An Assertion Language for Debugging SDN Applications

Ryan Beckett, X. Kelvin Zou, Shuyuan Zhang,
Sharad Malik, Jennifer Rexford, and David Walker
Princeton University
{rbeckett, xuanz, shuyuanz, sharad, jrx, dpw}@Princeton.edu

ABSTRACT

Software Defined Networking (SDN) provides opportunities for network verification and debugging by offering centralized visibility of the data plane. This has enabled both offline and online data-plane verification. However, little work has gone into the verification of time-varying properties (e.g., dynamic access control), where verification conditions change dynamically in response to application logic, network events, and external stimulus (e.g., operator requests).

This paper introduces an assertion language to support verifying and debugging SDN applications with dynamically changing verification conditions. The language allows programmers to annotate controller applications with C-style assertions about the data plane. Assertions consist of regular expressions on paths to describe path properties for classes of packets, and universal and existential quantifiers that range over programmer-defined sets of hosts, switches, or other network entities. As controller programs dynamically add and remove elements from these sets, they generate new verification conditions that the existing data plane must satisfy. This work proposes an incremental data structure together with an underlying verification engine, to avoid naively re-verifying the entire data plane as these verification conditions change. To validate our ideas, we have implemented a debugging library on top of a modified version of VeriFlow, which is easily integrated into existing controller systems with minimal changes. Using this library, we have verified correctness properties for applications on several controller platforms.

Categories and Subject Descriptors

C.2 [**Computer Communication Networks**]: Network Verification; D.3 [**Programming Languages**]: Verification Language Design

Keywords

Software Defined Network; incremental verification; stateful firewall

1. INTRODUCTION

SDNs are a promising approach for managing network complexity. However, subtle bugs introduced from complicated software remain problematic. Network operators need sophisticated tools and techniques to proactively catch these bugs before they can occur in production networks. Previous work in this area has focused on verifying static network invariants over dynamically changing networks [4, 6]. These prior works have found effective ways to ensure that invariants such as "host A can communicate with host B" are enforced even as the underlying network data plane changes. In contrast, we have designed efficient techniques for incrementally verifying *dynamic* network properties that describe the desired behavior of the evolution of the network rather than the behavior of the network at any given point in time. For example, an operator may wish to ensure that the network's stateful firewall functions correctly by verifying a property of the form: "host A can only communicate with host B after B first sends a message to A".

Because existing verification tools are isolated from the controller, operating exclusively on the data plane, they are only able to check that a property always or never holds for the network. For this reason, their enforcement mechanism can either be too strong (resulting in false positives) or too weak (missing violations of the property). For example, the programmer may want to suspend verification temporarily to push a set of rules into the data plane to form a path. However, existing tools will report unwanted property violations while invariants are temporarily violated.

To enable the verification of more expressive network properties and to avoid spurious warnings, we present an assertion language that allows programmers to formulate application-specific properties in terms of C-style assert statements. Control over the placement of assertions in the application enables the programmer to describe time-varying properties by relating properties to dynamic controller state as well as to refine the granularity at which properties are checked to avoid erroneously detecting transient property violations. As controller state changes, we can efficiently run incremental re-verification of relevant network properties. The main contributions of this paper are the following:

- We provide an assertion-based debugging/verification language to enable application developers to verify *dynamic* properties of controller applications via high-level program statements.

- We propose a verification procedure that combines the VeriFlow verification algorithm with an incremental

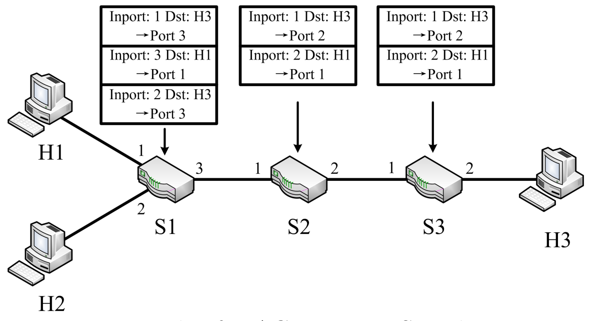
Inport: 1 Dst: H3 →Port 3
Inport: 3 Dst: H1 →Port 1
Inport: 2 Dst: H3 →Port 3
Inport: 1 Dst: H3 →Port 2
Inport: 2 Dst: H1 →Port 1
Inport: 1 Dst: H3 →Port 2
Inport: 2 Dst: H1 →Port 1
H1
H2
H3
S1
S2
S3
1
2
3
1
2
1
2

Figure 1: Example of MAC Learning Switch

```
def packet_in(self, event)
  pkt, sw, inport = parse(event)
  self.macTable[sw][pkt.src] = inport
  if pkt.dst not in self.macTable[sw]:
    flood()
  else:
    outport = self.macTable[sw][pkt.dst]
    install_rule(sw,
      inport=inport,
      dst=pkt.dst,
      outport=outport)
    packet_out(pkt, outport)
```

Figure 2: Buggy MAC learning application

data structure, to efficiently verify properties with dynamically changing verification conditions.

- We evaluate our tool on several controller applications from various controller platforms, and report its performance.

This paper is organized as follows: Section 2 introduces two examples to motivate the usefulness of debugging with assertions. Section 3 explains the language used to formulate assertions and specify dynamic network properties. The implementation details of our approach are provided in Section 4 and a performance evaluation is given in Section 5. An overview of related work is in Section 6, and we provide concluding remarks in Section 7.

2. DEBUGGING WITH ASSERTIONS

In this section, we will present two examples that demonstrate the usefulness of supporting controller-based network assertions and dynamically changing verification conditions.

2.1 MAC Learning Switch

Assume we have a simple network topology shown in Figure 1, and a MAC learning application written as in Figure 2. The MAC learning application maintains a table mapping destinations to output ports for each switch and reactively installs rules matching on packet destination and input port to forward packets according to this table. To see why this application is buggy, consider what happens when both H1 and H2 try to communicate with H3. Assume H1 and H3 first exchange a series of packets. The controller will learn the appropriate output port for both H1 and H3 at each switch upon seeing the first packet sent from each host. After seeing additional packets, the controller will install rules to send future packets out the appropriate output port.

Now, imagine that H2 tries to send a packet to H3. It will first send a packet to S1, and since S1 has no rule matching a packet from port 2, it will send the packet to the controller. The controller will learn about H2 for S1 before installing a rule to forward future packets from port 2, destined to H3, out port 3. The forwarding tables for each switch at this point are shown in Figure 1.

Bug. However, now when the packet sent from H2 to H3 reaches S2 and subsequently S3, it will not be sent to the controller since the controller has already installed matching rules on S2 and S3 for destination H3 and input port 1. Although the packet will arrive at H3 without going to the controller, S2 and S3 will never learn about H2. Whenever H3 now sends a packet to H2, S2 and S3 will forward the packet to the controller and the controller will flood the packet. While these packets will still reach their destination, the controller is consulted each time, leading to an elusive performance bottleneck. This kind of low-level reasoning about individual packets and switch state is difficult for most programmers.

Property. To see how assertions can catch subtle bugs of this kind, we now formulate a useful property to enforce for the MAC learning application. The main idea is to relate the state of the MAC table stored at the controller to paths in the network. A first guess might be: if the controller has learned about a host for a given switch, then that host should be able to reach that switch since it had to send a packet through the switch for the controller to learn about it. However, the controller may initially see a packet due to flooding rather than the presence of a path in the data plane. Although we cannot say much about the data plane when the controller has learned about a particular host, we can say something when it has *not* learned about a particular host. We can now try a variation of our first guess: if the controller has *not* learned about a host for a given switch, then that host should *not* be able to reach that switch.

In fact, using this assertion will uncover even the performance bug, raising an exception with the class of packets that violate the assertion. To see why this works, consider what happens when a packet is sent from H3 to H2. Since H2 is not in the MAC table for S3, any packet sent from H2 should never have gone through S3, and hence H2 should not be able to reach S3. However, H2 can reach S3 because the rule at S2 matches only on input port and destination.

Figure 3 shows how we can add this assertion to the program, where the host referred to in our property corresponds to the destination of the current packet at the controller, which we are looking up in the MAC table. The filter function here specifies that the verifier should only consider packets with data link source equal to the host under consideration. The variable `re` describes a path that starts at the host, takes some number of hops, and ends at the current switch. Finally, the `assert_now` statement instructs the verifier to check immediately that any packet passing the filter does not traverse a path described by `re`.

Fix. This bug occurs because the application installs rules that do not distinguish between packet source addresses, only input port and destination. By installing rules that match on source, input port, and destination rather than just input port and destination, this problem will be fixed. Note that existing data-plane verification tools like NetPlumber and VeriFlow [4, 6], are unable to detect this MAC learning misbehavior because it involves verifying properties

```
def packet_in(self, event)
  pkt, sw, inport = parse(event)
  self.macTable[sw][pkt.src] = inport
  if pkt.dst not in self.macTable[sw]:

    f = filter(dlSrc=pkt.dst)
    re = pkt.dst ^ star(DOT) ^ sw
    assert_now(Not(traverses(f, re))

    flood()
  else:
    outport = self.macTable[sw][pkt.dst]
    install_rule(sw,
      inport=inport,
      dst=pkt.dst,
      outport=outport)
    packet_out(pkt, outport)
```

Figure 3: Buggy MAC learning application with assertion

only after the destination of a packet is known to not be in the controller application's MAC table.

2.2 Stateful Firewall

We now turn our attention to a stateful firewall example. We will reuse the same topology shown in Figure 1 from the MAC learning example. Assume we have a set of hosts designated as clients consisting of H1 and H2 and a set of hosts designated as servers consisting of just H3. We wish to enforce the property that servers can only communicate with a client after the controller has first seen the client send a packet to the server. To implement this behavior, our stateful firewall application installs rules at S2 to drop and forward packets accordingly.

Like the MAC learning example, the correctness of this stateful firewall cannot be expressed via current verification tools because it requires knowledge of controller state, namely the set of hosts who have communicated. As this set changes, we would like to continue to verify that our property holds for the network. To accomplish this, we can assert the following formula using the `assert_continuously` command

```
...
f = Forall([c], clients,
      Forall([s], servers,
        iff((c,s) in sent, reachable(s,c))))

assert_continuously(f)
```

where `sent` is a set maintained in the controller application that tracks pairs of hosts that have communicated, and `reachable` is a library convenience function. The property states that a server should only be able to communicate with a client when the client has first communicated with the server. The `assert_continuously` statement at the start of the application also checks that this property holds after each new rule installation.

If any of the `clients`, `servers`, or `sent` sets are modified, our tool will generate new verification conditions that are verified incrementally. For example, when the controller sees some host initially send a packet to another host, we can add the following statement to the application:

```
insert(sent, (pkt.src,pkt.dst))
```

A	::=	`assert_now`(F)	`Assertion`
	$\mid$	`assert_continuously`(F)	
	$\mid$	`stop`(F)	
F	::=	$p \sim re$	`Formula`
	$\mid$	$\neg F$	
	$\mid$	$F \vee F$	
	$\mid$	$F \wedge F$	
	$\mid$	$\forall e \in S, F$	
	$\mid$	$\exists e \in S, F$	
	$\mid$	$e \in S$	
	$\mid$	$e = e$	
M	::=	`insert`$(S,e) \mid$ `remove`(S,e)	`Modification`
p	::=	`true`	`Predicate`
	$\mid$	$f = v$	
	$\mid$	`!p`	
	$\mid$	`p & p`	
	$\mid$	`p` $\vee$ `p`	
e	::=	$(x_1, \ldots, x_n)$	`Set element`
re	::=	$x \mid . \mid re\ re \mid re + re \mid re^*$	`Regular path`

Figure 4: Assertion language syntax

The insert statement adds the source-destination tuple to the `sent` set and incrementally re-evaluates all assertions dependent on the set.

3. PROPERTY LANGUAGE

Our assertion language supports debugging SDN applications by allowing programmers to annotate the applications with C-style assertions found in traditional programming languages. Using assertions, the programmer can express dynamic properties that describe the desired behavior of the evolution of the network.

Assertions. Several assertion primitives are available to the programmer in our language, as summarized in Figure 4. The `assert_now` statement acts as a traditional assert statement and checks the assertion at the exact point in time at which it is invoked. In the MAC learning example, this allowed the programmer to check an assertion only after ensuring that a given destination address was not in the MAC table for a switch. The `assert_continuously` and `stop` commands work together to control the scope of the verification process by checking assertions declared with the `assert_continuously` command after each new rule is emitted from the controller to the data plane until the `stop` command is invoked. They enable the description of the same properties as `assert_now`, but provide guidance to the verifier, as to the appropriate timing of verification. In the stateful firewall example we used `assert_continuously` to verify that a property always holds. However, if it required multiple rule installations to set up a path from a server to a client, we could temporarily disable verification with the `stop` command while a series of rules are installed before resuming verification.

Formulas. Formulas describe both invariants that hold for a network snapshot as well as how these invariants change over time. Here, the notion of time is twofold: a path invariant describes the permitted progression of a packet in a static network over time in terms of the packet's location, and a dynamic property describes the permitted progression of the network over time in terms of invariants.

We adopt a similar notation for describing path invariants as in previous work [4, 9, 10], which is based on regular expressions. A predicate selects the class of packets under consideration, and a corresponding regular expression describes the possible permitted paths that packets matching the predicate may traverse. For regular expressions here, a character x represents a single hop, or network location. The + operator means one path or the other, the * operator means a path is repeated zero or more times, consecutive regular expressions refers to path concatenation, and the dot symbol is a wild card that refers to any single hop. For example, the path invariant:

$$vlan = 1 \sim\ h_1\ .^*\ mb\ .^*\ h_2$$

states that packets with vlan=1, starting at h_1 must traverse a path that goes through middlebox mb and ends at h_2.

Quantifiers enable the programmer to make statements about groups of objects and let the verification engine continue to verify these statements efficiently as the groups change dynamically. For example, the following formula:

$$\forall g \in guests, \exists s \in servers,\ vlan = 1 \sim\ g\ .^*\ mb\ .^*\ s$$

builds on the previous formula to state that all guests in some set of guests can reach at least one server through a middlebox for any packet with vlan=1.

Set Modifications. In addition to providing quantified formulas ranging over sets of network entities, our language gives the programmer the ability to dynamically modify these sets, generating new verification conditions to be checked by the underlying VeriFlow engine as a result. Specifically, the programmer can insert a single element into a set or remove a single element from a set. For example, we can add a guest g to the set of guests with the statement:

$$\text{insert}(guests, g)$$

After each modification, our tool will incrementally re-verify every formula dependent on the modified set.

4. IMPLEMENTATION

4.1 Incremental Data Structure

As controller programs dynamically add and remove elements from sets, they generate new verification conditions that the existing data plane state must satisfy. To avoid naively re-checking the entire data plane after each such modification, we introduce an incremental data structure to process each modification, tracking changes to the underlying verification conditions for each property, and evaluating each property by reusing as much previous work as possible. We illustrate our data structure through an example.

Example. Consider a version of the earlier stateful firewall example where we have three clients and three servers. The data structure representing the formula for this example is shown in Figure 5. The formula is a type of expression tree with a value stored at each node to represent the validity of the corresponding sub formula. A circular node represents a "forall" quantifier in the formula whose children correspond to set elements, and whose result is the conjunction of the results of each of its children. A leaf node represents a concrete property that the data-plane verifier can check. Imagine initially that every sub formula (i.e, every node) in the tree has been evaluated to *true*

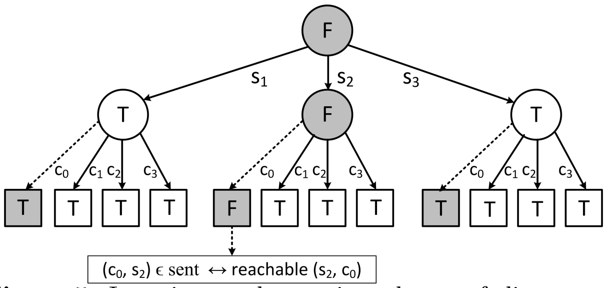

Figure 5: Inserting an element into the set of clients

Insertion. Now imagine that we are adding a new client c_0 to the `clients` set. After c_0 is added to the set of clients, each relevant node quantifying over the set of clients builds a new branch representing the sub formula for the added element. The new branch is evaluated with the data-plane verifier, and afterwards, the result is propagated up to the parent of the node. While the propagated result leads to a change in the result stored at the parent, the change will continue to propagate up the tree. For example, if the new sub formula:

$$(c_0, s_2) \in sent \iff reachable(s_2, c_0)$$

evaluates to $false$, then the parent "forall" node becomes $false$, and and hence the formula becomes $false$. The dark nodes in Figure 5 indicate where this propagation occurs.

Removal. Removing an element from a set proceeds in a similar fashion. The branch corresponding to the removed element is deleted and a simple check can determine whether or not the result of the parent node should change. If so, the change is propagated up to the parent node as before.

4.2 Combined Incremental Checking

Incremental verification occurs in multiple dimensions for our debugging tool. The tool incrementally verifies the same formula given a change in the underlying network data plane (i.e., a rule installation), and the tool incrementally verifies a new formula resulting from modifications to a programmer-defined set. To verify an assertion made via either the `assert_now` or or `assert_continuously` commands, we must check that the corresponding formula holds initially for the network data plane and this is accomplished by verifying the network formula against all existing packet classes in the network. As new rules are installed from the controller, the assertion is incrementally verified using the VeriFlow algorithm. On the other hand, when a relevant programmer-defined set is modified, we must now check that the new formula holds for the network configuration given that the old formula did. This is accomplished by checking the newly generated nodes in the tree against all packet classes and by propagating their results through the data structure.

Our modified VeriFlow engine and debugging library are responsible for coordinating these two types of change. In the case that *both a change in the formula and in the network configuration* occur, we can safely compose these two verification procedures together by first verifying the new formula against the old network by incrementally checking the new formula given the old formula, and then by verifying the new formula against the new network configuration by checking those packet classes that have been modified.

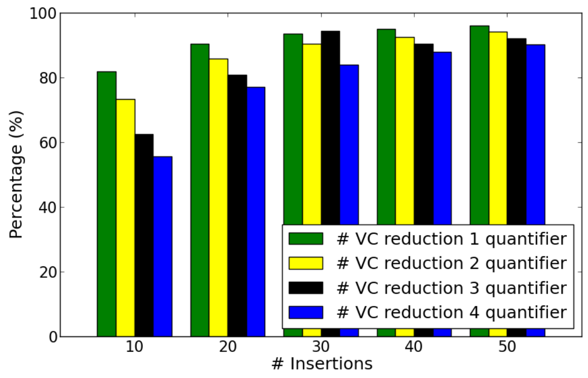

Figure 6: VC Reduction

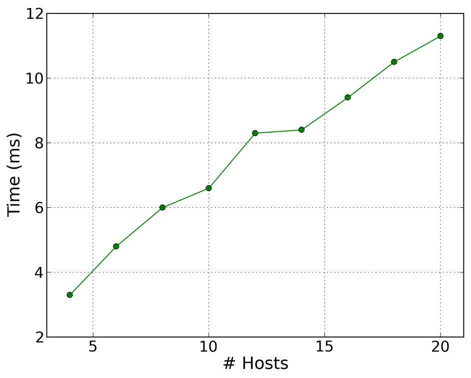

Figure 7: MAC Learning Average Assertion Overhead

In the case where an assertion is not currently active (i.e., `assert_continuously` after the `stop` command is invoked), the assertion is not re-checked after each rule installation and set modification, but instead, the relevant changes are buffered for that assertion. Later, if the formula is re-asserted, the debugging tool can reuse existing work by only verifying the relevant, new changes.

4.3 Precomputing Properties

Although regular expressions are expressive enough to describe many useful path invariants, their full power is often not needed. For a restricted subset of regular expressions, we can effectively precompute a formula so that evaluating new verification conditions as they arise is a constant-time operation. This restricted subset of regular expressions allows the use of the * operator only for the special case of $.^*$, and disallows the use of the + operator. For example, the regular expression $h_1\ .^*\ (mb_1 + mb_2)\ .^*\ h_2$ is disallowed whereas $h_1\ .^*\ mb_1\ .^*\ h_2$ is allowed.

To see why this restriction is useful, consider the path expression:

$$h_1\ .\ .\ mb_1\ mb_2\ .^*\ h_2$$

Checking that a packet traverses a path corresponding to this regular expression given a network forwarding graph can be broken down into a series of simple reachability queries. For the above example, we can check if h_1 can reach mb_1 in three hops, if mb_1 can reach mb_2 in a single hop, and if mb_2 can reach h_2 in any number of hops. We can either precompute reachability between all source-destination pairs, or compute reachability between a single source and all destinations on-the-fly.

For instance, we can precompute the above regular path expression on-the-fly by first checking if h_1 can reach mb_1 in three hops. Because we do not consider loops, we can efficiently find all other locations h_1 can reach in three hops or fewer using a simple breadth-first search. This process is repeated for each of the other reachability checks, and the results are stored in a cache. The cache is a function:

$$\textit{cache} : (\text{packet class, src, dst, hops}) \rightarrow \text{bool}$$

where hops is either an exact number or a lower bound. The precomputation here has the same time complexity for evaluating path expressions as using no precomputation.

Alternatively, we could precompute the path expression for all source and destination pairs. In this case, there is additional overhead to verify each property after a rule installation since we must now check the path expression for all sources, however any newly generated verification conditions are guaranteed to be evaluated in constant time.

5. PERFORMANCE

To evaluate the performance of our tool, we conducted a series of simple tests in Mininet [2], using the Pyretic [8] and Pox [1] platforms to measure the reduction in the number of verification conditions that our modified VeriFlow engine must verify over a naive re-evaluation strategy, as well as the overall run time of the tool. All tests were performed on an 8 core, 2.4 GHz machine with 8GB of RAM.

In general, the performance of the debugging tool is dominated by the number of newly generated verification conditions, since each must initially be checked against all existing packet classes in the network. As a result, verification time varies dramatically depending on the number of assertions, structure of the properties, and sizes of the user-defined sets.

Because our tool must enumerate objects in user-defined sets, the number of verification conditions for each relevant assertion after a set insertion is proportional to the product of the set sizes for nested quantifiers (set removal generates no new verification conditions). However, our incremental data structure is able to significantly reduce the number of newly generated verification conditions through the efficient reuse of previous computation. To demonstrate the effectiveness of our incremental data structure, we perform a number of random insertions into sets for various formulas of a given maximum quantifier nesting depth. The results are averaged over 10 iterations, and we record the percent reduction in the number of verification conditions that our VeriFlow engine must re-verify compared to a naive re-evaluation strategy. Figure 6 shows the results. The percent reduction in the number of verification conditions increases as the size of the user-defined sets increases in all cases.

For the MAC learning application, we varied the number of hosts in the network and measured the average verification time per assertion after running a ping between each pair of hosts. The results are shown in figure 7. The MAC learning application is expensive since it can generate a large number of new assertions, each of which must be checked against the entire data plane. As a result, the average verification time per assertion for the MAC learning application grows with the number of packet classes in the network, which increases roughly linearly with the number of hosts in this case.

By comparison, the stateful firewall application, which is conceptually more complex, generates relatively few new verification conditions due to the incremental checking of the single continuous assertion. As a result, verification time never exceeded 1ms for either set modifications or rule installations up to 20 hosts.

Depending on the particular use of assertions, they may or may not be feasible for real-time, online verification. However they are still useful for finding bugs in application logic in a testing environment where performance is not critical. For example, we found a flaw in the MAC learning application using a simple 3 switch, 3 host setup. Additionally, because checking a property against all packet classes in the network can result in a large performance overhead, and because each packet class can be checked independently for a given property, it would be possible to speed up verification by evaluating a property against packet classes in parallel.

6. RELATED WORK

Recently, we have seen many works on network verification and debugging. ANTEATER [7] translates high-level network invariants into instances of the boolean satisfiability problem (SAT), checks them against the network using a SAT solver, and reports counter-examples if violations are found. Header Space Analysis [5] builds a finite state machine out of the network state and topology and uses ternary symbolic simulation to traverse the state machine to verify properties. Both are offline static verifiers.

Our work is based on the VeriFlow framework [6], which builds a set of equivalence classes to represent the packets that observe the same network forwarding behavior. Whenever a networking event occurs (e.g. a rule installation), VeriFlow only needs to traverse the affected equivalence classes' forwarding graphs to detect property violations. Similar to VeriFlow, real time Header Space Analysis [4] leverages the fact that a network event only changes a small portion of the rule space, builds a dependency graphs between rules, and traverses the dependency graph to check properties.

OFRewind [11] helps debug network applications by replaying the events that lead to a particular situation. NetSight [3] is an extensible platform that records packet histories, enabling applications to retrieve packet histories of interest. Using NetSight, the authors implemented a network debugger based on packet traces. However this project focuses on the debugging at the packet level, whereas we focus on debugging via high-level assertion statements.

7. CONCLUSION

In conventional languages like C, the use of assertions is helpful for catching bugs before deployment. We believe the same is true in network programming environments. In this work, we developed an assertion language that supports verifying and debugging SDN applications with dynamically changing verification conditions. To efficiently check these changing verification conditions, we proposed a verification procedure that combines the VeriFlow verification algorithm with an incremental data structure. We have implemented our assertion language as a debugging library that is easily integrated into existing controllers with minimal changes. We evaluated our tool on several small examples, demonstrating its feasibility for network debugging.

8. ACKNOWLEDGEMENT

This work was supported in part by Princeton Fellowship, NSF grant TC-1111520, C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

9. REFERENCES

[1] Pox. `http://www.noxrepo.org/pox/about-pox/`.

[2] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (2012), CoNEXT '12, ACM, pp. 253–264.

[3] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 71–85.

[4] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 99–112.

[5] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 113–126.

[6] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 15–28.

[7] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with anteater. 290–301.

[8] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), nsdi'13, pp. 1–14.

[9] Reitblatt, M., Canini, M., Guha, A., and Foster, N. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013), pp. 109–114.

[10] Soulé, R., Basu, S., Kleinberg, R., Sirer, E. G., and Foster, N. Managing the network with merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013), pp. 24:1–24:7.

[11] Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on Annual Technical Conference* (2011), pp. 29–29.