Correct by Construction Networks using Stepwise Refinement

Submission #198

Abstract
Building software-defined network controllers is an exercise in software development and, as such, likely to introduce bugs. We present CoCoNet, a framework for SDN development that facilitates both the design and verification of complex networks using stepwise refinement to move from a high-level specification to the final network implementation.

A CoCoNet user specifies intermediate design levels in a hierarchical design process that delineates the modularity in complicated network forwarding and makes verification extremely efficient. For example, an enterprise network, equipped with VLANs, ACLs, and Level 2 and Level 3 Routing, can be decomposed cleanly into abstractions for each mechanism, and the resulting stepwise verification is over 200x faster than verifying the final implementation. CoCoNet further separates static network design from its dynamically changing configuration. The former is verified at design time, while the latter is checked at run time using statically defined invariants. We present six different SDN use cases including B4 and F10. Our performance evaluation demonstrates that CoCoNet is not only faster than existing verification tools but can also find many bugs statically before the network design has been fully specified.

1 Introduction
Software-defined networks (SDNs) are a popular and flexible means of implementing network control. In an SDN, a logically-centralized controller governs network behavior by emitting a stream of data-plane configurations in response to network events such as changing traffic patterns, new access control rules, intrusion detection, and so on. But decades of research and industry experience in software engineering have shown that writing bug-free software is far from trivial. By shifting to software, SDNs trade one form of complexity for another.

Data-plane verification has risen in popularity with SDNs. As the controller generates new forwarding configurations, tools like Header Space Analysis (HSA) and Veriflow [15, 16] verify that safety properties hold for each configuration in real time. Network operators can rest assured that access control violations, routing loops, and other common misconfiguration errors will be detected before being deployed.

This style of verification is an important safeguard, but falls short in several ways.

Design. Applying verification techniques early in the development cycle saves effort by catching bugs as soon as they are introduced. But correctness properties often depend on many mechanisms spanning many different levels of abstraction and time scales. Thus the entire controller must be implemented before data-plane verification can be utilized. Furthermore, data-plane verification catches bugs once the controller has been deployed in a live network, making it hard to fix the bug without disrupting network operation.

Debugging. Verifying detailed, whole-network configurations makes debugging difficult: It is difficult to pinpoint which part of the controller caused a particular property violation in the final configuration.

Scalability. Although existing tools verify one property for a realistic network in under a second, the number of checks can scale non-linearly with network size. For example, checking connectivity between all pairs requires a quadratic number of verifier invocations [26]. Thus practical verification at scale remains elusive.

Ideally, the controller software itself might be statically verified to guarantee it never produces configurations that violate safety properties. But proving arbitrary software programs correct is a frontier problem. Recent work has proposed full controller verification, but only for controllers with limited functionality [3].

We propose a middle ground—a correct-by-construction SDN design framework that combines static verification with runtime checks to efficiently verify complex SDNs, detecting most bugs at design time. Our framework, called CoCoNet, consists of an SDN
CoCoNet is based on two principles. First, it enables SDN design by stepwise refinement. A network programmer begins by specifying a high-level view which captures the network’s behavior from an end host perspective. Such a specification might say: “A packet sent by a host is delivered to the destination host if and only if the source is not blacklisted by the network security policy”, while eliding details such as forwarding or access control mechanisms. In essence, this high-level view specifies correct network behavior. The network engineer continues by refining the underspecified parts of the design, filling in pieces until sufficient detail exists to deploy the network. A refined specification may state: “End hosts are connected via Ethernet switches to zone routers, which forward packets between zones via the core network, while dropping packets that violate security policy.”

CoCoNet automatically verifies that each refinement preserves the behavior of the higher-level view of the network by reducing each refinement to a Boogie program and using the Corral verifier to check this program for refinement violations [18]. Bugs are immediately detected and localized to the step in which they are introduced. The refinement relation is transitive, and so CoCoNet guarantees that the lowest-level implementation refines the highest-level specification.

Second, CoCoNet separates static network design from its run-time configuration. While refinements specify static invariants on network behavior, dynamic configuration is captured by runtime-defined functions (RDFs). In the above example the hosts and exact security policy are not known at design time and serve as design parameters. They are specified as RDFs, i.e., functions that are declared but not assigned a concrete definition at design time. RDFs are generated and updated at run time by multiple sources: the SDN controller reporting a new host joining, the network operator updating the security policy, an external load balancer redistributing traffic among redundant links, etc. Upon receiving an updated RDF definition, the CoCoNet compiler generates a new data plane configuration.

To statically verify the design without knowing the exact configuration, CoCoNet relies on static assumptions. At design time, RDFs can be annotated with assumptions that constrain their definitions. For example, the topology of the network may be updated as links come up and down, but each refinement may only need to know that the topology remains connected. At run time, CoCoNet checks that RDF definitions meet their assumptions. This separation minimizes real-time verification cost: most of the effort has been done up-front at design time.

Hence, CoCoNet decomposes verification into two parts, as shown in Figure 1. Static verification guarantees correctness of all refinements; this verification is done once, before network deployment. Dynamic verification checks that behaviors supplied at run time (by updating RDFs) meet the assumptions each refinement makes about run-time behaviors.

Although we apply our techniques to SDNs in this paper, they are equally applicable to traditional networks. In particular, stepwise refinement may help find bugs in potentially messy interactions between mechanisms such as VLANs and ACLs, and check that forwarding state matches the assumptions made in the specification.

**Contributions** The main contribution of this paper is a network design and verification methodology based on step-wise refinement and separation of static and dynamic behavior. We evaluate this methodology by implementing it in the CoCoNet tool chain and applying it to design and verify six realistic network architectures. Our performance evaluation demonstrates that CoCoNet is faster than existing data-plane verification tools, while also being able to find many defects statically, even before the network design has been fully specified.

### 2 CoCoNet by Example

In this section, we introduce features of CoCoNet by implementing and verifying a variant of the enterprise network design described by Sung et al. [29], simplified for the sake of presentation. Figure 2 shows the intended network design. Hosts are physically partitioned into operational zones, such as administrative buildings, and grouped by owner into IP subnets symbolized by colors—hosts in each zone are often in the same subnet, but not always. Intra-subnet traffic is unrestricted and isolated by VLAN, but traffic between subnets is subject to an access control policy.
Each operational zone is equipped with a gateway router. Each router is also associated with a subnet and implements access control: Inter-subnet traffic must first traverse the gateway tied to its source subnet followed by the gateway associated with its destination subnet. The details of access control may change as the network runs, but all inter-subnet traffic must always traverse the gateways that implement access control. The path highlighted with a dashed blue line in Figure 2 illustrates traffic from a host in subnet 2 to one in subnet 1.

Our refinement strategy is illustrated in Figure 3. At a high level, the goals of the network are simple: Group hosts by subnet, allow intra-subnet traffic, and subject inter-subnet traffic to an access control policy (Figure 3a). The first refinement splits the network into operational zones and distributes access control checks across gateway routers (Figure 3b). The second and third refinements detail the L2 switching fabric inside zones and the core (Figure 3c and d).

We now formalize these refinements in the CoCoNet language, introducing key language features along the way. Figure 4 shows the high-level specification that matches Figure 3a.

Roles The main building blocks of CoCoNet specifications are roles, which specify arbitrary network entities: hosts, switches, routers, etc. A role accepts a packet, possibly modifies it and forwards to zero or more other roles. Roles are parameterized, so a single role can specify a set of similar entities, allowing a large network to be modeled with a few roles. An instance of the role corresponds to a concrete parameter assignment. A role has an associated characteristic function, which determines the set of its instances: Given a parameter assignment, the characteristic function returns true if and only if the corresponding instance of the role exists in the network.

We use separate roles to model input and output ports of hosts and switches. The input port specifies how the host or switch modifies and forwards packets. The output port specifies how the network handles packets generated by the host. Our high-level specification introduces HostIn and HostOut roles, which model the input and output ports of end hosts. Both roles are parameterized by the IP address of the host (parameters are given in square brackets in lines 17 and 26), with the characteristic function cHost (expression after the vertical bar), declared in line 9.

Policies A role’s policy specifies how its instances modify and forward packets. CoCoNet’s policy language is inspired by the Frenetic family of languages [11]: complex policies are built out of primitive policies using sequential and parallel composition. Primitive policies include filtering packets based on header values, updating header fields, and sending packets to other roles.

The HostOut policy in lines 18–23 first computes subnet IDs of the source and destination hosts and stores them in local variables, explained below. Next, it performs two security checks: (1) filter packets whose
source IP does not match the IP address of the sending host (line 21), and (2) filter packets sent across subnets based on the network’s security policy (line 21). Line 22 drops packets whose destination IP does not exist on the network. All other packets are sent to the input port of their destination host in line 23.

The send policy on line 23 is a key abstraction mechanism of CoCoNet. It can forward the packet to any instance of any role. While a send may correspond to a single hop in the network’s data plane, e.g., sending from an input to an output port of the same switch or between two connected ports of different switches, it can also forward to instances without a direct connection to the sender, thus abstracting multiple hops through network nodes not yet introduced at the current refinement level. CoCoNet’s final specification may only contain the former kind of send’s, which can be compiled directly to switch flow tables.

The HostIn policy in line 25 acts as a packet sink, dropping all packets delivered to it. Any packets sent by the host in response to previously-received packets are interpreted as new packets entering the network.

Variables The HostOut role illustrates three kinds of variables available to a policy: (1) the pkt variable, representing the packet processed by the role, which is passed as an implicit argument to each role and can be both read and modified by the policy; (2) read-only role parameters; and (3) local variables that store intermediate values while the role is processing the packet.

Functions Functions are pure (side-effect free) computations used in specifying the set of role instances and parameters; and (3) local variables that store intermediate values while the role is processing the packet.

The send policy on line 23 is a key abstraction mechanism of CoCoNet. It can forward the packet to any instance of any role. While a send may correspond to a single hop in the network’s data plane, e.g., sending from an input to an output port of the same switch or between two connected ports of different switches, it can also forward to instances without a direct connection to the sender, thus abstracting multiple hops through network nodes not yet introduced at the current refinement level. CoCoNet’s final specification may only contain the former kind of send’s, which can be compiled directly to switch flow tables.

The HostIn policy in line 25 acts as a packet sink, dropping all packets delivered to it. Any packets sent by the host in response to previously-received packets are interpreted as new packets entering the network.

Variables The HostOut role illustrates three kinds of variables available to a policy: (1) the pkt variable, representing the packet processed by the role, which is passed as an implicit argument to each role and can be both read and modified by the policy; (2) read-only role parameters; and (3) local variables that store intermediate values while the role is processing the packet.

Functions Functions are pure (side-effect free) computations used in specifying the set of role instances and defining policies. Function declarations can provide an explicit definition with their body (e.g., sameSubnet in Figure 4), or only a signature (e.g., cHost, cSubnet, acl and ip2subnet) without a definition. In the latter case, the body of the function can be defined by subsequent refinements, or the body can be dynamically defined and updated at run time, making the function a runtime-defined function (RDF).

Our top-level specification introduces four RDFs: cHost (discussed above); cSubnet, a characteristic function of the set of IP subnets (each subnet is given a unique identifier); ip2subnet, which maps end hosts to subnet IDs based on the IP prefix; and acl, the network security policy, which filters packets by header.

RDFs are a crucial part of CoCoNet’s programming model. They separate static network design from its runtime configuration. In our example, explicit definitions of RDFs are immaterial to the overall logic of the network operation—making those functions runtime-defined enables specifying the network design along with all possible runtime configurations it can produce.

At run time, RDFs serve as the network configuration interface. For example, by redefining RDFs in Figure 4, the operator can introduce new hosts and subnets, update the security policy, etc. However, not all possible definitions correspond to well-formed network configurations. In order to eliminate inconsistent definitions, CoCoNet relies on assumptions.

Assumptions Assumptions constrain the range of possible instantiations of functions—both explicit instantiations in a later refinement and runtime instantiations in the case of RDFs—without fixing a concrete instantiation. Consider the ip2subnet() function, which maps end hosts to subnets. We would like to restrict possible definitions of this function to map valid end host IP addresses to valid subnet IDs. Formally,

\[ \forall \text{addr} . \text{cHost}(\text{addr}) \Rightarrow \text{cSubnet}(\text{ip2subnet}(\text{addr})). \]

Line 13 states this assumption in the CoCoNet language.

In general, CoCoNet assumptions are in the fragment of first-order logic of the form \( \forall x_1 \ldots x_n . F(x_1, \ldots, x_n) \), where \( F \) is a quantifier-free formula using variables \( x \). This fragment has been sufficiently expressive for the systems we examine and allows for efficient verification.

Until a function is given a concrete definition, CoCoNet assumes that it can have any definition that satisfies all its assumptions. Refinements are verified with respect to these assumptions. When the function is defined in a later refinement step, CoCoNet statically verifies that the definition satisfies its assumptions. CoCoNet performs this verification at run time for RDFs.

Refinements A refinement replaces one or more roles with a more detailed implementation. It provides a new definition of the refined role and, typically, introduces additional roles, so that the composition of the refined role with the new roles behaves according to the original definition of the role.

Consider Refinement 1 in Figure 3b, which introduces zone routers. It refines the HostOut role to send packets to the local zone router, which sends them via the two gateway routers to the destination zone router and finally the destination host. The routers are modeled by four new roles, which model the two router ports facing core and zone networks (Figure 5).

Figure 7 illustrates this refinement, focusing on roles. Blue arrows show the packet path that matches the path in Figure 3b. Solid arrows correspond to hops between different network nodes (routers or hosts); dashed arrows show packet forwarding between incoming and outgoing ports of the same router. Both types of hops are expressed using the send operation in Figure 6, which shows the CoCoNet specification of this refinement.

Line 55 shows the refined specification of HostOut, which sends the packet directly to the destination only if it is on the same IP subnet and inside the same zone...
Router ports and corresponding roles. Figure 5: Router ports and corresponding roles.

(line 62); otherwise it sends to the local zone router (line 65). Router roles forward the packet based on its source and destination addresses. They encode the current path segment in the VLAN ID field of the packet, setting it to the source subnet ID when traveling to the source gateway (segments 1 and 2), 0 when traveling between source and destination gateways (segment 3), and destination subnet ID in segments 4 and 5. The security check is now split into two: The aclSrc() check performed by the outgoing gateway (lines 19 and 42) and the aclDst() check performed by the incoming gateway of the destination subnet (line 31). The assumption in line 10 guarantees that a conjunction of these two checks is equivalent to the global security policy expressed by the acl() function. This assumption enables CoCoNet to establish correctness of the refinement without getting into the details of the network security policy, which may change at run time.

Subsequent refinements detail the internal structure of core and zone networks. We only show the core network refinement (Figure 3c). For simplicity, Figure 8 specifies the core switching fabric as a single Ethernet switch with switch port number i connected to zone i router. This simplification is localized to a single refinement: As the network outgrows the single-switch design, the network programmer can later revise this refinement without affecting the high-level specification or other refinements.

The refined RouterCoreOut role (Figure 8, line 2) forwards packets to the core switch rather than directly to the destination router. The core switch input port (line 4) determines the destination router based on the VLAN ID and destination IP address (as one final simplification, we avoid reasoning about IP to MAC address mapping by assuming that switches forward packets based on IP addresses) and forwards the packet via the corresponding output port.

Putting it all together Figure 9 shows the final step in specifying our example network: adding the physical network elements (hosts, switches, and routers). Recall that a role can model any network entity, from an individual interface to an entire network segment. For the CoCoNet compiler to generate flow tables, it needs to know how roles combine to form each data-plane element. This is achieved using declarations in lines 1–5, which introduce parameterized hosts and switches (CoCoNet currently models routers as switches), specified in terms of their input/output port pairs. A port pair can represent multiple physical ports of the switch. We omit a detailed description of host and switch constructs, as these are incidental to the main ideas of this work.

Other language features CoCoNet supports multicast forwarding using the fork construct. For example,

\[
\text{fork}(\text{uint}<16> \text{ port}|\text{port}>0 \text{ and } \text{port}<n())
\]

\[
\text{send } \text{SwitchOut}[\text{port}]
\]

spawns a parallel copy of the send statement for each assignment to the port variable satisfying the fork condition (expression after the vertical bar). Each parallel thread operates on a private copy of the packet. Note that n() can be an RDF, in which case the number forked is determined at run time.

Underspecified behaviors can be expressed using nondeterminism. In the following snippet

\[
\text{havoc pkt.dstIP; assume pkt.dstIP | pkt.srcIP}
\]

the havoc statement nondeterministically picks a value for the dstIP field of the packet; the assume statement constrains the possible choices. Non-determinism is only allowed in high-level specifications and cannot occur in the final, most detailed, definition of any role.

3 Refinement-based verification

We informally present the semantics of CoCoNet specifications, the kinds of correctness guarantees that can be established through refinement-based verification, and the design of CoCoNet verification tools. See Appendix A for a more formal presentation.

Semantics We start with assigning semantics to roles as packet transformer functions. Let Pkt be the set of all possible packets, and Loc be the set of locations, where each location identifies a unique role instance in a CoCoNet specification. We define the set of located packets LPkt = \{(p, 1) | p ∈ Pkt, 1 ∈ Loc\}.

We define semantics of a role R as a partial function \([R] : LPkt \rightarrow 2^{LPkt}\) that takes a packet located at an instance of R and returns the set of all possible outputs that R can produce for this packet. Each output corresponds to one way of resolving non-determinism in the implementation of R. Deterministic roles produce a unique output. An output consists of a set of located packets, where each packet is generated by a send statement. In particular, a role that does not perform any send’s drops the packet, which corresponds to an empty set of packets. A unicast role produces output sets, each consisting of a single packet. Multicast roles, expressed via a combination of fork and send statements, produce output sets containing multiple packets.

We define refinement relation \(\sqsubseteq\) over roles:

**Definition 1** (Role refinement). Role \(\mathcal{R}\) refines role \(R\) \((\mathcal{R} \sqsubseteq R)\) if \(R\) and \(\mathcal{R}\) have identical parameter lists and characteristic functions and

\[\forall p \in \text{Domain}(\mathcal{R}), [\mathcal{R}](p) \subseteq [R](p). \quad (1)\]
A CoCoNet program defines a sequence of specifications, where a specification consists of a set of roles. Each `refine` block introduces a new specification obtained from the previous specification by providing new implementations for some of the roles and introducing new roles.

Next, we informally introduce the `inline` operation, which takes a role R and a set of roles \{P_1, ..., P_k\} and recursively inlines the implementation of \(P_1\) in \(R\) whenever \(R\) sends to \(P_1\). Consider the refinement in Fig-
use the Corral model checker [18] to solve it. We chose
Corral over other state-of-the-art model checkers due to
its expressive input language, called Boogie [19], which
enables a straightforward encoding of CoCoNet specifi-
cations. Given roles R and R, we would like to check
property (1) or, equivalently, \( \neg (\exists p, p', p' \in R(p) \wedge p' \not\in R(p)) \) (to simplify presentation, we assume that roles
are unicast, i.e., output exactly one packet). We encode
this property as a Boogie program:

\[
p' := \text{proc}(p); \text{assert(proc}(p, p'))
\]

Here, \( \text{proc} \) is a Boogie procedure that takes a lo-
cated packet \( p \) and non-deterministically returns one of
possible outputs of \( R \) on this packet; \( \text{proc}(p, p') \) returns
true iff \( p' \in R(p) \). We use Boogie’s havoc construct to
encode nondeterminism. We encode CoCoNet assump-
tions as Boogie axioms, and characteristic functions of
roles as procedure preconditions [19]. Violation of prop-
erty (1) triggers an assertion violation in this program.

Corral is a bounded model checker, i.e., it only detects
assertion violations that occur within a bounded number
of program steps. We sidestep this limitation by bound-
ing the maximal number of network hops introduced by
each refinement. This is a natural restriction in network
verification, as any practical network design must bound
the number of hops through the network. We introduce
a global counter incremented on every send operation
and generate an error when it exceeds a user-defined
bound, which is also used as a bound on the number of
program steps explored by Corral. Coincidentally, this
check guarantees that refinements do not introduce for-
warding loops.

**Verifying path properties** CoCoNet’s refinement-
based verification operates on a single role at a time and
never considers global forwarding behavior of the net-
work. Importantly, however, it guarantees that all such
behaviors are preserved by refinements, specifically, a
valid refinement can only modify a network path by in-
roducing intermediate hops into it; however, it cannot
modify paths in any other way, add or remove paths.

This invariant can be exploited to dramatically speed
up conventional property-based dataplane verification.
Consider, for example, the problem of checking pairwise
reachability between all end hosts. CoCoNet guarantees
that this property holds for the network implementation
if and only if it holds for its high-level specification. Of-
ten, the high-level specification is simple enough that
the desired property obviously holds for it. If, however,
the user does not trust the high-level specification, they
can apply an existing network verification tool such as
NetKAT, HSA, or Veriflow to it. Such verification can
be performed much more efficiently than checking an
equivalent property directly on the detailed low-level
implementation.

**Limitations** Because CoCoNet specifications describe
how individual packets are forwarded, it cannot verify
properties related to multiple packets such as stateful net-
work behaviors induced by say stateful firewalls. This
limitation is shared by virtually all current network veri-
fication tools, which verify dataplane snapshots.

However, stateful networks can be built on top of Co-
CoNet by encapsulating dynamic state inside RDFs. For
example, a stateful firewall specification may include a
function that determines whether a packet must be
blocked by the firewall. This function is computed by an
external program, potentially based on observed packet
history. CoCoNet can enforce statically defined invari-
ants over such functions. For example, with multiple
firewalls, it can enforce rule set consistency and ensure
that each entering packet is inspected by one firewall.

**Assumption checker** CoCoNet’s dynamic assumption
checker encodes all function definitions and assumptions
into an SMT formula and uses the Z3 SMT solver [6] to
check the validity of this formula.

4 Compiler

The CoCoNet compiler proactively compiles speci-
fications into switch flow tables; it currently supports
OpenFlow and P4 backends. Due to space limitations,
we only describe the OpenFlow backend.

The OpenFlow backend uses NetKAT as an interme-
diate representation and leverages the NetKAT compiler
to generate OpenFlow tables during the final compilation
step. Compilation proceeds in several phases. The first
phase computes the set of instances of each role by find-
ing all parameter assignments satisfying the characteris-
tic function of the role with the help of an SMT solver.
During the second phase, we specialize the implemen-
tation of each role for each instance by inlining function
calls and substituting concrete values for role parameters.

The third phase constructs a network topology graph,
where vertices correspond to hosts and switches, while
describes model physical links. To this end, the compiler
statically evaluates all instances whose roles are listed as
outgoing ports in host and switch specifications and
creates an edge between the outgoing port and the in-
coming port it sends to. The resulting network graph is
used in an emulator (Section 6).

During the fourth phase, instances that model input
ports of switches are compiled to a NetKAT program.
This is a straightforward syntactic transformation, since
NetKAT is syntactically and semantically close to the
subset of the CoCoNet language obtained after function
inlining and parameter substitution. During the final
compilation phase, the NetKAT compiler converts the
NetKAT program into OpenFlow tables to be installed
on network switches.
The resulting switch configuration handles all packets inline, without ever forwarding them to the controller. An alternative compilation strategy would be to forward some of the packets to the controller, which would enable more complex forms of packet processing that are not supported by the switch.

At run time, the CoCoNet compiler translates network configuration updates into updates to switch flow tables. Recompiling the entire network dataplane on every reconfiguration is both inefficient and unnecessary, since most updates only affect a small fraction of switches. While our current prototype does not support incremental compilation, it can be implemented as a straightforward extension.

5 Case studies

We show that real-world SDNs can benefit from refinement-based design by implementing and verifying six network architectures using CoCoNet. Our case studies cover both mainstream SDN applications such as network virtualization and emerging ones such as software-defined WANs and IXPs. The case studies have multiple sources of complexity including non-trivial routing logic, security constraints, fault recovery; they are hard to implement correctly using conventional tools. We present two studies in detail and briefly outline the remainder.

5.1 Case study 1: Software-defined WAN

We design and verify a software-defined WAN inspired by Google’s B4 [13] comprising geographically distributed datacenters connected by wide-area links (Figure 10). It achieves optimal link utilization by sending traffic across multi-hop tunnels dynamically configured by a centralized controller. In Figure 10, some traffic between datacenters 1 and 2 is sent via a tunnel consisting of underutilized links 1 and 2 instead of congested link 3. CoCoNet cannot reason about quality-of-service and relies on an external optimizer to choose tunnel configuration; however it can formalize the WAN architecture and enforce routing invariants, which ensure that optimizer configurations deliver packets correctly.

We specify end-to-end routing between end hosts in the WAN, including inter- and intra-datacenter routing. Local and global routing can be specified by different teams and integrated in a common CoCoNet specification. Our high-level specification (Figure 11) is trivial: it defines a set of hosts and requires that each packet be delivered to its destination, if it exists:

role HostOut[IP4 addr] | cHost(addr) =
    if cHost(pkt.dstIP) then send HostIn[pkt.dstIP]

Refinement 1 defines global routing and topology. It partitions hosts into subnets, localized within datacenters, and introduces WAN links across datacenters. It formalizes tunnel-based routing using two functions:

function tunnel(dcid_t src, dcid_t dst, Packet p): tid_t
function nexthop(tid_t tun, dcid_t dc): dcid_t

The former maps a packet to be sent from datacenter src to dst to ID of the tunnel to forward the packet through. The latter specifies the shape of each tunnel as a chain of datacenters. We define a recursive function distance(src, dst, tun), which computes the number of hops between two datacenters via tunnel tun. Correctness of global routing relies on the following assumption, which states that tunnels returned by the tunnel() function deliver packets to the destination in k hops or less, where k is a user-defined bound on the length of a tunnel:

assume (dcid_t src, dcid_t dst, Packet p)
    distance(src, dst, tunnel(p)) <= k()

Subsequent refinements detail intra-datacenter topology and routing. Specifically, we instantiate a fat-tree topology [1] within each datacenter: other topologies can be specified equally easily. Refinement 2 introduces groups of switches, called pods, within the datacenter fabric: each host is connected to a downstream port of a pod, which forwards packets to an upstream port of
the same pod, which, in turn, forwards to the destination pod. Pod behavior is underspecified by this refinement: the pod non-deterministically picks one of the upstream ports to send each packet through, giving rise to multiple paths, shown by blue and green arrows. This non-determinism is resolved by Refinement 3, which composes pods into two layers of switches. A bottom-layer switch picks a top-level switch to send to based on the hash of the packet’s destination address. Refinement 3 also takes advantage of path redundancy to route packets around failed links. The blue arrow in Figure 11d shows the normal path between top and bottom-layer switches within a pod; red arrows show the backup path taken in case of link failure. Finally, Refinement 4 details packet forwarding between pods via the core layer of switches.

5.2 Case study 2: Network virtualization

Network virtualization for multi-tenant datacenters is arguably the most important SDN application today [17]. It combines CPU and network virtualization to offer each client the illusion of running within its own private datacenter. Figure 12a shows the clients’ view of the datacenter as a collection of isolated LANs connected only by router nodes that have interfaces on multiple LANs. In reality, client workloads run inside virtual machines (VMs) hosted within physical servers connected by a shared network fabric (Figure 12b). Each server runs an instance of a software SDN switch, OpenVSwitch (OVS) [25], which isolates traffic from different tenants. Packets sent to VMs hosted on remote physical nodes are encapsulated and forwarded to remote OVS instances.

While the basic virtualization scheme is simple, industrial virtualization platforms, such as VMware NSX [17], have evolved into complex systems, due to numerous configuration options and feature extensions which are hard to understand and use correctly.

In this case study we untangle network virtualization with the help of refinement-based programming. We implement a basic virtualization scheme and a number of extensions in CoCoNet. Below we present two example extensions and show how CoCoNet separates the specification of various features from their implementation, thus helping users and developers of the framework to understand its operation, while also bringing the benefits of verification to network virtualization.

**Service chaining** Service chaining modifies the virtual forwarding to redirect packets through a chain of virtual middleboxes. Middlebox chains are formalized by the following RDF, which, based on packet headers and current packet location computes the virtual port to forward the packet to (the destination port or the next middlebox in the chain):

```java
function chain(Packet p, VPortId port): VPortId

// Service chaining required only a minor modification
// to the high-level specification: instead of forwarding the
// packet directly to its destination MAC address, we now
// forward it down the service chain:

role VHostOut[VPortId vport] | cVPort(vport) = ...
if pkt.dstMAC == hfffffffffffff(*bcast address*)
  then
    send VHostIn[vPortVNet(vport) == vnet]
    send VHostIn[chain(p, vport)]]
```

The implementation of this feature in the refined specification is, however, more complex: upon receiving a packet from a virtual host, OVS uses the chain() function to establish its next-hop destination. It then attaches a label to the packet encoding its last virtual location and sends the packet via a tunnel to the physical node that hosts the next-hop destination. OVS on the other end of the tunnel uses the label to determine which virtual host to deliver it to.

**Broadcast and ARP suppression** Virtual broadcast packets must be delivered to all VMs on the virtual network:

```java
role VHostOut[VPortId vport] | cVPort(vport) = ...
if pkt.dstMAC == hfffffffffffff(*broadcast address*) then
  fckr (VPortId vport | vPortVNet(vport) == vnet)
  then
    send VHostIn[vPortVNet(vport) == vnet]
```

This behavior is implemented via two cascading multicasts shown with dashed green arrow in Figure 12b. First, the OVS at the source multicasts the packet to all physical servers that host one or more VMs on the same virtual network. Second, the destination OVS delivers a copy of the packet to each local VM.

The ARP suppression extension takes advantage of the fact that most broadcast traffic consists of Address Resolution Protocol (ARP) requests. When ARP suppression is enabled for a virtual network, CoCoNet configures all OVS instances with a local table of IP-to-MAC address mappings, used to respond to ARP requests, locally.

Other extensions we have implemented include a decentralized information flow control model for networks and virtual-to-physical port forwarding.
5.3 Other case studies

Our third case study is a realistic version of the enterprise network, a simplified version of which was used in Section 2 [29]. In addition to features described in Section 2, we accurately model both MAC-based forwarding (within a VLAN) and IP-based forwarding across VLANs, implement support for arbitrary IP topologies that do not assume a central core network, and arbitrary level-2 topologies within each zone. We replace the standard decentralized routing protocols used in the original design with a SDN controller computing a centralized routing policy. This policy is expressed via RDFs, which are compiled to OpenFlow and installed on all switches.

The fourth case study implements the F10 fault-tolerant datacenter network design. F10 uses a variant of fat tree, extending it with the ability to globally reconfigure the network to reduce performance degradation due to link failures. In a traditional fat tree, a link failure may force the packet to take a longer path through the network, as shown in Figure 11d. F10 avoids this by reconfiguring all potentially affected switches to steer the traffic away from the affected region of the switching fabric. We implement and SDN version of F10, where the reconfiguration is performed by the central controller rather than a decentralized routing protocol.

Case study 5 implements a protocol called sTag [20]—a version of fat tree with source-based routing. The edge router attaches two tags to each packet: an mTag, which identifies switch ports to send the packet through at every hop, and a security tag that identifies the sender of the packet. The latter is validated by the last switch in the path, before delivering the packet to the destination.

Our final case study implements the iSDX software-defined Internet exchange point (IXP) architecture [12]. In iSDX, each autonomous system (AS) connected to IXP can define its own routing preferences. These preferences are encoded in the MAC address field of each packet sent from the AS to the IXP. The IXP decodes the MAC address and ensures that AS preferences do not conflict with the BGP routing database.

6 Implementation and Evaluation

We implemented CoCoNet in 4,700 lines of Haskell. We implemented, verified, and tested the six case studies described in Section 5 using the Mininet network emulator. CoCoNet, along with all case studies, is available under the Apache 2.0 license [5].

All experiments in this section were performed on a machine with a 2.5 GHz processor and 16 GB of RAM.

Table 1 summarizes our case studies, showing (1) total lines of CoCoNet code (LOC) excluding runtime-defined function definitions, (2) lines of code in the high-level specification, (3) number of refinements, (4) time taken by the CoCoNet static verifier to verify all refinements in the case study, and (5) time taken to verify the entire design in one iteration. The last column measures the impact of compositional verification: We combine all refinements and verify the combined specification against the high-level specification in one single step.

### Table 1: Summary of case studies.

<table>
<thead>
<tr>
<th>case study</th>
<th>LOC total (high-level)</th>
<th>#refines</th>
<th>verification time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAN virtualization enterprise</td>
<td>678</td>
<td>4</td>
<td>16 &gt;3600</td>
</tr>
<tr>
<td>F10</td>
<td>262</td>
<td>2</td>
<td>19 57</td>
</tr>
<tr>
<td>sTag</td>
<td>283</td>
<td>1</td>
<td>2 2</td>
</tr>
<tr>
<td>iSDX</td>
<td>190</td>
<td>2</td>
<td>3 3</td>
</tr>
</tbody>
</table>

Table 1: Summary of case studies. >3600 in the last column denotes experiments interrupted after one hour timeout.

The results show that CoCoNet verifies the static design of complex networks in a matter of seconds. Compositional verification was much faster than monolithic verification: a refinement that focuses on a single role has exponentially fewer states than the complete specification and is potentially exponentially faster to verify. Additional verification efficiency arises from parameterized specifications, which allow verifying all instances of a role at once.

#### Bug finding

We choose 3 (of many) examples from the case studies to show how CoCoNet detected subtle bugs early in our designs.

1. **Enterprise network:** When sending a packet between hosts on different subnets in the same zone, the zone router skipped access control checks at gateway routers (skipping path segments 2-3-4 in Figure 3b). Since this bug only manifests for some topologies and security policies, it is difficult to detect using testing or snapshot verification. The bug was detected early (in refinement 1), before L2 forwarding was introduced.

2. **WAN:** Consider the packet path in Figure 11d. Our implementation incorrectly sent the packet back to the core after hops 1 and 2, instead of sending it down via hop 3, causing a loop. This bug only manifests in response to a link failure, making it hard to catch by snapshot verification. It was detected only when verifying refinement 3, but the verifier localized the bug in space to the pod component.

3. **Virtualization:** This bug, discovered when verifying the sole refinement in this case study, is caused by the interplay between routing and security. The specification requires that neither unicast nor multicast packets can be exchanged by blacklisted hosts. The implementation filters out unicast packets at the source OVS; however, multicast packets were filtered at the destination and hence packets delivered to VMs hosted by the same server as the sender bypassed the security check.
Table 2: Number of hosts, switches, flowtable rules and size of NetKAT policy as a function of network’s scale parameter.

<table>
<thead>
<tr>
<th>Scale</th>
<th>Hosts</th>
<th>Switches</th>
<th>NetKAT Policy</th>
<th>Flowtable Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>11</td>
<td>1171</td>
<td>830</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>23</td>
<td>1986</td>
<td>3299</td>
</tr>
<tr>
<td>15</td>
<td>47</td>
<td>63</td>
<td>43948</td>
<td>31462</td>
</tr>
<tr>
<td>25</td>
<td>77</td>
<td>103</td>
<td>147608</td>
<td>89216</td>
</tr>
</tbody>
</table>

For each detected bug, Corral generated two witness traces, which showed how the problematic packet was handled by the abstract and refined implementations respectively. The two traces would differ in either how they modify the packet or in where they forward it.

Our encoding of refinements into Boogie guarantees the absence of false negatives, i.e., Corral does not miss any bugs (modulo defects in Corral itself). However, we have encountered three instances where Corral reported a non-existing bug. In all three cases this was caused by a performance optimization in Corral: by default, it runs the underlying Z3 SMT solver with a heuristic, incomplete, quantifier instantiation strategy. We were able to eliminate these false positives by reformulating some of our assumptions, namely, breaking boolean equivalences into pairs of implications.

6.2 CoCoNet vs NetKAT

CoCoNet verifies that each specification is a correct refinement. The NetKAT decision procedure for program equivalence [10] is the closest alternative to refinement verification. We compare CoCoNet against NetKAT on a parameterized model of the enterprise network case study [29]—we configure the network with three operational zones and scale the number of hosts and switches per zone. For an access control policy, we randomly blacklist communication between pairs of hosts. The topology of the operational zones and the router-to-router fabric are built from Waxman graphs. Table 2 summarizes the dimensions of our test network for a sample of scales.

We measure the full verification run time of CoCoNet, including the cost of static refinement verification and the cost of checking the assumptions of RDFs. We then perform an equivalent experiment using NetKAT. To this end, we translate each level of CoCoNet specification, along with definitions for the RDFs, into NetKAT, and use the NetKAT decision procedure to determine whether the lower-level specification exhibits a subset of the behaviors of the higher-level specification.

Figure 13 shows the verification run time in seconds as we increase the network scale. CoCoNet verification scales beyond that of NetKAT. CoCoNet performs much of the heavy lifting during static verification, taking advantage of all available design-time information captured in refinements, assumptions, and parameterized roles.

6.3 CoCoNet + HSA

At present, the easiest way to verify an arbitrary controller application is to verify reachability properties for each of the data-plane configurations it generates. As described in Section 3, CoCoNet can accelerate property-based verification: instead of checking path properties on the low-level data-plane configuration, one can check them more efficiently on the top-level CoCoNet specification, taking advantage of the fact that such properties are preserved by refinement.

We evaluate this using the Header Space Analysis (HSA) [15] network verifier. Using the same network scenarios as above, we use Frenetic to compile the NetKAT policy (translated from CoCoNet specification) into a set of OpenFlow flowtables. We produce the flowtables of the highest- and lowest-level specifications, shown below as Spec and Snapshot, respectively. We then apply HSA by creating the corresponding transfer functions and checking the all-pair reachability property on Spec and Snapshot, and measure the total run time. As can be seen in Figure 14, performing the verification on Spec and leveraging CoCoNet’s refinement verification results in dramatic improvement in verification performance, so that the cost of CoCoNet verification (red line) dominates the cost of HSA applied to the high-level specification (blue line).
7 Related Work

SDN controllers often use two-tier design: a controller emits a stream of data-plane configurations. There are many languages and verification techniques for both tiers, and some approaches that abandon the two-tiers.

Language design OpenFlow [24] is a data-plane configuration language: Controller frameworks like OpenDaylight [23], Floodlight [8], and Ryu [27] emit OpenFlow commands to update SDN-capable switches. The Frenetic family [2,9,21,28] introduces modular language design; they allow writing controller applications in a general purpose language and compiling to OpenFlow. VeriCon [3], FlowLog [22], and Maple [30] eschew the tiered structure entirely using custom languages that describe network behavior over time.

From a language design perspective, CoCoNet resides between a controller application and data-plane configuration language, but closer to the latter: CoCoNet receives a stream of definitions for run-time defined functions, and for each new definition produces a new data-plane configuration. However, refinements allow for modular development of invariant network behavior—in essence, CoCoNet characterizes how the pieces of the network fit together, reducing the burden on controller applications supplying RDF definitions.

Data-plane verification SDN verification takes two forms: controller verification and data-plane verification. Data-plane verification [14–16] verifies that a given set of safety properties (e.g., no black holes or forwarding loops) hold on a given data-plane configuration. Hence it must be reapplied to each configuration the controller produces. Further, checking reachability between host pairs scales quadratically with the number of hosts. Verification can be sped up by leveraging symmetries but the problem remains [26].

CoCoNet does not verify network properties directly. Rather, it guarantees that refinements are functionally equivalent, provided dynamically checked assumptions hold on RDF definitions. Often, reachability properties are “obvious” in high-level specifications: They hold by design and are preserved by functional equivalence, and so hold across refinements. If the design is not “obvious”, data-plane verification can be applied to the highest level CoCoNet specification, which is often dramatically simpler, enabling much faster property verification.

NetKAT [2] is a language with a decision procedure for program equivalence. This enables property verification but can also verify whether one NetKAT program is a correct refinement of another. However, NetKAT verification is not yet suitable for verifying the equivalence of large networks in near-real time. NetKAT lacks the abstractions—namely RDFs and assumptions—that allow some verification to be done statically.

NetKAT also lacks other language features that CoCoNet provides for stepwise refinement, including parameterized roles, in part because NetKAT is intended as a synthesis target emitted by the Frenetic controller. CoCoNet refinements, on the other hand, are human readable, even at scale. The definitions supplied to RDFs at run time are not, but their expected behavior is captured by assumptions annotated in the CoCoNet specification.

Controller verification VeriCon [3] and FlowLog [22] prove statically that a controller application always produces correct data-plane configurations. VeriCon reduces verification to SMT solving, while FlowLog uses bounded model checking. In both cases, scalability is a limiting factor. FlowLog also restricts expressivity to enable verification.

In contrast, CoCoNet statically verifies that refinements are functionally equivalent, but the refinement language is less expressive than either VeriCon or FlowLog—dynamic behavior is excluded, hidden behind RDFs. However, this combination of static and dynamic verification enables much greater scalability (see Section 6), while still providing strong guarantees about arbitrarily complex dynamic behavior hidden in RDFs.

Stepwise refinement Stepwise refinement for programming dates back to Dijkstra [7] and Wirth [31]. Despite its promise, refinement-based programming has had limited success in mainstream software engineering because: (1) developing formal specifications for non-trivial software systems is hard, (2) formalizing module boundaries for compositional verification is equally hard; even well designed software systems modules make implicit assumptions, and (3) verifying even simple software modules automatically is hard.

8 Conclusions

Our key discovery is that the factors that impede refinement based software engineering are not roadblocks to refinement-based network programming. First, even complex networks admit relatively simple high-level specifications. Second, boundaries between different network components admit much cleaner specifications than software interfaces. Finally, once formally specified, network designs can be efficiently verified.

CoCoNet can be seen as both a design assistant and a proof assistant: by imposing the refinement-based programming discipline on the network designer, it enforces more comprehensible designs that are also amenable to efficient automatic verification.

References


A Syntax and Semantics of CoCoNet

A.1 Syntax

The syntax of the CoCoNet system is given in Fig. 15. Let $\text{Id}$ be the set of identifiers, $\text{Pkt}$ the set of packets and $\text{Val}$ the set of values.

We suppose the existence of a countable set of identifiers, or variable names. Values comprise booleans $\text{true}$ and $\text{false}$, integers, tuples, and records of type $\text{id}$, written $\text{id}\{v\}$. Expressions comprise standard negation, binary operators $\otimes$, projection of fields $\text{id}.\text{id}$, construction of records $\text{id}\{e\}$, function call $\text{id}(e)$, built-in function calls $\text{id}!(e)$, variable call $\text{id}$, tuple construction $(e,\ldots,e)$, and call to the current packet being processed $\text{pkt}$. The semantics of a built-in function $\text{id}!$ is given by $\text{id}! \in \text{Val} \rightarrow \text{Val}$.

Statements allow filtering, which stops the computation if $e$ does not evaluate to $\text{true}$. Assumptions are slightly different in that they are not executable, but can be refined only if $e$ evaluates to $\text{true}$. Packet fields can be assigned explicitly with the $\text{:=}$ construct, or assigned to a nondeterministic value using $\text{havoc}$. Statements allow standard conditionals and let-bindings. Finally, a statement can send a packet to another role $\text{id}[e]$, or fork to multicast a packet across all variables $\text{args}$ satisfying condition $e$.

Declarations contain function definitions, both without a body to be refined later on or become user-defined functions, and with a body when defined explicitly. Declarations also contain role definitions: a role is parameterized by some arguments, and is only valid if some constraints on those arguments ($\otimes \text{e}$) and on the incoming packets ($\otimes \text{e}$) are true. Finally, assumptions allow restricting the future definitions of declared functions, both in future refinements and as user-defined functions.

Note that although types are part of the syntax, we drop them in the semantics to simplify notations.

Notes:

- The design and implementation of open vswitch.
- The OpenFlow protocol.
- A compiler and run-time system for network programming languages.
- OpenDaylight.
- Ryu.
- A Syntax and Semantics of CoCoNet.
A.2 Semantics

We give a denotational semantics of CoCoNet. The semantics of expressions, statements and declarations is given in terms of:

- a packet \( p \in Pkt \), a record of type \( Pkt \);
- a local environment \( \sigma \in \{id \rightarrow Val\} \), a partial function from identifiers to values, comprising \texttt{let}-defined variables; let \( Env \) be the set of local environments;
- a set of possible environments of functions \( \phi \). Functions take one argument (which can be a tuple). Each function’s denotational semantics is a (mathematical) function from a pair \((v,p)\) to a value \(v\). Each possible environment of functions is a partial function from identifiers to such denotational semantics. The set \( \phi \) is a set of such possible environments, representing all the possible function definitions. If \( \Phi \) represents the set of all the sets of possible environments of functions, we thus have

\[
\Phi = \mathcal{P}(\{id \rightarrow (Val \times Pkt) \rightarrow \mathcal{P}(\mathcal{P}(Pkt \times Env)))
\]

A.2.1 Semantics of expressions

The semantics of expressions is given in Figure 16, in terms of a triple \((\rho, \sigma, \phi)\). Expressions are nondeterministic, and thus their semantics is a set of possible output values.

Most of the semantics is standard. Note that the only nondeterminism is introduced by a function call \( id(e) \). Functions are defined in the environment \( \phi \), while built-in functions \( id! \) have their own semantics. The call \( pkt \) just returns the current packet \( p \) in all cases.

A.2.2 Semantics of statements

The semantics of statements is given in Figure 17, in terms of a quadruplet \((\rho, \sigma, \phi, \rho)\), and returns a set of
sets of pairs \((p, \sigma)\) to model both nondeterminism and multicasting.

The semantics of filter and assume only differ when \(\{\{(p, \sigma) \mid true \in [e](p, \sigma, \phi)\}\} = \emptyset\). In that case filter drops all packets (its semantics is \(\emptyset\)), whereas assume disallows refinements by denoting \(\emptyset\). The semantics of packet field updates (explicit or using havoc), conditionals, and let-bindings is standard.

The statement send is treated as a function call to the new role we are sending to, putting together all the nondeterministic behaviors of that role with a union. Finally, fork makes a cross-product on all the possibilities of each of the statements, generating all possible combinations of multicasting by picking one in each statement of as. Composition of statements \(a, as\) is defined using a similar cross-product to correctly handle both multicasting and nondeterminism.

A.2.3 Semantics of declarations

The semantics of declarations is given in Figure 18. A declaration updates the environments of functions \(\phi\) and roles \(p\). Constraints \(\mid e\) and \(\mid e\) on roles are considered true when unspecified.

A role declaration updates the role environment with a function \(r\). This function first checks whether the conditions \(e_3\) and \(e_4\) are fulfilled (first two lines); then, in the case where this definition is a refinement of an existing role, it checks whether the new role’s body is a valid refinement (third line); when those checks pass, \(r\) returns the semantics of the body \(as\) of the role.

A function declaration without an explicit body creates a possible function environment for any possible value of this function. When provided a body, those environments are restricted if a previous declaration existed (first line), otherwise an explicit definition is added (second line). Assumptions select the definitions of functions in \(\phi\) that agree with the assumption that is being considered.

The semantics of several declarations, refinements and finally whole specifications chains through the semantics of role declarations.
\[ \begin{align*}
&\text{role } \text{id}_1[\text{id}_2] = \text{role } \text{id}_1[\text{id}_2] | \text{true} / \text{true} \\
&\text{role } \text{id}_1[\text{id}_2] | e = \text{role } \text{id}_1[\text{id}_2] | e / \text{true} \\
&\text{role } \text{id}_1[\text{id}_2] / e = \text{role } \text{id}_1[\text{id}_2] | \text{true} / e \\
&\text{role } \text{id}_1[\text{id}_2] | e_3 / e_4 = \text{as}(\phi, \rho) = (\phi, \rho | \text{id}_1 \rightarrow r) \\
\end{align*} \]

where \( r = \lambda (p, v_2) \).

\[ \begin{align*}
\text{function } \text{id}_1(\text{id}_2)(\phi, \rho) &= (\{ \psi[\text{id}_1 \rightarrow f] \mid \psi \in \phi, f \in (\text{Val} \times \text{Pkt} \rightarrow \text{Val}), \text{id}_1 \notin \text{dom}(\psi) \}, \rho) \\
\text{function } \text{id}_1(\text{id}_2) = e(\phi, \rho) &= (\{ \psi \in \phi \mid \forall p, v_2. (\psi(\text{id}_1)(v_2, p) \in \text{as}(v_2/\text{id}_2)(p, \boxempty, \phi), \text{id}_1 \in \text{dom}(\psi)) \cup \{ \psi[\text{id}_1 \rightarrow f] \mid \psi \in \phi, \forall p, v_2.f(v_2, p) \in \text{as}(v_2/\text{id}_2)(p, \boxempty, \phi), \text{id}_1 \notin \text{dom}(\psi) \} \\
\text{assume } \text{args } e(\phi, \rho) &= (\{ \psi \mid \psi \in \phi, \forall \text{args}. \text{true} \in \text{as}()()\}, \rho) \\
\text{refine } \text{ids } \text{ds}(\phi, \rho) &= \text{ds}(\text{ids } \text{ds}(\phi, \rho)) \\
\text{refine } \text{ids } \text{ds}(\phi, \rho) &= \text{ds}(\phi, \rho) \\
\text{r, spec} &= \text{spec}(\text{r}(\phi, \rho))
\end{align*} \]

Figure 18: Semantics of declarations