Verifiably-Safe Software-Defined Networks for CPS

Richard Skowyra rskowyra@bu.edu
Andrei Lapets lapets@bu.edu
Azer Bestavros best@bu.edu
Assaf Kfoury kfoury@bu.edu

Computer Science Department
Boston University

ABSTRACT

Next generation cyber-physical systems (CPS) are expected to be deployed in domains which require scalability as well as performance under dynamic conditions. This scale and dynamicity will require that CPS communication networks be programmatic (i.e., not requiring manual intervention at any stage), but still maintain iron-clad safety guarantees. Software-defined networking standards like OpenFlow provide a means for scalably building tailor-made network architectures, but there is no guarantee that these systems are safe, correct, or secure.

In this work we propose a methodology and accompanying tools for specifying and modeling distributed systems such that existing formal verification techniques can be transparently used to analyze critical requirements and properties prior to system implementation. We demonstrate this methodology by iteratively modeling and verifying an OpenFlow learning switch network with respect to network correctness, network convergence, and mobility-related properties.

We posit that a design strategy based on the complementary pairing of software-defined networking and formal verification would enable the CPS community to build next-generation systems without sacrificing the safety and reliability that these systems must deliver.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol Verification—*cyber-physical systems*; C.2.1 [**Network Architecture and Design**]: *software-defined networking*

Keywords

Software-Defined Networking, OpenFlow, formal verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HiCoNS'13, April 9–11, 2013, Philadelphia, Pennsylvania, USA.
Copyright 2013 ACM 978-1-4503-1961-4/13/04 ...$15.00.

1. INTRODUCTION

As the capabilities of modern computer technology continue to improve, an increasing number of safety-critical tasks are integrated with or replaced by automatic control and sensing systems (e.g., self-driving cars, unmanned aerial vehicles, mobile sensor platforms, smart-grid technologies, GPS navigation systems, and so on). These systems are frequently distributed and often must be either large-scale or require elasticity of scale as utilization or population fluctuates. Furthermore, many of these cyber-physical systems must interoperate over very large collections of interacting devices (possibly federated and running under multiple authorities), and they must be robust enough to handle churn, mobility, and other potentially unstable or unpredictable conditions. In order to operate under these conditions, CPSs will inevitably need to rely on increasingly more sophisticated data and control networks while maintaining strong safety and reliability guarantees. These requirements suggest that CPS communication networks be both programmatic (i.e., not requiring manual intervention at any stage) [25] and flexible to changing operating conditions.

Let us consider communication networks that must link multiple mobile end-hosts which communicate intermittently but reliably. On a small scale these networks can be easily built using multiplexed wireless communication like 802.11 or 802.15.4 Zigbee routers. As scale increases and it becomes possible that end-host mobility may span multiple routers or access points, however, significant overhead may be imposed by a naive attempt to maintain an updated, consistent network state that tracks end-host locations. This complexity may be compounded by the need to maintain QoS guarantees with respect to delay, loss rate, power consumption, etc. Ideally, any such communication network for next-generation cyber-physical systems would use existing physical infrastructure, but leverage domain-specific network designs and protocols to maximize these guarantees.

Recent advances in software-defined networking, specifically the OpenFlow initiative, allow precisely this flexibility in both physical infrastructure and software [23]. OpenFlow networks outsource routing logic to a domain-specific software controller written by the network designer, which runs on a remote machine (ranging from commodity hardware to custom FPGAs) connected to each switch via a secure channel. OpenFlow-enabled switches send unknown or unhandled packets to this controller, which responds by installing flow-rules (next-hop rules which trigger based on packet headers) in one or more of switches across the net-

work. Compliant switches then route data-plane packets based on these flow rules.

OpenFlow routing hardware is supported by a number of major equipment vendors, often requiring only a firmware upgrade to existing, deployed routers [2]. OpenFlow has been used to implement a wide variety of network tools and protocols, including routing circuit-switch and packet-switched traffic over the same switch [10], wave-length path control in optical networks [21], in-network load balancers [26], wireless sensor networks [22], and wireless mesh networks [11].

While OpenFlow provides a powerful means for specifying control-plane logic and protocols, the resulting networks may not satisfy necessary safety conditions and QoS guarantees. In addition, controllers may contain not only implementation errors, but also critical design and logic flaws arising from insufficient or incorrect domain knowledge on the part of the designer, unexpected concurrency issues, misplaced assumptions about the operating environment, etc. These flaws could be extremely damaging if not detected before system deployment, especially in applications like vehicular control networks.

Fortunately, existing formal analysis and verification tools, applied to a model of the proposed system design, can be used to determine in a semi-automated manner that distributed systems built using OpenFlow do enforce their requirements in all cases. However, these tools are often limited to checking only properties in a small set of formal logics (LTL, relational calculus, process calculi, and so on). Real-world systems often have requirements spanning many such logics, all of which must be verified using different formalisms.

In this work, we present an infrastructure and associated tools for specifying and analyzing real-world, formally disparate properties of distributed systems, without requiring prior knowledge of any formal logics or languages. We provide an example using an OpenFlow-based network of learning switches to allow communication between mobile end-hosts. We investigate safety, stability, and probabilistic reliability properties, and use the result to iteratively design the system model until it verifiably satisfies all design requirements.

The rest of this paper is organized as follows. Section 2 describes the OpenFlow standard in more detail and discusses how communication systems controlled using OpenFlow programs may suffer from design or logical errors. Section 3 describes how these errors can be detected and fixed by analyzing formally specified properties of a model of the communication system prior to implementation. Section 4 describes an infrastructure for modeling distributed systems and automatically specifying and analyzing their properties. Section 5 describes related work, and Section 6 concludes.

2. OpenFlow

OpenFlow [23] is an emerging routing standard for software-defined networking that enforces a clean separation of the data and control planes. An OpenFlow switch routes data plane packets based on flow tables: ordered lists of rules guarded by a pattern to be matched over a packet header. These rules are installed by a controller, which is connected to each switch via a secure, dedicated link. Rules, at a minimum, can specify a port to route over, packet dropping, and forwarding of packets to the controller. If an incoming network packet's header does not match any flow rule, it is forwarded to the controller. Rules may also be set to expire after some time or duration, and be used to gather simple network statistics.

The OpenFlow controller is a software program running on a machine connected to each switch by a secure, dedicated control plane. The controller handles packets sent to it by OpenFlow switches and installs flow rules in the switches' flow tables. The functionality of the controller is determined completely by the application that it is being used to implement, but all controller programs must communicate with switches only by installing flow rules. Controllers can be written in a number of languages designed for the purpose. Popular choices include NOX/POX [16], Beacon [14], or Maestro [8], in addition to others [1].

An OpenFlow network provides a powerful, scalable infrastructure that can be used to tailor networks to specific tasks. Furthermore, all network design and planning beyond physical connectivity can be done in software, specifically in the design of the OpenFlow controller program. This allows for significantly richer network processing and routing functionality, but also introduces the possibility that (like any other software program) logical or design errors could lead to unsafe or incorrect behavior.

While network programming bugs may be tolerated in best-effort systems or those serving non-critical needs, many cyber-physical systems need iron-clad guarantees that a network routing mission-critical information will always (or with very high probability) meet its safety and correctness requirements. OpenFlow does not provide these guarantees, and so is not in its present form suitable for scalable network design in the cyber-physical realm. We propose that Verificare, a tool for formally verifiable distributed system design, can be used to bridge this gap.

3. DESIGN VERIFICATION

Significant prior work has been done on formal verification of software implementations [13]. Most of these techniques rely on the existence of a specification which is by assumption correct. The implemented software is then checked against this specification using formal techniques: the correct behavior of software is defined to be behavior which conforms to the software's specification, and bugs are defined as behaviors diverging from the specification.

However, as specifications (or languages for defining specifications) become more complex, another kind of correctness should be considered: how can a designer have confidence that the specification (or collection of specifications in multiple models) he defines is an accurate reflection of his intuitive and practical expectations regarding the correct behavior of the software? In particular, we are interested in some confirmation, from the perspective of a specification's designer, that the specification of a system correctly captures the behavior and requirements that the designer believes that it possesses. This is non-trivial, as there is no a priori correctness criterion against which a specification can be checked.

We address this issue by proposing that a specification should be constructed from a collection of formally modeled invariants, scenarios, and environments that are accompanied by familiar descriptions that a designer will recognize and understand within the context of the system's domain. On the back end, each invariant, scenario, or environment could potentially be represented in a distinct underlying

1. bind source address to source switch and port
2. if destination address is known:

   install fowarding rule to egress port for destination

3. if destination address is unknown:

   install forwarding rule to flood packets on all ports except source port

Figure 1: OpenFlow learning switch controller logic.

model or specification language that is appropriate for it. For example, the specification for a distributed file system which guarantees some level of availability could be constructed from a set of intuitive properties describing high-level notions of reliability; each intuitive property may map to multiple underlying specifications governing network behavior, data consistency, and data integrity.

We concentrate on the formal verification of distributed system design in the context of Verificare, a tool which allows specifications to be modeled and formally verified against properties chosen by the designer. This process is fast enough to allow rapid prototyping via formal verification, which can, for example, be used to iteratively model a specification as counter-examples to correctness claims are found.

Using the Verificare tool, it is possible to leverage OpenFlow's scalability without sacrificing confidence in the resulting system's safety and correctness. As an example application, we will model a network of OpenFlow learning switches, verify its key requirements, and demonstrate how this process can be used for rapid, iterative development of the specification. We chose to model a learning switch network because it is a well-known mobility solution that has a number of practical applications and deployment strategies, as well as a number of desirable properties which may be required to hold during its operation.

3.1 A Learning Switch Network

A learning switch is one that forms a routing table by binding the source address of network packets to the port on which those packets arrived. A network of these switches can be used to implement a communications network for mobile nodes such as autonomous mobile robots, warehouse floor staff, or mobile sensor platforms. As nodes move between locations, they connect to the network on different ports of different switches. These may, for example, be 802.11 wireless access points, 802.15.4 Zigbee routers, etc.

In order to ensure consistent network state, an OpenFlow controller maintains a network-wide routing table which it uses to install switch-specific packet forwarding rules. OpenFlow switches forward packets that don't match any forwarding rule to the switch, which records the packet's origin and installs a forwarding rule to describe its next hop (either to a specific switch port, or flooding to all switch ports but the origin). Specifically, the OpenFlow controller utilizes the logic in Figure 3.1 whenever it receives a packet sent to it by a switch.

In a traditional learning switch, every incoming network packet is sent to the controller, ensuring that the network state is as up-to-date as the last packet arrival. This minimizes the number of forwarding rules in the switch at any time, but can cause significant latency as all packets are forwarded to the controller for processing. In our example application, more forwarding rules are used to minimize the number of packets which must be sent to the controller without impacting the consistency of the logical and physical network states. Other designs are certainly possible, and in fact Verificare could be used to explore their tradeoffs in detail.

Our example network has the topology shown in Figure 2. Every switch has a dedicated channel to the OpenFlow controller, a bidirectional link to another switch, and four ports available for nodes to connect on. Nodes are mobile end-hosts which connect to the network on a switch port for some time, send and receive messages with other nodes, then disconnect for some time before reconnecting elsewhere.

While the example network is small compared to most real-world implementations, significant empirical evidence suggests that most violations of specified properties can be detected using a small scale model [3]. This approach of checking a scoped version of a model is standard in model-checking and related verification paradigms. By necessity the approach is not complete, so un-detected counterexamples may exist. The approach is sound, however, guaranteeing that any counter-example to a property found by the system will indeed violate that property.

The requirements that we will verify for this model are as follows.

1. *no-forwarding-loops*: Any packet that enters the network will eventually exit the network.
2. *no-blackholes*: Any packet that is sent will eventually be received.
3. *stable-correct-receiver*: If all nodes cease being mobile, eventually all packets that are received will be received by the intended recipient.
4. *stable-no-floods*: If all nodes cease being mobile, eventually no more packets will be flooded.
5. *bounded-loss-rate*: The expected packet loss rate of mobile nodes is below a specified bound.

The first two properties represent invariant safety and correctness requirements for the network: packets cannot be routed in infinite cycles or lost within the network. The third and fourth properties represent network convergence properties: once nodes remain stationary, the actual and perceived (by the controller) locations of each node will correctly converge. The final property represents a probabilistic expectation about loss rate in the face of node mobility. Note that we could also reason about the probability of specific loss rates (i.e., that the loss rate is below a specified bound with some probability) using the same formalisms discussed below.

3.2 Formalizing Requirements

In order to verify the properties expressed above, it is necessary to represent them as formulas in a formal logic that makes it possible to automatically and provably solve them over the domain of possible models. Specifically, in order to specify correct behavior of systems (or any set of defined algorithms), it is first necessary to define a collection of mathematical objects $\mathcal{M}$ that corresponds to the set of possible systems $\mathcal{S}$, as well as a mapping $f : \mathcal{S} \rightarrow \mathcal{M}$ from individual systems $s \in \mathcal{S}$ to objects $m \in \mathcal{M}$. It is

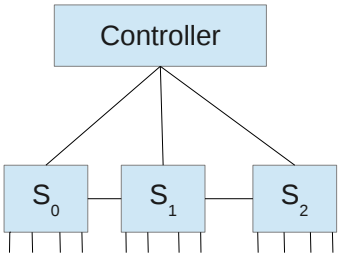

Figure 2: Learning switch network topology.

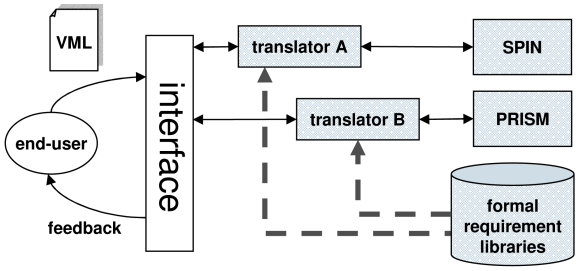

Figure 3: Verificare tool overview.

then necessary to establish some formal logical system $\mathcal{F}$, or *formalism*, that is sufficiently rich to express properties of objects in $\mathcal{S}$. Together $(f, \mathcal{M}, \mathcal{F})$ form one *model space* of the set of systems.

Given $(f, \mathcal{M}, \mathcal{F})$, it is possible to formally state what it means for a system to satisfy a property: a system $s \in \mathcal{S}$ satisfies some property φ iff it can be proven (either analytically or using a brute-force state space search) in the chosen formalism $\mathcal{F}$ that $f(s) \in \mathcal{M}$ satisfies a logical formula φ expressed in that formalism:

$$s \text{ satisfies requirement } \varphi \iff \varphi(f(s)) \in \mathcal{F}$$

However, adopting only one set of objects $\mathcal{M}$ and one formalism governing $\mathcal{M}$ that is sufficiently powerful to express all possible protocol properties is usually impractical. If some object $f(s) \in \mathcal{M}$ captured all possible aspects of a system $s \in \mathcal{S}$, determining whether that object satisfies some property may be intractable or undecidable.

A more tractable approach is to choose several different model spaces $(f_1, \mathcal{M}_1, \mathcal{F}_1), ..., (f_k, \mathcal{M}_k, \mathcal{F}_k)$ for systems, such that each model space can capture only some of the relevant properties of a system, and such that there exists a tractable or efficient algorithm for checking each property in its corresponding formalism.

In this work we consider two formalisms for describing properties of modeled systems: Linear Temporal Logic (LTL) and Probabilistic Computation Tree Logic (PCTL*) [5]. PCTL* formulas are logical statements governing probabilistic automata like discrete and continuous-time Markov chains; furthermore, these logical statements can contain probabilistic quantifiers. Assuming that there exists a mapping that can faithfully convert a protocol definition into a corresponding probabilistic automaton, it is then possible to construct a logical formula stating that the automaton (and thus, the protocol) satisfy, e.g., a certain expected packet loss rate.

LTL is a particular subset of PCTL*; LTL formulas are capable of expressing temporal properties about the existence of a state or set of states on the model which are of interest. For example, if there is a mapping that can faithfully convert a protocol definition into a Büchi automaton, then we can construct a logical formula stating that the automaton satisfies a property that will eventually (in some future state) always be true (over all states following it).

The first four properties defined above, *no-forwarding-loops*, *no-blackholes*, *stable-correct-receiver*, and *stable-no-floods*, are expressible as LTL formulas over a predicate which is true if and only if the corresponding model state is enabled. The final property, *bounded-loss-rate*, can be expressed in PCTL* using probabilistic quantifiers over the stationary distribution of the model represented as a Continuous-Time Markov Chain (CTMC).

We should note that in practice, *checking* that a given model of a system satisfies one or more properties usually amounts to an exhaustive search of a pruned state space. Individual states represent snapshots of the model in operation, and transitions represent valid ways in which a state can progress. Such an approach to checking properties is employed in other application domains, such as hardware processor design [6].

4. VERIFICARE

In this section, we describe how the critical requirements (described in Section 3.1) for a network of learning switches can be formally verified under multiple calculi using the Verificare tool. A key design principle of Verificare is rapid prototyping of a distributed system specification prior to its implementation. The tool consists of three related components, diagrammed in Figure 3.

- **Verificare Modeling Language (VML):** VML is a lightweight modeling language designed to permit rapid, iterative development of specifications. It is discussed in detail in Section 4.1.
- **Formal Requirement Libraries:** Verificare includes libraries of formal requirements which can be automatically bound to user-created VML code. These libraries are used to separate model development from requirement specification, and are described further in Section 4.3.
- **VML Translator:** Verificare offloads model verification to off-the-shelf verification engines using implemented translators to and from VML and tool-specific languages. These translators enable properties in a variety of logics to be checked, even if no individual tool supports all of the logics being used. Translation is discussed further in Sections 4.3 and 4.4.

In order to analyze a system under development, a designer uses VML to model relevant aspects of that system. Properties and requirements to be checked are selected from the formal requirement libraries and optionally augmented with user-defined requirements. The combined model and requirement specification is translated automatically to one or more backend verification engines; the output is then presented to the user as statements about the VML model.

By way of example, we first model an initial, naive version of the learning switch network in VML. We then describe how the properties to be verified are added to the model, and

```
 1 send(
 2  int link=0,
 3  set<host> dest=self.links[0],
 4  msgtype(a,b,...)
 5 )
 6
 7 recv(
 8  link=all,
 9  msgtype(var1, var2,...),
10  mfields='m'
11 )
```

Figure 4: VML send and receive primitives.

iteratively analyze and update the model until all necessary properties verifiably hold.

4.1 VML

Before modeling the network of learning switches, it is necessary to briefly introduce the Verificare Modeling Language, VML. VML is a specification modeling language inspired by Promela [17], Alloy [18], and Python. It is designed to capture sketches of specifications of distributed systems, and to abstract away implementation details in order to analyze architectural and design-specific properties.

VML supports typed sequences, sets, and dictionaries, in addition to integers and boolean values. In addition to basic indexing, insertion, and removal from these datastructures, VML allows simple containment checks, mapping of VML statements over sequences and sets, and comprehensions over sets. A non-deterministic `pick` primitive is also provided for sets, which allows a comprehension whose maximum size is bounded by the user. These features allow simple declarative reasoning about systems abstracted from implementation details, as is seen in the learning switch model.

In order to permit formal analysis of concurrency-related properties, control structures in VML are based on Dijkstra's Guarded Command Language [12]. VML supports `if` and `do` structures, each of which consists of a sequence of predicates guarding execution of a sequence of VML statements. If at any point multiple guards are true, the selection of which code block to execute is made non-deterministically.

4.1.1 The Network Abstraction

In addition to the features described above, VML provides a configurable network abstraction to simplify the modeling of distributed systems. Under this abstraction, a VML model consists of uniquely identified, independent, concurrent hosts communicating over a network. Each host has its own internal variable scope, state, and control logic.

Each host has a sequence, `links`, of sets of hosts, which denotes one or more links to the network over which some subset of other hosts are reachable. Messages sent over links are user-defined tuples prefixed with a message type.

Message sending is provided via the built-in `send` primitive in Figure 4.1.1, which sends a message to a (possibly empty) set of hosts over a specified link. If a host is not reachable over the specified link, it will not receive the message. The `dest` parameter may be any expression which evaluates to a set of hosts, such as a set comprehension. This allows simple reasoning about unicast (single hosts), multicast (set comprehensions) and broadcast (all hosts on the link) communication without needing separate implementations of each message-passing paradigm. If left empty,

```
 1 host opf_Controller():
 2  dict<hid><dict<hid><int>> routes
 3  loop true:
 4   recv(msg(saddr, daddr, sp)):
 5    all {switch | switch in routes.keys()}:
 6     if:
 7      (switch == m.src): #Bind source addr to source port
 8       routes[switch][saddr]=sp
 9      (switch != m.src): #Bind source addr to source switch
10       routes[switch][saddr]=routes[switch][m.src]
11   ?(!(daddr in routes[m.src].keys())):
12    routes[m.src][daddr]=-1
13   send(dest=sw_id, forwarding_rule(sp, saddr, daddr,
14        routes[switch][daddr])
15   send(dest=sw_iditch, msg(saddr, daddr))
```

Figure 5: Learning switch controller (naive model).

`send`'s `link` and `dest` parameters default to a host's first link and all hosts reachable via that link, respectively.

Messages are received using the `recv` primitive in Figure 4.1.1, which receives messages over one or more links. If messages are waiting on multiple links, the selection is made non-deterministically. By default, `recv` receives messages over all inks. The message type parameter imposes a criterion that only a message of type `msgtype` will be received; other messages waiting in the receiver will not trigger this statement.

The message must have as many tuple elements, with each of a matching type, as variables specified in the `recv` statement. After execution, each specified variable will contain the value of that message tuple field. Variables may be singletons, sets, or sequences. In the case of sequences, the length of the receiving sequence must be at least equal to the length of the sequence sent via the message. The optional `mfields` parameter provides a string to use as a prefix when accessing implicit message variables (such as the link a message arrived over).

4.2 A Learning Switch Network in VML

Recall that an Openflow network consists of three components: the controller, network switches which route packets based on flow tables, and end-hosts which send and receive packets. Our initial modeling attempt will specify a network of switches with flow tables, mobile end-hosts, and a controller with a standard learning switch functionality.

The learning switch network relies on bi-directional connections between ports, of which switches have several and end-hosts have one. This can easily be represented in the network abstraction by making each element of a host's `links` sequence correspond to a single port, restricting the size of the corresponding set of hosts per link to one, and ensuring mutual inclusion in two connected hosts' `links` sequences. Section 4.2.3, which models mobile end hosts, covers the process in detail. Similar syntax is used to set up the static topology shown in Figure 2 as the network's initial state.

4.2.1 The Openflow Controller

The naive VML model of an Openflow learning switch controller is presented in Figure 5. The `routes` datastructure provides a mapping of $(switch_id, destination_address) \mapsto port$, and allows the controller to look up the egress port from a switch to a destination address.

The remaining VML code is the controller's event handling loop. In lines 6-11, the controller updates its knowl-

```
 1 host opf_Switch():
 2  seq<pair<(int, int, int)><int>> flow_table
 3  int match=-2
 4  int saddr, daddr, p
 5  loop true:
 6   if:
 7    (len(self.links[0]) == 0):
 8     if:
 9      recv(msg(saddr, daddr)):
10       for rule in flow_table:
11        if:
12         ((m.link, saddr, daddr)==rule.first):
13          match=rule.second
14          break
15       if:
16        (match>-2):
17         if:
18          (match > -1]):
19           send(link=match, msg(saddr, daddr))
20          (match ==-1):
21           all {port | port in self.links
22                    && port != 0
23                    && len(links[port])}:
24            send(link=port, msg(saddr, daddr))
25        else:
26         send(link=0, msg(saddr, daddr, m.link))
27       else: skip
28    else:
29     recv(link=0, msg(sp, saddr, daddr)):
30      for rule in flow_table:
31       if:
32        ((sp, saddr, daddr)==rule.first):
33         match=rule.second
34         break
35      send(link=match, msg(saddr, daddr))
36     recv( link=0,
37           forwarding_rule(sp, saddr, daddr, dp)
38         ):
39      flow_table.insert(
40        0, pair((sp, saddr, daddr), dp)
41      )
```

Figure 6: VML model of an Openflow switch.

edge of the origin's current switch and port. The route is set to the source port in the case of the originating switch, or to the originating switch in the case of other switches.

At line 11, the controller checks the message destination against its network state. If the destination has been seen before, it instructs the switch to install the appropriate forwarding rule, which maps a network address to a port. If the destination has not been seen before, it sets the destination port to `-1`, which is interpreted by the switch as a flood. It then re-sends the packet to the querying switch for re-transmission.

4.2.2 The Openflow Switch

Openflow switches have no intelligence beyond the ability to match packet header fields against a table of forwarding rules and to contact the controller if a packet does not match any rule. For the purposes of this application, we modeled these forwarding rules as tuples of the form (*source port*, *sender address*, *destination address*). This ensures that a packet will be sent to the controller when the destination is unknown, or when the sender's current location (i.e., the specific binding of switch, source address, and port) is not already known to the controller. Responses from the controller are a forwarding rule that either specifies an output port or

```
 1 host end_host():
 2  hid switch
 3  int port
 4  loop true:
 5   if:
 6    (len(self.links[0])):
 7     if:
 8      recv(msg(saddr, daddr)): drop
 9     true:
10      daddr = pick 1 {n | n in _hosts
11                         && n is end_host }
12      send(msg(self.id, daddr))
13     true:
14      open_ports[switch].add(port)
15      _hosts[switch].links[port].remove(self.id)
16      self.links[0].remove(switch)
17    else:
18     switch = pick 1 {s | s in open_ports.keys()
19                       && len(open_ports[s])}
20     port = pick 1 {p | p in open_ports[switch]}
21     open_ports[switch].remove[port]
22     _hosts[switch].links[port].add(self.id)
23     self.links[0].add(switch)
```

Figure 7: VML model of a mobile end-host.

a flood instruction. Figure 6 presents the VML model of an Openflow switch.

Lines 8-27 model normal packet routing. The Openflow standards do not, to the authors' knowledge, explicitly state that packets arriving from the controller should be dealt with prior to packets arriving over normal links. We modeled it as such since the alternative allows, for example, indefinite delays on the installation of forwarding rules. The flow table is modeled as a sequence of tuples mapping to switch egress ports. This is an ordered ranking, which uses the `for` operator to iterate over the sequence by index order. The first rule which matches the packet's header tuple is triggered.

Lines 28-41 handle messages received from the controller. These are either forwarding rules, or forwarded packets that are re-sent after the installation of a forwarding rule.

4.2.3 The End-Host

End hosts are modeled as hosts that send and receive messages while connected to the network. Mobility is formalized by the ability to non-deterministically disconnect and to later reconnect on any open port. The end host VML model is presented in Figure 7.

Lines 7-12 handle message reception and sending. Lines 13-23 handle disconnection and re-connection, which consists of out-of-band adjustments to the network abstraction and a global `open_ports` dictionary tracking available ports. The semantics of the `send` function ensure that adding or removing an identifier to an element of a host's `links` sequence enables or disables the ability to send to that host. During the connection procedure, an end-host adds itself to the switch's link in order to receive messages, and adds the switch to its own link in order to send messages.

4.3 Specifying and Verifying Properties

In Verificare, properties to be formally verified are usually selected from a domain-specific library and not developed by the system designer. As distributed systems often cover multiple domains of expertise, it is unreasonable to assume that all systems designers will be well-versed in both

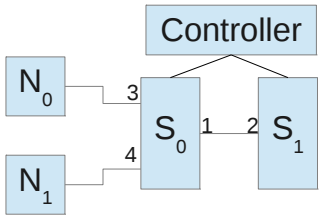

Figure 8: Modeled network topology.

Node	Controller		S_0	S_1
N_0	S_1	2	1	2
N_1	S_0	4	-	-

Figure 9: Forwarding loop network routing state.

the subtleties of relevant domains and the formal expression of those domains' traits. Furthermore, many critical properties (such as network connectivity, black-holes, etc.) are shared over many systems in the same domain. These can be formalized once and re-used many times.

```
1 host opf_Controller():
2   dict<hid><set<int>> end_host_ports
3   ...
4   loop true:
5     recv(msg(saddr, daddr, sp)):
6       if:
7         (sp in end_host_ports[m.src]):
8           ...
9         else: skip
```

Figure 10: Modified OpenFlow controller.

Properties are stored in libraries in two forms: as a high-level English-language statement, and as low-level formulas capturing that statement in one or more logics. While logical formulas are themselves general, the predicates and atoms used in a formula must be instantiated on a per-model basis. Network-related properties that consist entirely of statements about the state of the network abstraction and messages traversing it can be instantiated automatically. Others must be manually instantiated, with user-defined predicates to test relevant states and variables to bind as atoms.

The network safety properties *no-forwarding-loops* and *no-blackholes* can both be automatically instantiated, as they pertain only to the state of the network. The network convergence properties *stable-no-floods* and *stable-correct-receiver* primarily requires the user to label the relevant guarded code blocks and define a predicate to check the node identifier and destination address, respectively.

The network reliability property *bounded-loss-rate* requires the user to provide additional information, as explained in Section 4.4.2.

Once properties are selected and instantiated, the VML model is translated to one or more back-end verification engines using standard compiler implementation techniques [4]. These can range from single algorithms (e.g., non-interference checking) to off-the-shelf tools like Spin [17], Alloy [18], PRISM [20], and ProVerif [7]; the only requirement is that a

1. N_0 sends (N_0, N_1).
2. S_0 gets (N_0, N_1) on port 3, sends it to the controller.
3. The controller binds N_0 to port 3 on S_0.
4. The controller instructs S_0 to flood the message.
5. S_0 sends (N_0, N_1) to port 1.
6. S_1 gets ($N0$,$N1$) on port 2, sends it to the controller.
7. The controller binds N_0 to port 2 on S_1.
8. N_1 sends (N_1, N_0).
9. S_0 gets ($N1$, $N0$) on port 4, sends it to the controller.
10. The controller installs ($N_0 \rightarrow 1$) on S_0.
11. The controller re-sends the packet.
12. S_0 gets (N_1,N_0) from the controller, forwards it to port 1.
13. S_1 gets (N_1, N_0) on port 2 sends it to the controller.
14. The controller installs ($N_0 \rightarrow 2$) on S_1.
15. S_0 gets (N_1, N_0) from port 1,forwards it to port 1.
16. S_1 gets (N_1, N_0) on port 2, forwards it to port 2.

Figure 11: Execution trace for counter-example to *no-forwarding-loops*.

VML translator for the tool has been written.[1] Only properties that are checkable under that engine's formalism will be compiled along with the model. This may impose constraints on the order in which the user-selected properties can be checked. Packet loss ratios due to mobility, for example, should only be checked once it has been established that packets will not be lost when nodes are stationary.

4.4 Model Analysis

In this section, we analyze the naive model of a learning switch network with respect to the network safety and reliability properties defined above. The network safety properties are verified using the Spin model checker, which checks properties expressible in LTL (or more generally, any ω-regular property) via state-space search over a Büchi automaton. The network reliability properties are analyzed using PRISM, which performs probabilistic analysis of the system (modeled as a Continuous-Time Markov Chain) using user-specified transition rates. Note that these tools are property-specific, and are not a part of Verificare per se. Rather, the VML translator allows these tools to be treated as plugins to the system.

4.4.1 Network Safety and Convergence

The *no-blackholes*, *no-forwarding-loops*, *stable-no-floods* and *stable-correct receiver* properties were verified by translating the VML model to Promela, the modeling language used by the Spin model checker. Spin found a counter-example to the *no-forwarding-loop* property based on the naive controllers' mechanism for updating its routing table. In the following example, nodes are denoted using N_i and switches are denoted using S_i. Tuples denote a message sent from the first address to the second. The initial network state is as depicted in Figure 8. While the complete verification trace shown in Figure 11 is lengthy, relevant steps in the counter-example trace are as follows.

The final routing state is shown in Figure 9. The controller has bound N_0 to port 2 of S_1, which has resulted in flow rules that bind N_0 to port 1 on S_0 and port 2 on S_1. These rules will bounce a message for N_0 back and forth

[1]Translators for Spin and PRISM are currently under development. Others will be implemented in future work.

indefinitely. Note that this is also a violation of the *no-blackholes* property, as the message will never be received by the destination on port 3 of S_0. The root of the problem is in using a naive learning switch controller to manage multiple switches. Ports are expected to originate only traffic directly from nodes, not traffic forwarded from other switches. We introduced a new datastructure in the controller, `end_host_ports`, to track this distinction. The relevant portion of the modified controller is shown in Figure 10, in which a containment check within the sending switch's set of node ports is used as a guard over the route updating procedure.

After modifying the model to account for end-host ports, re-verification finds no counter-examples to network safety properties. A counter-example to *stable-no-floods* is found, however. Since rules never expire in the current model, a flood rule once installed will never be updated. Adding an expiration time (modeled as a non-deterministic option to expire) to flood rules and re-verifying the model now returns no counter-examples to the checked properties.

4.4.2 Network Reliability

Once network safety and convergence properties are verified, it is possible to analyze network reliability and performance with respect to average packet loss rates. Verificare currently uses PRISM to analyze such properties, as it can perform probabilistic analysis of Markovian processes. In this case, the system will be represented as a continuous-time Markov chain. Since transition rates are used to probabilistically change state and time, the VML translator requires guarded code blocks to be labeled with transition rates by the user. If a guard is not labeled, it is assumed to have a transition rate of 1. This allows the user to choose to only label guards that are relevant to the property under analysis.

For this example we chose to model packet loss rate only as a factor of mobility (the rate at which nodes join and leave) and send rate. Other potential factors could also be considered with minimal changes to the model, such as the rate at which forwarding rules expire, etc. The user would only have to assign rates the the respective guarded code blocks.

Given the structure of the VML model, mobility is modeled as two rates: a leave rate which models the frequency of disconnections, and a join rate which models the frequency of re-connections. This can be thought of as the rate at which nodes pass through service areas of access points and the amount of time to associate with the next access point, for example. The chart in Figure 12 uses a static join rate of 2, with leave and send rates as shown in the figure. In this analysis Verificare utilizes PRISM's capability to perform multiple verification runs using different parameter settings and to track the results.

5. RELATED WORK

We are unaware of any work wholly comparable to Verificare, but a number research communities intersect with specific aspects of the tool and its use. In this section, we compare relevant aspects of our work to work on provably safe and verifiable OpenFlow controllers, formal verification systems, and related work in the field of cyber-physical systems.

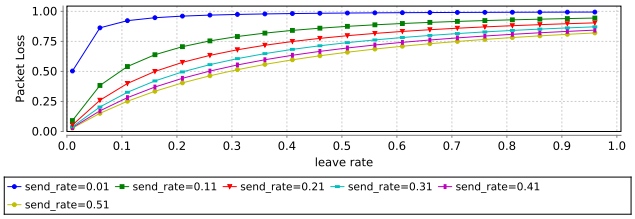

Figure 12: Packet Loss Rate

5.1 OpenFlow

In 2008, McKeown et al. proposed the OpenFlow standard [23]. A significant community in both industry and academia has since grown around this standard, including a number of researchers seeking to add provable or verifiable guarantees to OpenFlow controller programs.

Frenetic [15] is a declarative network programming language designed to allow safe programming of OpenFlow controllers. Frenetic provides a query abstraction that allows provably safe composition of controller functions, as well as modular controller design.

NICE [9] is a model checker specifically designed for OpenFlow controllers. It provides a library of common properties to be checked, and can analyze NOX [16] source code directly. NICE integrates symbolic execution with model checking to dramatically reduce the size of the state space to be searched by identifying equivalences among packet types.

Reitblatt et al. provide several formally verified consistent network update abstractions [24]. The authors define notions of per-packet and per-flow consistency, both at a high level and with respect to a mathematical network model presented in the paper. They define verifiable notions of trace invariants that allow model-checking of both notions of consistency.

Flog [19] is a logic-programming language that allows OpenFlow controller programs to be developed rapidly and in only a few lines of code. Flog breaks controllers into a flow-identification phase that specifies flows of interest, an information processing phase based on exhaustive triggering of inference rules, and a policy generation phase that generates forwarding rules to be installed on one or more switches.

5.2 Formal Verification

A key strength of Verificare is in the integration of off-the-shelf engines for formal verification of models. These verification tools each have a modeling language, property specification language, and formal system allowing automatic checking of properties over the model. Each tool excels at checking those properties which are capable of being expressed in its specification language.

The SPIN model checker [17] is designed to analyze concurrent processes communicating over channels. Models are written in Promela, which is a C-like language supporting non-determinism and providing a channel abstraction for modeling of inter-process communication. Properties are written in Linear Temporal Logic (LTL) or as never-claims, which are capable of expressing any ω-regular property. Spin translates the model and properties into Büchi automata, which are synchronously composed. The composed automaton's state space is then exhaustively searched for an instance that constitutes a reachable counter-example.

Alloy [18] is a declarative modeling and specification lan-

guage based on first-order logic, relational algebra, and set theory. Models and properties to be checked are not distinct in Alloy; properties are constraints over the space of model instances. Verification is done by translation of a scoped model to a Boolean formula, which is then passed to a SAT solver for satisfiability testing.

PRISM [20] is a verification tool that analyzes models written in a guarded-command language based in part on earlier languages [12]. Models correspond to probabilistic automata. Properties to be checked are written in Probabilistic Computation Tree Logic (PCTL*), which includes probabilistic and temporal quantifiers. PRISM also supports reward-based property verification, in which states and transitions can be labeled with rewards that are incremented whenever that state or transition is reached. PRISM has both model-checking (for qualitative verification of models) and numeric computation (for quantitative verification of models) libraries.

ProVerif [7] is a tool for analyzing cryptographic security properties of protocols. ProVerif models are represented as Horn clauses and verified using logical resolution. Checkable properties include reachability of defined states, observational equivalence of models, and correspondence.

6. CONCLUSIONS

In this work, we presented Verificare, a design and modeling tool for distributed systems. Verificare allows for real-world system properties from multiple domains to be verified against a system model or countered with example execution traces highlighting a property violation.

We also argued that Verificare can be used to bridge the gap between software-defined networking and QoS, safety, or reliability guarantees. This indicates a viable design methodology for the construction of scalable, verifiable communication networks for cyber-physical systems. We provided an example of this methodology in the form of an OpenFlow learning switch network. The system was modeled in Verificare, and network safety, convergence, and reliability properties were analyzed. Counter-examples to these properties were used to iteratively refine the system's design until all requirements were satisfied.

7. REFERENCES

[1] Openflow components, 2011.

[2] Open networking foundation members, 2012.

[3] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "small scope hypothesis". *Unpublished*, 2003.

[4] A. W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[5] C. Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[6] S. Berezin, E. Clarke, A. Biere, and Y. Zhu. Verification of out-of-order processor designs using model checking and a light-weight completion function. *Form. Methods Syst. Des.*, 20(2):159–186, Mar. 2002.

[7] B. Blanchet. Proverif automatic cryptographic protocol verifier user manual. *CNRS, Departement dInformatique, Ecole Normale Superieure, Paris*, 2005.

[8] Z. Cai, A. Cox, and T. Maestro. A system for scalable openflow control. Technical report, Technical Report TR10-08, Rice University, 2010.

[9] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[10] S. Das, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and L. Ong. Packet and circuit network convergence with openflow. In *Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference, 2010 Conference on (OFC/NFOEC)*, pages 1–3. IEEE, 2010.

[11] P. Dely, A. Kassler, and N. Bayer. Openflow for wireless mesh networks. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[12] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[13] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, 2008.

[14] D. Erickson. Beacon, 2012.

[15] N. Foster, R. Harrison, and M. Freedman. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011.

[16] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[17] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2005.

[18] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, Apr. 2002.

[19] N. P. Katta, J. Rexford, and D. Walker. Logic Programming for Software-Defined Networks. In *XLDI*, number 1, 2012.

[20] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *23rd International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[21] L. Liu, T. Tsuritani, I. Morita, H. Guo, and J. Wu. Openflow-based wavelength path control in transparent optical networks: a proof-of-concept demonstration. In *Optical Communication (ECOC), 2011 37th European Conference and Exhibition on*, pages 1–3. IEEE, 2011.

[22] A. Mahmud and R. Rahmani. Exploitation of openflow in wireless sensor networks. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 1, pages 594–600. IEEE, 2011.

[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[24] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication - SIGCOMM '12*, page 323, 2012.

[25] V. K. Sood, D. Fischer, J. M. Eklund, and T. Brown. Developing a communication infrastructure for the smart grid. In *Electrical Power & Energy Conference (EPEC), 2009 IEEE*, October 2009.

[26] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, pages 12–12. USENIX Association, 2011.