Correct by Construction Networks using Stepwise Refinement

Leonid Ryzhyk* — *VMware Research*
Nikolaj Bjørner — *Microsoft Research*
Marco Canini — *KAUST*
Jean-Baptiste Jeannin — *Samsung Research America*
Cole Schlesinger* — *Barefoot Networks*
Douglas B. Terry* — *Amazon*
George Varghese — *UCLA*

Abstract

Building software-defined network controllers is an exercise in software development and, as such, likely to introduce bugs. We present Cocoon, 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 Cocoon 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. Cocoon 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 Cocoon 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.

*Work performed at Samsung Research America.

Data-plane verification has risen in popularity with SDNs. As the controller generates new forwarding configurations, tools like Header Space Analysis (HSA) and Veriflow [16, 17] 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 [35].

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 [27]. 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 Cocoon, for Correct by Construction Networking, consists of an SDN programming language, a verifier for this language, and a compiler from the language to data-plane languages: NetKAT [2] and P4 [4].

Cocoon 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 is delivered to the destination host if and only if its sender 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."

Cocoon 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 [19]. Bugs are immediately detected and localized to the step in which they are introduced. The refinement relation is transitive, and so Cocoon guarantees that the lowest-level implementation refines the highest-level specification.

Second, Cocoon 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 Cocoon compiler generates a new data plane configuration.

To statically verify the design without knowing the exact configuration, Cocoon 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, Cocoon checks that RDF definitions meet their assumptions. This

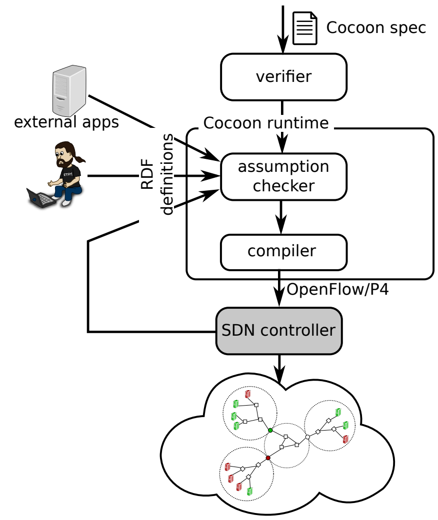

Figure 1: Cocoon architecture.

separation minimizes real-time verification cost: most of the effort has been done up-front at design time.

Hence, Cocoon 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.

Contributions. The main contribution of this paper is a new network design and verification methodology based on stepwise refinement and separation of static and dynamic behavior, and the Cocoon language and runtime, which support this methodology. Cocoon is a language carefully designed to be both amenable to stepwise refinement-based verification and also able to capture a wide variety of networking behavior. Its design enables:

- Writing complex specifications easily by phrasing them as high-level network implementations.
- Faster verification of functional correctness, with stepwise refinement naturally helping to localize the source of errors.
- Natural composition with other verification tools, like HSA [16], NetPlumber [15] and Veriflow [17], improving the speed at which they can verify network properties.

We evaluate Cocoon by using it to design and verify six realistic network architectures. Our performance evaluation demonstrates that Cocoon 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 Cocoon by example

In this section, we introduce features of Cocoon by implementing and verifying a variant of the enterprise

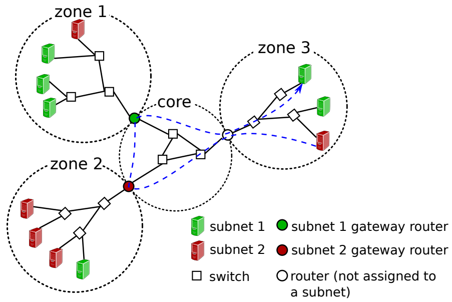

Figure 2: Example enterprise network.

network design described by Sung *et al.* [32], 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, which can also be assigned to implement access control for a subnet: 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.

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). Our refinement strategy, illustrated in Figure 3, follows the hierarchical struture of the network: the first refinement (Figure 3b) splits the network into operational zones connected via the core network, and distributes access control checks across gateway routers. The second and third refinements detail L2 switching inside zones and the core respectively (Figure 3c and d). We formalize these refinements in the Cocoon language, introducing language features along the way. Figure 4 shows the high-level specification that matches Figure 3a.

Roles The main building blocks of Cocoon 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,

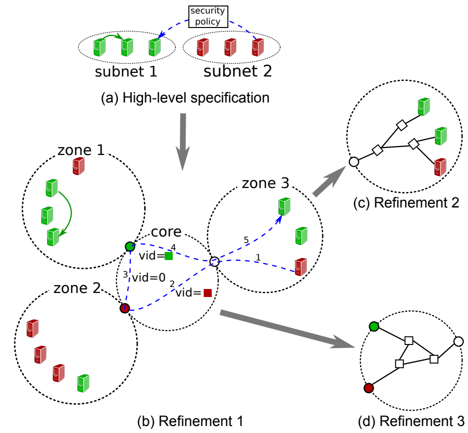

Figure 3: Refinement plan for the running example.

```
1 typedef uint<32> IP4
2 typedef uint<12> vid_t
3 typedef struct {
4   vid_t vid,
5   IP4 srcIP,
6   IP4 dstIP
7 } Packet
8
9 function cHost(IP4 addr): bool
10 function cSubnet(vid_t vid): bool
11 function acl(Packet p): bool
12 function ip2subnet(IP4 ip): vid_t
13 assume(IP4 addr) cHost(addr)=>cSubnet(ip2subnet(addr))
14 function sameSubnet(vid_t svid, vid_t dvid): bool =
15   svid == dvid;
16
17 role HostOut[IP4 addr] | cHost(addr) =
18   let vid_t svid = ip2subnet(pkt.srcIP);
19   let vid_t dvid = ip2subnet(pkt.dstIP);
20   filter addr == pkt.srcIP;
21   filter sameSubnet(svid, dvid) or acl(pkt);
22   filter cHost(pkt.dstIP);
23   send HostIn[pkt.dstIP]
24
25 role HostIn[IP4 addr] | cHost(addr) = filter false
```

Figure 4: High-level specification of the running example.

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. Cocoon's policy language is inspired by the Frenetic family of languages [12]: 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 out packets whose source IP does not match the IP address of the sending host (the `filter` operator drops packets that do not satisfy the filter condition) (line 21), and (2) drop packets sent across subnets that violate 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 Cocoon. 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. Cocoon'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 fields.

RDFs are a crucial part of Cocoon'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, Cocoon 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,

$\forall \mathtt{addr}.\mathtt{cHost}(\mathtt{addr}) \Rightarrow \mathtt{cSubnet}(\mathtt{ip2subnet}(\mathtt{addr}))$

Line 13 states this assumption in the Cocoon language.

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

Until a function is given a concrete definition, Cocoon 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, Cocoon statically verifies that the definition satisfies its assumptions. Cocoon 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 Cocoon specification of this refinement.

Line 55 in Figure 6 shows the refined specification of `HostOut`, which sends the packet directly to the destina-

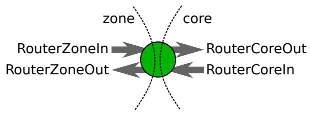

Figure 5: Router ports and corresponding roles.

tion only if it is on the same IP subnet and inside the same zone (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 Cocoon 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* connnected 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 Cocoon 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 (Cocoon currently models routers as switches), specified in terms or 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 Cocoon supports multicast forwarding using the `fork` construct. For example,

```
fork(uint<16> port|port>0 and port<n())
  send SwitchOut[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

```
havoc pkt.dstIP; assume pkt.dstIP != pkt.srcIP
```

the `havoc` statement non-deterministically 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 Cocoon specifications, the kinds of correctness guarantees that can be established through refinement-based verification, and the design of Cocoon verification tools. See Appendix A for a more formal presentation.

Semantics We start with assigning semantics to roles. Let $\mathtt{Pkt}$ be the set of all possible packets, and $\mathtt{Loc}$ be the set of *locations*, where each location identifies a unique role instance in a Cocoon specification. We define the set of *located packets* $\mathtt{LPkt} = \{(\mathtt{p},\mathtt{l}) \mid \mathtt{p} \in \mathtt{Pkt}, \mathtt{l} \in \mathtt{Loc}\}$.

We define semantics of a role $\mathtt{R}$ as a partial function $[\![\mathtt{R}]\!] : \mathtt{LPkt} \nrightarrow 2^{2^{\mathtt{LPkt}}}$ from a located packet to a set of sets of located packets. If Cocoon were deterministic, $[\![\mathtt{R}]\!]$ would just return the set of packets produced by role $\mathtt{R}$ on a given located packet. To model nondeterminism in the semantics, $[\![\mathtt{R}]\!]$ returns all possibilities of such sets of packets, thus forming a set of sets of packets.

We define refinement relation $\sqsubseteq$ over roles:

Definition 1 (Role refinement)**.** Role $\hat{\mathtt{R}}$ refines role $\mathtt{R}$ ($\hat{\mathtt{R}} \sqsubseteq \mathtt{R}$) iff $\hat{\mathtt{R}}$ and $\mathtt{R}$ have identical parameter lists and characteristic functions and

$$\forall \mathtt{p} \in \mathtt{Domain}([\![\mathtt{R}]\!]).[\![\hat{\mathtt{R}}]\!](\mathtt{p}) \subseteq [\![\mathtt{R}]\!](\mathtt{p}). \quad (1)$$

A Cocoon 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 $\mathtt{R}$ and a set of roles $\{\mathtt{P}_1 \ldots \mathtt{P}_k\}$ and recursively inlines the implementation of $\mathtt{P}_i$ in $\mathtt{R}$

```
refine HostOut {
  typedef uint<16> zid_t
  function cZone(zid_t zid): bool
  function zone(IP4 addr): zid_t
  assume (IP4 addr) cHost(addr) => cZone(zone(addr))
  function gwZone(vid_t vid): zid_t
  assume (vid_t vid) cSubnet(vid) => cZone(gwZone(vid))
  function aclSrc(Packet p): bool
  function aclDst(Packet p): bool
  assume (Packet p) acl(p) == (aclSrc(p) and aclDst(p))
  assume (vid_t vid) cSubnet(vid) => (vid != 0)

  role RouterZoneIn[zid_t zid] | cZone(zid) =
    let vid_t dvid = ip2subnet(pkt.dstIP);
    let vid_t svid = pkt.vid;
    filter cSubnet(dvid);
    if dvid != svid and gwZone(svid) == zid then {
      pkt.vid := 0;
      filter aclSrc(pkt)
    };
    send RouterCoreOut[zid]

  role RouterZoneOut[zid_t zid] | cZone(zid) =
    filter cHost(pkt.dstIP) and zone(pkt.dstIP) == zid;
    pkt.vid := 0;
    send HostIn[pkt.dstIP]

  role RouterCoreIn[zid_t zid] | cZone(zid) =
    let vid_t dvid = ip2subnet(pkt.dstIP);
    if pkt.vid == 0 then {
      filter aclDst(pkt);
      pkt.vid := dvid;
      if zone(pkt.dstIP) == zid then
        send RouterZoneOut[zid]
      else
        send RouterCoreOut[zid]
    } else if pkt.vid == dvid then {
      send RouterZoneOut[zid]
    } else {
      let vid_t svid = pkt.vid;
      pkt.vid := 0;
      filter aclSrc(pkt);
      send RouterCoreOut[zid]
    }
  role RouterCoreOut[zid_t zid]|cZone(zid)=
    if pkt.vid == 0 then {
      filter cSubnet(ip2subnet(pkt.dstIP));
      send RouterCoreIn[gwZone(ip2subnet(pkt.dstIP))]
    } else if pkt.vid != ip2subnet(pkt.dstIP) then {
      send RouterCoreIn[gwZone(ip2subnet(pkt.srcIP))]
    } else {
      filter cZone(zone(pkt.dstIP));
      send RouterCoreIn[zone(pkt.dstIP)]
    }
  role HostOut[IP4 addr] | cHost(addr) =
    let vid_t svid = ip2subnet(pkt.srcIP);
    let vid_t dvid = ip2subnet(pkt.dstIP);
    filter addr == pkt.srcIP;
    filter pkt.vid == 0;
    if svid==dvid and zone(pkt.dstIP)==zone(addr)then
    { filter cHost(pkt.dstIP);
      send HostIn[pkt.dstIP]
    } else {
      pkt.vid := ip2subnet(addr);
      send RouterZoneIn[zone(addr)]
    }
}
```

Figure 6: Refinement 1.

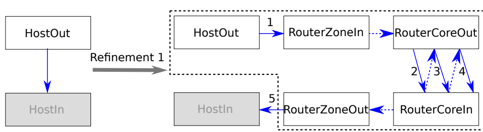

Figure 7: Refinement 1: the `HostOut` role is refined by introducing four router roles. The path from `HostOut` to `HostIn` is decomposed into up to 5 segments.

```
refine RouterCoreOut {
  role RouterCoreOut[zid_t zid] |cZone(zid) =
      send CoreSwitchIn[zid]
  role CoreSwitchIn[uint<16> port] | cZone(port) =
    if pkt.vid == 0 then {
      filter cSubnet(ip2subnet(pkt.dstIP));
      send CoreSwitchOut[gwZone(ip2subnet(pkt.dstIP))]
    } else if pkt.vid != ip2subnet(pkt.dstIP) then {
      send CoreSwitchOut[gwZone(ip2subnet(pkt.srcIP))]
    } else {
      filter cZone(zone(pkt.dstIP));
      send CoreSwitchOut[zone(pkt.dstIP)]
    }
  role CoreSwitchOut[uint<16> port] | cZone(port) =
    send RouterCoreIn[port]
}
```

Figure 8: Refinement 2.

whenever $\mathtt{R}$ sends to $\mathtt{P_i}$. Consider the refinement in Figure 7. When verifying this refinement, we would like to prove that the refined `HostOut` role combined with the newly introduced router roles is equivalent to the original `HostOut` role on the left. This combined role, indicated with the dashed line in Figure 7, is computed as $\mathtt{inline}(\mathtt{HostOut}, \{\mathtt{RouterZoneIn}, \ldots\})$.

We extend the refinement relation to specifications:

Definition 2 (Specification refinement)**.** Let $\mathtt{S} = \{\mathtt{R}_1, \ldots, \mathtt{R}_n\}$ and $\hat{\mathtt{S}} = \{\mathtt{R}'_1, \ldots, \mathtt{R}'_n, \mathtt{P}_1, \ldots, \mathtt{P}_k\}$ be two specifications, such that roles $\mathtt{R_i}$ and $\mathtt{R'_i}$ have identical names, parameter lists and characteristic functions. $\hat{\mathtt{S}}$ refines $\mathtt{S}$ ($\hat{\mathtt{S}} \sqsubseteq \mathtt{S}$) iff $\forall \mathtt{i} \in [1..n].\ \mathtt{inline}(\mathtt{R}'_\mathtt{i}, \{\mathtt{P}_1, \ldots, \mathtt{P}_k\}) \sqsubseteq \mathtt{R_i}$.

```
host Host[IP4 addr]((HostIn, HostOut))
switch ZoneRouter[zid_t zid](
    (RouterZoneIn,RouterZoneOut),
    (RouterCoreIn,RouterCoreOut))
switch CoreSwitch[]((CoreSwitchIn, CoreSwitchOut))
```

Figure 9: Declaring physical network elements: hosts and switches.

The following proposition is the foundation of Cocoon's compositional verification procedure:

Proposition 1. The $\sqsubseteq$ relation is transitive.

Hence, we can prove that the final specification is a refinement of the top-level specification by proving stepwise refinements within a chain of intermediate specifications.

We encode the problem of checking the refinement relation between roles into a model checking problem and use the Corral model checker [19] to solve it. We chose Corral over other state-of-the-art model checkers due to its expressive input language, called Boogie [20], which enables a straightforward encoding of Cocoon specifications. Given roles $\mathtt{R}$ and $\hat{\mathtt{R}}$, we would like to check property (1) or, equivalently, $\neg(\exists \mathtt{p}, \mathtt{p}'.\mathtt{p}' \in [\![\hat{\mathtt{R}}]\!](\mathtt{p}) \wedge \mathtt{p}' \notin [\![\mathtt{R}]\!](\mathtt{p}))$ (to simplify presentation, we assume that roles are unicast, i.e., output exactly one packet). We encode this property as a Boogie program:

$$\mathtt{p}' := \mathtt{proc}\hat{\mathtt{R}}(\mathtt{p});\ \mathtt{assert}(\mathtt{procR}(\mathtt{p}, \mathtt{p}'))$$

Here, $\mathtt{proc}\hat{\mathtt{R}}(\mathtt{p})$ is a Boogie procedure that takes a located packet $\mathtt{p}$ and non-deterministically returns one of possible outputs of $\hat{R}$ on this packet; $\texttt{procR}(p,p')$ returns true iff $p' \in R(p)$. We use Boogie's havoc construct to encode nondeterminism. We encode Cocoon assumptions as Boogie axioms, and characteristic functions of roles as procedure preconditions [20]. Violation of property (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 bounding 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 counter incremented by every send 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 forwarding loops.

Verifying path properties Cocoon's refinement-based verification operates on a single role at a time and, after the initial refinement, does not consider global forwarding behavior of the network. Importantly, however, it guarantees that all such behaviors are preserved by refinements, specifically, a valid refinement can only modify a network path by introducing 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 data plane verification. Consider, for example, the problem of checking pairwise reachability between all end hosts. Cocoon guarantees that this property holds for the network implementation if and only if it holds for its high-level specification. Often, 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. In Section 6.3, we show that such verification can be performed much more efficiently than checking an equivalent property directly on the detailed low-level implementation.

Limitations Because Cocoon specifications describe how individual packets are forwarded, it cannot verify properties related to multiple packets such as stateful network behaviors induced by say stateful firewalls. This limitation is shared by virtually all current network verification tools, which verify data plane *snapshots*.

However, stateful networks *can* be built on top of Cocoon 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. Cocoon can enforce statically defined invariants 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 Cocoon's dynamic assumption checker encodes all function definitions and assumptions into an SMT formula and uses the Z3 SMT solver [7] to check the validity of this formula.

4 Compiler

The Cocoon compiler proactively compiles specifications 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 intermediate 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 finding all parameter assignments satisfying the characteristic function of the role with the help of an SMT solver. During the second phase, we specialize the implementation 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 edges 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 incoming 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 Cocoon 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 Cocoon compiler translates network configuration updates into updates to switch flow tables. Recompiling the entire data plane on every reconfiguration is both inefficient and unnecessary, since most updates only affect a fraction of switches. For instance, a change in the network security policy related to a specific subnet in our running example requires reconfiguring the router assigned to this subnet only. While our current

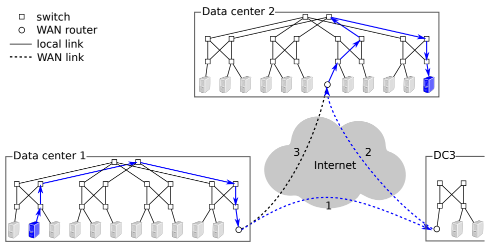

Figure 10: Case study 1: Software-defined WAN. Arrows show an example path between end hosts in different sites.

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 Cocoon. 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 [14] 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. Cocoon 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 Cocoon 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]
```

We refine the specification following the natural hierarchical structure of the network: we first decompose the WAN into datacenters (Refinement 1); these are further decomposed into pods (Refinement 2), which are in turn decomposed into three layers of switches (Refinements 3 and 4).

In more detail, Refinement 1 defines global routing

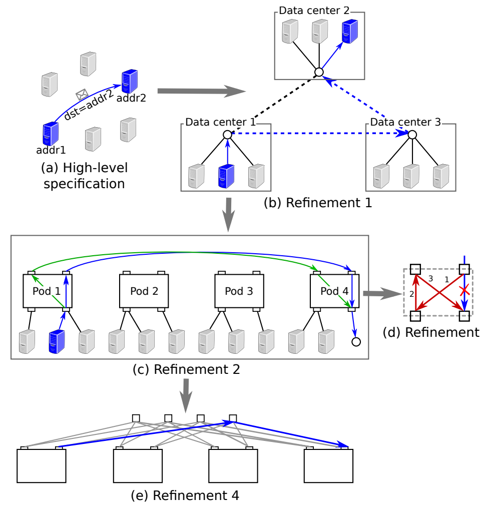

Figure 11: Refinement strategy for case study 1.

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 an assumption that tunnels returned by the `tunnel()` function deliver packets to the destination in `k` hops or less:

```
function distance(dcid_t src,dcid_t dst,tid_t tid):u8=
  case {
    src == dst: 8'd0;
    default: distance(nexthop(tid,src),dst,tid) + 1;}
assume(dcid_t src, dcid_t dst, Packet p)
  cDC(src) and cDC(dst)
  => distance(src,dst,tunnel(p)) <= k()
```

where `k` is a user-defined bound on the length of a tunnel and `cDC()` is the characteristic function of the set of datacenters.

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 decomposes pods into two layers of switches. A bottom-layer

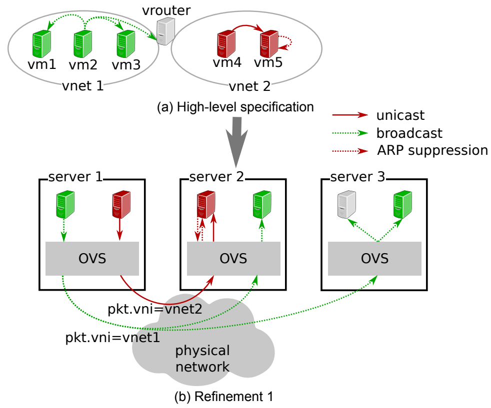

Figure 12: Refinement strategy for case study 2.

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 [18]. 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) [26], 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 [18], 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 Cocoon. Below we present two example extensions and show how Cocoon 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):

```
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) =
  ...
  (*send VHostIn[mac2VPort(vnet, pkt.dstMAC)]*)
  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 Broadcast packets must be delivered to all VMs on the virtual network:

```
role VHostOut[VPortId vport] | cVPort(vport) =
  ...
  if pkt.dstMAC == hffffffffffff(*bcast address*) then
    fork (VPortId vport | vPortVNet(vport) == vnet)
      send VHostIn[vhport.vhost, vhport.vport]
```

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, Cocoon 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 [32]. 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 [21]—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 [13]. 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.

5.4 Experience with the Cocoon language

We briefly report on our experience with the Cocoon language. We found the language to be expressive, as witnessed by the wide range of network designs covered by our case studies, concise (see Section 6), and effective at capturing the modularity inherent in any non-trivial network. Thus, the language achieves its primary design goal of enabling refinement-based design and verification of a large class of networks. At the same time, we found that by far the hardest part of programming in Cocoon is defining assumptions on RDFs necessary for static verification and runtime checking. Writing and debugging assumptions, such as the one shown in Section 5.1, may be challenging for engineers who are not accustomed to working with formal logic. Therefore, in our ongoing work we explore higher-level language constructs that will offer a more programmer-friendly way to specify assumptions.

6 Implementation and evaluation

We implemented Cocoon in 4,700 lines of Haskell. We implemented, verified, and tested the six case studies described in Section 5 using the Mininet network emulator. Cocoon, along with all case studies, is available under the open source Apache 2.0 license [6].

case study	LOC		#refines	verification time (s)	
	total	high-level		compositional	monolithic
WAN	305	18	6	10	>3600
virtualization	678	97	1	6	6
enterprise	342	50	4	16	>3600
F10	262	52	2	19	57
sTag	283	47	1	2	2
iSDX	190	21	2	3	3

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

All experiments in this section were performed on a machine with a 2.6 GHz processor and 128 GB of RAM. We measured single-threaded performance.

Table 1 summarizes our case studies, showing (1) total lines of Cocoon 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 Cocoon 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.

6.1 Static verification

The results show that Cocoon verifies the static design of complex networks in a matter of seconds with compositional verification being much faster than monolithic verification:[1] a refinement that focuses on a single role has an exponentially smaller state space than the complete specification and is potentially exponentially faster to verify. As we move through the refinement hierarchy, the low-level details of network behavior are partitioned across multiple roles and can be efficiently verified in isolation, while compositionality of refinements guarantees end-to-end correctness of the Cocoon program.

Bug finding We choose 3 (of many) examples from the case studies to show how Cocoon 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 re-

[1] The virtualization and sTag case studies only had one refinement; hence compositional and monolithic verification are equivalent in these examples.

Scale	Hosts	Switches	NetKAT Policy	Flowtable Rules
2	8	11	1,559	830
5	17	23	5,149	3,299
15	47	63	46,094	31,462
25	77	103	151,014	89,216
40	122	163	496,268	212,925

Table 2: Number of hosts, switches, size of NetKAT policy and flowtable rules as a function of network scale.

sponse 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.

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 eliminated these false positives by reformulating some of our assumptions, namely, breaking boolean equivalences into pairs of implications.

6.2 Cocoon vs. NetKAT

The NetKAT decision procedure for program equivalence [11] is the closest alternative to refinement verification. We compare Cocoon against NetKAT on a parameterized model of the enterprise network case study [32]—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 Cocoon, 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 Cocoon 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.

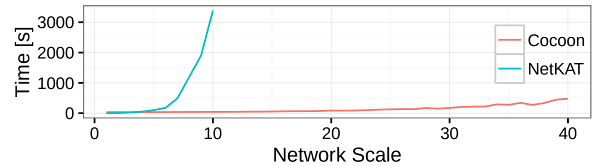

Figure 13: Comparison between Cocoon refinement verification vs. equivalence decision in NetKAT.

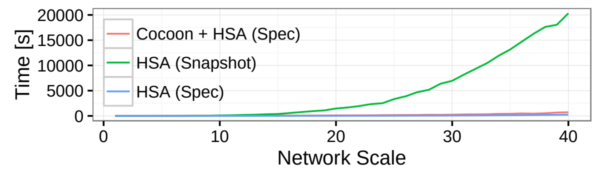

Figure 14: Comparison of high-level specification verification via HSA and Cocoon verification vs. snapshot data-plane verification via HSA.

Figure 13 shows the verification run time in seconds as we increase the network scale. Cocoon verification scales beyond that of NetKAT. Cocoon 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 Cocoon + 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, Cocoon 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 Cocoon specification, taking advantage of the fact that such properties are preserved by refinement.

We evaluate this using the Header Space Analysis (HSA) [16] network verifier. Using the same network scenarios as above, we use Frenetic to compile the NetKAT policy (translated from Cocoon 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 Cocoon's refinement verification results in dramatic improvement in verification performance, so that the cost of Cocoon 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 [25] is a data-plane configuration language: Controller frameworks like OpenDaylight [24], Floodlight [9], and Ryu [28] emit OpenFlow commands to update SDN-capable switches. The Frenetic family [2, 10, 22, 29] introduces modular language design; they allow writing controller applications in a general purpose language and compiling to OpenFlow. VeriCon [3], FlowLog [23], and Maple [33] eschew the tiered structure entirely using custom languages that describe network behavior over time.

Cocoon is a data-plane configuration language that inherits its sequential and parallel composition operators from NetKAT [2] while adding roles, refinements, RDFs, and assumptions. These features enable modular verification in addition to the modular program composition of NetKAT and other Frenetic languages.

Data-plane verification SDN verification takes two forms: data-plane verification and controller verification. The former checks that a given set of safety properties (e.g, no black holes or forwarding loops) hold on a given data-plane configuration [15–17]. 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 [27]. SymNet [31] is a network modeling language to perform efficient static analysis via symbolic execution of a range of network devices from switches to middleboxes.

Cocoon does not verify network properties directly. Rather, it guarantees that refinements are functionally equivalent, provided dynamically checked assumptions holds 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 Cocoon 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 Cocoon provides for stepwise refinement, including parameterized roles, in part because NetKAT is intended as a synthesis target emitted by the Frenetic controller. Cocoon refinements, on the other hand, are human readable, even at scale.

Controller verification VeriCon [3] and FlowLog [23] 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. NICE [5] uses model checking and symbolic execution to find bugs in Python controller programs but focuses on testing rather than verification.

In contrast, Cocoon 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 [8] and Wirth [34]. In the networking domain, Shankar and Lam [30] proposed an approach to network protocol design via stepwise refinement. 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.

Cocoon 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.

Although we apply our techniques to SDNs, we expect them to be equally applicable to traditional networks. In particular, stepwise refinement may help find bugs in potentially complex interactions between mechanisms such as VLANs and ACLs, and check that forwarding state matches the assumptions made in the specifications.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[3] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *PLDI*, 2014.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.

[5] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[6] Cocoon website. `https://github.com/ryzhyk/cocoon`.

[7] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS/ETAPS*, 2008.

[8] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3), Sept. 1968.

[9] Floodlight. `http://www.projectfloodlight.org/`.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[11] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A Coalgebraic Decision Procedure for NetKAT. In *POPL*, 2015.

[12] Frenetic. `http://frenetic-lang.org/`.

[13] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever. An Industrial-scale Software Defined Internet Exchange Point. In *NSDI*, 2016.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.

[16] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[17] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[18] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[19] A. Lal, S. Qadeer, and S. K. Lahiri. A Solver for Reachability Modulo Theories. In *CAV*, 2012.

[20] K. R. M. Leino. This is Boogie 2, June 2008. Manuscript KRML 178 `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[21] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese. Automatically verifying reachability and well-formedness in P4 Networks. Technical Report MSR-TR-2016-65, Microsoft Research, Sept. 2016.

[22] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *POPL*, 2012.

[23] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, 2014.

[24] OpenDaylight. `https://www.opendaylight.org/`.

[25] The OpenFlow protocol. `https://www.opennetworking.org/sdn-resources/openflow`.

[26] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[27] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling Network Verification Using Symmetry and Surgery. In *POPL*, 2016.

[28] Ryu. `https://osrg.github.io/ryu/`.

[29] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From Policies to Pipelines. In *ICFP*, 2014.

[30] A. U. Shankar and S. S. Lam. Construction of Network Protocols by Stepwise Refinement. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, 1990.

[31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: scalable symbolic execution for modern networks. In *SIGCOMM*, 2016.

[32] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards Systematic Design of Enterprise Networks. In *CoNEXT*, 2008.

[33] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.

[34] N. Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4), Apr. 1971.

[35] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering Why-Not Queries in Software-Defined Networks with Negative Provenance. In *HotNets*, 2013.

A Syntax and semantics of Cocoon

A.1 Syntax

The syntax of the Cocoon system is given in Fig. 15. Let `Id` be the set of identifiers, `Pkt` the set of packets and `Val` the set of values.

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

Statements allow filtering, which stops the computation if e does not evaluate to `true`. Assumptions are slightly different in that they are not executable, but can be refined only if e evaluates to `true`. Packet fields can be assigned explicitly with the $:=$ construct, or assigned

$$
\begin{array}{lrcl}
\text{integers} & n & & \\
\text{identifier} & id & & \\
 & ids & = & id \mid id, ids \\
\text{arguments} & args & = & \tau\ id \mid \tau\ id, args \\
\text{case body} & cbod & = & \cdot \mid e_1 : e_2; cbod \\
\text{value} & v & = & \texttt{true} \mid \texttt{false} \mid n \mid id\{v\} \\
 & & & (v,\dots,v) \\
\text{expression} & e & = & \texttt{not}\ e \mid e_1 \otimes e_2 \mid e\,\texttt{.}\,id \\
 & & & \mid id\,\texttt{(}e\texttt{)} \mid id\,\texttt{!}\,\texttt{(}e\texttt{)} \mid id\{e\} \\
 & & & \mid \texttt{pkt} \mid id \\
 & & & \mid \texttt{case}\ \{cbod\texttt{;}\ \texttt{default:}e\texttt{;}\} \\
 & & & \mid \texttt{(}e,\dots,e\texttt{)} \\
\text{type specs} & \tau & = & \texttt{uint <}n\texttt{>} \mid \texttt{bool} \mid id \mid \texttt{[}\tau\texttt{;}n\texttt{]} \\
 & & & \mid \texttt{struct}\ \{args\} \\
\text{statement} & a & = & \texttt{filter}\,e \mid \texttt{assume}\ e \\
 & & & \mid \texttt{pkt.}id\texttt{:=}e \mid \texttt{havoc pkt.}id \\
 & & & \mid \texttt{if}\ e\ \texttt{then}\ a_1 \\
 & & & \mid \texttt{if}\ e\ \texttt{then}\ a_1\ \texttt{else}\ a_2 \\
 & & & \mid \texttt{let}\ \tau\ id\ \texttt{=}\ e \\
 & & & \mid \texttt{send}\ id\texttt{[}e\texttt{]} \\
 & & & \mid \texttt{fork}\ \texttt{(}args\ \texttt{|}\ e\texttt{)}\ as \\
 & as & = & f \mid a, as \\
\text{role constraints} & cs & = & \cdot \mid \texttt{|}e \mid \texttt{/}e \mid \texttt{|}e\texttt{/}e \\
\text{declaration} & d & = & \texttt{typedef}\,\tau\,id \\
 & & & \mid \texttt{function}\,id\,\texttt{(}args\texttt{)}\ \texttt{:}\,\tau \\
 & & & \mid \texttt{function}\,id\,\texttt{(}args\texttt{)}\ \texttt{:}\,\tau = e \\
 & & & \mid \texttt{role}\ id\texttt{[}args\texttt{]}\ cs\ \ \texttt{=}\ as \\
 & & & \mid \texttt{assume}\ \texttt{(}args\texttt{)}\ e \\
 & ds & = & d \mid d, ds \\
\text{refinement} & r & = & \texttt{refine}\,ids\,ds \\
\text{spec} & spec & = & r \mid r, spec
\end{array}
$$

Figure 15: Cocoon Syntax.

to a nondeterministic value using `havoc`. Statements allow standard conditionals and let-bindings. Finally, a statement can send a packet to another role $id[e]$, or fork to multicast a packet across all variables *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 ($\mid e$) and on the incoming packets ($/\,e$) 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.

A.2 Semantics

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

- a packet $p \in \texttt{Pkt}$, a record of type `Pkt`;
- a local environment $\sigma \in (\texttt{Id} \rightharpoonup \texttt{Val})$, a partial function from identifiers to values, comprising `let`-defined variables; let `Env` be the set of local environments;
- a set of possible environments of functions ϕ. 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 ϕ is a set of such possible environments, representing all the possible function definitions. If Φ represents the set of all the sets of possible environments of functions, we thus have
$$\Phi = \mathscr{P}(\texttt{Id} \rightharpoonup (\texttt{Val} \times \texttt{Pkt}) \to \texttt{Val})$$
This enables the modelling of the nondeterminism introduced by functions defined only as signatures, and possibly restrained by assumptions;
- an environment of roles ρ, a partial function from identifiers to role semantics, where each role semantics is a (mathematical) function from a pair (v, p) to a set of sets of pairs (p, σ). Each set of pairs (p, σ) represents one possible execution, possibly returning multiple packets in the case of multicasting. We model nondeterminism by having the semantics return a set of those possible executions. Thus sets of sets enable the modeling of both nondeterminism and multicasting. If P represents the possible environments of roles, we thus have
$$P = (\texttt{Id} \rightharpoonup \texttt{Val} \times \texttt{Pkt} \to \mathscr{P}(\mathscr{P}(\texttt{Pkt} \times \texttt{Env})))$$

A.2.1 Semantics of expressions

The semantics of expressions is given in Figure 16, in terms of a triple (p, σ, ϕ). 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 ϕ, 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, σ) to model both nondeterminism and multicasting.

$$
\begin{aligned}
[\![e]\!](p,\sigma,\phi) &\in \mathscr{P}(\texttt{Val}) \\
[\![v]\!](p,\sigma,\phi) &= \{v\} \\
[\![\texttt{not}\ \ e]\!](p,\sigma,\phi) &= \{\neg v \mid v \in [\![e]\!]\} \\
[\![e_1 \otimes e_2]\!](p,\sigma,\phi) &= \{v_1 \otimes v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\
[\![e.id]\!](p,\sigma,\phi) &= \{v.id \mid v \in [\![e]\!]\} \\
[\![id(e)]\!](p,\sigma,\phi) &= \{f(id)(v,p,\phi) \mid v \in [\![e]\!], f \in \phi\} \\
[\![id!(e)]\!](p,\sigma,\phi) &= \{[\![id!]\!](v) \mid v \in [\![e]\!]\} \\
&\quad \text{where } [\![id!]\!] \in \texttt{Val} \to \texttt{Val} \\
[\![id\{e\}]\!](p,\sigma,\phi) &= \{id\{v\} \mid v \in [\![e]\!]\} \\
[\![\texttt{pkt}]\!](p,\sigma,\phi) &= \{p\} \\
[\![id]\!](p,\sigma,\phi) &= \sigma(id) \\
[\![\texttt{case}\ \ \{\cdot\texttt{;default:}e\texttt{;}\}]\!](p,\sigma,\phi) &= [\![e]\!](p,\sigma,\phi) \\
[\![\texttt{case}\ \ \{e_1 : e_2\texttt{;cbod;}\ \}]\!](p,\sigma,\phi) &= \\
\{v_2 \mid (\texttt{true}, v_2) \in [\![(e_1,e_2)]\!](p,\sigma,\phi)\} &\cup \\
\{v_2 \mid \texttt{false} \in [\![e_1]\!](p,\sigma,\phi), v_2 &\in [\![\texttt{case}\ \ \{\texttt{cbod;}\}]\!]\} \\
[\![(e_1,\ldots,e_n)]\!](p,\sigma,\phi) &= \{(v_1,\ldots,v_n) \mid v_i \in [\![e_i]\!]\}
\end{aligned}
$$

Figure 16: Semantics of expressions

The semantics of `filter` and `assume` only differ when $\{\{(p,\sigma) \mid \texttt{true} \in [\![e]\!](p,\sigma,\phi)\}\} = \{\varnothing\}$. In that case `filter` drops all packets (its semantics is $\{\varnothing\}$), whereas `assume` disallows refinements by denoting $\varnothing$. 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 multicasts 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 ϕ and roles ρ. Constraints $\mid e$ and $/\ e$ 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 prior declaration existed (first line), otherwise an explicit definition is added (second line). Assumptions select the definitions of functions in ϕ that agree with the assumption that is being considered.

$$
\begin{aligned}
[\![a]\!](p,\sigma,\phi,\rho) &\in \mathscr{P}(\mathscr{P}(\texttt{Pkt}\times\texttt{Env}))\\
[\![\texttt{filter}\ e]\!](p,\sigma,\phi,\rho) &= \{\{(p,\sigma)\mid \texttt{true}\in[\![e]\!](p,\sigma,\phi)\}\} && \text{can be } \{\varnothing\} \text{ but not } \varnothing\\
[\![\texttt{assume}\ e]\!](p,\sigma,\phi,\rho) &= \{\{(p,\sigma)\mid \texttt{true}\in[\![e]\!](p,\sigma,\phi)\}\} && \text{only if} \neq \{\varnothing\}\\
[\![\texttt{assume}\ e]\!](p,\sigma,\phi,\rho) &= \varnothing && \text{otherwise}\\
[\![\texttt{pkt}.id := e]\!](p,\sigma,\phi,\rho) &= \{\{(p[id\mapsto v],\sigma)\}\mid v\in[\![e]\!](p,\sigma,\phi)\}\\
[\![\texttt{havoc pkt}.id]\!](p,\sigma,\phi,\rho) &= \{\{(p[id\mapsto v],\sigma)\}\mid v\in\texttt{Val}\}\\
[\![\texttt{if}\ e\ \texttt{then}\ a_1]\!](p,\sigma,\phi,\rho) &= \{(p',\sigma')\in[\![a_1]\!](p,\sigma,\phi,\rho)\mid \texttt{true}\in[\![e]\!](p,\sigma,\phi)\}\cup\\
&\quad\ \{(p,\sigma)\mid \texttt{false}\in[\![e]\!](p,\sigma,\phi)\}\\
[\![\texttt{if}\ e\ \texttt{then}\ a_1\ \texttt{else}\ a_2]\!](p,\sigma,\phi,\rho) &= \{(p',\sigma')\in[\![a_1]\!](p,\sigma,\phi,\rho)\mid \texttt{true}\in[\![e]\!](p,\sigma,\phi)\}\cup\\
&\quad\ \{(p',\sigma')\in[\![a_2]\!](p,\sigma,\phi,\rho)\mid \texttt{false}\in[\![e]\!](p,\sigma,\phi)\}\\
[\![\texttt{let}\ id = e]\!](p,\sigma,\phi,\rho) &= \{\{(p,\sigma[id\mapsto v])\}\mid v\in[\![e]\!](p,\sigma,\phi)\}\\
[\![\texttt{send}\ id[e]]\!](p,\sigma,\phi,\rho) &= \bigcup\{\rho(id)(v,p)\mid v\in[\![e]\!](p,\sigma,\phi)\}\\
[\![\texttt{fork}(id\mid e)\ as]\!](p,\sigma,\phi,\rho) &= \bigotimes\{[\![as]\!](p,\sigma[id\mapsto v],\phi,\rho)\mid \texttt{true}\in[\![e]\!](p,\sigma[id\mapsto v],\phi), v\in\texttt{Val}\}\\
\text{where } \bigotimes\{A_1,\ldots,A_n\} &= \{a_1\cup\ldots\cup a_n\mid a_1\in A_1,\ldots,a_n\in A_n\}, a_i\in\mathscr{P}(\texttt{Pkt}\times\texttt{Env})
\end{aligned}
$$

$$
\begin{aligned}
[\![as]\!](p,\sigma,\phi,\rho) &\in \mathscr{P}(\mathscr{P}(\texttt{Pkt}\times\texttt{Env}))\\
[\![a,as]\!](p,\sigma,\phi,\rho) &= \bigcup\left\{\bigotimes\left\{[\![as]\!](p',\sigma',\phi,\rho)\mid (p',\sigma')\in A\right\}\mid A\in[\![a]\!](p,\sigma,\phi,\rho)\right\}
\end{aligned}
$$

Figure 17: Semantics of statements

$$
\begin{aligned}
[\![d]\!](\phi,\rho) &\in \Phi\times P\\
[\![\texttt{role}\ \ id_1\ [id_2]]\!] &= [\![\texttt{role}\ \ id_1\ [id_2]\mid \texttt{true}\ /\ \texttt{true}]\!]\\
[\![\texttt{role}\ \ id_1\ [id_2]\mid e]\!] &= [\![\texttt{role}\ \ id_1\ [id_2]\mid e\ /\ \texttt{true}]\!]\\
[\![\texttt{role}\ \ id_1\ [id_2]\ /\ e]\!] &= [\![\texttt{role}\ \ id_1\ [id_2]\mid \texttt{true}\ /\ e]\!]\\
[\![\texttt{role}\ \ id_1\ [id_2]\ \mid\ e_3\ /\ e_4 = as]\!](\phi,\rho) &= (\phi,\rho\,[id_1\mapsto r])\\
\text{where } r = \lambda(p,v_2).&\begin{cases}\varnothing \text{ if } \texttt{true}\notin[\![e_3[v_2/id_2]]\!](p,[\,],\phi)\\ \varnothing \text{ if } \texttt{true}\notin[\![e_4[v_2/id_2]]\!](p,[\,],\phi)\\ \varnothing \text{ if } id_1\in\mathsf{dom}(\rho) \text{ and } [\![as[v_2/id_2]]\!](p,[\,],\phi,\rho)\not\subseteq\rho(id_1)\\ [\![as[v_2/id_2]]\!](p,[\,],\phi,\rho) \text{ otherwise}\end{cases}\\
[\![\texttt{function}\ \ id_1(id_2)]\!](\phi,\rho) &= (\{\psi[id_1\mapsto f]\mid \psi\in\phi, f\in(\texttt{Val}\times\texttt{Pkt}\to\texttt{Val}), id_1\notin\mathsf{dom}(\psi)\},\rho)\\
[\![\texttt{function}\ \ id_1(id_2) = e]\!](\phi,\rho) &= (\{\psi\in\phi\mid \forall p,v_2.\psi(id_1)(v_2,p)\in[\![e[v_2/id_2]]\!](p,[\,],\phi), id_1\in\mathsf{dom}(\psi)\}\cup\\
&\quad\ \{\psi[id_1\mapsto f]\mid \psi\in\phi,\forall p,v_2.f(v_2,p)\in[\![e[v_2/id_2]]\!](p,[\,],\phi), id_1\notin\mathsf{dom}(\psi)\}\\
&\quad\ ,\rho)\\
[\![\texttt{assume}\ \ args\ e]\!](\phi,\rho) &= (\{\psi\mid\psi\in\phi,\forall args.\texttt{true}\in[\![e]\!](\{\ \},[args],\{\psi\})\},\rho)\\
\\
[\![d,ds]\!](\phi,\rho) &= [\![ds]\!]([\![d]\!](\phi,\rho))\\
\\
[\![\texttt{refine}\ \ ids\ ds]\!](\phi,\rho) &= [\![ds]\!](\phi,\rho)\\
\\
[\![r,spec]\!] &= [\![spec]\!]([\![r]\!](\phi,\rho))
\end{aligned}
$$

Figure 18: Semantics of declarations

The semantics of several declarations, refinements and finally whole specifications chains through the semantics of role declarations.