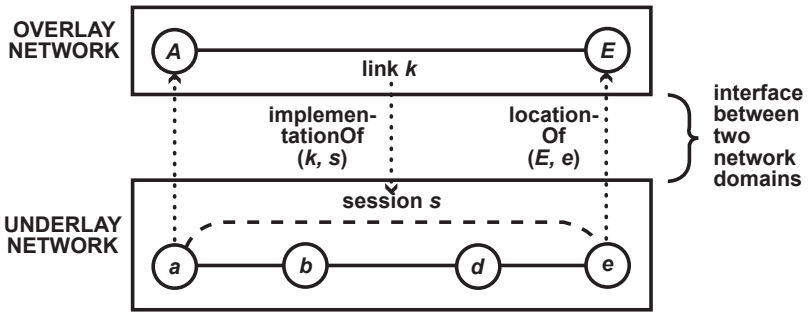

Fig. 6 Fundamental structures of network composition.

As Figure 6 shows, composition means that a session in the underlay *implements* a link in the overlay. Conversely, the overlay link *uses* the underlay session. The underlay is providing for the overlay a communication service with certain properties. These properties are guaranteed by the underlay session, and can be assumed to hold for the overlay link.

In the figure, when A sends a message on virtual link k, it actually offers the message to a with session identifier s (see Figure 3). Underlay member a encapsulates the message in the header for s in the underlay, and sends it through the underlay network. When e receives the message, it decapsulates (strips off the underlay header) and delivers it to E as part of session s.

Section 3.2 stated that the minimal contents of a header are source name, destination name, and session identifier. By convention, the names are usually associated with the forwarding protocol and the identifier with the session protocol. For instance, an Internet header might be described as "TCP over IP," where "over" (referring to layers) might also be "inside" (referring to the bit string). Encapsulation due to composition is similar, so that the headers on messages in Figure 6 could be described as "overlay header over/inside underlay header."

Figure 7 shows the geomorphic view of the classic Internet architecture. Each endpoint device (a cellphone and a Web server) hosts members of two networks, the Internet and a lower-level network with physical links. (The cellular network uses radio channels, the wide-area network uses optical fibers, and local-area links are simply wires.) At all higher levels of a graph of network composition, the links are virtual—they are implemented in software. The great benefit of the Internet is that its span is global, while each physical network has a limited span.

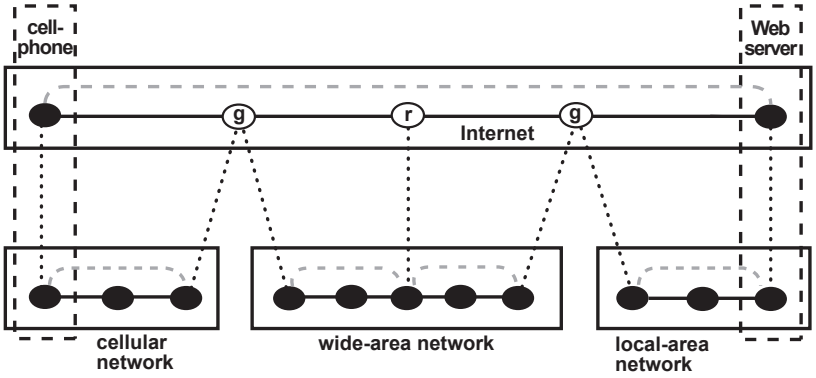

Fig. 7 Another view of the classic Internet architecture. The global Internet is composed with many local networks, and acts as a bridge between adjacent networks.

Note in Figure 7 the two Internet members marked *g* for *gateway*. From the viewpoint of the Internet they are merely forwarding messages from one link to another. Their significance is that each is attached to two different physical networks, so that each can forward messages from one physical network to the other. On each side of the figure, there is an Internet link between an endpoint and a gateway. Each link is implemented by a session in an "edge" or "access" network. The links between gateways, on the other hand, are implemented by sessions in a wide-area network.

In Figure 7 the Internet also has a member marked *r* for *router*. Routers are network members whose primary function is forwarding messages along desired paths; routers do not send or receive data-plane messages, but rather send and receive only through session protocols that help to implement the control plane.

The figure suggests that every cellphone served by the cellular network, when it is using what is known as "data service," has a direct virtual link to an Internet gate-

way. The reason for this architectural decision is to push the routing functions and forwarding needed to implement this link down to the cellular network, where the state, protocols, and control plane are specialized for cellular networks and different from what is normal for the Internet.

4.2 Domain state and behavior

Section 3.2 explained that *sessions, links,* and *routes* are part of the state of a network domain. As shown in Figure 6, composition of networks requires additional state that maps between members and communication channels across the network boundary. In the overlay, the required mapping is *implementationOf (k, s),* meaning that session *s* is the implementation of link *k*. In the underlay, the required mapping is *locationOf (om, um),* meaning that underlay member *um* is the location of overlay member *om*.

It is easiest to see the need for this extra state when links and sessions are dynamic, as they often are. In Figure 6, imagine that there is to be a new overlay session *so* between *A* and *E*. For architectural reasons, all the real implementation work for this session is to be performed in the underlay. So there will be a one-to-one correspondence between *so* and virtual link *k* in the overlay, and a one-to-one correspondence between *k* and session *s* in the underlay. Using a generic algorithm for compositional behavior in domains, here are the steps required in *A* and *a*:

1. *A* initiates session to *E*, creating session *so* state.
2. *A* initiates link to *E*, creating link *k* state and routing state. All messages in *so* will be routed to *k*.
3. *A* requests implementation of *k* from *a*. *a* uses *locationOf* to discover that the underlay location of *E* is *e*.
4. *a* initiates session to *e*, creating session *s* state. *a* replies to *A*'s request with result *s*, and *A* records that *implementationOf (k, s)*.
5. *A* sends a session-initiation message for *so*, which is routed to *k*. Because *k* is implemented by *s*, this means offering the message to *a* in session *s*.
6. *a* encapsulates the session-initiation message in the *s* header and sends it to *e*. It travels by pre-existing routing over static links.

When the session-initiation message arrives at *e* a similar but reverse process begins, which will establish similar state at the other endpoint, and enable correct delivery of messages in *so*.

4.3 Uses of composition

In Section 4.1, we saw composition used as a way to bridge smaller networks so that they can function together as a larger network. This was the original purpose of

composition in networks, but today there are many others. In this section there are examples of three of them.

4.3.1 Encryption

Figure 8 shows a *virtual private network (VPN)* layered on top of the Internet. To its users, the VPN looks exactly like the Internet, except that all the member names are in the part of the IP address space reserved for private use. The headers on messages through this network are TCP over IP. This particular VPN (and many, many others like it) are composed with (and share) the Internet, using the session protocol IPsec in tunnel mode. IPsec makes the virtual link between *B* and *C*, which traverses the public Internet, secure by means of authentication and encryption. Thus messages between *B* and *C* have headers with TCP over IP over IPsec over IP (which is part of the header stack in Figure 2). Presumably the links from *A* to *B* and *C* to *D* are implemented on safer private networks, which are not shown.

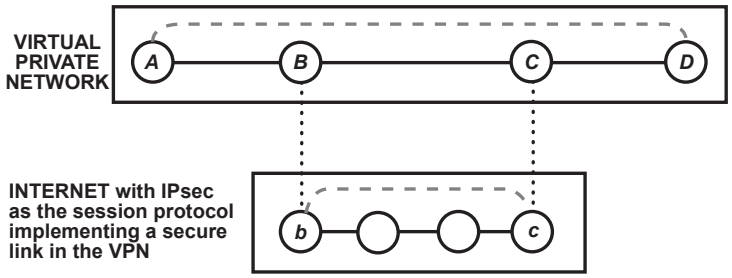

Fig. 8 Composition for the sake of security. There are two Internet-like networks, but the overlay network is private and has relatively few members.

4.3.2 Mobility

Another important reason for composition of networks is *mobility*, which means that a network must provide a persistent name for and connectivity to a device, even though that device is changing its physical connection to the network. To visualize mobility in networking terms, imagine that the cellphone in Figure 7 is continually moving from one cellular network or WiFi network to another, depending on the strength of radio signals in its vicinity.

Mobility is a big subject, covered thoroughly in [23], which organizes a survey of real mobility mechanisms by means of the geomorphic domain model. The survey shows that there are exactly two patterns for implementing mobility in networks. One pattern simply demands updates to routing as the links through which a member can be reached change. The other pattern uses composition of two layered networks

and the session protocol of the underlay. In this pattern, the session protocol of the underlay keeps overlay links alive as link endpoints change their attachments in the underlay.

There can be several instances of mobility in composed networks, at different levels and for different purposes. Do all of these implementations of mobility work together, or can they interfere with one another? Fortunately, in [24] it is proved that mobility mechanisms are indeed compositional, so that the presence of multiple implementations of mobility, in the same or interacting networks, do not interfere with each other's correct operation. This is a theorem about composed instances of our domain model, and a first original contribution to the theory of networking.

4.3.3 Cloud computing

A third example is that composition of networks is necessary for the virtualization of resources in cloud computing. Clouds often use several layers of virtualization to implement functions such as *service chaining, isolation,* and *quality-of-service (QoS) guarantees* for their tenants. At the same time, clouds must manage shared resources dynamically and effectively.

Consider the virtual link between *C* and *D* in Figure 8. Very likely it is implemented in a cloud, where the owner of the enterprise VPN is called a *tenant*. This link in a cloud is depicted in Figure 9. This link is implemented in the tenant's own (unshared, isolated) service network. The service network does *service chaining*, which means routing the tenant's messages through *network functions* such as *NF1* and *NF2*. Network functions, sometimes called "middleboxes," are network elements such as firewalls, intrusion detectors, load balancers, caches, and transcoders.

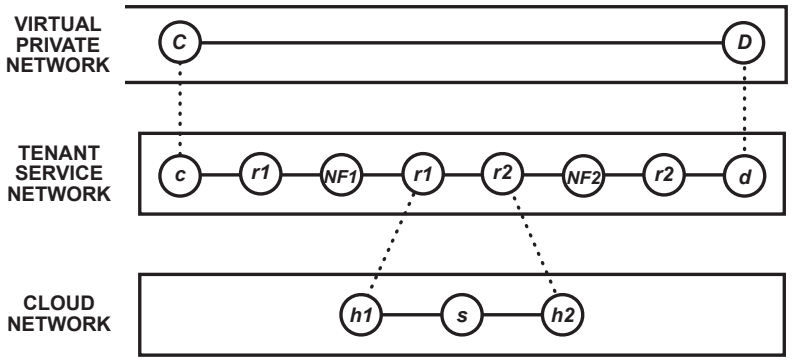

Fig. 9 Composition in a cloud. Layered networks provide special services, security, and quality-of-service guarantees, as well as simple transport.

In Figure 9, *C, c, NF1, NF2, d,* and *D* are all software running on virtual machines. *r1* and *r2* are tenant-specific routing functions running in the hypervisor

of virtualized machines. Because both these functions appear twice in the path, we know that *c* and *NF1* are on virtual machines hosted in the same virtualized machine, while *d* and *NF2* are also on the same machine. Links between virtual machines and hypervisor within the same machine are physical.

The two hypervisors also have interfaces (*h1* and *h2*) to the shared cloud network, which includes other switches such as *s* within a data center. A session in the cloud network implements a link in a tenant service network. The cloud provider may have service agreements with its tenants promising them a certain quality of service, *e.g.,* minimum bandwidth, on each tenant link. These agreements are enforced in the multiplexing of tenant-specific sessions onto the links of the cloud network. The cloud network may be a local-area network such as an Ethernet, or be implemented by one in the layer below.

The boundary between the VPN and tenant service network in Figure 9 is an important trust boundary, even though both networks are specific to a tenant. This is because members of the VPN run software that may be supplied by the tenant, while all the software below the VPN belongs to the cloud provider.

5 How the theory can contribute to network software

Expanding the design space: The possibility of disciplined composition greatly expands the design space for solving networking problems. It points the way to solutions that are too unusual to be discovered or considered with *ad hoc* approaches. In [23] we map out the design space for solving mobility problems, and in [24] show some previously undiscovered, efficient solutions to the problem of mobile-device users temporarily connected to a WiFi network that is moving, for example on a bus.

Software templates: In a hierarchy of layered virtual networks, every network is an instance of a network domain, and thus needs software to implement the behavior sketched out in Sections 3.2 and 4.2. Our theory of networking includes a generic algorithm for domain behavior that can be specialized for each domain. This specialization would not be difficult. Much of it would consist of defining data types and header formats, a process made easy by a domain-specific language such as P4 [2].

To a lesser extent, it may be possible to derive templates for the software of control planes. Control planes deal with diverse issues, from user requirements to resource allocation, security, and fault-tolerance. This makes the software more diverse, although all control planes have a common target in the data states of their networks.

Efficiency and optimization are top priorities in networking. Software templates based on the domain model are easily optimized in simple ways, such as removing vestigial structures. (For example, in any case where the mapping between two structures is one-to-one, one of them can probably be omitted.) They can also serve as the foundation for defining more sophisticated optimizations, which can then be re-used at many levels for many purposes, just as the templates are.

Compositional reasoning:

In today's networks, it is extremely difficult to get vital information about the messages traveling through the network. At the highest levels, this vital information includes the persistent identities of the source and destination, the nature of the communication (e.g., Web access *versus* real-time streaming), and the set of messages that should be logically grouped. At lower levels, this information includes how resource-allocation algorithms at different levels are interacting with each other on large groups of messages. This information is often disguised and transformed by many layers of complex network functionality implemented with *ad hoc* tables in network elements and *ad hoc* tags in packet formats. There is no easy way to understand what any of these tables and tags are for.

If stakeholders cannot understand what the traffic is, then they cannot know whether requirements are being satisfied. If stakeholders cannot understand the purpose of tables and tags, then they cannot understand whether the tables and tags are being populated or used correctly.

The study of many examples has shown that, when the examples are modeled compositionally, the *ad hoc* tables and tags are actually instances of the network state and header information presented in this paper (spread out over multiple composed layers.) Thus the domain theory provides a clear context and purpose for this information, and many consistency constraints among them[3] that can be used for verification.

The other benefit of the compositional framework is that it provides traceability of messages from top to bottom layers. Preliminary experiments have shown that this traceability can be exploited to prove theorems about security. For example, we can prove compositionally that all messages in a particular high-level grouping were transmitted along a particular low-level path, where the low-level path has security mechanisms built in. Traceability is also useful for fault diagnosis. For example, when messages are not reaching their high-level destinations as expected, traceability may help locate the misconfiguration problem at a lower level.

A further goal is to reason compositionally about performance and fault-tolerance. These problems appear to have many dimensions and be much more difficult, but additional structure almost always makes a hard problem somewhat easier.

6 Conclusions

In the introduction, benefits of a well-designed formal domain theory were listed. Some of them are already exemplified by the theory of networking, and we hope that future research and development will fulfill more of these aspirations.

The scarcity of domain theories is due partly to the difficulty of working theory-building into the schedule, budget, and process of software-development projects. It is also due to the fact that the skills necessary for development (especially mastery

[3] The topic of consistency constraints is not covered in this brief overview paper.

of detail) are somewhat different from the skills best-suited to model-building (especially extracting simplicity from complexity). It is also due to the lack of publicity for this approach and agenda.

Researchers in software engineering are in a position to help overcome all three obstacles. We are seeing this happen more and more, as researchers tie their work on programming and verification tools to concrete problems in specific domains. Researchers only need to take the extra step, which is to generalize and improve the domain models in terms of which these problems are stated. This will expand the technical challenges to be faced, but it will also magnify the potential benefits and satisfactions of the work.

Acknowledgements This paper is based on the results of long-term collaborations with Michael Jackson and Jennifer Rexford.

References

1. Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2014.
2. Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communications Review*, 44(3), July 2014.
3. David D. Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of SIGCOMM*. ACM, August 1988.
4. David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: Defining tomorrow's Internet. *IEEE/ACM Transactions on Networking*, 13(3):462–475, June 2005.
5. John Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.
6. Anja Feldmann. Internet clean-slate design: What and why? *ACM SIGCOMM Computer Communication Review*, 37(3):59–64, July 2007.
7. Timothy G. Griffin and João Luís Sobrinho. Metarouting. In *Proceedings of SIGCOMM*. ACM, August 2005.
8. Mark Handley. Why the Internet only just works. *BT Technology Journal*, 24(3):119–129, July 2006.
9. ITU. Information Technology—Open Systems Interconnection—Basic Reference Model: The basic model. ITU-T Recommendation X.200, 1994.
10. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006, 2012.
11. Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the 17th International Conference on Software Engineering*, pages 15–24. ACM Press, April 1995.
12. Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.
13. D. Kreutz, F. M. V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azoldolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, January 2015.

14. Timothy Roscoe. The end of Internet architecture. In *Proceedings of the 5th Workshop on Hot Topics in Networks*, 2006.
15. Oliver Spatscheck. Layers of success. *IEEE Internet Computing*, 17(1):3–6, 2013.
16. Philip J. Taylor and Timothy G. Griffin. A model of configuration languages for routing protocols. In *Proceedings of the 2nd ACM/SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*. SIGCOMM, 2009.
17. Geoffrey Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On static reachability analysis of IP networks. In *Proceedings of IEEE Infocom*. IEEE, March 2005.
18. Pamela Zave. Bridging the research-industry gap: The case for domain modeling. In *Proceedings of the 37th International Conference on Software Engineering/Workshop on Software Engineering Research and Industrial Practice*. IEEE, 2015.
19. Pamela Zave. A theory of networks: In the beginning . . . In *Proceedings of the 35th NATO International Summer School (Marktoberdorf)*. Springer LNCS to appear, 2015.
20. Pamela Zave. Theories of everything. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016.
21. Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.
22. Pamela Zave and Jennifer Rexford. The geomorphic view of networking: A network model and its uses. In *Proceedings of the 7th Middleware for Next Generation Internet Computing Workshop*. ACM Digital Library, 2012.
23. Pamela Zave and Jennifer Rexford. The design space of network mobility. In Olivier Bonaventure and Hamed Haddadi, editors, *Recent Advances in Networking*. ACM SIGCOMM, 2013.
24. Pamela Zave and Jennifer 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.