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].

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 \( p \leftarrow q_1, q_2, \ldots, q_n \), which can be read as \( q_1 \) and \( q_2 \) and ... and \( q_n \) implies \( p \). Here, \( p \) is the head of the rule, and \( q_1, q_2, \ldots, q_n \) 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:**
\[
\begin{align*}
d1 \ \text{reachable}(X, Y, C) & \leftarrow \text{link}(X, Y, C), \\
d2 \ \text{reachable}(X, Y, C) & \leftarrow \text{link}(X, Z, C_1), \ \text{reachable}(Z, Y, C_2), \ C_1 + C_2. \\
d3 \ \text{reachable}(X, Y, C) & \leftarrow \text{reachable}(X, Z, C_1), \ \text{link}(Z, Y, C_2), \ C_1 + C_2.
\end{align*}
\]

The program **Reachable** takes as input \( \text{link}(X, Y, C) \) 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, \( \text{link} \) tuples are stored based on the value of the \( X \) field. The program **REACHABLE** derives \( \text{reachable}(X, Y, C) \) 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 \( C_1 \), and \( Z \) knows about a path to \( Y \) with cost \( C_2 \), then, \( X \) can reach \( Y \) with cost \( C_1 + C_2 \). 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 \( \text{onehop} X Z C2 \) should be \( \text{onehop} Z Y C2 \), thus this program cannot derive three-hop paths.

**THREEHOPS (With a deliberate error in \( r2 \));**
\[
\begin{align*}
r1 \ \text{onehop}(X, Y, C) & \leftarrow \text{link}(X, Y, C). \\
r2 \ \text{twohops}(X, Z, C) & \leftarrow \text{link}(X, Z, C_1), \ \text{onehop}(X, Z, C_2), \ C = C_1 + C_2. \\
r3 \ \text{threehops}(X, Y, C) & \leftarrow \text{onehop}(X, Z, C_1), \ \text{twohops}(Z, Y, C_2), \ C = C_1 + C_2. \\
r4 \ \text{threehops}(X, Y, C) & \leftarrow \text{twohops}(X, Z, C_1), \ \text{onehop}(Z, Y, C_2), \ C = C_1 + C_2.
\end{align*}
\]

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 format 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 \( N_p \), and rule nodes, denoted \( N_r \). 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 or not. Each predicate with \( n \) attributes has \( n \) predicate nodes, denoted \( N_{p1}, \ldots, N_{pn} \). 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 has the head of that rule, or from a predicate node to a rule node where the predicate is in the rule body.

**Predicate type** \( \tau ::= \text{Pred} \mid \text{bt} \supset \tau \)

**Dependency graph** \( G ::= (N_p \ \text{List}, \ N_r \ \text{List}, \ E \ \text{List}) \)

**Predicate node** \( N_p ::= (nID, p\tau, cyce) \mid (nID, p\tau, ncyc) \)

**Rule node** \( N_r ::= (rID, hd, body, c) \)

**Edge** \( E ::= (rID, nID) \mid (nID, rID) \)

**Rule head** \( \text{hd} ::= \text{p(\text{E})} \)

**Rule body** \( \text{body} ::= p_1(x_1), \ldots, p_n(x_n) \)

**Rule constraints** \( c ::= e_1, \ bop e_2, c_1, c_2, c_3, c_4, c_v, c_2 | \exists x. c \)

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.
\[
\begin{align*}
r1 : p(x, y) & ::= q(x_1), s(y_1), x_1 = y_1, x = x_1, y = y_1. \\
r2 : p(x, y) & ::= q(x), s(y), x = y.
\end{align*}
\]
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:

$$\exists x_1, p_1(x_1) \land \cdots \land \exists x_n, p_n(x_n) \land c_p(x_1^* \cdots x_n^*)$$

$$\exists y_1, q_1(y_1) \land \cdots \land \exists y_m, q_m(y_m) \land c_q(x_1^* \cdots x_n, y_1 \cdots y_m)$$

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 $$q_i$$ to be derived before $$p_i$$s. 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: $$\forall x,y,z. \text{threehops } x \cdot y \cdot z \quad \exists x',y',z'. \text{twohops } x \cdot x' \cdot y \cdot z'$$

Q2: $$\forall x,y,z. \text{threehops } x \cdot y \cdot z \quad \exists x_1, x_2, z_1, z_2, z_3. \text{link } x \cdot x_1 \cdot z_1 \land \text{link } x_1 \cdot x_2 \cdot z_2$$

Q3: $$\exists x,y,z. \text{threehops } x \cdot y \cdot z$$

Q1 states that to derive threehops $$x \cdot y \cdot z$$, it is necessary to derive twohops $$x \cdot x' \cdot y$$ for some $$x'$$. 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**

$$\text{dpool ::= } \cdot | \text{dpool}(nID, p, \tau) \mapsto \Delta$$

**Entries**

$$\Delta ::= \Delta \quad \Delta(c, D)$$

**Derivation**

$$D ::= (BT, p(x)) | (rID, p(x), D \text{ List})$$

We write dpool to denote derivation pools. We write $$\Delta$$ to denote lists of pairs of a constraint and a derivation tree, denoted D. At a high-level, D can be instantiated to be a valid derivation of $$p(\tau)$$ using rules in the program, if c is satisfiable. A derivation tree, D, is inductively defined. The base tuples, denoted $$\text{BT}(c, p(x))$$, 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 $$p(\tau)$$, and the list of derivation trees for the body predicates of that rule (D List). We write dpool(p) to denote dpool(nID, p, $$\tau$$), which returns $$\Delta$$.

Figure 1 and Figure 2 present the main functions used for constructing a derivation pool from a dependency graph. The top-level function GENPOOL is defined in Figure 1. This function follows the topological order of the nodes in the dependency graph 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 GENPOOL, we remove one predicate node 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 a non-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 REMOVE 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 dpool. The GENDS function constructs derivations for that derivation pool (explained later).

1: function GENPOOL(G) 2: E ← G's edges 3: P ← G's predicate nodes that have no incoming edges 4: while E ≠ empty || P ≠ empty do 5: remove (nID, $$p\tau$$) from P 6: x ← fresh($$p\tau$$) 7: if x is a base tuple then 8: dpool← dpool([nID, p]) 9: else 10: d ← GENDS(G, dpool, (nID, p $$\tau$$)) 11: dpool← dpool ∪ d 12: (* done processing *) 13: P, E ← REMOVEINEDGES(P, E, G, nID) 14: end while 15: end function

16: function REMOVEINEDGES(P, E, G, nID) 17: remove outgoing edges of nID from E 18: for each rID with no edges of form (r, nID) in E do 19: remove edges (r, nID) from E 20: end for 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
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 \( \text{COMPLETED} \) generates a well-derivation for \( p \) using the rule ID and the list of derivations for \( q_i \). 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 3 shows the property query algorithm for non-recursive programs. The top-level function \( \text{CKPROP} \) 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 \( p_i \)s in \( P \) between lines 5 to 9. We omit the definition of \( \text{MERGEDERIVATION} \), as it is similar to \( \text{MERGEDLL} \).

The only difference is that we do not need to alpha-rename the derivations. Next, we check that for each possible derivation of \( p_i \)s in \( D \), all of \( q_i \)s appear in the derivation, and the constraint \( c_q \) holds (lines 10 to 14) using function \( \text{CKPROP} \). If for all possible derivations of \( p_i \)s, we can always find derivations of \( q_i \)s such that the constraint \( c_q \) holds, \( \varphi \) holds (line 14).

The function \( \text{CKPROP} \) 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 \( p_i \)s 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 \( \varphi \) is false, so \( \varphi \) 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 \( p_i \)s are derivable and constraint \( c_p \) is satisfiable. On line 20, function \( \text{UNIFY} \) identifies a list of occurrences of \( q_i \) in the derivation \( d \). That is, for each \( q_i(\vec{y}_i) \) appearing in \( d \), \( \text{UNIFY} \) returns the list of substitutions: \( (\vec{y}_i/\vec{x})::(\vec{y}_i/\vec{x})::::::(\vec{y}_n/\vec{x})::\text{nil} \), where \( \vec{x} \) is \( q_i \)’s arguments in \( \varphi \). The list map function returns the list of the list of occurrences for all the \( q_i \)s in \( Q \). We call it “\( \text{UNIFY} \)” because we unify the variables that are \( q_i \)’s arguments in \( \varphi \) 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 \( \text{UNIFY} \) will return an empty list nil.

---

**Figure 2.** Generate derivation pool for one predicate

**Figure 3.** Property query
function \text{CKPROPDC}(c_d, c_p, d, Q, c_q, B, c_o) 
if \text{CHECK SAT} c_d \land c_p = (\text{sat}, \sigma_p) then 
(* find all occurrences of $b$ *) 
\Sigma_b \leftarrow \text{LIST.MAP (UNIFY d) B} 
(* \Sigma_b is a list of substitutions *) 
\Sigma_b \leftarrow \text{MERGE LL} \Sigma_b 
(* \Sigma_b is the conjunction of $c_b \sigma$, where $\sigma_1 \in \Sigma_b^*) 
\Sigma_b \leftarrow \text{CON}(\Sigma_b') \Sigma_b 
(* \Sigma_b is a list of substitutions. Each substitution *) 
\Sigma_b \leftarrow \text{MERGE LL} \Sigma_b 
(* in \Sigma_b^* corresponds to one combination of $b$s in d *) 
\Sigma_b \leftarrow \text{MERGE LL} \Sigma_b 
(* find all occurrences of $q$ in d *) 
\Sigma_b \leftarrow \text{LIST.MAP (UNIFY d) Q} 
if \emptyset \subseteq \Sigma then 
(* check network constraints *) 
if \text{CHECK SAT} c_d \land c_p \land (\sigma_b) = (\text{sat}, \sigma^c) then 
return invalid(\sigma^c) 
else 
(* network constraints are not met *) 
return valid 
else 
\Sigma \leftarrow \text{MERGE LL} \Sigma 
(* find all possible combinations for $q_1, \cdots, q_m$ *) 
\Sigma \leftarrow \text{MERGE LL} \Sigma 
(* \Sigma is a list of substitutions each $\sigma$ in $\Sigma$ is a *) 
\Sigma \leftarrow \text{MERGE LL} \Sigma 
(* substitution for variables in one occurrence *) 
of $q_1$ to $q_m$ in $d$ for variables that appear in $Q^*$) 
for each $\sigma \in \Sigma$ do 
if \text{CHECK SAT} c_d \land c_p \land \neg c_q = (\text{sat}, \sigma_q) then 
continue 
else 
\sigma \leftarrow c_d \land c_p \land (\sigma_b) = (\text{sat}, \sigma^c) 
(* network constraints are met *) 
return valid 
(* None of the combinations of $q$ works. *) 
Next, check network constraints *) 
if \text{CHECK SAT} c_d \land c_p \land (\sigma_b) = (\text{sat}, \sigma_q) then 
return invalid(\sigma^c) 
else 
(* network constraints are not met *) 
return valid 
end function

Figure 4. Property query with network constraints

3.3 Network Constraints

Sometimes, the network being analyzed has certain network 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: $\forall z \exists ! b \left( x_1 = b \left( x_2, \ldots, x_n \right) \right)$, which $b$ 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 CKPROPDC (omitted here) is almost the same as CKPROP, except that it takes a network constraint ($\varphi_{net}$) as an additional argument and uses the function CKPROPDC, which additionally checks network constraints compared to CKPROP.

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 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 $\varphi_{net}$. 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 $c_b$s. To give 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 $c_b = y > 0 \land z > 0$.

Next, we collect the list of occurrences of $q$s, the same as before. If some $q$s 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 $q$s (line 21) as usual. For each substitution that makes all $q$s appear in $d$, we check whether $c_q$ is satisfiable. Between 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 $q$s appear in $d$. On line 34, we finish the loop and $c_q$ is not satisfiable for any of the substitutions that make $q$s appear in $d$. Again, we check the network constraints on $d$, and report an error only if $d$ satisfies the network constraint.

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 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 $\sigma^c \supset \sigma$ to mean that $\sigma^c$ extends $\sigma$. $B$ denotes a set of ground base tuples of prog. We write $\text{write prog, } \text{B} \vdash d(p)(t)$ to mean that $d$ is a derivation of $p(t)$ using program prog and base tuples $B$. We write $\text{write prog, } \text{B} \vdash d(p)(t)$ to mean that $\text{c}(d^t)$ is an entry in the derivation pool $dpool$ for the predicate $p$ and that $d^t$ is a derivation tree with $p(x)$ as the root.

Lemma 1 (Correctness of derivation pool construction).

$\text{DGRAPH}(\text{prog}) = \text{G}$ and \text{GENDPool}$(\text{G}) = \text{dpool}$

1. If $\text{prog, } \text{B} \vdash d(p)(t)$ then exists $\sigma$ and $(c, d^t(p)(x)) \in \text{dpool}(p)$ s.t. $d^t \sigma = d$ and $\sigma \subset c$.
2. If \((c, dp(x)) \in dpool(p)\) and \(\vdash \exists \sigma', \sigma \text{ s.t. } \sigma' \supseteq \sigma \text{ and } \text{prop}, B \equiv (\alpha \rightarrow dp(x) \sigma')\).

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).
\[
\varphi = \forall x_1, p_1(x_1) \land \ldots \land \forall x_n, p_n(x_n) \land c = c_1(x_1, \ldots, x_n) \triangleright \exists y_1, q_1(y_1) \land \ldots \land \exists y_m, q_m(y_m) \land c_2(x_1, \ldots, x_n, y_1, \ldots, y_m)
\]

\(\text{DGRAPH}(\text{prop}) = G \text{ and } \text{GENDPOOL}(G) = dpool\).

1. \(\Delta \cup B \text{ s.t. } \text{prop} \rightarrow \varphi \text{ implies } \text{CKPROP}(dpool, \varphi) = \text{invalid}(d, \sigma)\).
2. \(\text{CKPROP}(dpool, \varphi) = \text{invalid}(d, \sigma) \text{ implies exists } B \text{ s.t. } \text{prop} \cup B \not\equiv \varphi\).

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).
\[
\varphi = \forall x_1, p_1(x_1) \land \ldots \land \forall x_n, p_n(x_n) \land c = c_1(x_1, \ldots, x_n) \triangleright \exists y_1, q_1(y_1) \land \ldots \land \exists y_m, q_m(y_m) \land c_2(x_1, \ldots, x_n, y_1, \ldots, y_m)
\]

\(\text{DGRAPH}(\text{prop}) = G \text{ and } \text{GENDPOOL}(G) = dpool\).

1. \(B \cup \text{prop} \text{ and } \text{prop} \not\equiv \varphi \text{ implies } \text{CKPROP}(dpool, \varphi, \varphi_{\text{net}}) = \text{invalid}(d, \sigma)\).
2. \(\text{CKPROP}(dpool, \varphi, \varphi_{\text{net}}, \varphi) = \text{invalid}(d) \text{ implies exists } B \text{ s.t. } \text{prop} \cup B \not\equiv \varphi \text{ and } \text{prop} \not\equiv \varphi_{\text{net}}\).

**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 \(|\varphi| = m\) and \(|\text{prop}| = n\). The time complexity of our algorithm is \(O((|\text{prop}|W^4)nW^6)\). 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 \((c, \Delta)\), where \(\Delta\) 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.

<table>
<thead>
<tr>
<th>Constraint pool dpool</th>
<th>dpool, (nID, p;\tau) \rightarrow (c, \Delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Derivation</strong></td>
<td>(rec, p(x))</td>
</tr>
<tr>
<td><strong>Annotation</strong> A</td>
<td>(A, (nID, p;\tau)) \rightarrow (c, \Delta)</td>
</tr>
</tbody>
</table>

Derivation trees include a new leaf node \((\text{rec}, p(x))\), 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 \((\Delta, c)\), where \(\Delta\) are the arguments of \(p\) and \(c\) is the constraint which is satisfiable if and only if \(p\) is derivable.

The store 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 tuples 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 \(q1\) and \(q2\). Let’s assume that there is no edge from \(q1\) to \(r1\), as \(q1\) has been processed and \(q2\) 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 \(\text{REMOVEDGES}\) 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 \(q\), its 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 \((\text{rec}, q(x))\) as a place holder for derivations of \(q\) and use the constraint in \(A\) as the constraint for this derivation. The change is reflected in the \(\text{LOOKUP}\) function for collecting possible derivations of the body predicates (lines 21-23).

Finally, annotations need to be verified. The \(\text{GENDS}\) 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 \((c, \Delta)\)). 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 \(p\).

We consider two cases for a predicate on a cycle of the dependency graph: (1) there is an annotation for \(p\) in \(A\) and (2) there is no annotation for \(p\). For both cases, we need to collect all possible constraints for deriving \(p\) from \(\Delta\). Function \(\text{EXDIS}()\) computes the disjunction of constraints in \(\Delta\). 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

```plaintext
1: function GENDS(G, dpool, (nID, p;\tau))
2: while \(\Delta \neq \{\}\) do
3: for each rule with ID \(rID\) where \((rID, nID)\) in \(G\) do
4: \(\Delta \leftarrow \Delta \cup \text{GENDRULE}(G, dpool, (nID, p;\tau), rID)\)
5: if \((nID, p;\tau)\) is on a cycle then
6: (* gather all constraints *)
7: \((x, c) \leftarrow \text{EXDIS}(\Delta)\)
8: if \(A(p) = (\vec{y}, cA)\) then
9: (* check annotation *)
10: if CHECK SAT \(-cA(x/\vec{y})\epsilon c\) = (sat, _ ) then
11: return annotation_error
12: else
13: return \((cA, \Delta)\)
14: else
15: return \((c, \Delta)\)
16: end function
17: end function
18: function LOOKUP(dpool, q(x))
19: if \(q \in A\) then
20: \((\vec{y}, cA) \leftarrow A(q)\)
21: return \((\vec{y}/\vec{x}, cA, \text{rec}, q(x))::\text{nil}\)
22: else
23: \((\text{dpool}(q) = \Delta, \text{return L}\).\text{IST} \text{MAP} \text{EXTRACTD} \vec{x} \Delta\)
24: else
25: \((\text{dpool}(q) = \Delta, \text{return L}\).\text{IST} \text{MAP} \text{EXTRACTD} \vec{x} \Delta\)
26: end function
```

Figure 5. Construct derivation pools for recursive programs
of the constraints for all possible derivations of \( p \) (line 10). If this is the case, then the annotated constraint together with \( \Delta \) 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\_DISI (Lines 15). When \( p \) is not recursive, only \( \Delta \) is returned (line 17).

4.2 Property Query

We use the same property query algorithm for non-recursive programs. 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 \((\text{rec}, p(\vec{x}))\) may contain predicates needed by the antecedent of the property (the \( q_S \) in \( \varphi \)). Without expanding these derivations, the algorithm may report that \( \varphi \) is violated because \( q_S \) 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 \( q_S \) such that \( c_q \) holds, checking the network constraints on \( d \) requires us to expand \((\text{rec}, p(\vec{x}))\) 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.

4.3 Analysis of the Algorithms

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

\[
\sigma' \geq \sigma \\
\text{dpool} \vdash (\text{BT}, p(\vec{x})), \sigma \rightsquigarrow_0 (\text{BT}, p(\vec{x})), \sigma' \\
\forall j \in [1, n], \text{dpool} \vdash d_j, \sigma \rightsquigarrow_k d'_j, \sigma' \\
\text{dpool} \vdash (\text{ID}, p(\vec{x}), d_1::d_n::\langle \text{nil} \rangle), \sigma \\
\rightsquigarrow_{k+1} (\text{ID}, p(\vec{x}), d_1::d_n::\langle \text{nil} \rangle), \sigma' \\
\text{dpool}((p) = (c, \Delta)) \\
\text{dpool} \vdash (c, \Delta), (c, d_{ps}) \in \Delta \equiv c, \sigma' \\
\text{dpool} \vdash (c, \Delta), (c, d_{ps}) \in \Delta \equiv c, \sigma'' \\
\text{dpool} \vdash (\text{rec}, p(\vec{x})), \sigma \rightsquigarrow_{k+1} d_{ps}, \sigma''
\]

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 \( p ((\text{rec}, p(\vec{x})) \) using one of the possible derivations of \( p \) from \( \Delta \). Here, the unrolling can only use the derivation in \( \Delta \), whose constraint can be satisfied.

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 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 4 (Correctness of derivation pool construction (recursive)), \text{DGRAPH}(\text{prog}) = \mathcal{G}, \text{and \text{GENDPool}(\mathcal{G}, A) = \text{dpool}}

1. If \( \text{prog}, B \vdash d : p(i) \) then
   (a) either \( p \) is not on a cycle in the dependency graph and exists \( \sigma \) and \( (c, d', p(\vec{x})) \) \( \in \text{dpool}(p) \) s.t. \( \text{dpool} \vdash d, \sigma \rightsquigarrow [\Delta] d_i, \sigma = d_i \sigma \) and \( \equiv \sigma \).
   (b) or \( p \) is on a cycle in the dependency graph and exists \( \sigma \) and \( (c_p, \Delta_p) \in \text{dpool}(p) \) s.t. \( \text{dpool} \vdash (\text{rec}, p(\vec{x})), \sigma \rightsquigarrow [\Delta] d_i, \sigma = d_i \sigma \) and \( \equiv \sigma \).

2. (a) If \( (c, p(p(\vec{x})) \in \text{dpool}(p) \) and \( \equiv \sigma \), then \( \forall \forall n, m \leq n, \text{dpool} \vdash d, \sigma \rightsquigarrow_m^* d', \sigma' \), either \( d' \) does not contain \( \text{rec}, q(\vec{y}) \), and \( \exists B, \text{dpool}, B \vdash d' \sigma' \) or \( d' \) contains \( \text{rec}, q(\vec{y}) \), and replacing all of the \( (\text{rec}, q(\vec{y})) \) derivations with a derivation of \( \sigma' \) in \( d' \) results in a derivation for \( p(\vec{x}) \sigma' \).

   (b) If \( (c_p, \Delta_p p(\vec{x})) \in \text{dpool}(p) \) and \( \equiv \sigma_p \), then \( \forall \forall n, m \leq n, \text{dpool} \vdash (\text{rec}, p(\vec{x})), \sigma \rightsquigarrow_m d'', \sigma' \), either \( d'' \) does not contain \( \text{rec}, q(\vec{y}) \), and \( \exists B, \text{dpool}, B \vdash d'' \sigma' \) or \( d'' \) contains \( \text{rec}, q(\vec{y}) \), and replacing all of the \( (\text{rec}, q(\vec{y})) \) derivation with a derivations of \( \sigma' \) in \( d'' \) results in a derivation for \( p(\vec{x}) \sigma' \).

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),

\[
\varphi = \exists x_1, p_1(x_1) \land \ldots \land \exists x_n, p_n(x_n) \land c_p(x_1 \ldots x_n) \cup \\
\exists y_1, q_1(y_1) \land \ldots \land \exists y_m, q_m(y_m) \land c_q(x_1 \ldots x_n, y_1 \ldots y_m) \\
\text{DGRAPH(\text{prog}) = G}, \text{and \text{GENDPool(G, A) = dpool}}, \text{and C\text{\textit{PROP}}(dpool, \varphi) = valid implies \text{\textit{PROP}}(\text{\textit{PROP}})}
\]

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 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 rs3 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.

\[
\varphi_{\text{net1}} = \quad \text{initPacket}(\text{Host}, \text{Switch}, \text{Src}, \text{Dst}) \Rightarrow \\
\text{Host} \neq \text{Switch} \land \text{Host} = \text{Src} \land \\
\text{Host} \neq \text{Dst} \land \text{Switch} \neq \text{Dst}.
\]

\[
\varphi_{\text{net2}} = \quad \text{ofconn}(\text{Controller}, \text{Switch}) \Rightarrow \\
\text{Controller} \neq \text{Switch}.
\]

\[
\varphi_{\text{net3}} = \quad \text{swToHs}(\text{Switch}, \text{Host}, \text{Port}) \Rightarrow \\
\text{Switch} \neq \text{Host} \land \text{Switch} \neq \text{Port} \land \text{Host} \neq \text{Port}.
\]

\[
\varphi_{\text{net4}} = \quad \text{swToHs}(\text{Switch1}, \text{Host1}, \text{Port1}) \land \\
\text{swToHs}(\text{Switch2}, \text{Host2}, \text{Port2}) \Rightarrow \\
(\text{Switch1} = \text{Switch2} \land \text{Host1} = \text{Host2} \land \\
\text{Port1} = \text{Port2} \land \\
(\text{Switch1} = \text{Switch2} \land \text{Port1} = \text{Port2} \land \\
\text{Host1} = \text{Host2}).)
\]

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 swToHs). 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 swToHs).

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:

\[
\varphi_{\text{ESL2}} = \\
\forall \text{EndHost}, \text{Switch}, \text{SrcMac}, \text{DstMac}, \text{InPort}, \\
\text{OpPort}, \text{OutPort}, \text{Mac}, \text{Priority}, \\
\text{packet}(\text{EndHost}, \text{Switch}, \text{SrcMac}, \text{DstMac}) \\
\land \text{swToHs}(\text{Switch}, \text{EndHost}, \text{OpPort}) \\
\land \text{flowEntry}(\text{Switch}, \text{Mac}, \text{OutPort}, \text{Priority}) \\
\land \text{DstMac} \neq \text{EndHost} \\
\land \text{Mac} \neq \text{DstMac}
\]

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 \(\langle \text{Src} : \text{H6}, \text{Dst} : \text{H3} \rangle\) through port 2 from the upstream switch S2 (\(\varphi\)). 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 (\(\varphi\)). The controller then instructs S1 to broadcast the packet to all neighbors except S2 (\(\varphi\)). However, before Server H4 receives the broadcast packet, a new packet \(\langle \text{Src} : \text{H3}, \text{Dst} : \text{H4} \rangle\) could reach switch S1 (\(\varphi\)), triggering an ofPacket message to the controller (\(\varphi\)). The controller would then set up a new flow entry at switch S1, matching destination H3 (\(\varphi\)). It is possible that due to network delay, server H4 receives its copy of the broadcast packet just now (\(\varphi\)). Therefore, the execution trace generates packet (H4,S1,H6,H3), swToHs (S1,H4,1) (i.e. the link between S1 and H4), and flowEntry (S1,H3,2,1), with \(\text{Mac} = \text{DstMac} = \text{H3}\).

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:

\[
\varphi_{\text{ESL3}} = \\
\forall \text{EndHost}, \text{Switch}, \text{SrcMac}, \text{DstMac}, \text{OpPort}, \\
\text{packet}(\text{EndHost}, \text{Switch}, \text{SrcMac}, \text{DstMac}) \\
\land \text{swToHs}(\text{Switch}, \text{EndHost}, \text{OpPort}) \\
\land \text{DstMac} = \text{EndHost} \\
\land \text{switch}'(\text{Mac}, \text{OutPort}, \text{Priority}), \\
\text{flowEntry}(\text{switch}', \text{Mac}, \text{OutPort}, \text{Priority}) \\
\land \text{switch}' = \text{switch} \land \text{Mac} = \text{DstMac}
\]

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 (\(\varphi\)). Since there is no flow entry in the forwarding table to match the destination address, switch S1 informs the controller of the received packet (\(\varphi\)), and further broadcasts the packet under the controller’s instruction (\(\varphi\)). In this way, server H4 does receive a packet destined to it (\(\varphi\)), but switch S1 does not have a flow entry matching H4.
### 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(@OutNet, Switch, SrcMac, DstMac)**: OutNet received a packet from Switch that was sent by a host with mac address SrcMac to a target host with mac address DstMac.
- **swToHost(@Switch, OutNet, OutPort)**: Switch is connected to OutNet 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

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

Table 2. Ethernet Source Learning Rules

With further inspection, the above counterexamples, 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).

\[
\forall \text{WorkFW} =
\forall \text{Host, Port, Src, SrcPort, Switch, pktReceived(Host, Port, Src, SrcPort, Switch) \land Cntrl, trustedControllerMemory(@Cntrl, Switch, Src)}
\]

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 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 result** 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.

**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 config-
property of declarative programs. Notably, Wang et al. [47, 48] developed a proof system for proving correctness properties of networking protocols specified in NDL, 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 NDL 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 NDL. 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 AFSOR Young Investigator Award FA9550-12-1-0327.

References