Static Differential Program Analysis for Software-Defined Networks

Tim Nelson, Andrew D. Ferguson, and Shriram Krishnamurthi

Brown University

Abstract. Networks are increasingly controlled by software, and bad updates can bring down an entire network. Network operators therefore need tools to determine the impact of changes. To address this, we present *static differential analysis* of software-defined network (SDN) controller programs. Given two versions of a controller program our tool, Chimp, builds atop Alloy to produce a set of concrete scenarios where the programs differ in their behavior. Chimp thus enables network developers to exploit the power of formal methods tools without having to be trained in formal logic or property elicitation. Furthermore, we show that there are many interesting properties that one can state about the changes themselves. Our evaluation shows that Chimp is fast, returning scenarios in under a second on several real applications.

1 Introduction

Traditional networks run individually-configured, autonomous switches that are often closed, proprietary hardware. In a software-defined network (SDN) [7], switches defer control of their behavior—and by extension, of the network—to a logically centralized server (the "controller"), which may be anything from a single commodity machine to a distributed cluster. The controller executes programs that—by updating state, interacting with other programs, and sending instructions to switches—collectively implement the network's behavior, ranging from standard network operations to novel behaviors unseen in traditional networks. SDN has been adopted by companies such as VMware (for its virtual-network products [20]) and Google (for its backbone network [15]). Programs may be written in arbitrary languages; beyond traditional languages, this is leading to a resurgence of declarative languages—like Flog [16], NLog [20] from Nicira/VMware, and Flowlog [27]—which are the focus of this work.

In this paper, we target the *evolution* of controller software. Programs evolve for many reasons: due to a bug fix, feature update, refactoring, etc. Developers need robust techniques to manage evolution because mistakes can cause an entire network to malfunction. Techniques like testing and verification are, however, only as effective as the coverage provided by their inputs; they may require a knowledge of logic that operators may not have; and most of all, they *only check what was stated.* However, when we add a new feature, we do not write extensive properties or tests about parts of the system that are *unrelated.* Therefore, developers need techniques that have the ability to (perhaps unpleasantly) surprise.

We therefore present *differential analysis* for SDN controller programs, which presents the *semantic* or *behavioral* difference between two versions of a program. The core analysis only needs two versions; it does not require tests or logical properties. The output of our tool, Chimp (short for "*Ch*ange *imp*act"), is in terms of *scenarios*: concrete situations where the two programs differ. The overall goal of differential analysis is to help developers *transfer trust between versions*; if they had faith in the proper execution of an old version, the semantic differences help them focus on the only things they need to examine to extend that faith to the new version. (We contrast other forms of differential analysis in Sec. 8.)

Chimp also enables users to query differences and even verify properties of those differences. This enables many more use-cases. For instance, when a program is merely being refactored or otherwise cleaned up, there should be no behavioral change; a Chimp user can check such *differential properties*, and any counter-examples would need attention. Chimp can even contrast multiple ways of extending a program, i.e., compute the difference of differences.

This paper makes the following contributions:

1. it identifies the problem of static differential program analysis for SDNs (Sec. 2 and 3);
2. it discusses and overcomes potential challenges and pitfalls in this analysis (Sec. 2 and 5);
3. it demonstrates that this analysis can be done effectively (Sec. 6 and 7); and
4. it shows how traditional properties and differential analysis meet at *differential properties* (Sec. 2 and 4).

In short, Chimp represents a fruitful application of various formal methods to a novel and important domain.

2 Differential Analysis at Work

The work in this paper targets the class of declarative languages used to program SDNs. We focus on Flowlog [27], which is richer in expressiveness than most of the others, so that our work is most widely applicable. Beyond the details of tooling, we believe the core ideas of Chimp apply equally well to other languages (which we discuss in Sec. 8).

In an SDN, many traditional network operations are implemented entirely in software. We illustrate this with a well known networking example: Network-Address Translation (NAT), which is used widely (e.g., to multiplex multiple home machines over a shared router). Figure 1 shows two different implementations of NAT in Flowlog—an initial version without the underlined code, and a second version with it. For simplicity, this example involves a home router with only two ports (1=inside, 2=outside). The external interface uses the hardware, or MAC, address `00:00:00:00:00:FF` and is assigned the IP address `192.168.100.100`. The core ideas in this example are the same for larger devices.

Lines 1 and 2 define the controller's state schema. A database table, `nat`, stores the current NAT mappings; its first columns identify packets by source,

```
 1 TABLE nat(macaddr,ipaddr,tpport,tpport);
 2 VAR nextport: tpport = 10000;
 3
 4 ON tcp_packet(p) WHERE p.locPt = 1 AND
 5     nat(p.dlSrc,p.nwSrc,p.tpSrc,natport):
 6   DO forward(new) WHERE
 7     new.tpSrc = natport AND
 8     new.nwSrc = 192.168.100.100 AND
 9     new.locPt = 2 AND
10     new.dlSrc = 00:00:00:00:00:FF;
11
12 ON tcp_packet(p) WHERE p.locPt = 1 AND
13     NOT nat(p.dlSrc,p.nwSrc,p.tpSrc,ANY):
14   INSERT (p.dlSrc,p.nwSrc,p.tpSrc,nextport) INTO nat;
15   DO forward(new) WHERE
16     new.nwSrc = 192.168.100.100 AND
17     new.locPt = 2 AND
18     new.tpSrc = nextport AND
19     new.dlSrc = 00:00:00:00:00:FF;
20   INCREMENT nextport;
```

Packets arrive at a port (`locPt`) on a switch (`locSw`). Their headers contain a source (`dlSrc`) and destination (`dlDst`) hardware (or MAC) address. Transport Control Protocol (TCP) packets also have a source (`nwSrc`) and destination (`nwDst`) network address and a source (`tpSrc`) and destination (`tpDst`) service port.

Fig. 1. Network-address translation (NAT) in Flowlog. An initial version did not include the underlined portions, and so failed to translate MAC addresses as well as IP addresses.

and its final column gives an ephemeral TCP port to use in the translation. The original version identifies packets by IP address and initial TCP port; the modified version also uses MAC address. A variable (`nextport`) holds the next available TCP port for NAT to use, starting at 10000. Lines 4–10 handle outgoing packets for which a translation already exists, and lines 12–20 process outgoing packets that start new ones. We elide the code to handle return traffic, which only adds an additional 5 or 6 lines.

Each Flowlog program rule can be thought of as a database view over the program's current state and the incoming event. As in SQL or Datalog, Flowlog rule bodies constrain that view, dictating which actions the program can take. Lines 4–5 say that the forwarding action on lines 6–10 applies only for TCP packets arriving at port 1 (the internal port), where there is a matching row in the current `nat` table. It also binds the value in the final column of that row to the variable `natport`, which is used later. Lines 6–10 say to forward matching packets, but to first modify their IP source to `192.168.100.100` (the assigned IP address) and their TCP source to the value in `natport` (obtained from the `nat` table). In the modified version, the rule also sets the MAC source to `00:00:00:00:00:FF`. Lines 12–21 work similarly, but since there is no corresponding row in the `nat` table, they use the next available free port. They then insert the appropriate new row into the table and increment the `nextport` variable.

While the original only modifies a packet's IP and TCP fields, the second version (with underlines) also changes MAC sources to reflect that the modified

Pre-State: nat = {}, nextport = 10000

Incoming event: Packet with MAC src = `Mac 0`, IP src = `Ip 0`, TCP src = `TCP 0`

Program 1 Post-State:
nat = $\{(\text{Ip } 0, \text{TCP } 0, 10000)\}$
nextport = 10001
	Program 2 outputs packet with:
	MAC src = `00:00:00:00:00:FF`,
	IP src = `192.168.100.100`,
	TCP src = `10000`

Program 2 Post-State:
nat = $\{(\textbf{Mac 0}, \text{Ip } 0, \text{TCP } 0, 10000)\}$
nextport = 10001
	Program 1 does not output this packet.

Fig. 2. Scenarios returned by Chimp for the NAT change. In this case, both share the same pre-state and arrival event. The scenario on the left shows a *state-transition* difference caused by the new program storing the MAC source address. The scenario on the right shows that in the revised program, the packet's MAC address is modified. `Mac 0` denotes an arbitrary MAC address (and similarly for other fields).

packet comes from the outgoing interface—standard behavior for a NAT. In order to modify addresses consistently, the new program adds a column to the `nat` table that holds the source MAC address of each NAT flow. We will now use Chimp to analyze the semantic consequences of this edit.

If we view a Flowlog program as a function that processes events, a change may have two types of semantic impact: given the same input, either the programs produce different output (e.g., forward packets differently) or they transition to different states. Chimp defines a built-in analysis for each of these: (1) `chPolOut` ("change policy output") which generates scenarios that show any differing output behavior, and (2) `chStTrans` ("change state transitions"), which shows the differences in how the two programs evolve their state. Users may select from these (and other built-in analyses, which we discuss later) or construct their own using these as a starting point.

Chimp's output provides concrete *scenarios* that show how programs can express the behavior described. When seeking semantic differences via `chPolOut` or `chStTrans`, each scenario contains a *prestate* that shows the state of the two programs before they diverge, and a *trigger event* for the divergence. Scenarios for other analyses may contain different information as requested by the user.

Figure 2 shows output scenarios for both `chPolOut` and `chStTrans` on the NAT program edit. Both show a packet arriving at the internal interface; the scenario on the left shows a state-transition difference and the scenario on the right shows a behavioral difference. The `nat` tables in these scenarios are empty because their value is immaterial for this specific behavior, and Chimp is designed to only produce *minimal* scenarios, which greatly improves the quality and brevity of output (Sec. 7).

Schema Combination Every Flowlog program has a *schema*: a set of **`TABLE`** declarations, each of which includes a list of data-types that define that table's columns. A *schema clash* occurs whenever two programs declare different arities, types, or column orderings for the same table. The happens between the NAT programs (Figure 1, line 1), as the modified program adds a column to the `nat` table. This clash must be resolved in order for Chimp to have a consistent notion of "pre-state" for its analysis. To do so, Chimp creates a new version of the conflicted table for each clashing program. For the NAT example, it creates a separate three-column `nat1` table and four-column `nat2` table. Chimp then rewrites the original program to refer only to `nat1` and the modified version to `nat2`. This presents a new challenge: output scenarios will now contain both tables, and Chimp's search for scenarios will treat the two new tables independently. Since it searches all possibilities, Chimp will consider cases where (e.g.) the `nat1` table is empty but the `nat2` table is not. Scenarios where the two programs' states bear no relationship to one another may seem spurious—since the two tables were originally one, their contents should be somehow related.

Lockstep Constraints The programmer might assert that, since the new `nat` table is just an extension of the first, the two tables should be identical in the final three columns. Formally, they would like to restrict Chimp's search to scenarios where it holds that: "Every row in `nat2` (minus its first column) is also in `nat1`; every row in `nat1` is also in `nat2` (with some MAC address in the first column)" or, in logical form:

$$\forall i, p_1, p_2\ (\exists m\ nat_2(m, i, p_1, p_2)) \iff nat_1(i, p_1, p_2)$$

We call this a *lockstep constraint* because it expresses how two programs evolve their states together. It represents an intuition about the intent of the table change. Constraints like this may be added to an analysis, analogously to adding new conditions to a SQL statement. This process lends itself to iteration, with refinements growing ever more focused as the user zeroes in on surprising behavior.

Differential Properties Before we assert the lockstep constraint—and prevent Chimp from returning scenarios that violate it—we would like to validate the intuition it represents. To that end, we can phrase the lockstep constraint as a *differential property*, i.e., a property that spans the behavior of multiple programs, and check it in Chimp. When checking a lockstep constraint in this way, we refer to it as a lockstep property. We proceed inductively. To verify the base case, we check that the property always holds in the (empty) initial program state. The bi-implication in the above property makes this trivially true. It then suffices to check whether the programs can ever violate the property as their states evolve. To do so, we phrase the property as a custom analysis predicate (Sec. 4) and ask Chimp for counterexamples.

Perhaps surprisingly, Chimp produces a counterexample (Figure 3). This scenario shows a pre-state that respects the property (one row in each table), but a post-state that does not: a second entry, using a fresh external port, has been added to `nat2` but not to `nat1`. This means that either the revised program is wrong, or the property itself is incorrect (reflecting faulty intuition).

The revised program correctly creates a new entry for packets with a new MAC source, even if its IP source is already in the table. This is to be expected: since the MAC sources are distinct, the packets involve separate physical machines and must be handled separately. Thus, seeing this scenario corrects the programmer's

Pre-State:

nat1 =

`Ipaddr 0`	`Tpport 0`	`Tpport 2`

nat2 =

`Macaddr 0`	`Ipaddr 0`	`Tpport 0`	`Tpport 2`

Incoming event: TCP packet from:
MAC = `Macaddr 1`, IP = `Ipaddr 0`, TCP Port = `Tpport 0`

Program 1 Post-State: no change

Program 2 Post-State:

nat2 =

`Macaddr 0`	`Ipaddr 0`	`Tpport 0`	`Tpport 2`
`Macaddr 1`	`Ipaddr 0`	`Tpport 0`	`Tpport 0`

Fig. 3. Failure of the first NAT lockstep property. Abstract values `Macaddr 0`, `Tpport 0`, ... denote disjoint arbitrary addresses, ports, etc.

intuition and informs them that the new program has actually fixed a potential bug that they had not considered. Some reflection also leads to a more accurate constraint relating the two tables: "Every row in `nat1` is also in `nat2` (with some MAC address in the first column); for every row in `nat2` there is a row in `nat1` with the same source address and port,", or:

$$\forall i, p_1, p_2, m \; (nat_1(i, p_1, p_2) \implies \exists m' \; nat_2(m', i, p_1, p_2)) \\ \wedge (nat_2(m, i, p_1, p_2) \implies \exists p' \; nat_1(i, p_1, p'))$$

Chimp finds no counterexample to this new constraint, increasing confidence that it is correct. We now assert it in Chimp, forcing the two tables to be tightly coupled in each output scenario. As seen here, "obvious" intuitions about schema changes can be subtly wrong. Instead of assuming a standard lockstep constraint, or adding one automatically, Chimp lets users test their intuitions via analysis and then assert them explicitly. Errors revealed lead to *missing correctness properties* which can then be added to existing test- and property-suites.

3 Theory

Every Flowlog rule (an **`ON`** condition followed by a single action) is equivalent to a formula of first-order logic that defines the rule's meaning and enables formal reasoning. Figure 4 describes this translation in detail for rules, formulas, and terms; the translation for all rules produces the first-order theory of a Flowlog program. Flowlog's syntax is inspired by non-recursive Datalog with negation, and its logical semantics follows. Variables not explicitly quantified are interpreted universally, as in Datalog. The only exception is that the wildcard term `ANY` binds tighter than other terms; the formula `NOT R(ANY)` means $\neg\exists\, aR(a)$ (i.e, that the relation is empty). The translation inserts quantifiers to support this. Flowlog desugars rules with `OR` into multiple rules in the obvious way. The **`INCREMENT`** keyword is syntactic sugar for relational expressions plus **`INSERT`** and **`DELETE`** rules.

For each state relation symbol R, helper relations $plus_R$ and $minus_R$ (Figure 4) describe how that relation changes for each event received. If R is the relation

$$\begin{aligned}
[\![\texttt{ON in}(i)\texttt{: DO out}(o)\ \texttt{WHERE}\ F]\!] &= out(o) \leftarrow in(i) \wedge [\![\mathrm{F}]\!] \\
[\![\texttt{ON in}(i)\texttt{: INSERT}\ (o_1, ..., o_k)\ \texttt{INTO R WHERE}\ F]\!] &= plus_R(o_1, ...o_k) \leftarrow in(i) \wedge [\![\mathrm{F}]\!]) \\
[\![\texttt{ON in}(i)\texttt{: DELETE}\ (o_1, ..., o_k)\ \texttt{FROM R WHERE}\ F]\!] &= minus_R(o_1, ...o_k) \leftarrow in(i) \wedge [\![\mathrm{F}]\!])
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{NOT f}]\!] &= \neg[\![\texttt{f}]\!] \\
[\![\texttt{f1 AND f2}]\!] &= [\![\texttt{f1}]\!] \wedge [\![\texttt{f2}]\!] \\
[\![\texttt{t1 = t2}]\!] &= [\![\texttt{t1}]\!] = [\![\texttt{t2}]\!] \\
[\![\texttt{R(t1, ..., tk)}]\!] &= \exists any_1, ..., any_m\ R([\![\texttt{t1}]\!], ..., [\![\texttt{tk}]\!]) \\
&\quad \text{each } any_i \text{ has fresh index for every occurrence of } \texttt{ANY}.
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{c}]\!] &= c \text{ (for a constant } c) \\
[\![\texttt{x}]\!] &= x \text{ (for a variable } v) \\
[\![\texttt{x.fld}]\!] &= fld(x) \text{ (for a variable } x \text{ and packet field name } fld) \\
[\![\texttt{ANY}]\!] &= any_f \text{ (where } f \text{ is a fresh index)}
\end{aligned}$$

Fig. 4. Translation of Flowlog (rules, formulas, and terms) to FOL.

before an event arrives, then the new value of the relation will be:

$$(R \setminus minus_R) \cup plus_R$$

(That is, **INSERT** overrides **DELETE**.)

Flowlog disallows rule bodies that reference intensional relations (those defined by the program, e.g., `forward`, rather than stored in the program's state, e.g., `nat`). Also, rules must be *safe*: all variables (and output packet fields, in the case of a **DO** rule) in a rule's head and variables in negated body literals must appear in a non-negated literal in that rule's body.

Property-Checking The theory of a Flowlog program, Γ, is given by taking the union of the result of Figure 4 for each rule. It contains an implication for each rule, where each rule body dictates a class of input events and program states and each head gives a corresponding program behavior. For analysis, we take the theory's *Clark completion* [1, p. 407], Γ_c, which essentially adds reverse implications that define the ways each action could be caused.

Since Γ_c defines both the consequences of arriving events and the possible triggers for a given behavior, a first-order property ϕ holds if and only if $\Gamma_c \cup \{\neg\phi\}$ is unsatisfiable. For example, the program in Figure 1 always translates TCP packets' IP source to `192.168.100.100` only if the following formula is *unsatisfiable* in conjunction with the completion of the program's theory:

$$\exists p, p' \,.\, forward(p') \wedge tcp\_packet(p) \wedge nwsrc(p') \neq 192.168.100.100$$

Models that satisfy the conjunction are counterexamples to the property.

Differential Properties ϕ may also involve more than one program; it can express *differential* properties over multiple programs' collective behavior. The 2-program `chPolOut` analysis of Sec. 2, for example, corresponds to:

$$\begin{aligned}
\exists p, p' \,.\, tcp\_packet(p) \wedge (&forward_1(p') \wedge \neg forward_2(p') \vee \\
&forward_2(p') \wedge \neg forward_1(p'))
\end{aligned}$$

where each $forward_i$ represents the $forward$ relation of the i^{th} program. Chimp automatically performs this renaming for all output and state-modification relations, and we will use the notation freely when it is clear from context.

4 Flowlog to Alloy

Flowlog's runtime automatically updates the network's switches as needed; the language abstracts out the specifics of those updates and associated optimizations. Because of this *tierless* abstraction, Flowlog's first-order logic semantics can be used directly to reason about program behavior. Since Alloy supports predicate logic, producing an Alloy specification for a Flowlog program essentially involves following Figure 4. Chimp also defines several built-in analysis predicates.

Single-Program Predicates For every action that a program can take, Chimp creates an Alloy predicate representing when that action occurs. For instance, the packet-forwarding action for each program produces a predicate with signature[1]:

```
pred forward[st: State, e: EVpacket, out: EVpacket]
```

A `State` atom represents a database over the program's schema. Events (types starting with `EV`) are generalizations of packets that also include external events or notifications from other modules. Predicates are true or false on any given input. The `forward` predicate holds on inputs consisting of a `State st`, an input `Event e`, and an output `Event out` if and only if the original program would forward `e` with the modifications expressed in `out`. Figure 5 lists other predicates that Chimp creates for each program. The higher-level predicate `outpolicy` holds any time the program will respond to event `ev` by emitting `ev2` when in state `st`, and `transition` expresses when a state-transition is taken on an event.

Cross-Program Predicates Chimp also constructs basic cross-program differential analysis predicates (Figure 5) for each pair of programs given. To detect a difference in two programs' output (`chPolOut`), Chimp looks for an output (`outev`) that is produced by one program but not another:

```
pred chPolOut_1_2[st: State, ev: Event] {
  some out: Event |
   prog1/outpolicy[st,ev,out] && not prog2/outpolicy[st,ev,out] ||
   prog2/outpolicy[st,ev,out] && not prog1/outpolicy[st,ev,out] }
```

The names `prog1` and `prog2` denote the two programs; each has its own `outpolicy`. Since `outpolicy` is used, rather than any specific output action, this predicate is more general than the example formula in Sec. 3. `chStTrans` is defined similarly, checking for a mismatch in `plus_R` or `minus_R` behavior.

Custom Predicates Users may create their own analyses in the Alloy language, usually by building atop Chimp's built-in predicates. For instance, the first differential property from Sec. 2 can be expressed as:

[1] We have removed some machine-generated typing information and other Alloy-language foibles for brevity. Each Flowlog program produces a separate Alloy module.

Predicate	Arguments	True if the program in state s:
`plus_R`	$s : State, e : Event, t_0, ..., t_k$	adds $t_0, ..., t_k$ to `R` when receiving e
`minus_R`	$s : State, e : Event, t_0, ..., t_k$	removes $t_0, ..., t_k$ from `R` when receiving e
`<action>`	$s : State, e_1 : Event, e_2 : Event$	outputs e_2 (of type `<action>`) on e_1
`outpolicy`	$s : State, e_1, e_2 : Event$	outputs e_2 on receiving e_1
`transition`	$s : State, e : Event, s_2 : State$	transitions to s_2 on event e

Predicate	Arguments	True if the programs:
`chPolOut`	$s : State, e : Event$	have different output on event e in state s
`chStTrans`	$s : State, e : Event$	diverge in state on event e in state s
`rchChPolOut`	$s, s2 : State, e : Event$	`chPolOut` with reachability check for s
`rchChStTrans`	$s, s2 : State, e : Event$	`chStTrans` with reachability check for s

Fig. 5. Built-in predicates for each program and differential-analysis. `<action>` and `R` denote arbitrary output actions and table names.

```
pred lockstep_nat_condition[st1, st2: State] {
  all x1, x2, x3 : univ |
    (some x4 : univ | x4 -> x1 -> x2 -> x3 in st2.nat_2) iff
    (x1 -> x2 -> x3 in st1.nat_1) }

assert lockstep_nat_assert {
  all st, st1', st2': State, ev: Event |
    (lockstep_nat_condition[st, st] and
     prog1/transition[st,ev,st1'] and prog2/transition[st,ev,st2'])
    implies
      lockstep_nat_condition[st1', st2'] }
```

using the `transition` predicate from Figure 5. The `lockstep_nat_condition` predicate identifies pairs of "safe" states that satisfy the condition. The assertion seeks a scenario where the programs transition from a "safe" to an "unsafe" state. A single pre-state suffices since the two programs' `nat` tables are held separately.

5 Soundness and Completeness

As is standard for such tools (including Alloy), Chimp performs *bounded* scenario finding. Along with an analysis predicate, users provide bounds for each datatype, e.g. up to 6 switches, 4 IP addresses, and so on. The search is guaranteed to be *sound* (with a caveat below); it never returns a false positive. Given its boundedness, one might reasonably inquire whether it can issue false negatives.

Every rule body is in the $\exists^*\forall^*$ fragment of first-order logic, which is well-known [5, 33] to admit bounded satisfiability-checking. Positive instantiations of those bodies, as in the definition of `chPolOut`, are also in that fragment. However, negative instantiations (also used in `chPolOut` and others) are not.

A *cyclic* Flowlog program is one in which some `ANY` term appears together with a rule-body variable in a negated body atom. For instance, a program containing the body atom `NOT R(x, ANY)`, where `x` does not appear in the rule head, is cyclic. If a program is *acyclic*, elementary rewrites can break all $\forall\exists$ nesting in negated rule bodies. Thus, `chPolOut` and `chStTrans` over acyclic programs admit automatically-generated sufficient bounds. Acyclic Flowlog is expressive; every program in Sec. 7 is acyclic.

For cyclic programs, since both Flowlog and Alloy have a notion of types (in contrast to standard untyped first-order logic), it is possible [30] to strengthen the $\exists^*\forall^*$ condition to safely bound analysis involving limited $\forall\exists$ quantification. Chimp makes use of this information to produce bounds, and `chPolOut` and `chStTrans` are therefore complete even on many cyclic programs. Custom queries can of course introduce additional quantification that renders Chimp incomplete—e.g., as in the lockstep property of Sec. 2.

Thus, while Chimp is incomplete in general, many useful analyses have a bound under which Chimp is guaranteed to find any counterexamples; Chimp computes this bound automatically. Unlike prior work [30] on completeness in Alloy, which naively counts all well-typed ground terms, Chimp takes advantage of implicit disjunction in the analysis formulas. For instance, when seeking localized differences, there is no need to consider quantification in both `forward` and `plus_R` rules simultaneously. This produces tighter bounds that are sufficient to detect *any single semantic difference*, benefiting both performance and scenario brevity.

Where sufficient bounds cannot be established automatically, users provide bounds manually. As in Alloy, domain knowledge often eases this process. As Jackson [14] notes, even the most insidious bugs often occur on small example runs. Incomplete differential analyses can thus be viewed as a form of automated bug-finding that increases confidence in the program.

Soundness of Addition The original Flowlog-to-Alloy translator [27] did not support Flowlog's `add` primitive. While Alloy has a notion of integers, they are bounded by a user-provided bitwidth—a fact that made using Alloy integers impractical. Instead, Chimp represents addition via a ternary relation. By default, this under-approximates true addition, sacrificing a measure of soundness for tractability. In practice, we insert additional axioms for arithmetic as needed.

Pre-State Reachability By default, Chimp does not guarantee that prestates of scenarios it returns are reachable in real program runs. This does not render Chimp "unsound": such scenarios still witness a program state and input on which the two programs differ, and even an unreachable semantic difference can yield new insight into the programs. Nevertheless, it can be valuable to ignore unreachable scenarios and see a concrete execution trace that shows how the scenario can be reached. For these reasons, Chimp can enhance its search with full system traces. The reachability-aware analyses `rchChPolOut` ("reachable `chPolOut`") and `rchChStTrans` ("reachable `chStTrans`") behave much like their counterparts, but the scenarios they produce are augmented with separate system traces for each program. Separate traces are necessary since, if the programs' states eventually diverge, a single trace would be unable to capture behavioral differences that happen *after* divergence. Users may expect the state difference but not any subsequent deltas; a single trace could therefore conceal surprising differences. To use reachability-aware analyses, users must provide a maximum trace length to check up to. As (e.g.) a pre-state that requires 3 steps to reach will not be detected if the user-provided bound is 2 steps, reachability-awareness can cause a loss of completeness. It also negatively impacts performance, since longer traces mean a larger space of possible scenarios to search. It is up to the user to decide whether to make this tradeoff—losing completeness and performance in exchange for scenario provenance and reachability guarantees.

6 Scenario Minimization

Scenarios require user effort to understand, and needless detail increases the time taken to comprehend them as well as reducing the generality of each individual scenario. Because of this this, Chimp provably presents only minimal scenarios. Formally, let Γ be the first-order theory of a Flowlog program plus additional first-order constraints, such as the negation of properties (Sec. 3). Define the set of scenarios that satisfies Γ as $scns(\Gamma) = \{\mathbb{S} \mid \mathbb{S} \models \Gamma\}$ and the relation $\subseteq$ on scenarios to denote containment of relational facts, that is: $\mathbb{S}_1 \subseteq \mathbb{S}_2$ if and only if all facts $R(a_1, ..., a_n)$ true in $\mathbb{S}_1$ are also true in $\mathbb{S}_2$. Now the set of $\subseteq$-*minimal* scenarios for Γ is $mins(\Gamma) = \{\mathbb{S} \in scns(\Gamma) \mid \forall \mathbb{S}' \in scns(\Gamma) \,.\, \mathbb{S}' \subseteq \mathbb{S} \implies \mathbb{S} = \mathbb{S}'\}$.

In other words, minimal scenarios contain only the facts they need to satisfy the theory. For instance, removing any row in Figure 3 would either make the scenario inconsistent (i.e., not reflective of valid system behavior) or no longer satisfy the analysis predicate. Minimality also forces the use of abstract variables whenever possible. Chimp will not give a packet field or table cell a concrete value unless the scenario is contingent on that value. Otherwise, it will use an abstract value (e.g., `Macaddr 0` in Figure 3); this is key in reducing the number of scenarios given and improving the usefulness of each. To implement minimization, Chimp leverages Aluminum [28], a modified version of Alloy that iteratively removes unnecessary facts before presenting scenarios. As we will see, minimization can result in a drastic decrease in scenario size.

7 Evaluation

Our experiments include differential analyses across several programs: the NAT application from Sec. 2 (NAT); a learning-switch implementation (MAC); an address-resolution protocol (ARP) cache; a round-robin load-balancer (LB); a network-information base (NIB) that computes reachability and spanning-tree information; and a stolen-laptop detector (SL) that sends alerts if suspect addresses are seen on the network. Due to the conciseness of declarative, rule-based programming in this domain, these programs are each modest in size. Nevertheless, together they comprise a significant library of standard network functionality as well as some new behavior made possible by SDNs.

For NAT, we compare the two versions from Sec. 2 with the correct lockstep condition added, along with checking both lockstep properties. For MAC, we compare versions with and without support for host mobility. For ARP, we compare three consecutive diffs: two bugfixes (1→2 and 2→3), and a refactoring

(3→4). For LB we check a bug-fix involving initialization of the controller state. The bug manifests as improper forwarding behavior after initialization, and so we enable reachability-aware analysis here. For NIB, we check a fix to how network-reachability is calculated. Originally, SL sends notifications for every suspicious packet; we compare this to a buggy new version intended to rate-limit notifications (1→2) and that version to the correct new version (2→3), as well as examine the difference-of-differences between these changes: (1→2) vs. (1→3).

Performance and Scenario Counts Figure 6 reports the number of scenarios Chimp returns (under the corresponding bound in columns 3–7, which we discuss later), as well as Chimp's performance on each analysis. The first two columns name the program(s) and the analysis performed. The eighth column gives the number of scenarios found. It is Chimp's goal to present surprising scenarios to the user, but it cannot know ahead of time which scenarios will be most valuable. A small number of scenarios that nevertheless illustrate all potential semantic changes is therefore good in principle. Minimization plays a major role here, as even the stolen-laptop changes (with no more than 4 minimal scenarios) produce hundreds of non-minimal scenarios, many of which are (unnecessarily) as large as the bounds permit. Our experience indicates that the first scenario presented is generally interesting, especially for user-defined queries, and larger scenario-counts are to be expected when the programs differ broadly.

The final columns of Figure 6 report on runtime. Chimp first *translates* the problem to Boolean logic before *solving* to find a scenario. We report the time for both steps as the average and standard deviations of 10 runs; Chimp was started afresh each time to mitigate cache-warming effects. The solving time is the time to either produce the first scenario or complete the search without finding one. We measure performance on an Intel i5-2400 3.10 Ghz with 8GB RAM (i.e., a generic laptop). The search is largely CPU-bound, using no more than 1.5 GB of memory even on the larger analyses. Chimp returns scenarios fairly quickly—under a second, for most analyses—even when there are no results, and it must complete a search of the entire scenario-space.

Computed Bounds and Scenario Sizes Columns 3–7 of Figure 6 report on bounds and the size of scenarios that Chimp presents. We show bounds for each datatype separately. The **B** subcolumn reflects whether Chimp was able to compute a guaranteed-sufficient bound (Sec. 5); a ✗ indicates a bound could not be computed, in which case parenthetical values indicate bounds we manually provided to Chimp. Sometimes, even when a sufficient bound is available, a technical limitation in the Alloy engine—a cap on the number of potential facts value that we were unable to modify with reasonable effort—prevents us from using that bound, in which case we use a smaller number (indicated in parentheses). This is only a restriction imposed by our current toolchain, and not a fundamental limitation. As expected, Chimp is able to find a bound for each `chPolOut` and `chStTrans`, rendering its search complete on these rows.

Programs	Analysis	MAC		IP		TCP		Events		States		Scenario	Trans (ms)		Solve (ms)	
		B	S	B	S	B	S	B	S	B	S	Count	**Avg**	σ	**Avg**	σ
NAT	`chPolOut` & Lckstp 2	✗(4)	2	✗(4)	1	✗(4)	2	2	2	1	1	>1000	438	118	112	4
	Lckstp 1	✗(3)	2	✗(3)	2	✗(3)	3	✗(1)	1	✗(3)	2	54	42	2	104	7
	Lckstp 2	✗(3)	–	✗(3)	–	✗(3)	–	✗(1)	–	✗(3)	–	0	55	4	141	4
MAC	`chPolOut`	4	–	0	–	0	–	2	–	1	–	0	6	1	1	1
	`chStTrans`	4	1	0	0	0	0	1	1	3	2	4	19	14	11	8
ARP (1→2)	`chPolOut`	13(3)	3	6(3)	1	0	0	2	2	1	1	154	33	28	20	18
ARP (2→3)	`chPolOut`	13(3)	3	6(3)	1	0	0	2	2	1	1	102	35	28	23	19
ARP (3→4)	`chPolOut`	12(3)	3	6(3)	1	0	0	2	2	1	1	324	28	24	10	11
LB	`chStTrans`	4	0	0	0	0	0	1	1	3	2	1	45	31	46	37
	`chPolOut`	4	–	0	–	0	–	2	–	1	–	0	25	2	1	1
	`rchChStTrans`	✗(4)	2	0	0	0	0	✗(3)	3	✗(3)	3	3	167	8	6011	51
	`rchChPolOut`	✗(4)	1	0	0	0	0	✗(4)	4	✗(5)	4	8	330	151	89524	555
NIB	`chStTrans`	4	0	0	0	0	0	2	2	1	1	40	242	114	2605	168
	`chPolOut`	4	–	0	–	0	–	2	–	1	–	0	3	1	1	1
Stolen Laptop	`chPolOut`	4	0	0	0	0	0	2	2	1	1	2	14	2	11	2
(1→2)	`chStTrans`	3	1	0	0	0	0	1	1	3	2	3	21	17	10	8
Stolen Laptop	`chPolOut`	4	2	0	0	0	0	2	2	1	1	4	21	7	14	8
(2→3)	`chStTrans`	2	–	0	–	0	–	1	–	3	–	0	20	13	5	4
Stolen Laptop	$\Delta(p_1, p_3) - \Delta(p_1, p_2)$	4	1	0	0	0	0	2	2	1	1	4	26	21	16	13

Fig. 6. Bounds computed, scenario sizes, number of minimal scenarios found, and performance (average and standard deviation in ms) for each analysis. For each datatype, the **B** column denotes the bounds used. A number m by itself indicates the computed sufficient bound on that datatype, which was then used in analysis. $m(n)$ says that the computed bound was m, but that we lowered it to a more reasonable n. ✗(n) denotes that a sufficient bound could not be calculated, in which case we provided a reasonable bound n. Lower numbers indicate a smaller search space. A bound of 0 means that atoms of that type were provably unnecessary in the analysis. The **S** column gives, for each datatype, the median number of atoms of that type used across all scenarios that Chimp found. A "–" indicates that the analysis found no scenarios. For reachability-aware analyses, the maximum trace-length is equal to the bound on events.

A bound exists for the delta-of-delta analysis as well. In contrast, Chimp is unable to find sufficient bounds for checking the lockstep properties, as they use quantification in a more sophisticated way. As for reachability-aware tests, the `rchChPolOut` and `rchChStTrans` predicates admit a sufficient bound on scenario-size *per step*, thus requiring user input only to bound the number of events and states. Since the LB programs differ only in their initialization (i.e., state change, detected by `chStTrans`) `chPolOut` detects no functional differences. `rchChPol-Out`, however, entails a search for the *consequences* of that change beyond its immediate effect on program state; this adds significant complexity to the search.

Scenarios, especially after minimization (Sec. 6), may not need as many elements as the bound indicates. Therefore, the **S** column presents the median number of elements across all scenarios that Chimp returned. Where available, the **B** values are quite small. However, even relative to those, the **S** values are smaller; minimal scenarios contain no irrelevant output. We see that minimizing scenarios before presenting them often reduces scenario-size by more than 50 percent. Since simpler scenarios are quicker and easier to understand, this significantly assists the user in focusing on the critical components of the change.

8 Related Work

Controller programs operate at multiple *tiers* of execution, analogous to the multi-tier nature of web programs. In particular, controller programs generate persistent instructions for switches, making this a form of metaprogramming, which can be especially hard on a static analysis. Analysis is eased in *tierless* languages like Flowlog [27], which abstract out the details of how the controller interacts with the switches. To exploit tierlessness, Chimp is specifically targeted to Flowlog, but its core ideas are not limited to one language. VMware's Nlog [20] is, like Flowlog, based in non-recursive logic-programming and has relational state, making it a prime candidate for Chimp. CSDN [4], in spite of its imperative syntax, also has relational state and a trigger-action model similar to Flowlog's. Since CSDN is not tierless, analysis would need to model switch-rule updates explicitly, yet it is amenable to relational modeling. Flog [16] is another limited-power logic-programming SDN language with relational state. Flog allows recursion, and Chimp's underlying engine assumes a non-recursive logic; Chimp is nevertheless applicable to Flog's non-recursive fragment. Chimp's methods also apply to stateless, declarative policy languages like NetCore [26].

NetKAT [3] is an SDN programming language that supports efficient [10] program differencing. NetKAT programs can express *path*-based constraints, but do not support program state. Differencing is therefore a fundamentally different problem between the two languages. NetKAT also supports host-reachability analysis that depends on the network topology; Chimp is topology-independent, and checks whether program states (not network hosts) are reachable.

Differential program analysis is well studied outside the networking space. Early work by Horwitz [13] finds which portions of two programs correspond and where they can differ. Chimp's custom predicates enable more detailed analyses, as well as providing behavioral scenarios rather than annotated code.

More recent work includes SymDiff [21], which leverages satisfiability modulo theories (SMT) technology for program comparison; Differential Assertion Checking (DAC) [22], which checks properties relative to program changes; and Differential Symbolic Execution (DSE) [31], which combines symbolic execution and SMT-solving to summarize differences between Java methods. Chimp's scenarios are analogous to the output of these tools, except that they use relational program state. Also in contrast to these tools, Chimp addresses schema clashes and differences in potential program input types, as well as reasoning about lockstep behavior. Since Chimp targets limited-power, declarative languages for network-programming, it is able to make completeness guarantees that cannot generally be made for full-featured languages, and does so without necessitating symbolic execution.

Hawblitzel, et al. [12] give a framework for comparing pairs of imperative programs via theorem provers. Like their mutual summaries, Chimp's analysis predicates describe relations over differential behavior, although mutual summaries do not assume a shared prestate by default as Chimp's basic analyses do. Unlike mutual summaries, Chimp's predicates can involve any number of Flowlog programs, as in the 3-way delta-of-deltas comparison of Sec. 7. Finally, Hawblitzel et al. do not discuss performance or brevity of output.

Chimp is partly inspired by Margrave [29], which performs differential analysis on policies such as firewalls and routing tables. Margrave accepts a limited subset of Cisco's IOS configuration language and supports additional input via an intermediate policy language; it is not designed for the SDN domain. Its policies are strictly weaker than Flowlog's: they can *read* relational state but not modify it, and they lack the ability to express even the limited universal quantification of Flowlog's ANY keyword. Margrave uses Kodkod [37], the same engine underlying Alloy and Chimp, and could thus perform some, but not all of the analyses that Chimp can—for instance, Margrave has no support for reasoning about state-reachability. Margrave bounds its analyses by naively summing all terms in a policy [30]; Chimp's focus on single-rule variations produces tighter bounds. Also in the firewall space, Liu [23] addresses change-impact analysis for firewall policies, not full SDN programs.

Dougherty et al. [9] split a program's behavior into a system automaton and a dynamic policy that filters which transitions can be taken, then give algorithms for computing the difference of multiple policies with respect to the fixed system. These algorithms assume a common schema between policies, whereas Chimp allows for schema changes. Even more, since each Flowlog program defines its own transition system, Chimp's analysis must effectively work with multiple system automata. Finally, their work is not implemented.

Differencing techniques in the network space tend to focus on stateless forwarding policies rather than stateful programs. For instance, header-space analysis [18] could be used to compare static views of the network. In contrast, we are interested in analyzing controller *programs*, with state that changes over time.

DNA (Differential Network Analysis) [24] answers differential queries about reachability across multiple snapshots of network state (e.g., routing tables and ACLs). Chimp does not reason about network reachability, as its analysis is topology independent. Since Chimp analyzes programs, rather than snapshots of forwarding policy, it must be aware of state transitions between these snapshots, and its analysis is necessarily more complex. Like Chimp, the DNA tool minimizes its output using Boolean techniques, but Chimp's minimization also works over relational program states as well as packet headers.

Chimp is complementary to statistical tools like WISE [36], which estimates the impact of changes on response times in content-delivery-networks. Chimp's reasoning functions even in the absence of pre-existing logs, which machine-learning tools such as WISE require to train their classifiers.

In contrast to differential analysis, traditional property-verification for SDN programs is well studied. However, existing tools such as NICE [6], VeriCon [4], Verificare [34], and Flowlog's existing verification [27] lack *differential* reasoning capabilities. The same is true of recent proof-based verification efforts [8, 35] for SDN languages. Many other analyses [2, 11, 18, 19, 25, 32, 38] work over *fixed* network policies, often accepting raw forwarding tables as input. While powerful, applying these techniques to *stateful* SDN controller programs means resorting to dynamic methods in the running system, as in the case of NetPlumber [17] and VeriFlow [19]. Chimp analyzes stateful programs statically.

9 Conclusion

Chimp was designed with several core goals in mind: to handle dynamic program state, to produce concrete scenarios and support schema changes (Sec. 2), to rule out false negatives but allow reachability-checking if desired (Sec. 5), to provide minimal, general scenario output (Sec. 6), and to support both common and user-defined queries (Sec. 4). As our evaluation shows, Chimp's performance is good enough to be used as a regular part of the development cycle. The tool currently analyzes *controller programs*, independent of the network's topology. It would also be useful to reason about network-condition changes, such as host mobility [39], and their potential impact on behavior. Improving Chimp's handling of arithmetic by incorporating SMT-solver technology would also be an interesting avenue of future work.

Acknowledgements We are grateful to the anonymous reviewers for their helpful remarks. We thank Daniel J. Dougherty, Kathi Fisler, Rodrigo Fonseca, and Nate Foster for useful discussions, and the Frenetic and Alloy teams for creating excellent tools we could build upon. This work is partly supported by the NSF.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Al-Shaer, E., Al-Haj, S.: FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In: Workshop on Assurable and Usable Security Configuration (2010)
3. Anderson, C.J., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: Semantic foundations for networks. In: Principles of Programming Languages (POPL) (2014)
4. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: VeriCon: Towards verifying controller programs in software-defined networks. In: Programming Language Design and Implementation (PLDI) (2014)
5. Bernays, P., Schönfinkel, M.: Zum entscheidungsproblem der mathematischen Logik. Mathematische Annalen 99, 342–372 (1928)
6. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE way to test OpenFlow applications. In: Networked Systems Design and Implementation (2012)
7. Casado, M., Freedman, M.J., Pettit, J., Luo, J., McKeown, N., Shenker, S.: Ethane: Taking Control of the Enterprise. In: Conference on Communications Architectures, Protocols and Applications (SIGCOMM) (2007)
8. Chen, C., Jia, L., Zhou, W., Loo, B.T.: Proof-based verification of software defined networks. In: Open Networking Summit (2014)
9. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 4130, pp. 632–646 (2006)
10. Foster, N., Kozen, D., Milano, M., Silva, A., Thompson, L.: A coalgebraic decision procedure for NetKAT. In: Principles of Programming Languages (POPL) (2015)
11. Gutz, S., Story, A., Schlesinger, C., Foster, N.: Splendid isolation: A slice abstraction for software-defined networks. In: Workshop on Hot Topics in Software Defined Networking (2012)
12. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Towards modularly comparing programs using automated theorem provers. In: International Conference on Automated Deduction (2013)
13. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: Programming Language Design and Implementation (PLDI) (1990)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, second edn. (2012)
15. Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., Vahdat, A.: B4: Experience with a globally-deployed software defined WAN. In: Conference on Communications Architectures, Protocols and Applications (SIGCOMM) (2013)
16. Katta, N.P., Rexford, J., Walker, D.: Logic programming for software-defined networks. In: Workshop on Cross-Model Design and Validation (XLDI) (2012)
17. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: Networked Systems Design and Implementation (2013)
18. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: Networked Systems Design and Implementation (2012)

19. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: VeriFlow: Verifying network-wide invariants in real time. In: Networked Systems Design and Implementation (2013)
20. Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Gude, N., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., Zhang, R.: Network Virtualization in Multi-tenant Datacenters. In: Networked Systems Design and Implementation (2014)
21. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: International Conference on Computer Aided Verification (2012)
22. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Foundations of Software Engineering (2013)
23. Liu, A.X.: Change-impact analysis of firewall policies. In: European Symposium on Research in Computer Security (2007)
24. Lopes, N., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: DNA pairing: Using differential network analysis to find reachability bugs. Tech. Rep. MSR-TR-2014-58, Microsoft Research (April 2014)
25. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P.B., King, S.T.: Debugging the data plane with Anteater. In: Conference on Communications Architectures, Protocols and Applications (SIGCOMM) (2011)
26. Monsanto, C., Foster, N., Harrison, R., Walker, D.: A compiler and run-time system for network programming languages. In: Principles of Programming Languages (POPL) (2012)
27. Nelson, T., Ferguson, A.D., Scheer, M.J.G., Krishnamurthi, S.: Tierless programming and reasoning for software-defined networks. In: Networked Systems Design and Implementation (2014)
28. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: International Conference on Software Engineering (2013)
29. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: USENIX Large Installation System Administration Conference (2010)
30. Nelson, T., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Toward a more complete Alloy. In: International Conference on Abstract State Machines, Alloy, B, and Z (2012)
31. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: Foundations of Software Engineering (2008)
32. Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G.: A security enforcement kernel for OpenFlow networks. In: Workshop on Hot Topics in Software Defined Networking (2012)
33. Ramsey, F.P.: On a problem in formal logic. Proceedings of the London Mathematical Society 30, 264–286 (1930)
34. Skowyra, R., Lapets, A., Bestavros, A., Kfoury, A.: A verification platform for SDN-enabled applications. In: International Conference on Cloud Engineering (2014)
35. Stewart, G.: Computational verification of network programs in Coq. In: Certified Programs and Proofs (2013)

36. Tariq, M.M.B., Bhandankar, K., Valancius, V., Zeitoun, A., Feamster, N., Ammar, M.H.: Answering "what-if" deployment and configuration questions with WISE: Techniques and deployment experience. IEEE/ACM Transactions on Networking (Feb 2013)
37. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 632–647 (2007)
38. Xie, G.G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G., Rexford, J.: On static reachability analysis of IP networks. In: IEEE Conference on Computer Communications (2005)
39. Zave, P., Rexford, J.: The design space of network mobility. In: Bonaventure, O., Haddadi, H. (eds.) Recent Advances in Networking. ACM SIGCOMM (2013)