15-812 Report: Automated Verification of Safety Properties of Declarative Networking Programs

Lay Kuan Loh

Abstract

Networks are complex systems that are ridden with errors. Such errors can lead to disruption of services, which may have grave consequences. Verification of networks is key to eliminating errors and building robust networks. In this paper, we propose an approach to verify networks using declarative networking. In declarative networking, networks are specified in NDLog, a declarative language.

We focus on analyzing safety properties. We develop a technique to statically analyze NDlog programs. First, we build a dependency graph of the predicates of NDlog programs; then, we build a summary data structure called a constraint pool to represent all possible derivations and their associated constraints for predicates in the program; finally, properties specified in first-order logic are checked on the data structure with the help of the SMT solver Z3. We proved the correctness of our algorithm.

To evaluate our approach, we built a prototype tool, and showed the effectiveness of the tool in validating/debugging several SDN applications. We demonstrated that the tool can unveil different problems in the process of SDN application development, ranging from software bugs, incomplete topological constraints and incorrect property specification.

Contents

1 Introduction **2**

2 Overview **3**
- 2.1 Declarative Networking . . . 3
- 2.2 Analysis Overview . . . 3
- 2.3 Example Constraint Pool . . . 5

3 Analyzing Non-recursive Programs **6**
- 3.1 Derivation Pool Construction . . . 6
- 3.2 Property Query . . . 9
- 3.3 Network Constraints . . . 12
- 3.4 Correctness . . . 12

4 Extension to Recursive Programs **23**
- 4.1 Derivation Pool for Recursive Predicates . . . 23
- 4.2 Property Query . . . 24
- 4.3 Correctness . . . 25

5 Case Studies **30**
- 5.1 Verification process . . . 30
- 5.2 Ethernet Source Learning . . . 30
- 5.3 Firewall . . . 32
- 5.4 Load Balancing . . . 36
- 5.5 Address Resolution Protocol . . . 38
- 5.6 Discussion . . . 39

6 Related Work **44**

7 Conclusion **45**

1 Introduction

As more and more services are offered over the Internet, ensuring the security and stability of networks has become increasingly important. Unfortunately, networks are complex systems that are ridden with errors. Such errors can lead to disruption of services, which may have grave consequences. Verification of networks is key to eliminating errors and building robust networks.

Much work on network verification has focused on verifying topological-specific network configurations [23, 33, 18, 37]. Practical testing tools for finding undesired behavior in protocol implementation have also been proposed [25, 16]. With the emerging technology of software-defined networks (SDN), modeling networks as programmable software has gained unprecedented popularity. Researchers began to apply program verification techniques to the verification of SDNs [8, 9].

Our goal is to develop a general automated technique that can be applied to network verification. The first step towards that goal is to find the right abstraction for networks. This paper is based off joint work from [11].

Declarative networking [29] is one of the first research effort to demonstrate that high-level languages can be used to program networks. In declarative networking, network protocols are written in a declarative language NDLog, which is a distributed Datalog. Declarative networking techniques have been used in several domains including fault tolerance protocols [45], cloud computing [3], sensor networks [13], overlay network compositions [34], anonymity systems [44], mobile ad-hoc networks [36, 27], wireless channel selection [26], network configuration management [12], and forensic analysis [55, 53, 54]. An open-source declarative networking system called *RapidNet* [43] has been integrated with the ns-3 [39] simulator, so protocols can be tested. It has also been shown that network verification can be carried out using the declarative network framework [48, 47, 10]. In summary, NDLog is a great intermediary language for bridging the gap between network specification, verification, and implementation, so we use NDLog as our specification language for networks.

Unfortunately, all of the verification tools related to NDLog require manual proofs, which makes verification very labor intensive. What is worse is that when the proofs cannot be constructed, it is nontrivial to find out what went wrong.

Either there are bugs in the program, or the invariants used in the proofs are not correct. There is little tool support for identifying problems under these circumstances. In this paper, we develop an automated static analysis technique to analyze the safety properties of NDLog programs. When properties do not hold, our tool provides a concrete counterexample to further aid program debugging. The properties that we are interested in include invariants of the network and desirable behavior of nodes in the network. For instance, we would like to know if every forward entry corresponds to a route announcement packet, or if a successfully delivered packet indicates proper forwarding table setup in the switches that the packet traverses. One observation we have is that a large fragment of the interesting properties of networks can be expressed in a simple fragment of first-order logic. Leveraging this limited expressive power, we are able to develop static analysis for NDLog programs.

Our static analysis examines the structure of the NDLog program and builds a summary data structure for all derivations of that program. Properties specified in the restricted format of first-order logic are checked on the summary data structure with the help of the SMT solver Z3 [50]. The challenge is how to deal with recursive programs. For such programs, the number of possible derivations for recursive predicates is infinite. We use a concise representation for recursive predicates, so all possible derivations can be finitely represented. To evaluate our analysis, we built a prototype tool, and verified several safety properties of a number of SDN controller programs, where the SDN's controller program and switch logic are specified in NDLog.

This paper makes the following technical contributions.

- We developed algorithms for automatically analyzing a class of safety properties of NDLog programs.
- We proved the correctness of our algorithms.
- We implemented a prototype tool and verified a number of safety properties of SDN controller programs.

The rest of this paper is organized as follows. In Section 2, we review declarative networks and NDLog, and describe our analysis at a high-level. Then, we explain our algorithm for non-recursive programs in Section 3. Next, we extend the algorithm to handle recursive programs in Section 4. The case studies are described in Section 5. We discuss related work in Section 6 and then conclude.

2 Overview

We first review declarative networking and NDLog through examples. Then, we present an overview of our analysis.

2.1 Declarative Networking

Declarative networks are specified using *Network Datalog* (NDLog), which is a distributed recursive query language used for querying network graphs. Declarative queries are a natural and compact way to implement a variety of routing protocols and (overlay) networks. For example, traditional routing protocols such as path vector and distance-vector protocols can be expressed in a few lines of code [31], and the Chord distributed hash table in 47 lines of code [30]. When compiled and executed, these perform efficiently relative to imperative implementations.

NDLog is based on Datalog [42]. A Datalog program consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and ... and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants), or Boolean expressions that involve function symbols (including arithmetic) applied to attributes, which we call *constraints*.

Datalog rules can refer to one another in a mutually recursive fashion. Commas are interpreted as logical conjunctions. The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. The following example NDLog program computes full reachability between any pair of nodes. In the runtime, derived predicates are stored as tuples in database tables, so we use predicate and tuple interchangeably for the rest of this paper.

Reachable:
```
d1 reachable(@X,Y,C) :- link(@X,Y,C).
d2 reachable(@X,Y,C) :- link(@X,Z,C1),
                        reachable(@Z,Y,C2), C=C1+C2.
d3 reachable(@X,Y,C) :- reachable(@X,Z,C1),
                        link(@Z,Y,C2), C=C1+C2.
```

The program Reachable takes as input `link(@X,Y,C)` tuples, where each tuple corresponds to a copy of an entry in the neighbor table, and represents an edge from the node itself (`X`) to one of its neighbors (`Y`) of cost `C`. NDLog supports a *location specifier* in each predicate, expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, `link` tuples are stored based on the value of the `X` field. The program Reachable derives `reachable(@X,Y,C)` tuples, where each tuple represents the fact that `X` has a path to reach `Y` with cost `C`. Rule `d1` derives `reachable` tuples from direct links. Rule `d2` and `d3` compute transitive reachability: if there exists a link from `X` to `Z` with cost `C1`, and `Z` knows about a path to `Y` with cost `C2`, then transitively, `X` can reach `Y` with cost `C1+C2`. Rule `d3` is similar to `d2`

As our driving example, we will use the following non-recursive set of rules that compute one, two, and three hop reachability information within a network. Notice that there is an error in rule r2, where `onehop X Z C1` should be `onehop Z Y C1`. This program cannot derive three-hop paths.

ThreeHops:
```
r1 onehop(@X,Y,C) :- link(@X,Y,C).
r2 twohops(@X,Z,C) :- link(@X,Z,C1), onehop(@X,Z,C2),
                      C = C1+C2.
r3 threehops(@X,Y,C) :- onehop(@X,Z,C1),
                        twohops(@Z,Y,C2), C=C1+C2.
r4 threehops(@X,Y,C) :- twohops(@X,Z,C1),
                        onehop(@Z,Y,C2), C=C1+C2.
```

2.2 Analysis Overview

The static analysis mainly consists of two processes: a process that summarizes all derivations of predicates in an auxiliary data structure, which we call a *derivation pool*, and a process that queries properties on the derivation pool. NDLog programs are represented abstractly as dependency graphs. Recursive programs are more complicated than non-recursive programs, so we explain the algorithms for non-recursive programs first, before we discuss extensions to support recursive programs. The dependency graph and the properties to be checked are of the same form for both recursive and non-recursive programs. Next, we formally define the dependency graph and the format of the properties.

Dependency graph We build dependency graphs for NDLog programs. A dependency graph has two types of nodes, predicate nodes, denoted Np, and rule nodes, denoted Nr. Each predicate node corresponds to a tuple in the program. A predicate node consists of a unique ID for the node, the name of the predicate and its type, and a tag indicating whether the predicate is on a cycle in the graph. The tag cyc means that the node is on a cycle and ncyc means the opposite.

Each rule node corresponds to a rule in the program. A rule node consists of a unique ID, the head of the rule, the body of the rule, which is a list of predicates, and the constraints. The edges, denoted E, are directional. Each edge points either from a rule node to the predicate node which is the head of that rule node, or from a predicate node to a rule node where the predicate is in the rule body.

Predicate type	τ	$::= \mathsf{Pred} \mid \mathsf{bt} \supset \tau$
Dependency graph	$\mathcal{G}$	$::= (Np\ \mathsf{List}, Nr\ \mathsf{List}, E\ \mathsf{List})$
Predicate node	Np	$::= (nID, p : \tau, \mathsf{cyc}) \mid (nID, p : \tau, \mathsf{ncyc})$
Rule node	Nr	$::= (rID, hd, body, c)$
Edge	E	$::= (rID, nID) \mid (nID, rID)$
Rule head	hd	$::= p(\vec{x})$
Rule body	$body$	$::= p_1(\vec{x_1}), \cdots, p_n(\vec{x_n})$
Rule constraints	c	$::= e_1\ bop\ e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \exists x.c$

To make variable substitutions easier, each predicate takes unique variables as arguments. For instance, the following two NDLog rules are equivalent, but we use `r1` as the normal form.

```
r1: p(x,y) :- q(x1), s(y1), x1=y1, x=x1, y=y1.
r2: p(x,y) :- q(x), s(y), x=y.
```

The dependency graph for ThreeHops is shown in Figure 1, where boxes represent nodes in the graph and arrows represent edges in the graph.

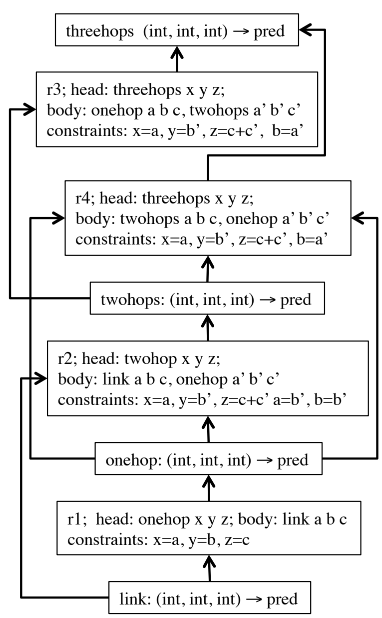

Figure 1: Dependency graph for ThreeHops (buggy)

Properties We focus on safety properties, which state that bad things haven't happened yet. We use trace-based semantics of NDLog [40, 10]. The advantage of trace-based semantics over fixed point semantics is that the order in which predicates are derived can be clearly specified using traces. Fixed point semantics only care about what are derivable in the end, and are not precise enough to capture transient faults that appear only in the middle of the execution of network protocols.

To make it possible for automated analysis, we restrict the form of the properties to be the following.

$$\varphi = \forall \vec{x_1}, p_1(\vec{x_1}) \wedge \forall \vec{x_2}, p_2(\vec{x_2}) \cdots \forall \vec{x_k}, p_k(\vec{x_k}) \wedge c_p(\vec{x_1}, \cdots \vec{x_k}) \supset \exists \vec{y_1} q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m} q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots \vec{x_k}, \vec{y_1}, \cdots \vec{x_m})$$

The meaning of the property is the following: if all of the predicates p_i are derivable, and their arguments satisfy constraint c_p, then each of the predicate q_j must be in one of the derivations of p_i, and the constraint c_q must be true. We implicitly require q_is to be derived before p_is. A lot of the correctness properties can be specified using formulas of this form. For instance, we can specify the following three properties of our ThreeHops program:

Q1: $\forall x, y, z, \mathsf{threehops}\ x\ y\ z \supset \exists x', z', \mathsf{twohops}\ x\ x'\ z'$
Q2: $\forall x, y, z, \mathsf{threehops}\ x\ y\ z$
$\supset \exists x_1, x_2, z_1, z_2, z_3, \mathsf{link}\ x\ x_1\ z_1 \wedge \mathsf{link}\ x_1\ x_2\ z_2$
$\wedge\ \mathsf{link}\ x_2\ y\ z_3$
Q3: $\exists x, y, z, \mathsf{threehops}\ x\ y\ z$

Q1 states that to derive threehops $x\ y\ z$, it is necessary to derive twohops $x\ x'\ z'$, for some x' and z'. Q1 does not hold because there are two ways to derive threehops and one of them does not contain such twohops tuple as a sub-derivation. Q2 states that to derive a threehops tuple, three links connecting those two nodes are necessary. Q2 should hold. Q3 states that threehops tuple is derivable for some x, y, and z.

2.3 Example Constraint Pool

A simplified derivation pool for onehop, twohops, and threehops is shown below. To ease presentation, we rewrite the derivation pool using equality constraints. onehop has only one derivation, using rule r1. A derivation $\mathcal{D}$ is a tuple consisting of four fields: the name of the last rule in the derivation; the conclusion of the derivation; the constraint associated with this derivation; and the list of derivations of the premises of the last rule. We instantiate the rules with concrete variables. The constraint in $\mathcal{D}$ is true, denoted $\top$; as there is no constraint in r1. The predicate twohops also has only one derivation, using r2. The premises of r2 are link and onehop. Since link is a base tuple, we simply represent its derivation as the tuple itself. The sub-derivation of onehop is the same as in the previous case. The constraint for deriving onehop is the conjunction of three constraints: c_1 is the constraint for deriving onehop, c_2 for the base tuple link, and c_3 the rule constraint of rule r2. Here c_2 is true, because no constraint is imposed on base tuples.

onehop
$\mathcal{D}$: $(r1, \mathsf{onehop}\ x_1\ x_2\ x_3, \{\mathsf{link}\ x_1\ x_2\ x_3\})$
$c = \top$
twohops
$\mathcal{D}$: $(r2, \mathsf{twohops}\ x_1\ x_2\ x_3$
$\{\mathsf{link}\ x_1\ x_2\ y_3, (r1, \mathsf{onehop}\ x_1\ x_2\ z_3, \{\mathsf{link}\ x_1\ x_2\ z_3\}\})$
$c = \top \wedge \top \wedge x_3 = y_3 + y_3$
threehops
$\mathcal{D}_1$: $(r3, \mathsf{threehops}\ x_1\ x_2\ x_3,$
$\{(r1, \mathsf{onehop}\ x_1, y_2, y_3, \{\mathsf{link}\ x_1\ y_2\ y_3\})$
$(r2, \mathsf{twohops}\ y_2\ x_2\ s_3,$
$\{\mathsf{link}\ y_2\ x_2\ t_3, (r1, \mathsf{onehop}\ y_2\ x_2\ u_3, \{\mathsf{link}\ y_2\ x_2\ u_3\})\})\})$
$c = \top \wedge \top \wedge \top \wedge s_3 = t_3 + u_3 \wedge x_3 = y_3 + s_3$
$\mathcal{D}_2$: $(r4, \mathsf{threehops}\ x_1\ x_2\ x_3,$
$\{(r2, \mathsf{twohops}\ x_1\ y_1\ s_3,$
$\{\mathsf{link}\ x_1\ y_1\ t_3, (r1, \mathsf{onehop}\ x_1\ y_1\ u_3, \{\mathsf{link}\ x_1\ y_1\ u_3\}\})$
$(r1, \mathsf{onehop}\ y_1, x_2, y_3, \{\mathsf{link}\ y_1\ x_2\ y_3\})\})$
$c = \top \wedge \top \wedge \top \wedge s_3 = t_3 + u_3 \wedge c_5 = x_3 = y_3 + s_3$

Tuple threehops has two derivations, one uses r3, the other uses r4. Both derivations contain sub-derivations of onehop and twohops. The constraints for deriving threehops include constraint for deriving twohops, onehop, and the rule constraint of r3 (r4).

3 Analyzing Non-recursive Programs

In this section, we first explain how to compute the derivation pool for a non-recursive NDLog program. Then, we show how to check property properties. Next, we show how to incorporate network constraints into our property checking algorithm. Finally, we prove the correctness of our algorithm.

3.1 Derivation Pool Construction

For a non-recursive program, its derivation pool maps each predicate to the set of all derivation trees rooted at that predicate. It is formally defined as follows.

$$\begin{array}{lll} \textit{Derivation pool} & \textit{dpool} & ::= \cdot \mid \textit{dpool}, (\textit{nID}, p{:}\tau) \mapsto \Delta \\ \textit{Entries} & \Delta & ::= \cdot \mid \Delta, (c, \mathcal{D}) \\ \textit{Derivation} & \mathcal{D} & ::= (\mathsf{BT}, p(\vec{x})) \mid (\textit{rID}, p(\vec{x}), \mathcal{D}\ \mathsf{List}) \end{array}$$

We write *dpool* to denote derivation pools. We write Δ to denote lists of pairs of a constraint and a derivation tree, denoted $\mathcal{D}$. At a high-level, $\mathcal{D}$ can be instantiated to be a valid derivation of $p(\vec{t})$ using rules in the program, if c is satisfiable. A derivation tree, $\mathcal{D}$, is inductively defined. The base tuples, denoted $(\mathsf{BT}, p(\vec{x}))$, are the leaf nodes. A non-leaf node consists of the unique rule ID of the last rule of the derivation, the conclusion of that rule ($p(\vec{x})$), and the list of derivation trees for the body predicates of that rule ($\mathcal{D}$ List).

Figure 2 and 3 present the main functions used for constructing a derivation pool from a dependency graph. The top-level function GenDPool is defined in Figure 2. This function follows the topological order of the nodes in the dependency graph $\mathcal{G}$.

We keep track of a working set P, which is the set of nodes whose derivations can be summarized currently. We also keep track of the set of edges that the function has not traversed yet. The function terminates when all of the edges in the dependency graph have been traversed and the derivations for all of the predicates in the dependency graph are built. In the body of GenDPool, we remove one predicate node p from P, and build all derivations for it. A base tuple's only possible derivation is one with itself as the leaf node. The constraint associated with this derivation is the trivial true constraint $\top$ (Line 8). When p is not a base tuple, derivations for tuples that p's derivations depend on have been stored in *dpool*. The GenDs function constructs derivations for p given the dependency graph and the current derivation pool (explained later).

After the derivations for a predicate p are constructed, outgoing edges from p are removed (Line 13), so predicates that depend on p can be processed in later iterations. Function RemoveEdges removes outgoing edges from p, and outgoing edges from rule nodes that now do not have incoming edges. This may result in predicates enqueued into P for the next iteration of processing.

```
function GenDPool(𝒢)
    E ← 𝒢's edges
    P ← 𝒢's predicate nodes that have no incoming edges
    while E ≠ empty || P ≠ empty do
        remove (nID, p : τ) from P
        x⃗ ← fresh(p : τ)
        if p is a base tuple then
            dpool← dpool[(nID, p) ↦ {(⊤, (BT, p(x⃗)))}]
        else
            d ← GenDs(𝒢, dpool, (nID, p:τ))
            dpool← dpool ∪ d
        (* done processing p, remove edges *)
        P, E ←removeEdges(P, E, G, nID)
    end while
end function

function removeEdges(P, E, G, nID)
    remove outgoing edges of nID from E
    for each rID with no edges of form (_, rID) in E do
        remove edges (rID, nID) from E
        for each (nID, p : τ) with no incoming edges in E do
            add (nID, p : τ) to P
end function
```

Figure 2: Construct derivation pools for non-recursive programs

function GenDs($\mathcal{G}$, *dpool*, $(nID, p : \tau)$)
  $\Delta \leftarrow \{\}$
  for each rule with ID *rID* where (*rID*,*nID*) in $\mathcal{G}$ **do**
    $\Delta \leftarrow \Delta \cup$GenDRule($\mathcal{G}$, *dpool*, $(nID, p : \tau)$,*rID*)
  return Δ
end function

function GenDRule($\mathcal{G}$, *dpool*, $(nID, p{:}\tau)$, *rID*)
  $(p(\vec{y}), Q, c) \leftarrow \mathcal{G}(rID)$
  (* $Q = (q_1, q_2, \cdots, q_m)$
    D is the list of list of derivations for $(q_1, q_2, \cdots, q_m)$ *)
  $D \leftarrow$ List.map (LookUp *dpool*) Q
  $D' \leftarrow$List.FoldRight MergeDLL D nil
  $\vec{x} \leftarrow fresh(p(\vec{y}))$
  return List.Map (completeD c *rID* $p(\vec{y})$ $\vec{x}$) D'
end function

function MergeD(dc_i, dc_{2i})
  (* dc_{2i} *is a derivations for* q_n *to* q_{i+1}
    dc_i *is a possible derivation of* q_i *)
  $(\sigma_{2i}, c_{2i}, d_{2i}) \leftarrow dc_{2i}$
  $(c_i, d_i) \leftarrow dc_i$
  (* σ_i *substitutes new vars in* q_i *for old ones* *)
  $(\sigma_i, c'_i, d'_i) \leftarrow fresh(c_i, d_i)$
  return $(\sigma_i \cup \sigma_{2i}, c'_i \wedge c_{2i}, d'_i :: d_{2i})$
end function

function LookUp(*dpool*, $q(\vec{x})$)
  return List.Map (extractD $\vec{x}$) $dpool(q)$
end function

function extractD($\vec{x}$, (c, d))
  $(rID, p(\vec{y}), dl) \leftarrow d$
  return $(\vec{y}/\vec{x}, c, d)$
end function

function completeD(c_r, *rID*, $p(\vec{y})$, $\vec{x}$, d)
  $(\sigma, c, dl) \leftarrow d$
  return $((c \wedge c_r)\sigma[\vec{x}/\vec{y}], (rID, p(\vec{x}), dl))$
end function

Figure 3: Generate derivation pool for one predicate

```
function MergeDLL(dc_li, dc_l2i)
    (* dc_l2i is the list of derivations for q_n, q_{n-1}, ..., q_i
       dc_li is the list of derivation of body tuple q_i *)
    a ← List.map (MergeDL dc_l2i) dc_li
    return List.flatten(a)
end function

function MergeDL(dc_l2i, dc_i)
    (* dc_l2i is the list of derivations for q_n, q_{n-1}, ..., q_i
       dc_i is a possible derivation of body tuple q_i *)
    return List.map (MergeD dc_i) dc_l2i
end function
```

Figure 4: List merge functions

Function GenDs (Figure 3) takes the dependency graph, the derivation pool that has been constructed so far, and a predicate p, as arguments, and returns all derivation pool entries for p. The body of GenDs calls GenDRule to construct derivations for each rule that derives p. The function GenDRule makes use of List map and fold operations to construct all possible derivations of p from a rule of the form $r:\ p(\vec{x})\,{:}\!-q_1(\vec{y_1}), ..., q_n(\vec{y_n}), c$. *dpool* has already stored all possible derivations for each q_i. We need to compute all combinations of the derivations for q_is. The LookUp function on line 11 collects the list of derivations for one body tuple and the list map function returns the list of derivations for all body tuples. More precisely, the LookUp function returns a list of tuples of the form (σ, c, d), where d is a derivation, c is the constraint associated with that derivation, and σ is a variable substitution. The domain of σ is q_i's arguments in the rule node, and the range of σ is q_i's arguments in the conclusion of the derivations. We need these substitutions because we alpha-rename the derivations. The constraint in the rule node needs to use the correct variables. Line 12 uses list fold operation to generate all possible derivations. Function MergeDLL and MergeDL in Figure 4 are helper functions to generate the list of derivations. Function MergeD is the function that takes as arguments, the list of derivations from q_m to q_{i+1} and one derivation for q_i, and prepends the derivation for q_i to the list of derivations from q_m up to q_i. Here, the substitutions need to be merged and the resulting constraint is the conjunction of the two constraints. Finally on line 14, function completeD generates a well-formed derivation for p using the rule ID and the list of derivations for q_is. The constraint associated with this derivation of p is the conjunction of constraints for the derivation of q_i and the constraint in the rule body. The substitutions are applied to the constraint c, because all derivations are alpha-renamed and use fresh variables.

3.2 Property Query

Figure 5 shows the property query algorithm for non-recursive programs. The top-level function CkProp takes the derivation pool and the property as arguments. One line 3, we separate the property into the list of predicates to the left of the implication (P), the constraint to the left of the implication (c_p), the list of predicates to the right of the implication (Q), and the constraint to the right of the implication c_q. Next, similar to the derivation pool construction, we construct all possible combinations of the derivations of all the p_is in P between lines 5 to 9. We omit the definition of MergeDerivation, as it is similar to MergeDLL. The only difference is that we do not need to alpha-rename the derivations. Next, we check that for each possible derivation of p_is in D, all of q_is appear in the derivation, and the constraint c_q holds (lines 10 to 14) using function CkPropD. If for all possible derivations of p_is, we can always find derivations of q_is such that the constraint c_q holds, φ holds (line 14).

```
function CkProp(dpool, φ)
    (* P is p_1 ··· p_n and Q is q_1 ··· q_m *)
    (P, c_p, Q, c_q) ← φ
    (* get the list of list of derivations for p_1, ··· , p_n *)
    L ← LookUp(dpool, P)
    (* combine all possible derivations for p_1 ··· p_n
       Each entry in D also include substitutions that replace
       free variables in p_i with the variable in the derivation *)
    D ←MergeDerivation L
    for each (σ, c, d) in D do
        z ←CkPropD(c, c_p σ, d, Q, c_q σ)
        if z = invalid(d, σ_r) then
            return invalid(d, σ_r)
    return valid
end function

function CkPropD(c_d, c_p, d, Q, c_q)
    if Check sat c_d ∧ c_p = (sat, σ_p) then
        (* find all occurrences of q in d *)
        Σ ←List.map (Unify d) Q
        if nil ∈ Σ then
            (* some q_i does not appear in d *)
            return Invalid(d,σ_p)
        else
            (* find all possible combinations for q_1...q_m
               Σ_q is a list of substitutions each σ in Σ_q is a
               substitution for variables in one occurrence
               of q_1 to q_m in d for variables that appear in Q *)
            Σ_q ←MergeLL Σ
            for each σ_q ∈ Σ_q do
                if Check sat c_d∧c_p∧¬c_q σ_q = (sat,σ_c) then
                    continue
                else
                    return valid
            (* None of the combinations of q works *)
            return invalid(d,σ_p)
    else
        return valid
end function
```

Figure 5: Property query

1: **function** CkPropDC(c_d, c_p, d, Q, c_q, B, c_b)
2: **if** Check sat $c_d \wedge c_p =$ true **then**
3: (* *find all occurrences of* b
4: Σ_b *is a list of list of substitutions* *)
5: $\Sigma_b \leftarrow$List.map (Unify d) B
6: (* Σ'_b *is a list of substitutions. Each substitution* *)
7: (* *in* Σ'_b *corresponds to one combination of* $b_i s$ *in* d *)
8: $\Sigma'_b \leftarrow$MergeLL Σ_b
9: (* c'_b *is the conjunction of* $c_b\sigma_i$*, where* $\sigma_i \in \Sigma'_b$ *)
10: $c'_b \leftarrow$Conj(Σ'_b, c_b)
11: (* *find all occurrences of* q *in* d *)
12: $\Sigma \leftarrow$List.map (Unify d) Q
13: **if** nil$\in \Sigma$ **then**
14: (* *check network constraints* *)
15: **if** Check sat $c_d \wedge c_p \wedge (c'_b) = (\mathsf{sat}, \sigma^c)$ **then**
16: **return** invalid($d\sigma^c$)
17: **else**
18: (* *network constraints are not met* *)
19: **return** valid
20: **else**
21: $\Sigma_1 \leftarrow$MergeLL Σ
22: (* *find all possible combinations for* $q_1 ... q_m$
23: Σ_1 *is a list of substitutions each* σ *in* Σ_1 *is a*
24: *substitution for variables in one occurrence*
25: *of* q_1 *to* q_m *in* d *for variables that appear in* Q *)
26: **for** each $\sigma \in \Sigma_1$ **do**
27: **if** Check sat $c_d \wedge c_p \wedge \neg c_q\sigma = (\mathsf{sat}, \sigma_q)$ **then**
28: **continue**
29: **else**
30: $c \leftarrow c_d \wedge c_p \wedge c_q\sigma \wedge (c'_b)$
31: **if** Check sat $c = (\mathsf{sat}, \sigma^c)$ **then**
32: (* *network constraints are met* *)
33: **return** valid
34: (* *None of the combinations of* q *works.*
35: *Next, check network constraints* *)
36: **if** Check sat $c_d \wedge c_p \wedge (c'_b) = (\mathsf{sat}, \sigma^c)$ **then**
37: **return** invalid($d\sigma^c$)
38: **else**
39: (* *network constraints are not met* *)
40: **return** valid
41: **else**
42: **return** valid
43: **end function**

Figure 6: Property query with network constraints

The function CkPropD checks that in the list of derivations d, with constraints c_d, whether all the predicates in Q appear in d, and c_q is true. On Line 18, we first check whether all the p_is are derivable and constraint c_p is satisfiable. If the conjunction of the derivation constraint c_d and c_p is not satisfiable, then the precedent of φ is false, so φ is trivially true for that derivation. So, we return valid in the else branch (line 38). If the conjunction is satisfiable, then there are substitutions for variables so that all the p_is are derivable and the constraint c_p is satisfiable. Next, we need to check whether all q_is are derivable. On line 20, function Unify identifies a list of occurrences of q_i in the derivation d. That is, for each $q_i(\vec{y_i})$ appearing in d, Unify returns the list of substitutions: $(\vec{y_1}/\vec{x})::(\vec{y_2}/\vec{x})\cdots::(\vec{y_n}/\vec{x})::\mathsf{nil}$, where $\vec{x}$ is q_i's arguments in φ. The list map function returns the list of the list of occurrences for all the q_is in Q. We call it "UNIFY" because we unify the variables that are q_i's arguments in φ with q_i's arguments in the derivation d. This substitution will be applied to constraint c_q later. If some q_i does not appear in d, then UNIFY will return an empty list nil. Therefore, on line 21, we check whether each q_i will appear at least once in d. If it is not the case, then we return invalid with the current derivation and one satisfying substitution that makes p_is true for constructing a counterexample. Otherwise, we check whether the constraint c_q can be satisfied. Before doing so, on line 25, we first compute the list of all possible combinations of occurrences of q_is. Again, this function is similar to MERGEDLL and we omit the details. Now on line 30 for each possible appearance of q_is in d, Σ_q is a list of substitutions, each of which, when applied to c_q, makes c_q use the same variables as those in the derivation. We ask whether the negation of c_q together with the derivation constraint and the constraint on the arguments of p_is are satisfiable. If this is not satisfiable, then we know that there exists a substitution for variables so that the property φ holds. Otherwise, we return the derivation and the satisfying substitution that makes p_is and q_is derivable, but c_q false for counterexample construction.

3.3 Network Constraints

Sometimes, the network being analyzed has some constraints, for instance, every node in the network has only one outgoing link. We call these constraints *network constraints*. Our property query algorithm needs to take into consideration, these network constraints. If we ignore these constraints, the counterexample generated by the tool may not be useful as the counterexample could violate the network constraints.

Network constraints that our analysis can handle have similar form as the properties: $\forall \vec{x_1}.b_1(\vec{x_1}) \wedge ...\forall \vec{x_k}.b_k(\vec{x_k}) \supset c$, where, b_i is a base tuple. Figure 6 shows the algorithm for checking properties on networks with constraints. For ease of explanation, we explain the case with only one network constraint. Extending the algorithm to handle multiple constraints is straightforward.

The top-level function CKPROPC is almost the same as CKPROP, except that it takes a network constraint (φ_{net}) as an additional argument and uses the function CKPROPDC, which additionally checks network constraints compared to CKPROPD. The function CKPROPDC takes as additional arguments, a list base tuples B and the constraint c_b in the network constraint. In the body of CKPROPDC, we first check whether the constraint on p_is is satisfiable. If it is not, then this derivation does not violate the property we are checking. Next, between lines 3 to 10, we find all occurrences of the base tuples in the constraint φ_{net}. We find all possible combinations of substitutions for arguments of these base tuples as they appear in the derivation d. For each occurrence of the base tuples, the constraint c_b needs to be true, so we compute the conjunction of all the c_bs. To given an example, if the constraint is $\forall b(x) \supset x > 0$. If d has two occurrences of b, $b(y)$ and $b(z)$, then $c_b' = y > 0 \wedge z > 0$.

Next, we collect the list of the occurrences of q_is, the same as before. If some q_is do not appear in d (line 13), we additionally check whether this derivation d satisfies the network constraint (line 15). If it is the case, then we find a counterexample. Otherwise, d does not violate the property being checked.

Then, we compute the combination of all possible occurrences of q_is (line 21) as usual. For each substitution that makes all q_is appear in d, we check whether c_q is satisfiable. On lines 30 to 33, c_q is satisfiable, so we need check that the network constraint is satisfied. If this is the case, d satisfies the property being checked. Otherwise, we have to try the next substitution that makes all q_is appear in d. On line 34, we finished the loop and c_q is not satisfiable for any of the substitutions that make q_is appear in d. Again, we check the network constraints on d, and report an error only if d satisfies the network constraint.

3.4 Correctness

We first prove that our derivation pool construction is correct. Lemma 1 states that an entry for a predicate p in the derivation pool maps to a valid derivation of p if the constraints of that derivation is satisfiable; and that if a predicate p is derivable, then there must be a corresponding entry in the derivation pool. The semantics of NDLog programs are bottom up, so a set of base tuples B is needed to start the execution of the program. We write $\sigma' \geq \sigma$ to mean that σ' extends σ. B refers to the base tuples of *prog*.

Lemma 1 (Correctness of derivation pool construction)**.**
$GenDPool(prog) = dpool$

1. *If* $prog, B \vDash d'{:}p(\vec{t})$ *then exists* σ *and* $(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in dpool(p)$ *s.t.* $d(\vec{x_d})\sigma = d'$ *and* $\vDash c(\vec{x_c})\sigma$.

2. *If* $(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in dpool(p)$ *and* $\vDash c(\vec{x_c})\sigma$*, then exists* B*,* σ' *s.t.* $\sigma' \geq \sigma$ *and* $prog, B \vDash d(\vec{x_d})\sigma'{:}p(\vec{x})\sigma'$.

Proof. 1. Proof by induction on the structure of the derivation d'.

Base case: $d' = (\mathsf{BT},\ p(\vec{t}))$

Since p is a base tuple, By Line 7 of Function GenDPool, its entry in *dpool* is given by

$(\top, (\mathsf{BT}, p(\vec{x}))) \in dpool(p)$

By Line 6 of Function GenDPool,

$\vec{x}$ are fresh variables for the arguments of p

Let $\sigma = \vec{t}/\vec{x}$, then

$(\mathsf{BT}, p(\vec{x}))\sigma = (\mathsf{BT}, p(\vec{t}))$ and

$\models \top\sigma$

Inductive case: $d' = (rID,\ p(\vec{t}),\ (d'_1{:}q_1(\vec{t_1}))::...::(d'_n{:}q_n(\vec{t_n}))::\mathsf{nil})$

rID has form `p(u) :- q1(u1),...,qn(un),c(u1,...,un)`

c is a constraint that may comprise the arguments of $p, q_1, \ldots, q_n$

It would be more accurate to write $c(\vec{t_c})$ where $\vec{t_c} \subseteq \{\vec{t}, \vec{t_1}, \cdots, \vec{t_n}\}$;

However we write $c(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$ for clarity in later parts of the proof

(1) $\models c(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$

By assumption,

$prog, B \models (rID,\ p(\vec{t}),\ (d'_1{:}q_1(\vec{t_1}))::...::(d'_n{:}q_n(\vec{t_n}))::\mathsf{nil}){:}p(\vec{t})$

Therefore for $1 \leq i \leq n$,

(2) $prog, B \models d'_i{:}q_i(\vec{t_i})$

By the Inductive Hypothesis,

(3) $\exists \sigma_i$ where $\sigma_i = [\vec{t_{di}}/\vec{x_{di}}]$ such that

$(c_i(\vec{x_{ci}}), d_i(\vec{x_{di}}){:}q_i(\vec{x_i})) \in dpool(q_i)$,

$(d_i(\vec{x_{di}}){:}q_i(\vec{x_i}))\sigma_i = d'_i{:}q_i(\vec{t_i})$,

$\models c_i(\vec{x_{ci}})\sigma_i$

(4) $\vec{x_{ci}} \subseteq \vec{x_{di}}$, $\vec{x_i} \subseteq \vec{x_{di}}$, $\vec{x_i} \subseteq \vec{x_{ci}}$

(5) $\vec{t_i} \subseteq \vec{t_{di}}$.

By Freshness Lemma (Lemma 2),

(6) $\vec{x}, \vec{x_{d1}}, \ldots, \vec{x_{dn}}$ are fresh

By GenDPool,

(7) $(c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}), (rID, p(\vec{z}), d_1(\vec{z_{d1}}){:}q_1(\vec{z_1})::...::d_n(\vec{z_{dn}}){:}q_n(\vec{z_n})::\mathsf{nil})) \in dpool(p)$

(8) $\vec{z_i} \subseteq \vec{z_{di}}$, $\vec{z_{ci}} \subseteq \vec{z_{di}}$, $\vec{z_i} \subseteq \vec{z_{ci}}$

By Freshness Lemma (Lemma 2),

$i \neq j \rightarrow \vec{z_{di}} \cap \vec{z_{dj}} = \emptyset$

By (3), (4), (5), (6), (7), (8), we can define

(9) $\sigma = \bigsqcup_{i=1}^{n} [\vec{x_{di}}/\vec{z_{di}}]\sigma_i$

$= \bigsqcup_{i=1}^{n} [\vec{x_{di}}/\vec{z_{di}}][\vec{t_{di}}/\vec{x_{di}}]$

$= \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}]$

where $\vec{z} \subseteq \{\vec{z_{d1}}, \ldots, \vec{z_{dn}}\}$

Using (7) where $(c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}), (rID, p(\vec{z}), d_1(\vec{z_{d1}}){:}q_1(\vec{z_1})::...::d_n(\vec{z_{dn}}){:}q_n(\vec{z_n})::\mathsf{nil})) \in dpool(p)$

And (8), we know that $\vec{z_{ci}} \subseteq \vec{z_{di}}$, thus

(10) $c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n})\sigma$

$= c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n}) \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}]$

$= c(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$

By (1), $\models c(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$

Therefore by (10),

(11) $\models c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n})\sigma$

By (3), $\models c_i(\vec{x_{ci}})\sigma_i$, (where $\sigma_i = [\vec{t_{di}}/\vec{x_{di}}]$).

By (9), $\sigma = \bigsqcup_{i=1}^{n} [\vec{x_{di}}/\vec{z_{di}}]\sigma_i$

(12) $c_i(\vec{z_{ci}})\sigma = c_i(\vec{z_{ci}}) \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}] = c_i(\vec{t_{ci}})$

By (12),
  (13) $\models \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}})\sigma$

By (8), $\sigma = \bigsqcup_{i=1}^{n}[\vec{t_{di}}/\vec{z_{di}}]$

By (8), $\sigma = \bigsqcup_{i=1}^{n}[\vec{t_{di}}/\vec{z_{di}}]$
By (11) and (13), we get
  (14) $\models (c(\vec{z}, \vec{z_1}, ..., \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}))\sigma$

By (3), $(d_i(\vec{x_{di}}){:}q_i(\vec{x_i}))\sigma_i = d_i'{:}q_i(\vec{t_i})$
  (15) $(d_i(\vec{z_{di}}){:}q_i(\vec{z_i}))\sigma$
    $= (d_i(\vec{z_{di}}){:}q_i(\vec{z_i})) \bigsqcup_{i=1}^{n}[\vec{t_{di}}/\vec{z_{di}}]$
    $= d_i'{:}q_i(\vec{t_i})$

By (7) and (12),
  $(rID, p(\vec{z}), d_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}...{::}d_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}))\sigma$
  $= (rID, p(\vec{t}), d_1'{:}q_1(\vec{t_1}){::}...{::}d_n'{:}q_n(\vec{t_n}){::}\mathsf{nil}))$

2. Proof by the structure of d

Base Case $(\top, (\mathsf{BT}, p(\vec{x}))) \in \mathit{dpool}(p))$
Define $B = \{p(\vec{x})\}$.
Choose $\sigma = \{\}$
Then there exists $\sigma' = [\vec{t}/\vec{x}]$ where $\sigma' \geq \sigma$, such that
  $\mathit{prog}, B \models (\mathsf{BT}, p(\vec{x}))\sigma'{:}p(\vec{x})\sigma'$
  Which is equivalent to $\mathit{prog}, B \models (\mathsf{BT}, p(\vec{t})){:}p(\vec{t})$

Inductive case
$(c_p(\vec{x_{cp}}), (rID, p(\vec{x}), ((d_1(\vec{x_{d1}}){:}q_1(\vec{x_1})){::}...{::}(d_n(\vec{x_{dn}}){:}q_n(\vec{x_n})){::}\mathsf{nil})){:}p(\vec{x})) \in \mathit{dpool}(p)$
where $\vec{x_i} \subseteq \vec{x_{di}}$, $\vec{x_{cp}}$ are variables to be determined
Given $(c_p(\vec{x_{cp}}), (rID, p(\vec{x}), ((d_1(\vec{x_{d1}}){:}q_1(\vec{x_1})){::}...{::}(d_n(\vec{x_{dn}}){:}q_n(\vec{x_n})){::}\mathsf{nil})){:}p(\vec{x})) \in \mathit{dpool}(p)$
  (1) For $1 \leq i \leq n$, $(c_i(\vec{z_{ci}}), d_i(\vec{z_{di}}){:}q_i(\vec{z_i})) \in \mathit{dpool}(q_i)$
    where $\vec{z_i} \subseteq \vec{z_{di}}$, $\vec{z_{ci}} \subseteq \vec{z_{di}}$, $\vec{z_i} \subseteq \vec{z_{di}}$
By Freshness Lemma (Lemma 2)
  $i \neq j \rightarrow \vec{z_{di}} \cap \vec{z_{dj}} = \emptyset$

By GenDPool,
  (2) $c_p(\vec{z_{cp}}) = c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}})$
    where $\vec{z_i} \subseteq \vec{z_{ci}}$, $\vec{z_i} \subseteq \vec{z_{di}}$, $\vec{z_{ci}} \subseteq \vec{z_{di}}$
rID has form `p(u) :- q1(u1),...,qn(un),c(u1,...,un)`
c is a constraint that may comprise the arguments of $p, q_1, \ldots, q_n$
It would be more accurate to write $c_r(\vec{t_c})$ where $\vec{t_c} \subseteq \{\vec{t}, \vec{t_1}, \cdots, \vec{t_n}\}$,
However we write $c_r(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$ for clarity in later parts of the proof
By [Freshness Lemma],
  (3) $\vec{z_{d1}}, \ldots, \vec{z_{dn}}$ are fresh variables

By assumption, there is some σ such that
  (4) $\models c_p(\vec{z_{cp}})\sigma$
Rewrite (4) to get
  (5) $\models (c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}))\sigma$

By Line 13 of function GenDRule
  (6) $\vec{x}$ are fresh variables for the arguments of p
Using (6), we can define
  (7) $\sigma = \bigsqcup_{i=1}^{n}[\vec{t_{di}}/\vec{z_{di}}]$
    where $\vec{t_i} \subseteq \vec{t_{di}}$, $\vec{t_i} \subseteq \vec{t_{ci}}$, $\vec{t_{ci}} \subseteq \vec{t_{di}}$

and $\vec{z} \subseteq \{\vec{z_{d1}}, \ldots, \vec{z_{dn}}\}$

By (5), $\models (c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}))\sigma$,
Using conjunction elimination,
(8) $\models c_i(\vec{z_{ci}}))\sigma$
By σ as in (7),
(9) $c_i(\vec{z_{ci}})\sigma$
$= c_i(\vec{z_{ci}}) \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}]$
$= c_i(\vec{t_{ci}})$
By (9), we can choose
(10) $\sigma_i = [\vec{t_{ci}}/\vec{z_{ci}}]$ such that
$\models c_i(\vec{z_{ci}})\sigma_i$

By Induction Hypothesis, for $1 \leq i \leq n$,
(11) exists B_i, σ'_i where $\sigma'_i \geq \sigma_i$, such that
$prog, B_i \models d_i(\vec{z_{di}})\sigma'_i{:}q_i(\vec{z_i})\sigma'_i$
By Freshness Lemma (Lemma 2),
(12) $(i \neq j) \rightarrow (\vec{z_{di}} \cap \vec{z_{dj}} = \emptyset)$
By (12),
(13) $(i \neq j) \rightarrow (dom(\sigma'_i) \cap dom(\sigma'_j) = \emptyset)$

By (13), we can define
$\sigma' = \bigsqcup_{i=1}^{n} [\vec{x_{di}}/\vec{z_{di}}]\sigma'_i$
By construction,
$\sigma' \geq \sigma$

By (11), for $1 \leq i \leq n$, $prog, B_i \models d_i(\vec{z_{di}})\sigma'_i{:}q_i(\vec{z_i})\sigma'_i$, therefore
(14) $prog, \bigcup_{i=1}^{n} B_i \models (d_1(\vec{x_{d1}}){:}q_1(\vec{x_1}))::\ldots::(d_n(\vec{x_{dn}}){:}q_n(\vec{x_n}))::\mathsf{nil})\sigma'$

By applying the rule *rID* to (14), we construct
$prog, \bigcup_{i=1}^{n} B_i \models d_p\sigma'{:}p(\vec{x})\sigma'$
where $d_p = (rID,\ p(\vec{x}),\ (d_1(\vec{x_{d1}}){:}q_1(\vec{x_1}))::\ldots::(d_n(\vec{x_{dn}}){:}q_n(\vec{x_n}))::\mathsf{nil})$

□

Lemma 2 (Freshness)**.** *If $(c, d{:}p(\vec{x})) \in dpool(p)$, then the variables in $(c, d{:}p(\vec{x}))$ are fresh.*

Proof.

By Induction on the structure of d

Base Case: $(c, (\mathsf{BT}, p(\vec{x})){:}p(\vec{x})) \in dpool(p)$
By Line 8 of GenDPool,
$(c, (\mathsf{BT}, p(\vec{x})) \in dpool$
By Line 6 of GenDPool
There are fresh variables $\vec{x}$ for the arguments of p
$c = \top$ has no variables
Therefore the variables in $(c, (\mathsf{BT}, p(\vec{x})){:}p(\vec{x}))$ are fresh.

Inductive Case:
$(c_p, (rID, p(\vec{z}), (d_1(\vec{z_{d1}}){:}q_1(\vec{z_1}))::\ldots::(d_n(\vec{z_{dn}}){:}q_n(\vec{z_n}))::\mathsf{nil}){:}p(\vec{z})) \in dpool(p)$

By Functions GenDPool and GenDRule,
(1) $c_p(\vec{z}, \vec{z_{c1}}, \ldots, \vec{z_{cn}}) = c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}})$
where c_r is the constraint for *rID*
and $\vec{z_i} \subseteq \vec{z_{ci}}$

By I.H., for all $1 \leq i \leq n$,

(2) $(c_i(\vec{x_{ci}}), d_i(\vec{x_{di}}){:}q_i(\vec{x_i})) \in \mathit{dpool}(q_i)$
where $\vec{x_{ci}} \subseteq \vec{x_{di}}$, and $\vec{x_i} \subseteq \vec{x_{di}}$.
Where $\vec{x_{di}}$ are fresh variables

By Function MergeD (Line 23), for all $1 \leq i \leq n$,
exists σ_i where $\sigma_i = [\vec{z_{di}}/\vec{x_{di}}]$ such that
(3) $c_i(\vec{x_{ci}})\sigma_i = c_i(\vec{z_{ci}})$
(4) $(d_i(\vec{x_{di}}){:}q_i(\vec{x_i}))\sigma_i = d_i(\vec{z_{di}}){:}q_i(\vec{z_i})$
where $\vec{z_{ci}} \subseteq \vec{z_{di}}$, $\vec{z_i} \subseteq \vec{z_{di}}$
and $\vec{z_{di}}$ is fresh
Therefore
$(i \neq j) \supset (\sigma_i \sqcup \sigma_i = \emptyset)$

Line 12 of Function GenDRule uses Functions MergeDLL and MergeD
returns a list of possible combinations of derivations of $q_1, \ldots, q_n$ with of form
(5) $(\bigsqcup_{i=1}^{n} \sigma_i,\ \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}),\ d_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil})$
By (3) and (4),
$\bigsqcup_{i=1}^{n} \sigma_i$ substitutes new variables $\vec{z_{di}}$ for old ones $\vec{x_{di}}$
$\bigwedge_{i=1}^{n} c_i(\vec{z_{ci}})$ is composed of fresh variables
$d_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}$ is also composed of fresh variables
By (5) we can define
(6) $\sigma = \bigsqcup_{i=1}^{n} \sigma_i$
The derivation of p is
(7) $((c_r(\vec{x}, \vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}))\sigma, (\mathit{rID}, p(\vec{z}), d_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}))$
By (6) and (7)
All the variables in $(c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}), (\mathit{rID}, p(\vec{z}), d_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}))$ are fresh
□

Using the result of Lemma 1, we prove our property checking algorithm is correct with regard to the formula semantics.

Theorem 3 (Correctness of property query)**.**

$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \forall \vec{x_2}.p_2(\vec{x_2}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n}) \supset$
$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$
$\mathsf{DPool}(\mathit{prog}) = \mathit{dpool}$

Note that it would be more accurate to write
$c_p(\vec{x_{cp}})$*, where* $\vec{x_{cp}} \subseteq \vec{x_1}, \ldots, \vec{x_n}$ *and*
$c_q(\vec{x_{cq}})$*, where* $\vec{x_{cq}} \subseteq \vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}$
However we write $c_p(\vec{x_1}, \cdots, \vec{x_n})$ *and* $c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$ *for reasons of clarity when performing substitutions*

1. $\mathit{prog}, B \nvDash \varphi$ *implies* $\textsc{CkProp}(\mathit{dpool}, \varphi) = \mathsf{invalid}(d, \sigma)$*,* $d\sigma$ *is a list of derivations for* $p_1(\vec{t_1}), \cdots, p_n(\vec{t_n})$ *and either the derivations do not contain every* q_i*s, or for every combination of* q_1 *to* q_m*,* c_q *is not satisfiable.*

2. $\textsc{CkProp}(\mathit{dpool}, \varphi) = \mathsf{invalid}(d, \sigma)$ *implies exists* B *s.t.* $\mathit{prog}, B \nvDash \varphi$*.*

Proof. Proof of 1.

By assumption $\mathit{prog}, B \nvDash \varphi$
which is equivalent to
$\mathit{prog}, B \nvDash \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \forall \vec{x_2}.p_2(\vec{x_2}) \wedge \cdots \wedge \forall \vec{x_n}.p_k(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n}) \supset$
$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$
By semantics of $\supset$ this means that
(1) $\mathit{prog}, B \vDash \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \forall \vec{x_2}.p_2(\vec{x_2}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n})$
(2) $\mathit{prog}, B \nvDash \exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$

By (1), and assuming that $x_1, \dots, x_n$ are unique variables,
there exists substitution

(3) $\sigma_p = \bigsqcup_{i=1}^{n} [\vec{t_i}/\vec{x_i}]$ such that

(4) $\mathit{prog}, B \vDash (p_1(\vec{x_1}) \wedge \cdots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n}))\sigma_p$
which is equal to $\mathit{prog}, B \vDash p_1(\vec{t_1}) \wedge \cdots \wedge p_n(\vec{t_n}) \wedge c_p(\vec{t_1}, \cdots, \vec{t_n})$

By (2),

(5) $\nexists \sigma \geq \sigma_p$ such that $\mathit{prog}, B \vDash (q_1(\vec{y_1}) \wedge \cdots \wedge q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{y_m}))\sigma$

By Correctness of Derivation Pool (Lemma 1),
Given that $\vDash p_i(\vec{x_i})\sigma_p$, for $1 \leq i \leq n$,

(6) exists σ_i such that
$(c_i(\vec{u_{ci}}), d_i(\vec{u_{di}}){:}p_i(\vec{u_i})) \in \mathit{dpool}(p_i)$,
$\vDash c_i(\vec{u_{ci}})\sigma_i$
$d_i(\vec{u_{di}})\sigma_i$ is a proof of $p_i(\vec{u_i})\sigma_i$

(7) $\vec{u_i} \subseteq \vec{u_{di}}$, $\vec{u_{ci}} \subseteq \vec{u_{di}}$, $\vec{u_i} \subseteq \vec{u_{ci}}$

By Freshness Lemma (Lemma 2)

(8) $u_{d1}, \dots, u_{dn}$ are fresh variables

By (6) and (7)

(9) $\sigma_i = [\vec{t_{di}}/\vec{u_{di}}]$ for some constant $\vec{t_{di}}$

By (3),
$\vec{t_i} \subseteq \vec{t_{di}}$,

The algorithm returns valid under two cases

subcase 1:

$\nexists \sigma$ such that $\mathit{prog}, B \vDash (c_p(\vec{x_1}, \cdots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_{pi}(u_{ci}))\sigma$
Failed the check on Line 18 of CkPropD and return "valid" on Line 38

However, we can construct such a σ
By (4), $\mathit{prog}, B \vDash (p_1(\vec{x_1}) \wedge \cdots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n}))\sigma_p$

(10) $\mathit{prog}, B \vDash c_p(\vec{x_1}, \cdots, \vec{x_n})\sigma_p$

By (9), $\sigma_i = [\vec{t_{di}}/\vec{u_{di}}]$
By (6), for each $1 \leq i \leq n$,

(11) $\vDash c_{pi}(\vec{u_{ci}})\sigma_i$

Using (10) and (11), we can define

(12) $\sigma = \sigma_p \sqcup \bigsqcup_{i=1}^{n} \sigma_i$

Therefore

(13) $\mathit{prog}, B \vDash (c_p(\vec{x_1}, \cdots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_{pi}(u_{ci}))\sigma$
and (13) contradicts the assumption of this subcase

subcase 2:

Every element in D in CkProp is "invalid" in CkPropD

(14) The unification on Line 20 of CkPropD is successful

By (14), for each $1 \leq j \leq m$, there exists $1 \leq k \leq n$ such that
$q_j(z_j) \in d'_k{:}p_k(\vec{t_k})$
By CkPropD,

(7) $c_p(\vec{x_1}, \dots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_{pi}(\vec{u_{ci}}) \wedge \neg c_q(\vec{x_1}, \dots, \vec{x_n}, \vec{y_1}, \dots, \vec{y_m})$ is unsat

By (7),
there exists a substitution σ' such that

(8) $\mathit{prog}, B \vDash c_q(\vec{x_1}, \dots, \vec{x_n}, \vec{y_1}, \dots, \vec{y_m})\sigma'$

By (10), $prog, B \models c_p(\vec{x_1}, \cdots, \vec{x_n})\sigma_p$
By (8) and (10),
(9) $\sigma' \geq \sigma_p$

By (8) and (9),
(10) exists $\sigma' \geq \sigma_p$ such that $prog, B \models (q_1(\vec{y_1}) \wedge \cdots \wedge q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}))\sigma'$
Recall that (5) means that
$\nexists \sigma \geq \sigma_p$ such that $prog, B \models (q_1(\vec{y_1}) \wedge \cdots \wedge q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{y_m}))\sigma$
(10) contradicts (5)

Proof of 2.

By assumption, CkProp returns invalid, hence
(1) $prog, B \not\models \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \forall \vec{x_2}.p_2(\vec{x_2}) \wedge \cdots \wedge \forall \vec{x_n}.p_k(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n}) \supset$
$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$
Therefore
(2) $prog, B \models \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \forall \vec{x_2}.p_2(\vec{x_2}) \wedge \cdots \wedge \forall \vec{x_n}.p_k(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n})$
(3) $prog, B \not\models \exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$

By (2) and Correctness of Derivation Pool (Lemma 1),
there exists a substitution σ_p such that
(4) $prog, B \models (p_1(\vec{x_1}) \wedge \ldots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n}))\sigma_p$

subcase 1:
The test on line 21 of CkPropD fails
Some q_i in $q_1, \ldots, q_m$ is not found in the derivations of $p_1, \ldots, p_n$, thus
(5) $prog, B \not\models \exists \vec{y_i}.q_1(\vec{y_i})$
Given (5), this implies that
(6) $prog, B \not\models \exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$
By (6), the consequent of φ is invalid
Since the antecedent of φ is assumed to be valid, therefore
$prog, B \not\models \varphi$

subcase 2:
for every unification of q_i
(7) $c_p(\vec{x_1}, \cdots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_{pi}(\vec{u_{ci}}) \wedge \neg c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})\sigma_p$ is satisfiable
By (7), $c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$ is unsat, so
(8) $prog, B \not\models \exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots, \vec{x_n}, \vec{y_1}, \cdots, \vec{x_m})$
By (8), the consequent of φ is invalid
Since the antecedent of φ is assumed to be valid, therefore
$prog, B \not\models \varphi$

$\square$

When network constraints are provided, we prove that the property checking algorithm is correct with regard to the network constraints on base tuples.

Theorem 4 (Correctness of property query with constraints)**.**

$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \forall \vec{x_2}.p_2(\vec{x_2}) \wedge \ldots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}) \supset$
$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \exists \vec{y_2}.q_2(\vec{y_2}) \wedge \ldots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{x_m})$
$\varphi_{net} = \forall \vec{z_1}.b_1(\vec{z_1}) \wedge \ldots \wedge \forall \vec{z_k}.b_k(\vec{z_k}) \supset c_{net}(\vec{z_1}, \ldots, \vec{z_k})$
Where $b_1, \ldots, b_k$ are base tuples.

It would be more accurate to write
$c_p(\vec{x_p})$ *where* $\vec{x_p} \subseteq \{\vec{x_1}, \ldots, \vec{x_n}\}$
$c_q(\vec{y_q})$ *where* $\vec{y_q} \subseteq \{\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{x_m}\}$
$c_{net}(\vec{z_{net}})$ *where* $\vec{z_{net}} \subseteq \{\vec{z_1}, \ldots, \vec{z_k}\}$

However we write $c_p(\vec{x_1}, \ldots, \vec{x_n})$, $c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{x_m})$, and $c_{net}(\vec{z_1}, \ldots, \vec{z_k})$ for clarity in substitutions

GENDPOOL(*prog*) = *dpool*,

1. *$B \vDash \varphi_{net}$ and $prog, B \nvDash \varphi$ implies $CkPropC(dpool, \varphi_{net}, \varphi) = \mathsf{invalid}(d, \sigma)$, $d\sigma$ is a list of derivations for $p_1(\vec{t_1}), ..., p_n(\vec{t_n})$ and either the derivations do not contain every q_is, or for every combination of q_1 to q_m, c_q is not satisfiable.*

2. *$CkPropC(dpool, \varphi_{net}, \varphi) = \mathsf{invalid}(d)$ implies exists B s.t. $prog, B \nvDash \varphi$ and $B \vDash \varphi_{net}$.*

Proof. **Proof of 1.**

By assumption,
$prog, B \nvDash \varphi$
which is equivalent to
$prog, B \nvDash \forall\vec{x_1}.p_1(\vec{x_1}) \wedge \forall\vec{x_2}.p_2(\vec{x_2}) \wedge \ldots \wedge \forall\vec{x_n}.p_k(\vec{x_n}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}) \supset$
$\exists\vec{y_1}.q_1(\vec{y_1}) \wedge \ldots \wedge \exists\vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{x_m})$
By semantics of $\supset$ this means that
(1) $prog, B \vDash \forall\vec{x_1}.p_1(\vec{x_1}) \wedge \forall\vec{x_2}.p_2(\vec{x_2}) \wedge \ldots \wedge \forall\vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \cdots, \vec{x_n})$
(2) $prog, B \nvDash \exists\vec{y_1}.q_1(\vec{y_1}) \wedge \ldots \wedge \exists\vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{x_m})$

By (1), and assuming that $x, x_1, \ldots, x_n$ are fresh variables,
there exists substitution σ_p
where $\sigma_p = [\vec{t}/\vec{x}] \sqcup \bigsqcup_{i=1}^{n}[\vec{t_i}/\vec{x_i}]$ such that
(3) $prog, B \vDash (p_1(\vec{x_1}) \wedge \ldots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}))\sigma_p$
(4) $\nexists\sigma \geq \sigma_p$ s.t. $prog, B \vDash (q_1(\vec{y_1}) \wedge \ldots \wedge q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}))\sigma$

By Correctness of Derivation Pool (Lemma 1),
Given that by (3), $prog, B \vDash (p_1(\vec{x_1}) \wedge \ldots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}))\sigma_p$
for $i \in \{1, 2, \ldots, n\}$
(5) exists $\sigma_i = [\vec{t_{di}}/\vec{z_{di}}]$ such that
$(c_i(\vec{z_{ci}}), d_i(\vec{z_{di}}){:}p_i(\vec{z_i})) \in dpool(p_i)$
$\vDash c_i(\vec{z_i})\sigma_i$,
$d_i\sigma_p$ is a derivation of $p_i(\vec{z_i})\sigma_p$
where $\vec{z_i} \subseteq \vec{z_{ci}}$, $\vec{z_i} \subseteq \vec{z_{di}}$, $\vec{z_{ci}} \subseteq \vec{z_{ci}}$
and $\vec{t_i} \subseteq \vec{t_{ci}}$, $\vec{t_i} \subseteq \vec{t_{di}}$, $\vec{t_{ci}} \subseteq \vec{t_{ci}}$
By Freshness Lemma 2,
$\vec{z_{d1}}, \ldots, \vec{z_{dn}}$ are fresh
By assumption,
$\vDash \varphi_{net}$
Which is equivalent to
$\vDash \forall\vec{z_1}.b_1(\vec{z_1}) \wedge \ldots \wedge \forall\vec{z_k}.b_k(\vec{z_k}) \supset c_{net}(\vec{z_1}, \ldots, \vec{z_k})$
Therefore by semantics,
(6) $\vDash \forall\vec{z_1}.b_1(\vec{z_1}) \wedge \ldots \wedge \forall\vec{z_k}.b_k(\vec{z_k})$
$\vDash c_{net}(\vec{z_1}, \ldots, \vec{z_k})$
By (6), there is some σ_{net} such that
$\vDash (b_1(\vec{z_1}) \wedge \ldots \wedge b_k(\vec{z_k}))\sigma_{net}$
Therefore
(7) $\vDash c_{net}(\vec{z_1}, \ldots, \vec{z_k})\sigma_{net}$

The algorithm returns valid under several cases

subcase 1: Line 42 of CKPROPDC returns "valid" each time
$c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}})$ is unsat
We show a contradiction
By (3),

$prog, B \models (p_1(\vec{x_1}) \wedge \ldots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}))\sigma_p$
Using conjunction elimination, we have $prog, B \models c_p(\vec{x_1}, \ldots, \vec{x_n})\sigma_p$
By (5), for $i \in \{1, 2, \ldots, n\}$,
$\models c_i(\vec{x_{ci}})\sigma_i[dom(\sigma_i)/\vec{z_{di}}]$
is equal to $\models c(\vec{x_{ci}})[\vec{t_{di}}/\vec{x_{di}}][\vec{x_{di}}/\vec{z_{di}}]$
is equal to $\models c(\vec{x_{ci}})[\vec{t_{di}}/\vec{x_{di}}]$
is equal to $\models c(\vec{t_{ci}})$
Combining the two, we get
$\sigma = \sigma_p \cup \bigsqcup_{i=1}^{n} \sigma_i[dom(\sigma_i)/\vec{z_{di}}]$
We have a satisfying substitution
$(c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}))\sigma$
Which is equal to $c_p(\vec{t_1}, \ldots, \vec{t_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{t_{ci}})$

subcase 2: Line 33 of CkPropDC returns "valid" each time
By Line 21 of CkPropDC,
(8) Each $q_1, \ldots, q_m$ has some $d_k(\vec{x_{dk}}) : p_k(\vec{x_k})$ such that
$q_j(y_j) \in d_k(\vec{x_{dk}})$
By Line 27 of CkPropDC,
(9) $c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \neg c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m})$ is unsat
By Line 30 of CkPropDC,
(10) $c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \neg c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k})$

In this case there exists a substitution $\bar{\sigma} = \bigcup_{i=1}^{n}[\vec{t_i}/\vec{x_i}] \cup \bigcup_{i=1}^{m}[\vec{t_i}/\vec{y_i}]$ such that
(11) $prog, B \models (q_1(\vec{y_1}) \wedge \ldots \wedge q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}))\bar{\sigma}$
By subcase 1, we know that there is some $\sigma \geq \sigma_p$ such that $(c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}))\sigma$

Let $\sigma' = \sigma \cup \bar{\sigma}$
Then $\sigma' \geq \sigma_p$ is a satisfying substitution for $c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \neg c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m})$
we arrive at a contradiction of (9)

subcase 3: Line 19 of CkPropDC returns "valid" each time

Since the test on Line 13 of CkPropDC passes, therefore
Some q_i does not appear in d

The test on Line 15 of CkPropDC fails, therefore
(12) $c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k})$ is unsat

We can derive a σ''' which satisfies (12)
By (3), σ satisfies $c_p(\vec{x_1}, \ldots, \vec{x_n})$
By (5), $\bigsqcup_{i=1}^{n} \sigma_i[dom(\sigma_i)/\vec{z_{di}}]$ satisfies $\bigwedge_{i=1}^{n} c_i(\vec{x_{ci}})$
By (7), $\models c_{net}(\vec{z_1}, \ldots, \vec{z_k})\sigma_{net}$
Define $\sigma''' = \sigma \cup \bigsqcup_{i=1}^{n} \sigma_i[dom(\sigma_i)/\vec{z_{di}}] \cup \sigma_{net}$
Then $\models (c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}))\sigma'''$
which is a contradiction of (12)

subcase 4: Line 19 of CkPropDC returns "valid" each time
By Line 2 of CkPropDC,
There is some σ_p such that $\models (c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}))\sigma_p$
By Line 36 of CkPropDC,
there is no $\sigma^{(4)}$ such that
(13) $\models (c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}))\sigma^{(4)}$
By (7), $\models c_{net}(\vec{z_1}, \ldots, \vec{z_k})\sigma_{net}$
We can construct $\sigma^{(4)}$ by taking
$\sigma^{(4)} = \sigma_p \cup \sigma_{net}$

Which contradicts (13)

Proof of 2.
By assumption
CkProp returns invalid

By CkProp and Correctness of Derivation Pool (Lemma 1),
(1) there exists substitution σ_p such that
$prog, B \vDash (p_1(\vec{x_1}) \wedge \ldots \wedge p_n(\vec{x_n}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}))\sigma_p$

subcase 1: Line 19 returns "valid"
(2) one of q_i is not found in the derivation d
The test on Line 15 of CkPropDC fails, thus
(3) $c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k})$ is sat

By (2),
there is no σ_q such that
(4) $prog, B \vDash (q_1(\vec{y_1}) \wedge \ldots \wedge q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}))\sigma_q$

By (3),
There is some σ_p such that
(5) $\vDash (c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}))\sigma_p$
By conjunction elimination of (5),
(6) $\vDash (c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}))\sigma_p$

By conjunction elimination of (5),
(7) $\vDash \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k})\sigma_p$
By (7),
(8) $\vDash (b_1(\vec{z_1}) \wedge \cdots \wedge b_k(\vec{z_k}) \supset \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k})\sigma_p$
By (8),
exists B such that
$B \vDash \varphi_{net}$

By (4), (6), (8),
$prog, B \nvDash \varphi$

subcase 2: Line 37 of CkPropDC returns "invalid"

The test on Line 13 of CkPropDC passed
Every unification of q_i is found in derivation d

The test on Line 27 of CkPropDC passed
(8) $\bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \neg c_q(\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m})$ is sat

The test on Line 36 of CkPropDC passed
(9) $\bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}$ is sat
By (9),
There is some σ_b such that
$\bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}) \wedge \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}$ is sat
By conjunction elimination,
(10) $\vDash (\bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}))\sigma_b$
By (10),
(11) $\vDash (b_1(\vec{z_1}) \wedge \cdots \wedge b_k(\vec{z_k}) \supset \bigwedge_{i=1}^{k} c_{net}(\vec{z_1}, \ldots, \vec{z_k}))\sigma_b$
By (10) and (11),
exists B such that
(12) $B \vDash \varphi_{net}$

By (8),

(13) There is no σ_p such that $\models c_q\sigma_p$

(14) There is some σ_p such that $\models (\bigwedge_{i=1}^{n} c_i(\vec{x_{ci}}) \wedge c_p(\vec{x_1}, \ldots, \vec{x_n}))\sigma_p$

By (12), (13), (14),

$prog, B \not\models \varphi$

$\square$

```
function GenDs(𝒢, dpool, (nID, p : τ))
    Δ ← {}
    for each rule with ID rID where (rID,nID) in 𝒢 do
        Δ ← Δ∪GenDRule(𝒢, dpool, (nID, p : τ),rID)
    if (nID, p : τ) is on a cycle then
        (* gather all constraints *)
        (x⃗, c) ← Ex_DisJ(Δ)
        if A(nID, p : τ) = c_A then
            (* check annotation *)
            if Check sat ¬(c_A[x⃗/fv(c_A)] ⇔ c) then
                return annotation_error
            else
                return (c_A, Δ)
        else
            return (c, Δ)
    else
        return Δ
end function

function LookUp(dpool, q(x⃗))
    if q ∈ A then
        (y⃗, c_A) ← A(q)
        return (y⃗/x⃗, c_A, (rec, q(y⃗)) :: nil)
    else
        if dpool(q) = Δ then
            return List.Map (extractD x⃗) Δ
        elsedpool(q) = (c, Δ)
            y⃗ ← fv(Δ)
            return (y⃗/x⃗, c, (rec, q(y⃗)) :: nil)
end function
```

Figure 7: Construct derivation pools for recursive programs

4 Extension to Recursive Programs

The dependency graph for a recursive program contains cycles. The derivation pool construction algorithm presented in Figure 2 does not work for recursive programs because it relies on the topological order of nodes in the dependency graph. We need to augment our data structures and algorithms to handle recursive programs.

4.1 Derivation Pool for Recursive Predicates

When p is recursively defined, *dpool* maps p to a pair (c, Δ), where Δ has the same meaning as before. The additional constraint c is an invariant of p: c is satisfiable if and only if p is derivable.

$$\begin{array}{llll} \textit{Constraint pool} & \textit{dpool} & ::= & \cdots \mid \textit{dpool}, (\textit{nID}, p{:}\tau) \mapsto (c, \Delta) \\ \textit{Derivation} & \mathcal{D} & ::= & \cdots \mid (\mathsf{rec}, p(\vec{x})) \\ \textit{Annotation} & A & ::= & \cdot \mid A, (\textit{nID}, p{:}\tau) \mapsto (\vec{x}, c) \end{array}$$

Derivation trees include a new leaf node $(\mathsf{rec}, p(\vec{x}))$, where p appears on a cycle in the dependency graph. This leaf node is a place holder for the derivation of the recursive predicate p.

We write A to denote annotations for recursive predicates. It is provided by the user. A maps a predicate p to a pair $(\vec{x}, c)$, where $\vec{x}$ is the arguments of p and c is the constraint which is satisfiable if and only if p is derivable.

```
1: function RemoveEdges(P, E, G)
2:     remove outgoing edges of nID from E
3:     for each rID with no edges of form (_, rID) in E do
4:         remove edges (rID, nID) from E
5:         if (nID, p : τ) has no incoming edges in E then
6:             add (nID, p : τ) to P
7:         if every (nID′, q : τ′) s.t. (nID′, rID), (rID, nID) ∈ E, (nID′, q : τ′) in A then
8:             add (nID, p : τ) to P
9: end function
10:
11: function Ex_Disj(Δ)
12:     if Δ = nil then return ({}, ⊤)
13:     else
14:         ((c1, (rID, p(y⃗), dl)), Δ′) ← Δ
15:         (_, c2) ← Ex_Disj(Δ′, Γ)
16:         return (y⃗, ∃(fv(c1)\y⃗).c1 ∨ c2)
17: end function
```

Figure 8: Helper functions for constructing derivation pool entries for recursive predicates

The structure of the derivation pool construction remains the same. We highlight the changes in Figure 7. The main difference is that now when a cycle is reached, the annotations are used to break the cycle. The working set P, which contains the set of nodes that can be processed next, includes not only predicate nodes that do not have incoming edges, but also include nodes that depend on only body tuples that have annotations. Consider the following scenario: Rule $r1$ derives p and has two body tuples q_1 and q_2. Let's assume that there is no edge from q_1 to $r1$, as q_1 has been processed and q_2 has an annotation in A. In this case, we will place p in the working set. The above mentioned change is encoded in the new RemoveEdges function (lines 7-8) in Figure 7.

The second change is in constructing derivation pool entries for a predicate p. In the non-recursive case, each derivation tree of a predicate p corresponds to the application of a rule to the list of derivation trees for the body tuples of that rule. In the recursive case, if one of the body tuples, say q, is on a cycle, when we process p, q's entries in *dpool* have not been constructed. However, the constraint under which q can be derived is given in the annotation A. In this case, we use $(\mathsf{rec}, q(\vec{x}))$ as a place holder for derivations for q, and use the constraint in A as the constraint for this derivation. The change is reflected in the LookUp function for collecting possible derivations of the body predicates (lines 21-23).

Finally, annotations need to be verified. The GenDs function checks the correctness of the annotations after all the predicates have been processed (lines 5-15). For a recursive predicate, the derivation pool maps it to a summary constraint and a list of possible derivations (a pair (c, Δ)). The requirement of the summary constraint for p is that it has to be satisfiable if and only if there is at least one derivation for the recursive predicate p. That is, this summary constraint has to be logically equivalent to the disjunction of the constraints associated with all possible derivations of p in Δ. We consider two cases for a predicate on a cycle of the dependency graph: (1) there is an annotation for it in A and (2) there is no annotation. For both cases, we need to collect all the possible constraints for deriving p from Δ. Function EX_Disj in Figure 8 computes the disjunction of constraints in Δ. Each constraint is existentially quantified over the arguments that do not appear in p. For case (1), we need to check that the annotation is logically equivalent to the disjunction of the constraints for all possible derivations of p (Lines 20). If this is the case, then the annotated constraint together with Δ is returned; otherwise, an error is returned, indicating that the invariant doesn't hold. For case (2), we return the disjunctive formula returned by EX_Disj (Lines 15). When p is not recursive, only Δ is returned (line 17).

4.2 Property Query

We use the same property query algorithm for non-recursive program. This obviously has limitations, because the derivations of recursive predicates are not expanded. The imprecision of the analysis comes from the following two sources. The first is that derivations represented as $(\mathsf{rec}, p(\vec{x}))$ may contain predicates needed by the antecedent of the property (the q_is in φ). Without expanding these derivations, the algorithm may report that φ is violated because q_is cannot be found, even though this is not the case in reality. The second is that network constraints cannot be accurately checked. When we find a suitable derivation d that contains all the q_is such that c_q holds, checking the network constraints on d requires us to expand $(\mathsf{rec}, p(\vec{x}))$s in d. The algorithm may report that the property holds, even though, the witness it finds does not satisfy the network constraints. Similarly, when the algorithm reports that the property does not hold, the counterexample may not satisfy the network constraints. For the analysis to be precise, we would need annotations for recursive predicates to provide invariants for recursive predicates. Our case studies do not require annotations. Expanding the algorithm to handle recursive predicates precisely remains our future work.

function CKPROPC($dpool$, φ_{net}, φ)
    (* P *is* $p_1 \cdots p_n$ *and* Q *is* $q_1 \cdots q_m$ *)
    $(P, c_p, Q, c_q) \leftarrow \varphi$
    (* B *is* $b_1 \cdots b_n$, *where* $b_i s$ *are base tuples* *)
    $(B, c_b) \leftarrow \varphi_{net}$
    (* *get the list of list of derivations for* $p_1, \cdots, p_n$ *)
    $L \leftarrow$ LOOKUP($dpool$, P)
    (* *combine all possible derivations for* $p_1 \cdots p_n$
        Each entry in D *also include substitutions that replace*
        free variables in p_i *with the variable in the derivation* *)
    $D \leftarrow$MERGEDERIVATION L
    for each (σ, c, d) in D **do**
        $z \leftarrow$CKPROPDC(c,$c_p\sigma$, d, Q, $c_q\sigma$, B, $c_b\sigma$)
        if $z = \mathsf{invalid}(d)$ **then**
            return $\mathsf{invalid}(d)$
    return valid
end function

Figure 9: Top-level property function with network constraints

4.3 Correctness

Similar to the non-recursive case, we prove the correctness of derivation pool construction and the query algorithm. Because derivations of recursive predicates are summarized as $(\mathsf{rec}, p(\vec{x}))$, the correctness of the derivation pool construction needs to consider the unrolling of $(\mathsf{rec}, p(\vec{x}))$. First, we define a relation $dpool \vdash d, \sigma \rightsquigarrow_k d', \sigma'$ to mean that a derivation d with the substitution σ can be expanded using derivations in $dpool$ to another derivation d' of depth k and a new substitution σ'.

$$\frac{\sigma' \geq \sigma}{dpool \vdash (\mathsf{BT}, p(\vec{x})), \sigma \rightsquigarrow_0 (\mathsf{BT}, p(\vec{x})), \sigma'}$$

$$\frac{\forall j \in [1, n], dpool \vdash d_j, \sigma \rightsquigarrow_k d'_j, \sigma'}{dpool \vdash (rID, p(\vec{x}), d_1 :: \cdots d_n :: \mathsf{nil}), \sigma \quad \rightsquigarrow_{k+1} (rID, p(\vec{x}), d'_1 :: \cdots d'_n :: \mathsf{nil}), \sigma'}$$

$$\frac{dpool(p) = (c, \Delta) \qquad (c_i, d_{pi}) \in \Delta \qquad \vDash c_i\sigma' \qquad dpool \vdash d_{pi}, \sigma' \rightsquigarrow_k d'_{pi}, \sigma''}{dpool \vdash (\mathsf{rec}, p(\vec{x})), \sigma \rightsquigarrow_k d'_{pi}, \sigma''}$$

The first rule applies to the base tuples. Here, no unrolling is needed and the depth of the derivation is 0. The second rule unrolls the premises of a derivation d. The depth of d' is $k+1$. The last rule is the key rule that unrolls the derivation of recursive predicate p $((\mathsf{rec}, p(\vec{x})))$ using one of the possible derivations of p from Δ. Here, the unrolling can only use the derivation in Δ, whose constraint can be satisfied.

Lemma 5 shows that the derivation pool construction algorithm is correct with respect to an unrolling of the derivation. If a predicate p is derivable, then the derivation pool should have an entry for p that can be unrolled into that derivation. In the other direction, for every entry in the derivation pool, it either unrolls into a finite derivation, or can be further unrolled. This lemma allows the unrolling to be infinite.

Lemma 5 (Correctness of derivation pool construction (recursive))**.**
GENDPOOL($prog$, A) $= dpool$
where `rid p(u) :- q1(u1),...,qn(un),c(u1,...,un)`

1. *If $prog, B \vDash d{:}p(\vec{t})$*

   (a) *either p is not on a cycle in the dependency graph and exists σ and $(c, d' : p(\vec{x})) \in dpool(p)$ s.t. $dpool \vdash d', \sigma \rightsquigarrow_{|d|} d_1, \sigma,\ d = d_1\sigma$ and $\vDash c\sigma$.*

   (b) *or p is on a cycle in the dependency graph and exists σ s.t. $dpool \vdash (\mathsf{rec}, p(\vec{x})), \sigma \rightsquigarrow_{|d|} d_1, \sigma,\ d_1\sigma = d$ and $\vDash c_p\sigma$.*

2. (a) *If $(c, d{:}p(\vec{x})) \in dpool(p)$ and $\vDash c\sigma$, then $\forall n, \exists m, m \le n$, $dpool \vdash d, \sigma \rightsquigarrow_m d', \sigma'$, either d' does not contain $(\mathsf{rec}, q(\vec{y}))$, and exists B, s.t. $prog, B \vDash d'\sigma' : p(\vec{x})\sigma'$ or d' contains $(\mathsf{rec}, q(\vec{y}))$, and replacing all of the $(\mathsf{rec}, q(\vec{y}))$ derivations with a derivation of $q\sigma'$ in d' results in a derivation for $p(\vec{x})\sigma'$*

   (b) *If $(c_p, \Delta : p(\vec{x})) \in dpool(p)$ and $\vDash c_p\sigma$ then $\forall n, \exists m, m \le n$, $dpool \vdash (\mathsf{rec}, p(\vec{x})), \sigma \rightsquigarrow_m d', \sigma'$, either d' does not contain $(\mathsf{rec}, q(\vec{y}))$, and exists B, s.t. $prog, B \vDash d'\sigma' : p(\vec{x})\sigma'$ or d' contains $(\mathsf{rec}, q(\vec{y}))$, and replacing all of the $(\mathsf{rec}, q(\vec{y}))$ derivation with a derivations of $q\sigma'$ in d' results in a derivation for $p(\vec{x})\sigma'$*

Proof.
1. Proof by induction of the structure of d

Base case: $d = (\mathsf{BT}, p(\vec{t}))$

CASE (A): p is not on a cycle in $\mathcal{G}$
By Line 8 of GENDPOOL,
  $(\top, (\mathsf{BT}, p(\vec{x}))) \in dpool(p)$
By Line 6 of GENDPOOL,
  $\vec{x}$ are fresh variables for the arguments of p
Choose $\sigma = \{\}$
Choose $\sigma' = [\vec{t}/\vec{x}]$
Since $\sigma' \ge \sigma$
  $dpool \vdash ((\mathsf{BT}, p(\vec{x})), \sigma) \rightsquigarrow_0 ((\mathsf{BT}, p(\vec{x})), \sigma')$
  $(\mathsf{BT}, p(\vec{t})) = (\mathsf{BT}, p(\vec{x}))\sigma'$
  $\vDash \top\sigma'$

CASE (B): p is not on a cycle since it is a base tuple

Inductive case: $d = (rID, p(\vec{t}), (d_1{:}q_1(\vec{t_1}))::\ldots::(d_n{:}q_n(\vec{t_n}))::\mathsf{nil})$

It would be more accurate to write $c(\vec{t_c})$ where $\vec{t_c} \subseteq \{\vec{t}, \vec{t_1}, \cdots, \vec{t_n}\}$
However, we write $c(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$ for clarity when performing substitutions
  (1) $\vDash c(\vec{t}, \vec{t_1}, \cdots, \vec{t_n})$

By Inductive Hypothesis, for each $d_i{:}q_i(\vec{t_i})$, where $1 \le i \le n$,
  (2) exists σ_i where $\sigma_i = [\vec{t_{di}}/\vec{x_{di}}]$ such that
    either $(c_i(\vec{x_{ci}}), d'_i(\vec{x_{di}}){:}q_i(\vec{x_i})) \in dpool(q_i)$, $(d'_i(\vec{x_{di}}), \sigma_i) \rightsquigarrow_k (d'_i(\vec{x_{di}}), \sigma_i)$, $d'_i(\vec{x_{di}})\sigma_i = d_i$, $\vDash c_i(\vec{x_{ci}})\sigma_i$
    or $(c_i(\vec{x_{ci}}), \Delta_i(\vec{x_{\Delta i}}){:}q_i(\vec{x_i})) \in dpool(q_i)$, $((\mathsf{rec}, q_i(\vec{x_i})), \sigma_i) \rightsquigarrow_k (d'_i(\vec{x_{di}}), \sigma_i)$, $d'_i(\vec{x_{di}})\sigma_i = d_i$, $\vDash c_i(\vec{x_{ci}})\sigma_i$
    where $\vec{x_i} \subseteq \vec{x_{di}}$, $\vec{x_i} \subseteq \vec{x_{ci}}$, $\vec{x_{ci}} \subseteq \vec{x_{di}}$, $\vec{x_{di}} \subseteq \vec{x_{\Delta i}}$
    and $\vec{t_i} \subseteq \vec{t_{di}}$
  By Freshness Lemma (Lemma 2),
    (3) $\vec{z_{d1}}, \ldots, \vec{x_{dn}}$ is fresh

By GENDPOOL, GENDRULE function
  (4) $(c(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}), (rID, p(\vec{z}), d'_1(\vec{z_{d1}}){:}q_1(\vec{z_1})::\ldots::d'_n(\vec{z_{dn}}){:}q_n(\vec{z_n})::\mathsf{nil})) \in dpool$
    will be returned as a possible derivation of p
      where $\vec{z_i} \subseteq \vec{z_{di}}$, $\vec{z_i} \subseteq \vec{z_{ci}}$, $\vec{z_{ci}} \subseteq \vec{z_{di}}$
      and $\vec{z} \subseteq \{\vec{z_{d1}}, \ldots, \vec{z_{dn}}\}$
      and d'_i is $(\mathsf{rec}, q_i(\vec{z_i}))$ if q_i is on a cycle
  By Freshness Lemma (Lemma 2),
    (5) $\vec{z_{d1}}, \ldots, \vec{z_{dn}}$ is fresh

By (3) and (5), we know that x_{di} and z_{di} are fresh.
Define

(6) $\sigma = \bigsqcup_{i=1}^{n} \sigma_i[\mathsf{dom}(\sigma_i)/\vec{z_{di}}]$
$= \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{x_{di}}][\mathsf{dom}([\vec{t_{di}}/\vec{x_{di}}])/\vec{z_{di}}]$
$= \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{x_{di}}][\vec{x_{di}}/\vec{z_{di}}]$
$= \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}]$
where $\vec{z} \subseteq \{\vec{z_{d1}}, \ldots, \vec{z_{dn}}\}$

By (1), we know that $\vDash c(\vec{t}, \vec{t_1}, ..., \vec{t_n})$
By (6), we have $\sigma = \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}]$
(7) $\vDash c(\vec{z}, \vec{z_1}, ..., \vec{z_n})\sigma$
By (2), we know that $\vDash c_i(\vec{x_{ci}})\sigma_i$
By (6), we have $\sigma = \bigsqcup_{i=1}^{n} [\vec{t_{di}}/\vec{z_{di}}]$
(8) $\bigwedge_{i=1}^{n} c_i(\vec{z_i})\sigma$
(9) $(d'_i(\vec{z_{di}}), \sigma) \rightsquigarrow_k (d'_i(\vec{z_{di}}), \sigma)$
By (7) and (8),
(10) $\vDash (c(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_i}))\sigma$

Assume p is not on a cycle
By the definition of $\rightsquigarrow$
(11) $\mathit{dpool} \vdash (d'_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d'_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}, \sigma) \rightsquigarrow_{k+1} (d'_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d'_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}, \sigma)$
By (9), (10), (11),
the conclusion holds

Now assume p is on a cycle
By Function LookUp,
(12) $\mathit{dpool}(p) = (c_p(\vec{z_p}), \Delta_p(z_{\vec{\Delta}p}))$
By (4),
$(c(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}}), (\mathit{rID}, p(\vec{z}), d'_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d'_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil})) \in \Delta_p(z_{\vec{\Delta}p})$
Using (2),
Applying the last $\rightsquigarrow$ rule,
(13) $\mathit{dpool} \vdash ((\mathsf{rec}, p(\vec{z})), \sigma) \rightsquigarrow_k (d'_1(\vec{z_{d1}}){:}q_1(\vec{z_1}){::}\ldots{::}d'_n(\vec{z_{dn}}){:}q_n(\vec{z_n}){::}\mathsf{nil}, \sigma)$
By the above,
the conclusion holds

2. Proof by induction on n.

Base case $n = 0$
trivially true since base tuples are not on cycles by definition

Inductive case $n = k + 1$

Subcase (a)
$(c_p(\vec{z_p}), (\mathit{rID}, p(\vec{z}), (d'_1{:}q_1(\vec{z_1})){::}\ldots{::}(d'_n{:}q_n(\vec{z_n})){::}\mathsf{nil} : p(\vec{z}))) \in \mathit{dpool}(p)$
where $\vec{z_i} \subseteq \vec{z_{di}}$, $\vec{z_i} \subseteq \vec{z_{ci}}$, $\vec{z_{ci}} \subseteq \vec{z_{di}}$

By GenDPool and GenDRule function, for $1 \leq i \leq n$,
(1) $(c_i(x_{ci}), d'_i(x_{di}){:}q_i(\vec{x_i})) \in \mathit{dpool}(q_i)$,
or $(c_i(x_{ci}), \Delta_i(x_{\vec{\Delta}i}){:}q_i(\vec{x_i})) \in \mathit{dpool}(q_i)$
where $\vec{x_i} \subseteq \vec{x_{di}}$, $\vec{x_i} \subseteq \vec{x_{ci}}$, $\vec{x_{ci}} \subseteq \vec{x_{di}}$, $\vec{x_{di}} \subseteq x_{\vec{\Delta}i}$
By Freshness Lemma (2),
$\vec{x_{d1}}, \ldots, \vec{z_{dn}}$ are fresh

By GenDPool,
(2) $c_p(\vec{z_p}) = c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}})$
It would be more accurate to write $c(\vec{z_c})$ where $\vec{z_c} \subseteq \{\vec{z}, \vec{z_1}, \cdots, \vec{z_n}\}$

However, we write $c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n})$ for clarity when performing substitutions
By Freshness Lemma (2)
$\vec{z}, \vec{z_{c1}}, \ldots, \vec{z_{cn}}$ are fresh

By assumption,
$\models c_p(\vec{z_p})\sigma$
which is equal to saying that
(3) $\models c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n})\sigma$

By (1) and (3), for $1 \leq i \leq n$,
$\models c_i(\vec{x_{ci}})[\vec{z_{di}}/\vec{x_{di}}]\sigma$

By Inductive Hypothesis, for $1 \leq i \leq n$,
EITHER (I) d_i does not contain any rec nodes
exists B_i such that
(4) $\mathit{prog}, B_i \models d'_i(\vec{x_{di}})\sigma_i{:}q_i(\vec{x_i})\sigma_i$
Applying the the second rule of $\rightsquigarrow$,
(5) $\mathit{prog}, \bigcup_{i=1}^{n}(B_i\sigma) \models (d'_1(\vec{x_{d1}}){::}\ldots{::}d'_n(\vec{x_{di}}){::}\mathsf{nil})\sigma{:}p(\vec{x})\sigma$
Therefore

By Inductive Hypothesis, for $1 \leq i \leq n$,
OR (II) $\{d_{j_1}, \ldots, d_{j_w}\} \subset \{d_1, \ldots, d_k\}$ contain a rec node
there exists (6) $\mathit{dpool} \vdash ((\mathsf{rec}, q_{j_\ell}(y_{j_\ell})), \sigma) \rightsquigarrow_k (d'_{j_\ell}(\vec{y_{d\ell}}), \sigma)$
For elements in $\{d_1, \ldots, d_k\} \backslash \{d_{j_1}, \ldots, d_{j_w}\}$,
(7) exists B_i such that $\mathit{prog}, B_i \models d'_i(\vec{y_{di}})\sigma_i{:}q_i(\vec{y_i})\sigma_i$
Applying the the second rule of $\rightsquigarrow$, and using (6) and (7),
$\mathit{dpool} \vdash ((\mathsf{rec}, p(\vec{x})), \sigma) \rightsquigarrow_{k+1} (d'_1(\vec{x_{d1}}){::}\ldots{::}d'_n(\vec{x_{dn}}){::}\mathsf{nil}, \sigma)$

Subcase (b) $(c_p(\vec{z_p}), \Delta(\vec{z_{\Delta p}})) \in \mathit{dpool}(p)$

By Assumption,
(8) $\models c_p(\vec{z_p})\sigma$
By GenDPool,
(9) $c_p(\vec{z_p}) = c_r(\vec{z}, \vec{z_1}, \ldots, \vec{z_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{z_{ci}})$
where $\vec{z_i} \subseteq \vec{z_{ci}}$
It would be more accurate to write $c(\vec{z_c})$ where $\vec{z_c} \subseteq \{\vec{z}, \vec{z_1}, \cdots, \vec{z_n}\}$
However, we write $c(\vec{z}, \vec{z_1}, \cdots, \vec{z_n})$ for clarity when performing substitutions
By Freshness Lemma (2)
$\vec{z_{c1}}, \ldots, \vec{z_{cn}}$ are fresh
Therefore we can define
(10) $\sigma = \bigsqcup_{i=1}^{n}[\vec{t_{ci}}/\vec{z_{ci}}]$
By (8) and (9), we have
(11) $\models c_r(\vec{t}, \vec{t_1}, \ldots, \vec{t_n}) \wedge \bigwedge_{i=1}^{n} c_i(\vec{t_{ci}})$

By GenDs checks summary constraint for recursive predicate on Line 5
(12) exists $\sigma' \geq \sigma$
where $\sigma' = \bigsqcup_{i=1}^{n}[\vec{t_{di}}/\vec{z_{di}}]$
$(c_i(\vec{z_{ci}}), d'_i(\vec{z_{di}})) \in \Delta(\vec{z_{\Delta p}})$ and
$\models c_i(\vec{z_{ci}})\sigma'$

For each $1 \leq i \leq n$,
Apply I.H. on k,
(13) $d'_i(\vec{x_{di}})\sigma'$ is a derivation of $p(\vec{x_i})\sigma'$

Apply the second rule of $\rightsquigarrow_k$ to (13) to obtain
(14) $\mathit{dpool} \vdash ((\mathit{rID}, p(\vec{z}), d'_1(\vec{z_{d1}}){::}\ldots{::}d'_n(\vec{z_{dn}}){::}\mathsf{nil}), \sigma') \rightsquigarrow_{k+1} ((\mathit{rID}, p(\vec{z}), d'_1(\vec{z_{d1}}){::}\ldots{::}d'_n(\vec{z_{dn}}){::}\mathsf{nil}), \sigma')$

Apply the last rule of $\rightsquigarrow_k$ to (14)

(15) $\mathit{dpool} \vdash ((\mathsf{BT}, p(\vec{z})), \sigma') \rightsquigarrow_{k+1} ((rID, p(\vec{z}), d_1'(\vec{z_{d1}})::\ldots::d_n'(\vec{z_{dn}})::\mathsf{nil}), \sigma')$

The conclusion holds.

□

As we discussed in Section 4.2, we cannot show a general correctness theorem without annotations for recursive predicates. We can only prove the soundness of the algorithm when there is no network constraint.

Lemma 6 (Soundness of property query)**.**
$\varphi = \forall\vec{x_1}, p_1(\vec{x_1}) \wedge \forall\vec{x_2}, p_2(\vec{x_2}) \cdots \forall\vec{x_k}, p_k(\vec{x_k}) \wedge c_p(\vec{x_1}, \cdots \vec{x_k}) \supset$
$\exists\vec{y_1} q_1(\vec{y_1}) \wedge \cdots \wedge \exists\vec{y_m} q_m(\vec{y_m}) \wedge c_q(\vec{x_1}, \cdots \vec{x_k}\vec{y_1}, \cdots \vec{x_m})$ $\textsc{GenDPool}(\mathit{prog}, A) = \mathit{dpool}$*,* $\textsc{CkProp}(\mathit{dpool}, \varphi) =$ yes *implies* $\mathit{prog} \vDash \varphi$.

5 Case Studies

In this section, we conduct case studies of our tool by applying it to software-defined networking (SDN), an emerging networking technique that allows network administrators to program the network through well-defined interfaces (e.g. OpenFlow protocol [35]). SDNs intentionally separate the control plane and the data plane of the network.

A centralized controller is introduced to monitor and manage the whole network. The controller provides an abstraction of the network to network administrators, and establishes connections with underlying switches.

Recently, declarative programming languages have been introduced to SDN to write controller applications that configure the network [37]. Like any program, these applications are not guaranteed to be bug-free. We show the effectiveness of our tool in validating/debugging several SDN applications. We demonstrate that the tool can unveil different problems in the process of SDN application development, ranging from software bugs, incomplete topological constraints and incorrect property specification. All verifications in our case study are completed within one second.

5.1 Verification process

We first provide a high-level description of the verification process. When analyzing a property, the user is expected to provide three types of inputs: (1) formal specification of the property in accordance with the format requirement of our framework; (2) formal specification of initial network constraints, such as topological constraints and switch default setup; and (3) formal specification of invariants on recursive tuples.

Out tool accepts the above user specifications along with the NDLog program as inputs. It first checks the correctness of the invariants on recursive tuples. After invariants are validated, the tool runs the main algorithm for verification, and outputs either "True" if the property holds, or "False" if the property is not valid. For invalid properties, the tool also generates a concrete counter example to help the programmer debug the program.

5.2 Ethernet Source Learning

The first case study we consider is Ethernet source learning, which allows switches in a network to remember the location of end hosts through incoming packets. More specifically, three kinds of entities are deployed in the network: (1) **end hosts** (servers or desktops) at the edge of the network that send packets to the network through connected switches, (2) **switches** that forward a packet if the packet matches a flow entry in the forwarding table, or relay the packet to the controller for further instruction if there is a table miss, and (3) **a controller** that connects to all switches in the network. The controller learns the position of an end host through packets relayed from a switch, and installs a corresponding flow entry in the switch for future forwarding.

Figure 11 presents then NDLog encoding of Ethernet Source Learning ($prog_{ESL}$). Table 1 lists the safety properties for the program that we looked for.

Encoding In a typical scenario, an end host initiates a packet and sends it to the switch it connects (rh1). The switch recursively looks up its forwarding table to match against the received packet (rs1, rs2). If a flow entry matches the packet, the packet is forwarded to the port indicated by the "Action" part of the entry (rs3). Otherwise, the switch wraps the packet in an OpenFlow message, and relays it to the controller for further instruction (rs5). On receiving the OpenFlow message, the controller first extracts the location information of the source address in the packet (the OpenFlow message registers incoming port for each packet), and installs a flow entry matching the source address in the switch (rc1). The controller then instructs the switch to broadcast the mis-matched packet to all its neighbors other than the upstream neighbor who sent the packet (rc2). Rules rs5 and rs6 specify the reaction of the switch corresponding to Rules rc1 and rc2 respectively — the switch either inserts a flow entry into the forwarding table (rs5) or broadcasts the packet (rs6) as instructed.

Network constraints We inject the following basic network constraints when verifying properties. The constraints enforce the topology on which we run Ethernet source learning. We demand that an end host always initiates packets using its own address as source, and the switch it connects to cannot be the source or the destination (constraints on initPacket). In addition, the controller cannot share addresses with switches (constraints on ofconn). And a switch cannot have a link to itself (constraints on single swToHst). Also, each switch should have only one link connecting the neighbor host, and no two hosts can connect to the same port of a switch (constraints on any two swToHsts). The network constraints are given below.

Property	Property description	Formal Specification	Result
φ_{ESL_1}	If the switch has a routing entry for a host with MAC address A, it has received a packet sourced from that host in the past.	$\forall Switch, Mac, OutPort, Priority,$ $\mathsf{flowEntry}(Switch, Mac, OutPort, Priority)$ $\wedge Mac = A \supset$ $\exists Nei, DstMac,$ $\mathsf{packet}(Switch, Nei, Mac, DstMac)$	true
φ_{ESL_2}	If an EndHost has received a packet that is not destined for its MAC address, then the switch does not have a routing entry for that EndHost's MAC address.	$\forall EndHost, Switch, SrcMac, DstMac, InPort,$ $OPort, Outport, Mac, Priority,$ $\mathsf{packet}(EndHost, Switch, SrcMac, DstMac)$ $\wedge \mathsf{swToHst}(Switch, EndHost, OPort)$ $\wedge \mathsf{flowEntry}(Switch, Mac, Outport, Priority)$ $\wedge DstMac \neq EndHost \supset$ $Mac \neq DstMac$	false
φ_{ESL_3}	If EndHost has received a packet destined for it, then the switch has a flow entry for the EndHost.	$\forall EndHost, Switch, SrcMac, DstMac, OPort,$ $\mathsf{packet}(EndHost, Switch, SrcMac, DstMac)$ $\wedge \mathsf{swToHst}(Switch, EndHost, OPort)$ $\wedge DstMac = EndHost \supset$ $\exists Switch', Mac, Outport, Priority,$ $\mathsf{flowEntry}(Switch', Mac, Outport, Priority)$ $\wedge Switch' = Switch \wedge Mac = DstMac$	false
φ_{ESL_4}	If the switch has a flowEntry for a host with mac address Mac, then there has been a flow table miss in the past for that particular host	$\forall Switch, Mac, Outport, Priority,$ $\mathsf{flowEntry}(Switch, Mac, Outport, Priority) \supset$ $\exists Switch', SrcMac, DstMac, InPort, Priority,$ $\mathsf{matchingPacket}(Switch', SrcMac, DstMac, InPort, Priority')$ $\wedge Switch' = Switch \wedge SrcMac = Mac$ $\wedge InPort = Outport \wedge Priority' = 0$	true

Table 1: Results of checking safety properties of $prog_{ESL}$ on our tool

$$
\begin{array}{ll}
\varphi^{ESL}_{net_1} & \mathsf{initPacket}(Host, Switch, Src, Dst) \supset \\
& \quad Host \neq Switch \wedge Host = Src \wedge \\
& \quad Host \neq Dst \wedge Switch \neq Dst. \\
\varphi^{ESL}_{net_2} & \mathsf{ofconn}(Controller, Switch) \supset \\
& \quad Controller \neq Switch. \\
\varphi^{ESL}_{net_3} & \mathsf{swToHst}(Switch, Host, Port) \supset \\
& \quad Switch \neq Host \wedge Switch \neq Port \wedge Host \neq Port. \\
\varphi^{ESL}_{net_4} & \mathsf{swToHst}(Switch1, Host1, Port1) \wedge \\
& \mathsf{swToHst}(Switch2, Host2, Port2) \supset \\
& \quad (Switch1 = Switch2 \wedge Host1 = Host2 \supset \\
& \qquad Port1 = Port2) \wedge \\
& \quad (Switch1 = Switch2 \wedge Port1 = Port2 \supset \\
& \qquad Host1 = Host2).
\end{array}
$$

Verification results We verify a number of properties that are expected to hold in a network running the Ethernet Source Learning program. We discuss two properties that generate counter examples in detail. A summary of all the properties we verified can be found in Table 9.

The first property specifies that whenever an end host receives a packet not destined to it, the switch it connects have no matching flow entry for the destination address in the packet. Formally:

$$\forall EndHost, Switch, SrcMac, DstMac, InPort,$$
$$OPort, Outport, Mac, Priority,$$
$$\mathsf{packet}(EndHost, Switch, SrcMac, DstMac)$$
$$\wedge \mathsf{swToHst}(Switch, EndHost, OPort)$$
$$\wedge \mathsf{flowEntry}(Switch, Mac, Outport, Priority)$$
$$\wedge DstMac \neq EndHost \supset$$
$$Mac \neq DstMac$$

Though this property is seemingly true, our tool returns a negative answer, along with a counterexample shown in Figure 12. The counter example reveals a scenario where an endhost (H4) receives a broadcast packet destined to another machine (H3) (Execution trace (1) in Figure 12), but the switch it connects to (S1) has a flowEntry that matches the destination MAC address in the packet (Execution trace (2) in Figure 12).

In the counter example, switch S1 receives a packet $\langle Src : H6, Dst : H3\rangle$ through port 2 from the upstream switch S2 (①). Since S1 does not have a flow entry for the destination address H3, it relays the packet wrapped in an OpenFlow message (i.e. ofPacket) to the controller C1(②). The controller then instructs S1 to broadcast the packet to all neighbors except S2 (③). However, before Server H4 receives the broadcast packet, a new packet $\langle Src : H3, Dst : H4\rangle$ could reach switch S1(④), triggering an ofPacket message to the controller (⑤). The controller would then set up a new flow entry at switch S1, matching on destination H3 (⑥,⑦). It is possible that due to network delay, server H4 receives its copy of the broadcast packet just now(⑧). Therefore, the execution trace generates packet (H4,S1,H6,H3), swToHst (S1,H4,1) (i.e. the link between S1 and H4), and flowEntry (S1,H3,2,1), with $Mac == DstMac$ ($H3 = H3$).

Our tool also generates a counterexample for another seemingly correct property. This second property specifies that whenever an end host receives a packet destined to it, the switch it connects to has a flowEntry matching the end host's MAC address. Formally:

$$\forall EndHost, Switch, SrcMac, DstMac, OPort,$$
$$\mathsf{packet}(EndHost, Switch, SrcMac, DstMac)$$
$$\wedge \mathsf{swToHst}(Switch, EndHost, OPort)$$
$$\wedge DstMac = EndHost \supset$$
$$\exists Switch', Mac, Outport, Priority,$$
$$\mathsf{flowEntry}(Switch', Mac, Outport, Priority)$$
$$\wedge Switch' = Switch \wedge Mac = DstMac$$

The generated counter example shown in Figure 14 shows that a packet could reach the correct destination by means of broadcast — a corner case that can be easily missed with manual inspection. In the counter example, switch S1 receives a packet destined to server H4(①). Since there is no flow entry in the forwarding table to match the destination address, switch S1 informs the controller of the received packet (②), and further broadcasts the packet under the controller's instruction (③). In this way, server H4 does receive a packet destined to it (④). but switch S1 does not have a flow entry matching H4.

With further inspection, we found that the above counter examples, in essence, are attributed to incorrect specification of network properties, rather than bugs in the programs. In the first case, a stricter property would specify that a received broadcast message indicates an *earlier* table miss. While in the second one, the property fails to consider the possibility of specific broadcast messages in the execution. We further discuss the implication of these counter examples with another counter example produced in the firewall case study.

5.3 Firewall

We also use our tool to verify properties of a stateful firewall. A stateful firewall is usually deployed at the edge of a corporate network to filter untrusted packets from the Internet. Compared to a stateless firewall, which makes decision purely based on specific fields of a packet, a stateful firewall allows richer access control depending on flow history. For example, the firewall can allow traffic from an outside end host to reach machines inside the local domain only if the communication was initiated by the internal machines. We implement a SDN-based stateful firewall, which can set up filtering policies under the instruction of the controller. The controller registers traffic traversal information and installs appropriate filtering entries.

Our firewall case study is based on a program from [8] that has been modified slightly to test our counterexample genration process. We present our NDLog implementation of the program ($prog_{WeakFW}$) in Figure 13. Key tuples generated at each node executing the program are listed in Table 4. We summarize the program in Table 5. The firewall forwards traffic from trusted hosts in the local domain without interference (r1), and also notifies the controller of the destination address in the packet (r2). When the firewall receives a packet from the Internet,

Predicate	Description
ofconn(@*Controller*, *Switch*)	*Controller* is able to communicate with *Switch*
ofPacket(@*Controller*, *Switch*, *InPort*, *SrcMac*, *DstMac*)	*Switch* does not have a hit in its flow entry table for a packet that appeared on it, send by host with mac address *SrcMac*, to target host with mac address *DstMac*. Therefore, *Switch* forwarded the packet to *Controller* to ask it how to proceed.
flowMod(@*Switch*, *SrcMac*, *InPort*)	Controller generates and sends this tuple to switch Switch to allow it to install host with mac address *SrcMac* into its flow entry table.
matchingPacket(@*Switch*, *SrcMac*, *DstMac*, *InPort*, *Priority*)	A packet that appeared on switch *Switch* via port *InPort*, from host with mac address *SrcMac*, with target host of mac address *DstMac*, and priority *Priority*
packet(@*OutNei*, *Switch*, *SrcMac*, *DstMac*)	*OutNei* received a packet from *Switch* that was sent by a host with mac address *SrcMac* to a target host with mac address *DstMac*
swToHst(@*Switch*, *OutNei*, *OutPort*)	*Switch* is connected to *OutNei* via port *OutPort*
hstToSw(@*Host*, *Switch*, *OutPort*)	*Host* is connected to switch *Switch* via port *OutPort*
maxPriority(@*Switch*, *TopPriority*)	packets arriving on *Switch* have a priority of at most *TopPriority*, where a larger priority number indicates greater urgency
initPacket(@*Host*, *Switch*, *SrcMac*, *DstMac*)	*Host* with mac address *SrcMac* sends out a packet to a target host with mac address *DstMac* to *Switch*
recvPacket(@*Host*, *SrcMac*, *DstMac*)	*Host* with mac address *DstMac* has received a packet address to it, which was sent out by host with mac address *SrcMac*

Table 2: Predicates in φ_{ESL}

it relays the packet to the controller for further decision (r4). If the source address was once registered at the controller, the controller would install a flow entry in the firewall (r5), allowing packets of the same flow to access the internal domain in the future (r3).

This is realized as follows. Two types of hosts are connected to a switch: (i) trusted hosts (within the organization) via port 1; and (ii) untrusted hosts (outside the organization) via port 2. Packets from trusted hosts are always forwarded to untrusted hosts. Packets from untrusted hosts are forwarded to trusted hosts only if the source host has previously received a packet from a trusted host. The auxiliary relation tr records the trusted hosts for each switch. We use bold font to denote OpenFlow commands. The program is executed in an infinite loop with two type of events: pktIn events that are annotated with commands, and pktFlow events whose semantics is determined by the current content of the flow table [8].

We check the property φ_{WeakFW} (shown below), which states that the destination of a packet received by an trusted machine must be trusted by the controller. Our tool finds a counterexample for it.

$$\begin{array}{l}\forall Host, Port, Src, SrcPort, Switch,\\ \quad \mathsf{pktReceived}(Host, Port, Src, SrcPort, Switch) \supset\\ \quad \exists Controller,\\ \quad\quad \mathsf{trustedControllerMemory}(Controller, Switch, Src)\end{array}$$

The network constraints for this modified version of firewall are shown below. They are the same as those given in firewall, but with an additional link tuple.

Role	Rule	Summary
Controller	rc1	Controller installs a flow entry on the switch to match on the source address of the incoming packet
	rc2	Controller instructs the switch to broadcast the unmatching packet to all neighbors except the upstream neighbor
Switch	rs1	Receives a new packet and starts address look-up in the local flow table
	rs2	Recursively matches the packet with each flow entry
	rs3	If a matching is found for the packet, forwards the packet accordingly
	rs4	If no flow entry matches the packet, relays the packet to the controller for further inspection
	rs5	Updates the local flow table under the instruction of the controller
	rs6	Broadcasts a packet under the instruction of the controller
End Host	rh1	Initializes a packet and sends it to the connected switch
	rh2	Receives a packet from the connected switch

Table 3: Summary of φ_{ESL} encoding

$$
\begin{array}{ll}
\varphi^{WeakFW}_{net_1} & \mathsf{connection}(Switch, Controller) \supset \\
& \quad Switch \neq Controller \\
\varphi^{WeakFW}_{net_2} & \mathsf{pktIn}(Switch, Src, SrcPort, Dst) \supset \\
& \quad Switch \neq Src \wedge Switch \neq SrcPort \\
& \quad \wedge Switch \neq Dst \wedge Src \neq SrcPort \\
& \quad \wedge Src \neq Dst \wedge SrcPort \neq Dst \\
\varphi^{WeakFW}_{net_3} & \mathsf{pktIn}(Switch1, Src1, SrcPort1, Dst1) \\
& \wedge\ \mathsf{pktIn}(Switch2, Src2, SrcPort2, Dst2) \\
& \wedge\ Switch1 \neq Switch2 \wedge Src1 = Src2 \supset \\
& \quad SrcPort1 = SrcPort2 \\
\varphi^{WeakFW}_{net_4} & \mathsf{link}(Switch, Dst, PortDst) \supset \\
& \quad Switch \neq Dst \wedge Switch \neq PortDst \\
& \quad \wedge\ Dst \neq PortDst
\end{array}
$$

$$
\begin{array}{ll}
\varphi^{WeakFW}_{net_5} & \mathsf{link}(Switch1, Dst1, PortDst1) \\
& \wedge\ \mathsf{link}(Switch2, Dst2, PortDst2) \supset \\
& \quad (Switch1 = Switch2 \wedge Dst1 = Dst2 \\
& \qquad \supset PortDst1 = PortDst2) \\
& \quad \wedge\ (Switch1 = Switch2 \wedge PortDst1 = PortDst2 \\
& \qquad \supset Dst1 = Dst2)
\end{array}
$$

```
#define TRUSTED_PORT 1
#define UNTRUSTED_PORT 2

/* a packet from a trusted host via TRUSTED_PORT
 * appeared on switch without a forwarding rule
 * Forward packets from all trusted sources
 */
r1 pktReceived(@Dst, Uport, Src, Tport, Switch) :-
  pktIn(@Switch, Src, Tport, Dst),
  link(@Switch, Dst, Uport),
  Tport == TRUSTED_PORT.

r2 trustedControllerMemory(@Controller,
                           Switch, Dst) :-
  pktIn(@Switch, Src, Tport, Dst),
  connection(@Switch, Controller),
  Tport == TRUSTED_PORT.

/*
 * a packet from with a forwarding rule appears
 * on the switch Forward according to the rule
 * The packet may be from a trusted/untrusted source
 */
r3 pktReceived(@Dst, PortDst, Src,
               PortSrc, Switch) :-
  pktIn(@Switch, Src, PortSrc, Dst),
  link(@Switch, Dst, PortDst),
  perFlowRule(@Switch, Src, PortSrc, Dst).


/*
 * Packet from untrusted host appeared on
 * switch Send it to the controller to check
 * if it is trusted
 */
r4 pktFromSwitch(@Controller, Switch,
                 Src, Uport, Dst) :-
  pktIn(@Switch, Src, Uport, Dst),
  connection(@Switch, Controller),
  Uport == UNTRUSTED_PORT.

r5 perFlowRule(@Switch, Src, Uport, Dst) :-
  pktFromSwitch(@Controller, Switch, Src, Uport, Dst),
  trustedControllerMemory(@Controller, Switch, Src),
  Uport == UNTRUSTED_PORT,
  Tport := TRUSTED_PORT.
```

Figure 13: NDLog implementation of $prog_{WeakFW}$

Verification results We verify a number of properties about the stateful firewall. We discuss one example here; the property specifies that source destinations of all packets reaching internal machines are trusted by the controller:

$$\begin{array}{l}\varphi_{WeakFW} = \\ \quad \forall Host, Port, Src, SrcPort, Switch, \\ \quad\quad \mathsf{pktReceived}(Host, Port, Src, SrcPort, Switch) \supset \\ \quad \exists Controller, \\ \quad\quad \mathsf{trustedControllerMemory}(@Controller, Switch, Src)\end{array}$$

Surprisingly, our tool gives a counterexample for this property (Figure 15), which depicts the scenario that an internal machine H3 sends a packet to another internal machine H4 in the same domain through the firewall F1. Because the controller C1 never registers local machines, the property is violated.

In spite of its simplicity, we find the counterexample interesting, because it can be interpreted in different ways; each corresponds to a different approach to fixing the problem. The counterexample can be viewed as a revelation of a program bug. The programmer can add a patch to the program and re-verify the property over the updated program. Alternatively, the counterexample could be linked to incomplete specification of network constraints that internal machines should never send internal traffic to the firewall. The fix would then be to insert extra constraints over base tuples of the program. In addition, the problem could also stem from the property specification, since

Predicate	Description
pktReceived(@*Dst*, *DstPort*, *Src*, *SrcPort*, *Switch*)	*Dst* has received a packet via the Switch through port *DstPort*, that was originally send by host *Src* through port *SrcPort*
pktIn(@*Switch*, *Src*, *SrcPort*, *Dst*)	A packet sent by host *Src* through port *SrcPort* with target host *Dst* appeared on the switch
trustedControllerMemory(@*Controller*, *Switch*, *Host*)	*Controller* stores a link between *Switch* an (untrusted) *Host*.
connection(@*Switch*, *Controller*)	There is a connection between *Switch* and *Controller*
perFlowRule(@*Switch*, *Src*, *SrcPort*, *Dst*, *DstPort*)	*Switch* stores in its memory that untrusted host *Src* is allowed to send packets to trusted host *Dst*
pktFromSwitch(@*Controller*, *Switch*, *Src*, *SrcPort*, *Dst*)	*Switch* asks *Controller* to check if untrusted host *Src* is allow to send a packet to host *Dst*
link(@*Switch*, *Dst*, *PortDst*)	*Switch* is linked to *Dst* via *PortDst*

Table 4: Tuples for $prog_{WeakFW}$

Rule	Summary
r1	a packet from a trusted host, with a destination host whose trustworthiness is unknown, appeared on switch without a forwarding rule. Forward the packet to the destination host regardless.
r2	A packet from a trusted host appeared on switch without a forwarding rule. Insert the target host *Dst* of the packet into trusted controller memory.
r3	A packet from with a forwarding rule appears on the switch, which forwards it according to its flow table
r4	A packet from an untrusted host appeared on switch, which sends it to the controller to check if it can forward the packet to its intended destination
r5	Controller checks a packet originally sent by an untrusted host, found that there is a previous link between that untrusted host and the switch, and tells the switch that it can forward the packet by inserting a per flow rule into the switch for that untrusted host

Table 5: Summary of $prog_{WeakFW}$ encoding

users may only care about traffic from outside the domain. In this case, we can change the property specification, to specify that if a packet is from an *external* machine, then the source address must be registered at the controller before. In real deployment, it is up to the programmer to decide which interpretation is most appropriate.

5.4 Load Balancing

The third case study we examine is load balancing. When receiving packets to a specific network service (e.g. web page requests), a typical load balancer splits the packets on different network paths to balance traffic load. The strategies for load balancing are various, e.g. static configuration or congestion-based adjustment. In our case study, we implement a load balancer which load balances traffic towards a specific destination address, and determines the path of a packet based on the hash value of its source address.

Figure 17 presents our implementation of load balancer implemented in NDLog ($prog_{LB}$). Key tuples generated at each node executing the program are listed in Table 7. We summarize the program in Table 6.

When the load balancer receives a packet, it first inspects its destination address. If the destination address matches the address that the load balancer is responsible for, the load balancer would generate a hash value of the source destination of the packet (r1). The hash value is used to select the server to which the packet should be routed. The load balancer replaces the original destination address in the packet with the address of the selected server, and forwards the packet to the server (r2). In addition, the load balancer has a default rule that forwards traffic not destined to the designated address without interference (r3).

We demonstrate that the load balancer could potentially break flow affinity property (i.e. packets received on different servers cannot share the same source address). When a machine is connected to two load balancers and sends packets of the same flow to both of the load balancers, one of the flow will match on the default rule of a load balancer and may potentially be directed to a different server. The property we are trying to check is:

Event	Rule	Summary
Initialize Packets	r1	A load balancer receives a packet that a client has sent out.
A packet appearing on a load balancer is destined to the load balancer's designated server	r2	A load balancer has received a packet to be sent to its designated destination. It hashes the source and uses that result modulo the number of servers to get a number corresponding to a server.
	r3	The load balancer matches the integer obtained by hashing to obtain a server to send the packet to.
Packet appearing on a load balancer is not to be sent to its designated server	r4	The load balancer forwards the packet directly to the destination as prescribed by the packet.

Table 6: Summary of $prog_{LB}$ encoding

$$\begin{aligned}&\forall Server1, Server2, Src1, Src2,\\&\quad \mathsf{recvPacket}(Server1, Src1, ServiceAddr)\\&\quad \wedge \mathsf{recvPacket}(Server2, Src2, ServiceAddr)\\&\quad \wedge Server1 \neq Server2 \supset\\&\qquad Src1 \neq Src2\end{aligned}$$

A counterexample is shown in figure 16.

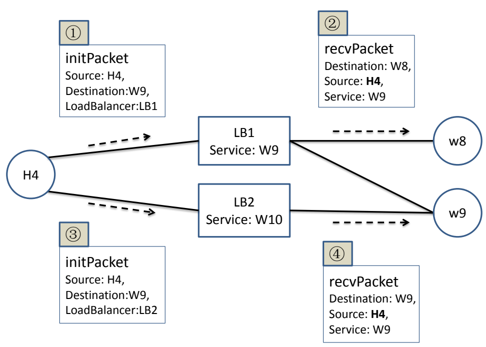

Figure 16: A counter example for property φ_{LB}

The network constraints are the following:

$$\begin{aligned}&\varphi^{LB}_{net_1} && \mathsf{initPacket}(v1, v2, v3) \supset\\& && \quad v1 \neq v2 \wedge v2 \neq v3 \wedge v1 \neq v3\\&\varphi^{LB}_{net_2} && \mathsf{designated}(v4, v5) \supset\\& && \quad v4 \neq v5\\&\varphi^{LB}_{net_3} && \mathsf{designated}(v9, v10) \wedge \mathsf{designated}(v11, v12)\\& && \wedge\, v9 = v11 \supset\\& && \quad v10 = v12\\&\varphi^{LB}_{net_4} && \mathsf{serverMapping}(v6, v7, v8) \supset\\& && \quad v6 \neq v7 \wedge v7 \neq v8 \wedge v6 \neq v8\end{aligned}$$

$$\begin{aligned}&\varphi^{LB}_{net_5} && \mathsf{serverMapping}(v13, v14, v15)\\& && \wedge\, \mathsf{serverMapping}(v16, v17, v18)\\& && \wedge\, v13 = v16 \wedge v14 = v17 \supset\\& && \quad v15 = v18\\&\varphi^{LB}_{net_6} && \mathsf{serverMapping}(v13, v14, v15)\\& && \wedge\, \mathsf{serverMapping}(v16, v17, v18)\\& && \wedge\, v13 = v16 \wedge v15 = v18 \supset\\& && \quad v14 = v17\end{aligned}$$

initPacket(@*Client*, *Server*, *LoadBalancer*)	*Client* sends out a packet to *LoadBalancer* with intended destination *Server*.
packet(@*LoadBalancer*, *Client*, *Server*)	*LoadBalancer* received a packet from *Client* that has destination *Server*
designated(@*LoadBalancer*, *DesignatedDst*)	For packets arriving on *LoadBalancer* with destination address *DesignatedDst*, *LoadBalancer* determines it path of a packet based on the hash value of its source address.
hashed(@*LoadBalancer*, *Client*, *ServerNum*, *Server*)	*LoadBalancer* had received a packet whose destination address matches the address that it is responsible for. *LoadBalancer* generates a hash value of the source address of *Client* to obtain an integer *ServerNum*. *ServerNum* is uniquely mapped to *Server*, to which the packet is to be routed.
serverMapping(@*LoadBalancer*, *Server*, *ServerNum*)	*LoadBalancer* stores the bijective mappings of each destination server to a unique number, *ServerNum*
recvPacket(@*Server*, *Client*, *ServiceAddr*)	*Server* has received a packet from source *Client* via *LoadBalancer*.

Table 7: Tuples for $prog_{LB}$

Verification result The property that we verify for load balancing is called flow affinity, that is, if two servers receives packets requesting the same service—which means the packets share the same initial destination address—the source addresses of the packets must be different.

The property does not hold in the given protocol specification, and a counterexample is given by our tool. In the counterexample, two load balancers responsible for different network service could co-exist in the network, and if a server sends packets to both load-balancers, requesting the same service, it is possible that the packets are routed to different servers.

Similar to the case in the firewall, the programmer can fix the counter example of the load balancer by patching the program, adding network assumption (e.g. assuming no server is connected to two load-balancers), or changing property specification (e.g. "load-balanced packets that are forwarded out of different ports of the load balancer do not share the same source address").

5.5 Address Resolution Protocol

The final case study we focus on is the Address Resolution Protocol (ARP) in an Ethernet network. End hosts use ARP to request the destination MAC address corresponding to an IP address they want to communicate to. Traditionally, the ARP requests are broadcast through the domain. In our case study, we replace the broadcast with a centralized controller that answers ARP requests.

Figure 18 presents an implementation of our NDLog encoding of SDN-based ARP ($prog_{ARP}$). Key tuples generated at each node executing the program are listed in Table 10. We summarize the program in Table 8.

In a typical execution, an end host initiates an ARP request and broadcasts it to the network (rh1). When a switch receives the packet, it informs the controller of the ARP request (rs1). The controller first remembers the location of the requested end host (rc1). Then it registers the mapping between the source IP address and source MAC address as indicated in the request for future address resolution (rc2, rc3). After this, the controller looks up the destination IP address in the mapping table (rc4). If the mapping is found, the controller generates an ARP reply message, and instructs the relaying switch to sends back the message (rc5). The switch will reply to the end host as instructed (rs2), so that the end host gets the correct destination MAC address and finishes ARP (rh2).

The purpose of Address Resolution Protocol (ARP) is to find out the MAC address of a device in a Local Area Network (LAN). When a source device want to communicate with another device, source device checks its Address Resolution Protocol (ARP) cache to find it already has a resolved MAC Address of the destination device. If it is there, it will use that MAC Address for communication. If ARP resolution is not there in local cache, the source machine will generate an Address Resolution Protocol (ARP) request message and broadcast it to the LAN. The message is received by each device on the LAN since it is a broadcast. Each device compare the Target Protocol Address (IPv4 Address of the machine to which the source is trying to communicate) with its own Protocol Address (IPv4 Address). Those who do not match will drop the packet without any action.

Role	Rule	Summary
Host	rh1	Host sends an ARP request to a switch that is directly connected to it
	rh2	Host receives an ARP reply from the connected switch and stores the message
Controller	rc1	Receives an ARP request and registers the location of the source address
	rc2	Receives an ARP request and extracts core information related to address resolution
	rc3	Record the mapping between source IP address and source MAC address
	rc4	Looks up destination IP address of the ARP request in the local ARP cache, and generates an ARP reply packet to answer to request
	rc5	Wraps the ARP reply packet inside an OpenFlow message instructing the switch to relay the ARP reply back to the requesting host
Switch	rs1	Receives an ARP request message and relays it to the controller for address resolution
	rs2	Follows the controller's instruction and relays the ARP reply back to the requesting host

Table 8: Summary of $prog_{ARP}$ encoding

Property	Property description	Formal Specification	Result
φ_{ARP_1}	If any controller sends an ARP response for IP address IP_A, then some end host had sent a broadcast ARP request message for IP_A.	$\forall Controller, IP_A, Mac_A, DstIP, DstMac,$ $\mathsf{arpReplyCtl}(Controller, IP_A, Mac_A, DstIP, DstMac) \supset$ $\exists Qmac,$ $\mathsf{arpRequest}(Host, DstIp, DstMac, IP_A, Qmac)$ $\wedge\ Qmac = 255$	true
φ_{ARP_2}	If any controller has a map between IP address IP_A and MAC address Mac_A, then host A has sent a broadcast ARP request.	$\forall Controller, IP_A, Mac_A,$ $\mathsf{arpMapping}(Controller, IP_A, Mac_A) \supset$ $\exists Host, SrcIP, SrcMac, DstIP, DstMac,$ $\mathsf{arpReply}(Host, IP_A, Mac_A, DstIp, DstMac)$ $\wedge\ DstMac = 255$	true

Table 9: Results of checking safety properties of $prog_{ARP}$ on our tool

Verification results We verify a number of safety properties on ARP, and all these properties prove to be true. The detailed results can be found in Table 9.

5.6 Discussion

We conclude the case studies by briefly discussing the experience and insights that we obtain when performing the case studies.

Cause of property violation The counter examples we discuss above reveal a common pattern: when a predicate in the program has multiple derivations, proving properties over the predicate becomes harder. The situation is even worse when a property involves multiple predicates, each with multiple derivations. The increased complexity of predicate derivations makes it error-prone for human programmers to write correct programs or specify correct properties, and serves as the core cause of property violation. Naturally, the fixes we proposed for counter examples generally fall into two categories: (1) enriching the property specification to include the missing derivations, or (2) changing the program to remove the uncovered derivations.

Iterative application development Another observation is that reasonable network assumption (e.g. topological constraints) helps prune scenarios that would not appear in actual executions, and generate insightful counter examples. For example, a counter example may suggest a topology where a switch has a link to itself. A programmer may start with trivial network assumptions and let the tool guide the exploration of corner cases and gradually add (implicit) network assumptions that are not obvious to the programmer. In fact, our tool enables the programmer to *iteratively* develop applications. The generated counter examples could help the programmer understand (1) applicable domain of the program (feedback of missing network constraints); (2) implementation correctness (feedback of bugs in the program); and/or (3) expected behavior of the program (feedback of incorrect

Predicate	**Description**
packet(@$Switch$, $Host$, $DstMac$, $DstIp$, $SrcMac$ $SrcIp$, $Arptype$)	$Switch$ has received an ARP message of $Arptype$ (Request/Reply) from $Host$. The message is from ($SrcMac$, $SrcIp$) to ($DstMac$, $DstIp$).
packetIn(@$Controller$, $Switch$, $InPort$, $DstMac$, $DstIp$ $SrcMac$, $SrcIp$, $Arptype$)	Initializes the packet above.
linkHst(@$Host$, $Switch$, $Port$)	$Host$ is connected to $Switch$ via $Port$
linkSwc(@$Switch$, $Host$, $InPort$)	$Switch$ is connected to $Host$ via $InPort$
arpRequest(@$Host$, $SrcIp$, $SrcMac$, $DstIp$, $DstMac$)	An ARP request message at @$Host$, querying the corresponding MAC address of $DstIp$, $SrcIp$ and $SrcMac$ represent the IP address and MAC address of $Host$. $DstMac$ uses broadcast address in Ethernet.
hostPos(@$Controller$, $SrcIp$, $Switch$, $InPort$)	The controller registers the information that the host with Source IP SrcIp is connected the port InPort of Switch.
ofconnCtl(@$Controller$, $Switch$)	$Controller$ has a connection to $Switch$
arpMapping(@$Controller$, $SrcIp$, $SrcMac$)	$Controller$ remembers that the host of IP address $SrcIp$ has the MAC address $SrcMac$.
arpReqCtl(@$Controller$, $SrcIp$, $SrcMac$, $DstIp$, $DstMac$)	An ARP request message at Controller, querying the corresponding MAC address of $DstIp$, from the host with IP address $SrcIp$ and MAC address $SrcMac$.
arpReplyCtl(@$Controller$, $DstIp$, $DstMac$, $SrcIp$, $SrcMac$)	An ARP reply message answering $SrcMac$ of $SrcIp$ to the host with IP address $DstIp$ and MAC address $DstMac$,
packetOut(@$Switch$, $Controller$, $Port$, $DstMac$, $DstIp$, $SrcMac$, $SrcIp$, $Arptype$)	An OpenFlow message sent from Controller to Switch, to send an ARP packet of type Arptype from $SrcIp$, $SrcMac$ to $DstIp$, $DstMac$
flowEntry(@$Switch$, $Arptype$, $Prio$, $Actions$)	A flow entry of priority Prio at Switch that applies Actions to packets of type Arptype.

Table 10: Tuples for $prog_{ARP}$

property specification). After the programmer fix the problem, s/he can redo the verification repeatedly until the specified property holds.

```
/*Controller program*/
/*Install rules on switch*/
rc1 flowMod(@Switch, SrcMac, InPort) :-
  ofconn(@Controller, Switch),
  ofPacket(@Controller, Switch, InPort, SrcMac, DstMac).

/*Instruct the switch to send out the unmatching packet*/
rc2 broadcast(@Switch, InPort, SrcMac, DstMac) :-
  ofconn(@Controller, Switch),
  ofPacket(@Controller, Switch, InPort, SrcMac, DstMac).


/*Switch program*/
/*Query the controller when receiving unknown packets */
rs1 matchingPacket(@Switch, SrcMac, DstMac, InPort, TopPriority) :-
  packet(@Switch, Nei, SrcMac, DstMac),
  swToHst(@Switch, Nei, InPort),
  maxPriority(@Switch, TopPriority).

/*Recursively matching flow entries*/
rs2 matchingPacket(@Switch, SrcMac, DstMac, InPort, NextPriority) :-
  matchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
  flowEntry(@Switch, MacAdd, OutPort, Priority),
  Priority > 0,
  DstMac != MacAdd,
  NextPriority := Priority - 1.

/*A hit in flow table, forward the packet accordingly*/
rs3 packet(@OutNei, Switch, SrcMac, DstMac) :-
  matchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
  flowEntry(@Switch, MacAdd, OutPort, Priority),
  swToHst(@Switch, OutNei, OutPort),
  Priority > 0,
  DstMac == MacAdd.

/*If no flow matches, send the packet to the controller*/
rs4 ofPacket(@Controller, Switch, InPort, SrcMac, DstMac) :-
  ofconn(@Switch, Controller),
  matchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
  Priority == 0.

/*Insert a flow entry into forwarding table*/
/*(TODO): We assume all flow entries are independent, which is not general*/
rs5 flowEntry(@Switch, DstMac, OutPort, Priority) :-
  flowMod(@Switch, DstMac, OutPort),
  ofconn(@Switch, Controller),
  maxPriority(@Switch, TopPriority),
  Priority := TopPriority + 1.

/*TODO: should be a_MAX<Priority> in the head tuple*/
rs6 maxPriority(@Switch, Priority) :-
  flowEntry(@Switch, DstMac, OutPort, Priority).

/*Following the controller's instruction, send out the packet as broadcast*/
rs7 packet(@OutNei, Switch, SrcMac, DstMac) :-
  broadcast(@Switch, InPort, SrcMac, DstMac),
  swToHst(@Switch, OutNei, OutPort),
        OutPort != InPort.


/*Host program*/
/*Packet initialization*/
rh1 packet(@Switch, Host, SrcMac, DstMac) :-
  initPacket(@Host, Switch, SrcMac, DstMac),
  hstToSw(@Host, Switch, OutPort).

/*Receive a packet*/
rh2 recvPacket(@Host, SrcMac, DstMac) :-
  packet(@Host, Switch, SrcMac, DstMac),
  hstToSw(@Host, Switch, InPort).
```

Figure 11: NDLog implementation of $prog_{ESL}$

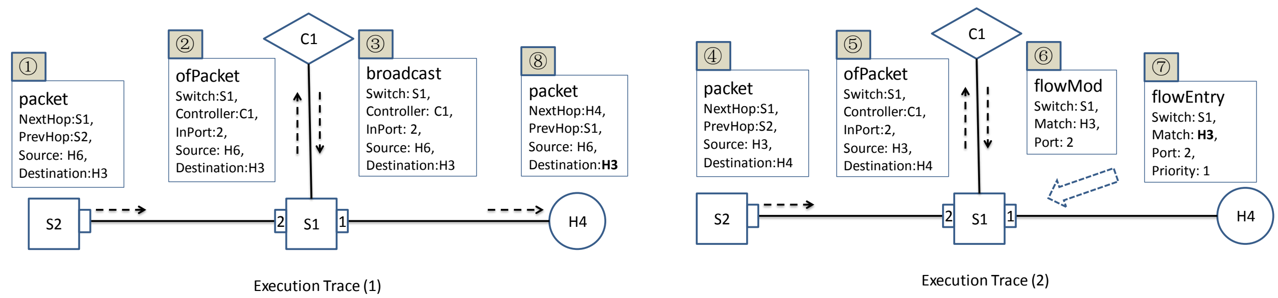

Figure 12: A counter example for property φ_{ESL_2}

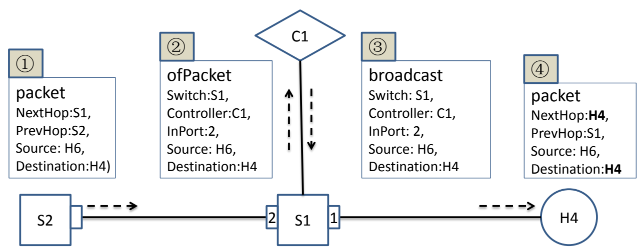

Figure 14: A counter example for property φ_{ESL_3}

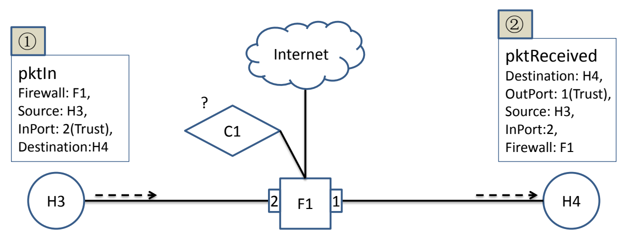

Figure 15: A counter example for property φ_{WeakFW}

```
/* total number of possible servers that the
 * load balancers can send a packet to */
#define NUM_SERVERS 5


/* Initialize Packets*/

r1 packet(@LoadBalancer, Client, Server) :-
  initPacket(@Client, Server, LoadBalancer).


/* Packet appearing on LoadBalancer is to be
 * sent to its designated server */

r2 hashed(@LoadBalancer, Client, ServerNum, Server) :-
    packet(@LoadBalancer, Client, Server),
  designated(@LoadBalancer, DesignatedDst),
  DesignatedDst == Server,
  Value := f_hashIp(Client),
  ServerNum := 1+f_modulo(Value, NumServers),
  NumServers := NUM_SERVERS.

r3 recvPacket(@Server, Client, ServiceAddr) :-
  hashed(@LoadBalancer, Client, ServerNum, ServiceAddr),
  serverMapping(@LoadBalancer, Server, ServerNum).


/* Packet appearing on LoadBalancer is NOT to be
 * sent to its designated server */

r4 recvPacket(@Server, Client, Server) :-
  packet(@LoadBalancer, Client, Server),
  designated(@LoadBalancer, DesignatedDst),
  Server != DesignatedDst,
  ServiceAddr := Server.
```

Figure 17: NDLog implementation of $prog_{LB}$

```
/* constants */
#define BROADCAST "ff:ff:ff:ff:ff"
#define ALL_PORT 0
#define ARP_TYPE "ARP"
#define IPV4_TYPE "IPV4"
#define CONTROLLER "controller"
#define ARP_REQUEST 1
#define ARP_REPLY 2
#define ARP_PRIO 1

/* Host program */

// Send ARP request to directly connected switch
rh1 packet(@Switch, Host, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  linkHst(@Host, Switch, Port),
  arpRequest(@Host, SrcIp, SrcMac, DstIp, DstMac),
  Host == SrcIP, Arptype := ARP_REQUEST, DstMac == BROADCAST.

// Received packet from switch and extract ARP reply packets
rh2 arpReply(@Host, SrcIp, SrcMac, DstIp, DstMac) :-
  linkHst(@Host, Switch, Port),
  packet(@Host, Switch, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  Arptype == ARP_REPLY, Type == ARP_TYPE, DstMac == Host.

/* Controller program */

// Register host position
rc1 hostPos(@Controller, SrcIp, Switch, InPort) :-
  ofconnCtl(@Controller, Switch),
  packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  Arptype == ARP_REQUEST, DstMac == BROADCAST.

// Recover ARP request
rc2 arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac) :-
  packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  ofconnCtl(@Controller, Switch), Arptype == ARP_REQUEST.

// Learn ARP mapping
rc3 arpMapping(@Controller, SrcIp, SrcMac) :-
  arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac).

// Generate ARP reply
rc4 arpReplyCtl(@Controller, DstIp, Mac, SrcIp, SrcMac) :-
  arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac),
  arpMapping(@Controller, DstIp, Mac).

// Send out packet_out message
rc5 packetOut(@Switch, Controller, Port, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  arpReplyCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac),
  ofconnCtl(@Controller, Switch),
  hostPos(@Controller, DstIp, Switch, Port), Arptype := ARP_REPLY.

/*Switch program*/

rs1 packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  ofconnSwc(@Switch, Controller),
  packet(@Switch, Host, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  linkSwc(@Switch, Host, InPort),
  flowEntry(@Switch, Arptype, Prio, Actions),
  Prio == ARP_PRIO, Actions == CONTROLLER, DstMac == BROADCAST.

rs2 packet(@Host, Switch, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  packetOut(@Switch, Controller, OutPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  linkSwc(@Switch, Host, OutPort), Arptype == ARP_REPLY.
```

Figure 18: NDLog implementation of $prog_{ARP}$

6 Related Work

Network verification. In recent years, formal verification has received much attention in the network community. There has been a cloud of prior work on network verification focusing on several different aspects. One aspect is the verification of network configurations, where the proposed solutions detect network configuration errors either 1) through static analysis of the configuration file [18, 2, 17, 38, 49], or 2) by analyzing snapshots of the data plane—reflecting the aggregate impact of all configurations—during system execution [23, 22, 33, 51]. These solutions rely heavily on application-specific network models and property specifications, which limits its adoption in more general scenarios. The second aspect is to leverage proof-based and model-checking techniques to verify the correctness of both the design and implementation of network protocols [48, 19, 25, 16, 47]. Such solutions often demand participation of system administrators during the verification phase, and require domain-specific expertise. The third aspect focuses on security properties, such as origin and route authenticity properties, in secure networking protocols that use cryptographic primitives [5, 6, 14, 10, 52].

Most closely related to ours is the work on verifying network protocol design using declarative networking [48, 47, 10]. The general approach of the prior work share similarities with the one of ours—both model the network behavior using trace semantics, and properties are specified and verified on the trace-based model. However, the proposed solution in this paper enables automated static analysis of safety properties and generates counterexamples for debugging purposes, whereas the prior work relies on manual proofs and therefore can handle a richer set of properties.

SDN verification. One special case of network verification is SDN verification [8, 9, 24, 1, 21, 41, 46]. For example, VeriCon [8] defines its own special language for modeling SDN controller and switches [8]. A hoare-logic is developed on this language to prove properties of SDN controllers. The proof obligations are translated to constraints and solved by the SMT solver. NICE is a testing tool for SDN controllers written in Python [9]. NICE combines symbolic execution of the controller programs with state-exploration-based model checking. An alternative approach is to verify network configurations generated by SDN controllers in realtime, instead of verifying the protocols directly [24, 33]. For instance, Anteater reduced SDN data plane verification into SAT problems so that SAT solvers can solve them effectively in practice [33].

All of these tools are specially designed to analyze SDN controllers or dataplanes. We use NDLog as the intermediary language for modeling network protocols and software-defined networks. Modeling and verifying SDN controllers is one example application of our analysis; the analysis applies straightforwardly to other network protocols as well. On the other hand, in the current state, we can only check simple safety properties, while VeriCon and NICE can handle more expressive properties.

Verification of declarative programs. Declarative languages have been proposed to model systems in a variety of domains such as networks, mobile agent planning, and algorithms for graph structures (e.g., Network Datalog (Ndlog) [28], MELD [7], Linear Meld [15], Netlog [20], DAHL [32], Dedalus [4]). However, there has been few work on analyzing low-level correctness properties of declarative programs. Notably, Wang et al. [47, 48] developed a proof system for proving correctness properties of networking protocols specified in NDlog, where programs are translated into equivalent first-order logic axioms, that is, all the body tuples are derivable if and only if the head tuple is derivable.

7 Conclusion

In this paper, we presented an automated approach to analyzing and debugging network protocols using declarative networking. By focusing on a specific class of safety properties, we are able to analyze NDLog programs with few annotations. Our algorithm reduces property checking to constraint solving that can be automatically checked by the SMT solver Z3. We prove the correctness of our algorithms and implement a prototype tool on top of RapidNet, a compilation and execution framework for NDLog. As our case studies, we analyzed a number of real-world SDN network protocols and their safety properties. The case study demonstrated that, when a given safety property is violated, our tool can also provide meaning counterexamples to help debug the protocol specification.

Future Work Currently, our tool can only check for safety properties where "bad things haven't happened yet". We want to extend it to be able to check for liveness properties, where "Something good eventually happens".Since we already derive the provenance for head tuples of NDLog rules, another potential area to be explored is provenance related topics.

References

[1] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *SafeConfig*, 2010.

[2] Ehab Al-Shaer and Hazem Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM*, 2004.

[3] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Eurosys*, 2010.

[4] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[5] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. Modeling and verifying ad hoc routing protocols. In *CSF*, 2010.

[6] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. Deciding security for protocols with recursive tests. In *CADE*, 2011.

[7] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *IROS*, 2007.

[8] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI*, 2014.

[9] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A nice way to test openflow applications. In *NSDI*, 2012.

[10] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. In *FORTE*, 2014.

[11] Chen Chen, Lay Kuan Loh, Limin Jia, Wenchao Zhou, and Boon Thau Loo. Automated verification of safety properties of declarative networking programs. Submitted to PPDP, 2015.

[12] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Co-NEXT*, 2010.

[13] David Chiyuan Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The Design and Implementation of a Declarative Sensor Network System. In *SenSys*, 2007.

[14] Véronique Cortier, Jan Degrieck, and Stéphanie Delaune. Analysing routing protocols: four nodes topologies are sufficient. In *POST*, 2012.

[15] Flávio Cruz, Ricardo Rocha, Seth Copen Goldstein, and Frank Pfenning. A linear logic programming language for concurrent programming over graph structures. *TPLP*, 14(4-5):493–507, 2014.

[16] Dawson Engler and Madanlal Musuvathi. Model-checking large network protocol implementations. In *NSDI*, 2004.

[17] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *NDSI*, 2005.

[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[19] Alwyn Goodloe, Carl A. Gunter, and Mark-Oliver Stehr. Formal prototyping in early stages of protocol design. In *WITS*, 2005.

[20] Stéphane Grumbach and Fang Wang. Netlog, a rule-based language for distributed programming. In *PADL*, 2010.

[21] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[22] P Kazemian, M Chan, H Zeng, G Varghese, N McKeown, and S Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[24] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *HotSDN*, 2012.

[25] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[26] Changbin Liu, Ricardo Correa, Harjot Gill, Tanveer Gill, Xiaozhou Li, Shivkumar Muthukumar, Taher Saeed, Boon Thau Loo, and Prithwish Basu. PUMA: Policy-based Unified Multi-radio Architecture for Agile Mesh Networking. In *COMSNETS*, 2012.

[27] Changbin Liu, Ricardo Correa, Xiaozhou Li, Prithwish Basu, Boon Thau Loo, and Yun Mao. Declarative policy-based adaptive mobile ad hoc networking. *IEEE/ACM Trans. Netw.*, 20(3):770–783, 2012.

[28] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[29] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. In *CACM*, 2009.

[30] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[31] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[32] Nuno P. Lopes, Juan A. Navarro, Andrey Rybalchenko, and Atul Singh. Applying prolog to develop distributed systems. In *ICLP*, 2010.

[33] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[34] Yun Mao, Boon Thau Loo, Zachary Ives, and Jonathan M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *Co-NEXT*, 2008.

[35] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[36] Shivkumar C. Muthukumar, Xiaozhou Li, Changbin Liu, Joseph B. Kopena, Mihai Oprea, Ricardo Correa, Boon Thau Loo, and Prithwish Basu. RapidMesh: declarative toolkit for rapid experimentation of wireless mesh networks. In *WINTECH*, 2009.

[37] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[38] Timothy Nelson, Christopher Barratt, Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[39] Network Simulator 3. `http://www.nsnam.org/`.

[40] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining Distributed Logic Programs Incrementally. In *PPDP*, 2011.

[41] P Porras, S Shin, V Yegneswaran, M Fong, M Tyson, and G Gu. A security enforcement kernel for openflow networks. In *HotSDN*, 2012.

[42] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[43] RapidNet. http://netdb.cis.upenn.edu/rapidnet/.

[44] Micah Sherr, Andrew Mao, William R. Marczak, Wenchao Zhou, Boon Thau Loo, and Matt Blaze. A3: An extensible platform for application-aware anonymity. In *NDSS*, 2010.

[45] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT Protocols Under Fire. In *NSDI*, 2008.

[46] R. W Skowyra, A Lapets, A Bestavros, and A Kfoury. Verifiably-safe software-defined networks for cps. In *HiCoNS*, 2013.

[47] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *PADL*, 2009.

[48] Anduo Wang, Boon Thau Loo, Changbin Liu, Oleg Sokolsky, and Prithwish Basu. A Theorem Proving Approach towards Declarative Networking. In *TPHOLs*, 2009.

[49] L Yuan, H Chen, J Mai, C. N. Chuah, Z Su, and P Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *SRSP*, 2006.

[50] Z3. http://z3.codeplex.com/.

[51] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, , and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI*, 2014.

[52] Fuyuan Zhang, Limin Jia, Cristina Basescu, Tiffany Hyun-Jin Kim, Yih-Chun Hu, and Adrian Perrig. Mechanized network origin and path authenticity proofs. In *CCS*, 2014.

[53] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure Network Provenance. In *SOSP*, 2011.

[54] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed Time-aware Provenance. In *VLDB*, 2013.

[55] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *SIGMOD*, 2010.