Automated Verification of Safety Properties of Declarative Networking Programs

Chen Chen
University Of Pennsylvania
chenche@seas.upenn.edu

Lay Kuan Loh
Carnegie Mellon University
lkloh@cmu.edu

Limin Jia
Carnegie Mellon University
liminjia@cmu.edu

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

Boon Thau Loo
University Of Pennsylvania
boonloo@cis.upenn.edu

Abstract

Networks are complex systems that unfortunately 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, where 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 derivation 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 build a prototype tool and demonstrate the effectiveness of the tool in validating and debugging several SDN applications.

Keywords Declarative networking, static analysis

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 [18, 22, 33, 38]. Practical testing tools for finding undesired behavior in protocol implementation have also been proposed [16, 25]. With the emerging technology of software-defined networking (SDN), modeling networks as programmable software has gained unprecedented popularity. Researchers began to apply program verification techniques to the verification of SDNs [8, 9].

PPDP'15, July 14 - 16, 2015, Siena, Italy.
Copyright © 2015 ACM. ISBN 978-1-4503-3516-4/15/07$15.00.
DOI:http://dx.doi.org/10.1145/2790449.2790516

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. Declarative networking [31] is one of the first research efforts 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 [4], sensor networks [13], overlay network compositions [34], anonymity systems [44], mobile ad-hoc networks [27, 36], wireless channel selection [26], network configuration management [12], and forensic analysis [53–55]. 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 [10, 47, 48]. 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 (soundness and completeness) of our algorithms for non-recursive programs and proved the soundness of our algorithms for recursive programs.
- 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.

Due to space constraints, we omit many technical details. They can be found in our companion technical report [11].

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 [29], and the Chord distributed hash table in 47 lines of code [28]. When compiled and executed, these NDLog programs 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 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 `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, `X` can reach `Y` with cost `C1+C2`. Rule `d3` is similar to `d2`.

As our driving example, we will use the following *erroneous* program. The following non-recursive set of rules computes one-, two-, and three-hop reachability information within a network. There is an error in rule `r2`, where `onehop X Z C2` should be `onehop Z Y C2`, thus this program cannot derive three-hop paths.

THREEHOPS *(With a deliberate error in r2)*:
```
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 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.
```

Properties We focus on safety properties, which state that bad things have not happened yet. We use trace-based semantics of NDLog [10, 40]. 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 is 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 properties to be the following:

$$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$$
$$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$$

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 a 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.

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 properties. Next, we show how to incorporate network constraints into our property checking algorithm. Finally, we prove the correctness of our algorithm and analyze its time complexity.

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.

Derivation pool	*dpool*	$::= \cdot \mid dpool, (nID, p{:}\tau) \mapsto \Delta$
Entries	Δ	$::= \cdot \mid \Delta, (c, \mathcal{D})$
Derivation	$\mathcal{D}$	$::= (\mathsf{BT}, p(\vec{x})) \mid (rID, p(\vec{x}), \mathcal{D}\ \mathsf{List})$

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). We write $dpool(p)$ to denote $dpool(nID, p{:}\tau)$, which returns Δ.

Figure 1 and 2 present the main functions used for constructing a derivation pool from a dependency graph. The top-level function GenDPool is defined in Figure 1. 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).

```
1:  function GenDPool($\mathcal{G}$)
2:      $E \leftarrow \mathcal{G}$'s edges
3:      $P \leftarrow \mathcal{G}$'s predicate nodes that have no incoming edges
4:      while E $\neq$ empty || P $\neq$ empty do
5:          remove $(nID, p{:}\tau)$ from $P$
6:          $\vec{x} \leftarrow fresh(p{:}\tau)$
7:          if $p$ is a base tuple then
8:              $dpool \leftarrow dpool[(nID, p) \mapsto \{(\top, (\mathsf{BT}, p(\vec{x})))\}]$
9:          else
10:             $d \leftarrow$ GenDs($\mathcal{G}$, $dpool$, $(nID, p{:}\tau)$)
11:             $dpool \leftarrow dpool \cup d$
12:         (* done processing p, remove edges *)
13:         $P, E \leftarrow$removeEdges($P$, $E$, $G$, $nID$)
14:     end while
15: end function
16:
17: function removeEdges($P$, $E$, $G$, $nID$)
18:     remove outgoing edges of $nID$ from $E$
19:     for each $rID$ with no edges of form $(\_, rID)$ in $E$ do
20:         remove edges $(rID, nID)$ from $E$
21:         for each $(nID, p{:}\tau)$ with no incoming edges in $E$ do
22:             add $(nID, p{:}\tau)$ to $P$
23: end function
```

Figure 1. Construct derivation pools for non-recursive programs

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 GenDs (Figure 2) 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 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 sub-

```
1:  **function** GENDS($\mathcal{G}$, $dpool$, $(nID, p{:}\tau)$)
2:      $\Delta \leftarrow \{\}$
3:      **for** each rule with ID $rID$ where $(rID, nID)$ in $\mathcal{G}$ **do**
4:          $\Delta \leftarrow \Delta \cup$GENDRULE($\mathcal{G}$, $dpool$, $(nID, p{:}\tau)$,$rID$)
        **return** $\Delta$
5:  **end function**
6:
7:  **function** GENDRULE($\mathcal{G}$, $dpool$, $(nID, p{:}\tau)$, $rID$)
8:      $(p(\vec{y}), Q, c_r) \leftarrow \mathcal{G}(rID)$
9:      (* $Q = q_1 \cdots q_m$)
10:         $D$ is the list of list of derivations for $q_1 \cdots q_m$ *)
11:     $D \leftarrow$ LIST.MAP (LOOKUP $dpool$) $Q$
12:     $D' \leftarrow$LIST.FOLDRIGHT MERGEDLL $D$ nil
13:     $\vec{x} \leftarrow fresh(p(\vec{y}))$
14:     **return** LIST.MAP (COMPLETED $c_r$ $rID$ $p(\vec{y})$ $\vec{x}$) $D'$
15: **end function**
16:
17: **function** MERGED($dc_i$, $dc_{2i}$)
18:     (* $dc_{2i}$ is a derivation for $q_n \cdots q_{i+1}$
19:         $dc_i$ is a possible derivation of $q_i$ *)
20:     $(\sigma_{2i}, c_{2i}, d_{2i}) \leftarrow dc_{2i}$
21:     $(\sigma_i, c_i, d_i) \leftarrow dc_i$
22:     (* $\sigma_i'$ substitutes new vars in $q_i$ for old ones *)
23:     $(\sigma_i', c_i', d_i') \leftarrow fresh(c_i, d_i)$
24:     **return** $(\sigma_i\sigma_i' \cup \sigma_{2i}, c_i' \wedge c_{2i}, d_i'{::}d_{2i})$
25: **end function**
26:
27: **function** LOOKUP($dpool$, $q(\vec{x})$)
28:     **return** LIST.MAP (EXTRACTD $\vec{x}$) $dpool(q)$
29: **end function**
30:
31: **function** EXTRACTD($\vec{x}$, $(c, d)$)
32:     $(rID, p(\vec{y}), dl) \leftarrow d$
33:     **return** $(\vec{y}/\vec{x}, c, d)$
34: **end function**
35:
36: **function** COMPLETED($c_r$, $rID$, $p(\vec{y})$, $\vec{x}$, $d$)
37:     $(\sigma, c, dl) \leftarrow d$
38:     **return** $(c \wedge c_r[\vec{x}/\vec{y}]\sigma, (rID, p(\vec{x}), dl))$
39: **end function**
```

Figure 2. Generate derivation pool for one predicate

stitutions are applied to the constraint c, because all derivations are alpha-renamed and use fresh variables.

3.2 Property Query

Figure 3 shows the property query algorithm for non-recursive programs. The top-level function CKPROP takes the derivation pool and the property as arguments. On 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).

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

```
1:  **function** CKPROP($dpool$, $\varphi$)
2:      (* $P$ is $p_1 \cdots p_n$ and $Q$ is $q_1 \cdots q_m$ *)
3:      $(P, c_p, Q, c_q) \leftarrow \varphi$
4:      (* Get the list of list of derivations for $p_1 \cdots p_n$ *)
5:      $L \leftarrow$ LOOKUPREC($dpool$, $P$)
6:      (* Combine all possible derivations for $p_1 \cdots p_n$
7:          Each entry in $D$ also include substitutions that replace
8:          free variables in $p_i$ with the variable in the derivation *)
9:      $D \leftarrow$MERGEDERIVATION $L$
10:     **for** each $(\sigma, c_d, d)$ in $D$ **do**
11:         $z \leftarrow$CKPROPD$(c_d, c_p\sigma, d, Q, c_q\sigma)$
12:         **if** $z =$ invalid$(d, \sigma_r)$ **then**
13:             **return** invalid$(d, \sigma_r)$
14:     **return** valid
15: **end function**
16:
17: **function** CKPROPD$(c_d, c_p, d, Q, c_q)$
18:     **if** CHECK SAT $c_d \wedge c_p$ = (sat, $\sigma_p$) **then**
19:         (* find all occurrences of q in d *)
20:         $\Sigma \leftarrow$LIST.MAP (UNIFY $d$) $Q$
21:         **if** nil $\in \Sigma$ **then**
22:             (* some $q_i$ does not appear in d *)
23:             **return** invalid$(d, \sigma_p)$
24:         **else**
25:             (* Find all possible combinations for $q_1 \cdots q_m$
26:                 $\Sigma_q$ is a list of substitutions of form $\sigma_{q_1}{::}\cdots{::}\sigma_{q_\lambda}$
27:                 $\sigma_{q_\ell} \in \Sigma_q$ is a substitution for variables in one
28:                 occurrence of $q_1$ to $q_m$ in d, for variables that
29:                 appear in $Q$ *)
30:             $\Sigma_q \leftarrow$MERGELL $\Sigma$
31:             (* $c_q' = \bigwedge_{\ell=1}^{\lambda} \neg c_q\sigma_{q\ell}$ *)
32:             $c_q' \leftarrow$CONJ$(\Sigma_q, \neg c_q)$
33:             $c_a \leftarrow c_p \wedge c_d \wedge c_q'$
34:             **if** CHECK SAT $c_a =$ (sat, $\sigma_a$) **then**
35:                 **return** invalid($d$, $\sigma_a$)
36:             **else return** valid
37:     **else**
38:         (* Constraints for $p_1 \ldots p_n$ and $c_p$ are unsat *)
39:         **return** valid
40: **end function**
```

Figure 3. Property query

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: $(\vec{y_1}/\vec{x}){::}(\vec{y_2}/\vec{x}){::}\cdots{::}(\vec{y_n}/\vec{x}){::}$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 29, we first compute the list of all possible combinations of occurrences of q_is. Again, the function MERGELL is similar to MERGEDLL and we

```
1:  function CkPropDC($c_d, c_p, d, Q, c_q, \beta, c_b$)
2:      if Check sat $c_d \wedge c_p = (\mathsf{sat}, \sigma_p)$ then
3:          (* find all occurrences of b
4:            $\Sigma_b$ is a list of list of substitutions *)
5:          $\Sigma_b \leftarrow$List.map (Unify $d$) $\beta$
6:          (* $\Sigma'_b$ is a list of substitutions. Each substitution
7:            in $\Sigma'_b$ corresponds to one combination of $b_i$s in $d$ *)
8:          $\Sigma'_b \leftarrow$MergeLL $\Sigma_b$
9:          (* Given $\Sigma'_b = \sigma_{b1}{::}\cdots{::}\sigma_{b\mu}$, $c'_b = \bigwedge_{\ell=1}^{\mu} c_b\sigma_{b\ell}$ *)
10:         $c'_b \leftarrow$Conj($\Sigma'_b, c_b$)
11:         (* find all occurrences of $q$ in $d$ *)
12:         $\Sigma \leftarrow$List.map (Unify $d$) $Q$
13:         if $\mathsf{nil} \in \Sigma$ then
14:             (* check network constraints *)
15:             if Check sat $c_d \wedge c_p \wedge (c'_b) = (\mathsf{sat}, \sigma_c)$ then
16:                 (* Network constraints are met *)
17:                 return invalid($d, \sigma_c$)
18:             else return valid
19:         else
20:             (* Find all possible combinations for $q_1 \cdots q_m$
21:               $\Sigma_q$ is a list of substitutions of form $\sigma_{q_1}{::}\cdots{::}\sigma_{q_\lambda}$
22:               $\sigma_{q_\ell} \in \Sigma_q$ is a substitution for variables in one
23:               occurrence of $q_1$ to $q_m$ in d, for variables that
24:               appear in $Q$ *)
25:             $\Sigma_q \leftarrow$MergeLL $\Sigma$
26:             (* $c'_q = \bigwedge_{\ell=1}^{\lambda} \neg c_q\sigma_{q\ell}$ *)
27:             $c'_q \leftarrow$Conj($\Sigma_q, \neg c_q$)
28:             $c_s \leftarrow c_d \wedge c_p \wedge c'_q \wedge c'_b$
29:             if Check sat $c_s = (\mathsf{sat}, \sigma_s)$ then
30:                 (* Network constraints are met *)
31:                 return invalid($d, \sigma_s$)
32:             else return valid
33:     else
34:         (* Constraints for $p_1 \ldots p_n$ and $c_p$ are unsat *)
35:         return valid
36: end function
```

Figure 4. Property query with network constraints

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 certain *network constraints* constraints; for instance, every node in the network has only one outgoing link. 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: $\forall\vec{x_1}.b_1(\vec{x_1}) \wedge \cdots \wedge \forall\vec{x_k}.b_k(\vec{x_k}) \supset c_b(\vec{x_1}\cdots\vec{x_k})$, where b_i is a base tuple. Figure 4 shows the algorithm for checking properties on networks with constraints. For clarity, we explain the case with one network constraint. Extending the algorithm to handle multiple constraints is straightforward.

The top-level function CkPropC (omitted here) 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 $\forall x, b(x) \supset x > 0$. 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 in derivation d (line 26) as usual, and find the substitutions that make q_is appear in d. We compute the conjunction of all $\neg c_q$s (line 28). If the conjunction of c_d, c_p, the conjunction of all the c_bs found in lines 3-10, and the conjunction of all the $\neg c_q$s is satisfiable, then networks constraints are met although d does not satisfy the property being checked, and we report an error (lines 30-34).

3.4 Analysis of the Algorithms

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 function DGraph generates a dependency graph for *prog*, which can be straightforwardly defined. 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 denotes a set of ground base tuples of *prog*. We write $prog, B \vDash d{:}p(\vec{t})$ to mean that d is a derivation of $p(\vec{t})$ using program *prog* and base tuples B. We write $(c, d'{:}p(\vec{x})) \in dpool(p)$ to mean that (c, d') is an entry in the derivation pool *dpool* for the predicate p and that d' is a derivation tree with $p(\vec{x})$ as the root.

Lemma 1 (Correctness of derivation pool construction)**.** DGraph$(prog) = \mathcal{G}$ *and* GenDPool$(\mathcal{G}) = dpool$

1. *If* $prog, B \vDash d'{:}p(\vec{t})$, *then* $\exists\sigma$, $\exists(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in dpool(p)$ *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* $\vDash c(\vec{x_c})\sigma$ *where* $\mathsf{dom}(\sigma) = \vec{x_c}$, *then* $\forall\sigma'$ *s.t.* $\sigma' \geq \sigma$ *and* $\mathsf{dom}(\sigma') = \vec{x_d}$, $\exists B$ *s.t.* $B = \{b \,|\, b$ *is a base tuple and appears in* $d(\vec{x_d})\sigma'\}$ *and* $prog, B \vDash d(\vec{x_d})\sigma'{:}p(\vec{x})\sigma'$.

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

Theorem 2 (Correctness of property query)**.**
Given a property of the following form:

$$\varphi = \forall\vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall\vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1}\cdots\vec{x_n}) \supset \exists\vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists\vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1}\cdots\vec{x_n}, \vec{y_1}\cdots\vec{y_m})$$

and DGraph$(prog) = \mathcal{G}$ *and* GenDPool$(\mathcal{G}) = dpool$, *then*

1. CkProp$(dpool,\varphi)$=valid *implies* $\forall B$, $prog, B \vDash \varphi$
2. *and* CkProp$(dpool,\varphi)$=invalid(d,σ) *implies* $\exists B$ *s.t.* $prog, B \nvDash \varphi$.

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

Theorem 3 (Correctness of property query with constraints)**.**
Given two properties of of the following forms:

$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$
$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$
$\varphi_{net} = \forall \vec{u_1}.b_1(\vec{u_1}) \wedge \cdots \wedge \forall \vec{u_k}.b_n(\vec{u_k}) \supset c_b(\vec{u_1} \cdots \vec{u_k})$
and DGRAPH$(prog) = \mathcal{G}$ *and* GENDPOOL$(\mathcal{G}) = dpool$*, then*
(1) CKPROPC$(dpool, \varphi_{net}, \varphi) =$ valid *implies* $\forall B$*, either* $prog, B \vDash \varphi$ *or* $B \nvDash \varphi_{net}$
(2) and CKPROPC$(dpool, \varphi_{net}, \varphi) =$ invalid(d, σ) *implies* $\exists B$ *s.t.* $prog, B \nvDash \varphi$ *and* $B \vDash \varphi_{net}$.

Time complexity. We give an upper bound on the time complexity of the property query algorithm (Figure 3). Given an NDLog program with R rules; each rule contains at most W body tuples. Also assume $|Q| = m$ and $|P| = n$. The time complexity of our algorithm is $O((R^{nW^R})n^m W^{Rn})$. In practice, R and W are usually small. For example, in our case study, R is bounded by 11 and W is bounded by 5. In this case, R and W can be viewed as constants.

4. Extension to Recursive Programs

The dependency graph for a recursive program contains cycles. The derivation pool construction algorithm presented in Figure 1 does not work for recursive programs because it relies on the topological order of nodes in the dependency graph. In this section, we show how 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.

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

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 p. We write A to denote annotations for recursive predicates, 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.

The structure of the derivation pool construction remains the same. We highlight the changes in Figure 5. 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 includes 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.

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 summary constraint is satisfiable if and only if there is at least one derivation for the recursive predicate p. Thus, the summary constraint must 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 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 (line 10). 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

```
 1: function GENDS($\mathcal{G}$, $dpool$, $(nID, p{:}\tau)$)
 2:     $\Delta \leftarrow \{\}$
 3:     for each rule with ID $rID$ where $(rID, nID)$ in $\mathcal{G}$ do
 4:         $\Delta \leftarrow \Delta \cup$GENDRULE($\mathcal{G}$, $dpool$, $(nID, p{:}\tau)$, $rID$)
 5:     if $(nID, p{:}\tau)$ is on a cycle then
 6:         (* gather all constraints *)
 7:         $(\vec{x}, c) \leftarrow$ EX_DISJ($\Delta$)
 8:         if $A(p) = (\vec{y}, c_A)$ then
 9:             (* check annotation *)
10:             if CHECK SAT $\neg(c_A[\vec{x}/\vec{y}] \leftrightarrow c) = (\mathsf{sat}, \_)$ then
11:                 return annotation_error
12:             else return $(c_A, \Delta)$
13:         else return $(c, \Delta)$
14:     else return $\Delta$
15: end function
16:
17: function LOOKUP($dpool$, $q(\vec{x})$)
18:     if $q \in A$ then
19:         $(\vec{y}, c_A) \leftarrow A(q)$
20:         return $(\vec{y}/\vec{x}, c_A, (\mathsf{rec}, q(\vec{y}))::\mathsf{nil})$
21:     else
22:         if $dpool(q) = \Delta$ then
23:             return LIST.MAP (EXTRACTD $\vec{x}$) $\Delta$
24:         else
25:             $dpool(q) = (c_q(\vec{y}), \Delta_q)$
26:             return $(\vec{y}/\vec{x}, c, (\mathsf{rec}, q(\vec{y}))::\mathsf{nil})$
27: end function
28:
29: function LOOKUPREC($dpool$, $q(\vec{x})$)
30:     if $dpool(q) = \Delta$ then
31:         return LIST.MAP (EXTRACTD $\vec{x}$) $\Delta$
32:     else
33:         $dpool(q) = (c_q(\vec{y}), \Delta_q)$
34:         return LIST.MAP (EXTRACTD $\vec{x}$) $\Delta_q$
35: end function
36:
37: function REMOVEEDGES($P$, $E$, $G$)
38:     remove outgoing edges of $nID$ from $E$
39:     for each $rID$ with no edges of form $(\_, rID)$ in $E$ do
40:         remove edges $(rID, nID)$ from $E$
41:         if $(nID, p : \tau)$ has no incoming edges in $E$ then
42:             add $(nID, p : \tau)$ to $P$
43:         if every $(nID', q : \tau')$ s.t. $(nID', rID), (rID, nID) \in$
    $E$, $(nID', q : \tau')$ in $A$ then
44:             add $(nID, p : \tau)$ to $P$
45: end function
```

Figure 5. Construct derivation pools for recursive programs

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 programs. Because the derivations of recursive predicates are not expanded, this has the following limitations: (1) 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. (2) Network constraints cannot be accurately checked. Given a 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, weneed annotations for recursive predicates to provide invariants for recursive predicates. Our case studies do not require annotations; future work is to expand the algorithm to handle recursive predicates precisely.

4.3 Analysis of the Algorithms

Correctness. We prove the correctness of derivation pool construction and soundness of the query algorithm. Because derivations of recursive predicates are summarized as $(\mathsf{rec}, p(\vec{x}))$, proving correctness requires us to consider the unrolling of $(\mathsf{rec}, p(\vec{x}))$.

First, we define a relation $\mathit{dpool} \vdash d_k, \sigma_k \leadsto_{k+1} d_{k+1}, \sigma_{k+1}$ (for $k \geq 0$) to mean that a derivation d_k with the substitution σ_k can be unrolled using derivations to dpool to another derivation d_{k+1} and a new substitution σ_{k+1}. The rules defining this relation allow the $(\mathsf{rec}, \_)$ leafs in the derivation to be gradually expanded, starting from the root and moving up the tree.

We write d_k to denote the derivation after unrolling a derivation d for a sequence of k steps: $d, \sigma_0 \leadsto_0 d, \sigma_0 \leadsto_1 d_1, \sigma_1 \leadsto_2 \ldots \leadsto_i d_i, \sigma_i \leadsto_{i+1} d_{i+1}, \sigma_{i+1} \leadsto_{i+1} \ldots \leadsto_k d_k, \sigma_k$. The $\leadsto_{i+1}$ rules ensure that each d_i has no $(\mathsf{rec}, \_)$ leafs from the root up to the $i-1^{th}$ level. Step $d_i, \sigma_i \leadsto_{i+1} d_{i+1}, \sigma_{i+1}$ expands the $(\mathsf{rec}, \_)$ leafs at the i^{th} level of d_i, and thus d_{i+1} has no $(\mathsf{rec}, \_)$ leafs from the root up to the i^{th} level. We write $d_0, \sigma_0 \longmapsto d_k, \sigma_k$ as shorthand notation for the above sequence of steps.

$$\textsc{Base}\ \frac{}{\mathit{dpool} \vdash d, \sigma \leadsto_0 d, \sigma}$$

$$\textsc{WkInd}\ \frac{\begin{array}{c} \mathit{dpool} \vdash d, \sigma \leadsto_k d, \sigma \\ k < n \qquad d \text{ does not contain } (\mathsf{rec}, \_) \text{ as subderivations} \end{array}}{\mathit{dpool} \vdash d, \sigma \leadsto_n d, \sigma}$$

$$\textsc{RNrec}\ \frac{\forall i \in [1, n], \mathit{dpool} \vdash d_i, \sigma \leadsto_k d_i', \sigma_i \qquad \sigma' = \bigcup_{i=1}^{n} \sigma_i}{\begin{array}{c} \mathit{dpool} \vdash (\mathit{rID}, p(\vec{x}), d_1{::}\ldots{::}d_n{::}\mathsf{nil}), \sigma \\ \leadsto_{k+1} (\mathit{rID}, p(\vec{x}), d_1'{::}\ldots{::}d_n'{::}\mathsf{nil}), \sigma' \end{array}}$$

$$\textsc{RRec}\ \frac{\begin{array}{c} \mathit{dpool}(p) = (c_p, \Delta_p) \\ (c(\vec{z_c}), d(\vec{z_d}){:}p(\vec{z})) \in \Delta_p \qquad \vec{z_d}' = \mathit{fresh}(\vec{z_d} \backslash \vec{z}) \\ \vDash c(\vec{z_c})[\vec{z_d}'/(\vec{z_d} \backslash \vec{z})][\vec{x}/\vec{z}](\sigma \cup \sigma') \end{array}}{\mathit{dpool} \vdash (\mathsf{rec}, p(\vec{x})), \sigma \leadsto_1 d(\vec{z_d})[\vec{z_d}'/(\vec{z_d} \backslash \vec{z})][\vec{x}/\vec{z}], \sigma \cup \sigma'}$$

Rule BASE does not extend the derivation. Rule WKIND weakens the index from k to a larger number n when derivation d does not contain any recursive subderivations of form $(\mathsf{rec}, \_)$. Given a derivation $(\mathit{rID}, p(\vec{x}), d_1{::}\ldots{::}d_n{::}\mathsf{nil})$ with no $(\mathsf{rec}, \_)$ leafs from the root up to the $k-1^{th}$ level, and whose subderivations $d_1, \ldots, d_n$ that can be expanded to subderivations $d_1', \ldots, d_n'$ which have no $(\mathsf{rec}, \_)$ leafs from the root to the $k-1^{th}$ level, then rule RNREC expands $(\mathit{rID}, p(\vec{x}), d_1{::}\ldots{::}d_n{::}\mathsf{nil})$ to derivation $(\mathit{rID}, p(\vec{x}), d_1'{::}\ldots{::}d_n'{::}\mathsf{nil})$, which has no $(\mathsf{rec}, \_)$ leafs from the root up to the the k^{th} level.

The last rule RREC is the key rule that expands the derivation of the recursive predicate p at the root in one step. Recursive derivation $(\mathsf{rec}, p(\vec{x}))$ and substitution σ are expanded to derivation $d{:}p(\vec{x}) \in \Delta_p$. Constraint c is satisfiable for substitution $\sigma \cup \sigma'$.

Lemma 4 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 will have an entry for for p which can be unrolled into that concrete derivation. For every $\ell \in \mathbb{N}$, if there is a skeleton derivation d for p in the derivation pool and the corresponding constraint to derive d is satisfiable, then either we can unroll d to a concrete derivation d' for p within ℓ steps, where d' does not contain any subderivations of form $(\mathsf{rec}, q(\vec{x_q}))$, or after unrolling d for ℓ steps, the resultant derivation d' contains some recursive subderivations of form $(\mathsf{rec}, q(\vec{x_q}))$, and if every $(\mathsf{rec}, q(\vec{x_q}))$ can be unrolled to a concrete derivation, then the derivation d for predicate p can be unrolled to a concrete derivation. To save space, we state only the key points of the lemma and produce the full statement in our companion technical report [11].

Lemma 4 (Correctness of derivation pool construction (recursive))**.**
$\textsc{DGraph}(\mathit{prog}) = \mathcal{G}$, *and* $\textsc{GenDPool}(\mathcal{G}, A) = \mathit{dpool}$

1. *If* $\mathit{prog}, B \vDash d{:}p(\vec{t})$, *then either* p *is not on a cycle in* $\mathcal{G}$ *and* $\exists(c(\vec{x_c}), d'(\vec{x_d}'){:}p(\vec{x})) \in \mathit{dpool}(p)$, $\exists\sigma_d''$ *where* $\vDash c(\vec{x_c})\sigma_d''|_{\vec{x_c}}$ *s.t. using* σ_d'', $d'(\vec{x_d}')$ *can be unrolled into* d, *or* p *is on a cycle in* $\mathcal{G}$ *and* $\exists(c_p(\vec{x}), \Delta_p) \in \mathit{dpool}(p)$, $\exists\sigma_d''$ *where* $\vDash c_p(\vec{x})\sigma_d''|_{\vec{x}}$ *s.t. using* σ_d'', $(\mathsf{rec}, p(\vec{x}))$ *can be unrolled into* d.
2. $\forall \ell \in \mathbb{N}$,
   (a) *If* $(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in \mathit{dpool}(p)$ *and* $\vDash c(\vec{x_c})\sigma$, *either* $\exists d'(\vec{x_d})$ *s.t.* $d'(\vec{x_d}')$ *does not contain* $(\mathsf{rec}, \_)$, $d(\vec{x_d})$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $d'(\vec{x_d}')$ *can be unrolled to* d *with an appropriate extension of* σ, *or* $\exists d'(\vec{x_d}')$ *s.t.* $d'(\vec{x_d}')$ *contains* $(\mathsf{rec}, s(\vec{x_s}))$, $d(\vec{x_d})$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $\forall(\mathsf{rec}, s(\vec{x_s})) \in d'(\vec{x_d}')$, $\exists d_s, \ell_s$ *s.t.* $(\mathsf{rec}, s(\vec{x_s}))$ *can be unrolled to* d_s *with an appropriate extension of* $\sigma|_{\vec{x_s}}$ *implies* $\exists d''$ *s.t.* $\mathit{prog}, B \vDash d''$ *and* $d(\vec{x_d})$ *and be unrolled to* d'' *with an appropriate extension of* σ.
   (b) *If* $(c_{\mathsf{rec}:p}(\vec{x}), \Delta_p) \in \mathit{dpool}(p)$ *and* $\vDash c_{\mathsf{rec}:p}(\vec{x})\sigma$, *then either* $\exists d'(\vec{x_d})$ *s.t.* $d'(\vec{x_d}')$ *does not contain* $(\mathsf{rec}, \_)$, $(\mathsf{rec}, p(\vec{x}))$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $d'(\vec{x_d}')$ *can be unrolled to* d *with an appropriate extension of* σ, *or* $\exists d'(\vec{x_d}')$ *s.t.* $d'(\vec{x_d}')$ *contains* $(\mathsf{rec}, s(\vec{x_s}))$, $(\mathsf{rec}, p(\vec{x}))$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $\forall(\mathsf{rec}, s(\vec{x_s})) \in d'(\vec{x_d}')$, $\exists d_s, \ell_s$ *s.t.* $(\mathsf{rec}, s(\vec{x_s}))$ *can be unrolled to* d_s *with an appropriate extension of* $\sigma|_{\vec{x_s}}$ *implies* $\exists d''$ *s.t.* $\mathit{prog}, B \vDash d''$ *and* $(\mathsf{rec}, p(\vec{x}))$ *can be unrolled to* d'' *with an appropriate extension of* σ.

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 5 (Soundness of property query)**.**
$\varphi = \forall\vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall\vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$
$\quad \exists\vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists\vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$
$\textsc{DGraph}$*(prog)* = $\mathcal{G}$ *and* $\textsc{GenDPool}(\mathcal{G}, A) = \mathit{dpool}$ *and* $\textsc{CkProp}(\mathit{dpool}, \varphi) = \mathsf{valid}$ *implies* $\forall B, \mathit{prog}, B \vDash \varphi$.

Time complexity. The time complexity of the property query algorithm on recursive programs is the same as that of non-recursive programs. Observe that the height of a derivation in the derivation pool is still bounded by R (the number of rules in the program). This is because in the derivation pool construction algorithm (Figure 5), each rule node is processed at most once. Therefore a path in a derivation from the root predicate to any leaf predicate could have at most R rules.

5. Case Study

We apply our tool to the verification of software-defined networking (SDN) applications. SDN is 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 used to to write SDN controller applications [38]. Like any program, these applications are not guaranteed to be bug-free. We show the effectiveness of our tool in validating and debugging several SDN applications. We demonstrate that the tool can unveil 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 the form discussed in Section 2; (2) formal specification of initial network constraints (e.g., topological constraints and switch default setup); and (3) formal specification of invariants on recursive tuples.

Our tool takes 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.

Encoding We encode the behaviors of each component in NDLog. Due to space limitation, we omit the full program and just provide a summary of the program in Table 2.

In a typical scenario, an end host initiates a packet and sends it to the switch that it connects to (rh1). The switch recursively looks up its forwarding table to match against the received packet (rs1, rs2). If a flow entry matches the packet, it 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 mismatched 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 use the following basic network constraints to limit the topology of the network that runs Ethernet source learning.

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

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).

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 in detail.

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

$$\begin{array}{l}
\varphi_{ESL_2} = \\
\quad \forall EndHost, Switch, SrcMac, DstMac, InPort, \\
\qquad OPort, Outport, Mac, Priority, \\
\qquad\quad \mathsf{packet}(EndHost, Switch, SrcMac, DstMac) \\
\qquad\quad \wedge \mathsf{swToHst}(Switch, EndHost, OPort) \\
\qquad\quad \wedge \mathsf{flowEntry}(Switch, Mac, Outport, Priority) \\
\qquad\quad \wedge DstMac \neq EndHost \supset \\
\qquad Mac \neq DstMac
\end{array}$$

Though this property is seemingly true, our tool returns a negative answer, along with a counterexample shown in Figure 6. The counter example reveals a scenario where an endhost (H4) receives a broadcast packet destined to another machine (H3) (Execution trace (1) in Figure 6), but the switch it connects to (S1) has a flowEntry that matches the destination MAC address in the packet (Execution trace (2) in Figure 6).

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 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)

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 1. Predicates in Ethernet Source Learning

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 2. Ethernet Source Learning Rules

(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:

$$\begin{aligned}&\varphi_{ESL_3} = \\ &\quad \forall EndHost, Switch, SrcMac, DstMac, OPort, \\ &\qquad \mathsf{packet}(EndHost, Switch, SrcMac, DstMac) \\ &\qquad \wedge \mathsf{swToHst}(Switch, EndHost, OPort) \\ &\qquad \wedge DstMac = EndHost \supset \\ &\quad \exists Switch', Mac, Outport, Priority, \\ &\qquad \mathsf{flowEntry}(Switch', Mac, Outport, Priority) \\ &\qquad \wedge Switch' = Switch \wedge Mac = DstMac\end{aligned}$$

The generated counter example (Figure 7) 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, the above counter examples, 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.

5.3 Firewall

Our second case study is a stateful firewall, which 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.

Verification results We verify a number of properties about the stateful firewall. We discuss one property here (shown below).

$$\begin{aligned}&\varphi_{WeakFW} = \\ &\quad \forall Host, Port, Src, SrcPort, Switch, \\ &\qquad \mathsf{pktReceived}(Host, Port, Src, SrcPort, Switch) \supset \\ &\qquad \exists Cntrl, \mathsf{trustedControllerMemory}(@Cntrl, Switch, Src)\end{aligned}$$

The above property specifies that source destinations of all packets reaching internal machines are trusted by the controller. Surprisingly, our tool gives a counterexample for this property (Figure 8), which depicts the scenario that an internal machine H3 sends a packet to another internal machine H4 in the same domain

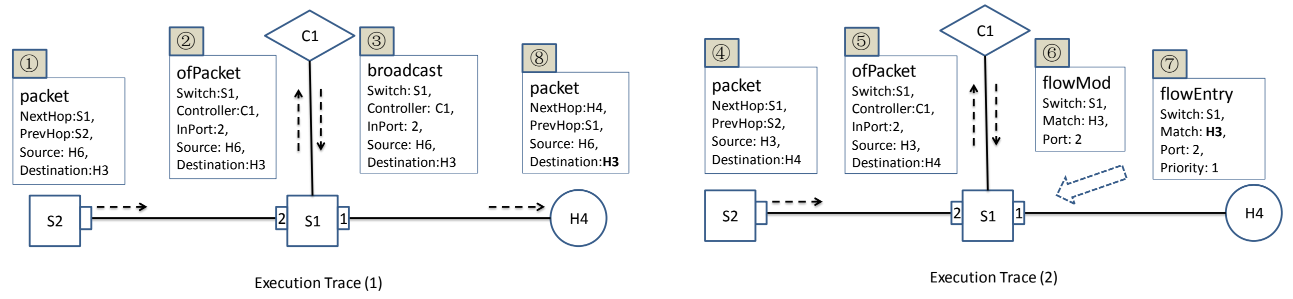

Figure 6. A counter example for property φ_{ESL_2}

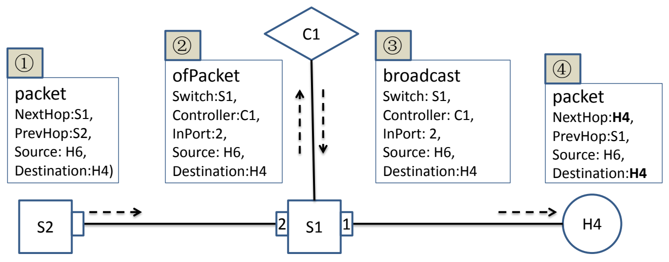

Figure 7. A counter example for property φ_{ESL_3}

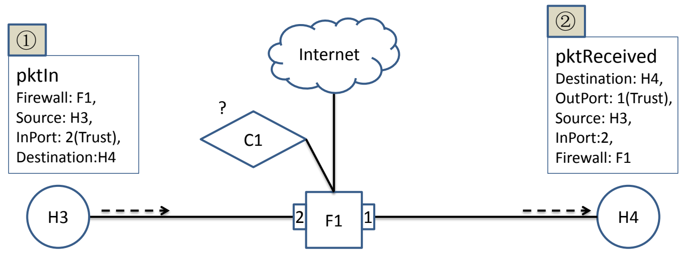

Figure 8. A counter example for property φ_{WeakFW}

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 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 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. There are a number of strategies for load balancing, 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.

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 of 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 Ethernet Address Resolution

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 that 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.

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 3.

5.6 Discussion

We discuss our experience of using the tool and insights obtained from 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.

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 3. Results of checking safety properties of $prog_{ARP}$ on our tool

Iterative application development Another observation is that reasonable network assumptions (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 property specification). After the programmer fix the problem, she or he can redo the verification repeatedly until the specified property holds.

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 [2, 17, 18, 37, 49], or 2) by analyzing snapshots of the data plane—reflecting the aggregate impact of all configurations—during system execution [22, 23, 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 [16, 19, 25, 47, 48]. 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, 10, 14, 52].

Most closely related to ours is the work on verifying network protocol design using declarative networking [10, 47, 48]. 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 [1, 8, 9, 21, 24, 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]. NetKAT is a high-level language designed specifically for programming SDN. Its semantics are based on Kleene algebra. The correctness properties of networks programming using NetKAT are tightly connected to the semantics of Kleene algebra, for instance, reachability, way points and traffic separation.

All of these tools are specially designed to analyze SDN controllers or data planes. Modeling and verifying SDN controllers is one example application of our analysis; our analysis can be applied to analyzing other distributed systems expressible in NDLog. On the other hand, in the current state, we can only check simple safety properties, while VeriCon, NICE, and NetKAT 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) [30], MELD [7], Linear Meld [15], Netlog [20], DAHL [32], Dedalus [3]). 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

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 analyzed formal properties of our algorithms and implemented a prototype tool on top of RapidNet, a compilation and execution framework for NDLog. Using our tool, we analyzed a number of real-world SDN network protocols. Our tool can unveil problems ranging from software bugs, incomplete topological constraints, and incorrect property specification. When a given safety property is violated, our tool can provide meaningful counterexamples to help debug the protocol specification.

8. Acknowledgment

We thank the anonymous reviewers for their invaluable comments. This work is supported in part by NSF CNS-1218066, CNS-1117052, CNS-1115706, CNS-0845552, CNS-1453392 and AFOSR Young Investigator Award FA9550-12-1-0327.

References

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

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

[3] P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Eurosys*, 2010.

[5] M. Arnaud, V. Cortier, and S. Delaune. Modeling and verifying ad hoc routing protocols. In *CSF*, 2010.

[6] M. Arnaud, V. Cortier, and S. Delaune. Deciding security for protocols with recursive tests. In *CADE*, 2011.

[7] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *IROS*, 2007.

[8] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI*, 2014.

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

[10] C. Chen, L. Jia, H. Xu, C. Luo, W. Zhou, and B. T. Loo. A program logic for verifying secure routing protocols. In *FORTE*, 2014.

[11] C. Chen, L. K. Loh, L. Jia, W. Zhou, and B. T. Loo. Automated verification of safety properties of declarative networking programs. Technical Report CMU-CyLab-15-002, CyLab, Carnegie Mellon University, Jun 2015.

[12] X. Chen, Y. Mao, Z. M. Mao, and J. van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Co-NEXT*, 2010.

[13] D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The Design and Implementation of a Declarative Sensor Network System. In *SenSys*, 2007.

[14] V. Cortier, J. Degrieck, and S. Delaune. Analysing routing protocols: four nodes topologies are sufficient. In *POST*, 2012.

[15] F. Cruz, R. Rocha, S. C. Goldstein, and F. Pfenning. A linear logic programming language for concurrent programming over graph structures. *TPLP*, 14(4-5):493–507, 2014.

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

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

[18] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[19] A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *WITS*, 2005.

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

[21] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[22] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[23] 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.

[24] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *HotSDN*, 2012.

[25] C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[26] C. Liu, R. Correa, H. Gill, T. Gill, X. Li, S. Muthukumar, T. Saeed, B. T. Loo, and P. Basu. PUMA: Policy-based Unified Multi-radio Architecture for Agile Mesh Networking. In *COMSNETS*, 2012.

[27] C. Liu, R. Correa, X. Li, P. Basu, B. T. Loo, and Y. Mao. Declarative policy-based adaptive mobile ad hoc networking. *IEEE/ACM Trans. Netw.*, 20(3):770–783, 2012.

[28] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[29] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[30] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[31] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. In *CACM*, 2009.

[32] N. P. Lopes, J. A. Navarro, A. Rybalchenko, and A. Singh. Applying prolog to develop distributed systems. In *ICLP*, 2010.

[33] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[34] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *Co-NEXT*, 2008.

[35] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[36] S. C. Muthukumar, X. Li, C. Liu, J. B. Kopena, M. Oprea, R. Correa, B. T. Loo, and P. Basu. RapidMesh: declarative toolkit for rapid experimentation of wireless mesh networks. In *WINTECH*, 2009.

[37] T. Nelson, C. Barratt, D. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[38] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

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

[40] V. Nigam, L. Jia, B. T. Loo, and A. 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] R. Ramakrishnan and J. 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] M. Sherr, A. Mao, W. R. Marczak, W. Zhou, B. T. Loo, and M. Blaze. A3: An extensible platform for application-aware anonymity. In *NDSS*, 2010.

[45] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. 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] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *PADL*, 2009.

[48] A. Wang, B. T. Loo, C. Liu, O. Sokolsky, and P. 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] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, , and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI*, 2014.

[52] F. Zhang, L. Jia, C. Basescu, T. H.-J. Kim, Y.-C. Hu, and A. Perrig. Mechanized network origin and path authenticity proofs. In *CCS*, 2014.

[53] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *SIGMOD*, 2010.

[54] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *SOSP*, 2011.

[55] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed Time-aware Provenance. In *VLDB*, 2013.