StateKeeper

Generalizing Reachability

Vishrant Vasavada
UCLA
vasavada@cs.ucla.edu

Ricky Lee
UCLA
ricky.lee@ucla.edu

ABSTRACT

Today there are many tools that are built using formal methods to verify networks. The majority of these tools check network specifications and configurations to identify only a limited class of failures such as *Reachability*, *Forwarding Loops* and *Slicing*. They don't take into account the rate-limiting rules that affect the throughput of the system and also link delays. Both of these quantities are responsible for the Quality of Service (QoS). We believe that without a decent throughput and affordable delay, a network is pretty much in a dead state even if it passes the tests for reachability, forwarding detection, and other such failure classes. We developed StateKeeper, a tool based on ideas from Atomic Predicates Verifier. StateKeeper keeps track of both network performance and network reachability. In our experiments, we factor in link delay and throughput as our performance metrics. StateKeeper will allow network operators to verify Quality of Service at each port along a route by showing information about the state of each step.

1 INTRODUCTION

Network verification is a daunting and error-prone task. Operators making mistakes on configuration is the most common reason for network failure. Researchers have proposed mainly two different approaches towards detecting such configuration errors: *static analysis*[1] and *analyzing dataplane snapshots*[2][3]. Unfortunately, all of these approaches only target limited failure classes such as *reachability*, *forwarding loops*, *multipath consistency*, etc. Most of these deal with forwarding and routing behavior in network checking for route validity and path visibility faults. They do not account for rate limiting rules that could heavily impact the throughput and performance of the system.

Customers want more than just reachability to be satisfied. Ultimately, it is all about Quality of Service levels such as guaranteed bandwidth and bounded delays. One might think that analysis of quantitative properties of the network is a run-time analysis. However, static analysis of configurations might also eliminate quantitative bugs. Rate limiting traffic is a very common practice to avoid packet losses and stressing router queues. For example, Google Researchers developed Trickle to rate limit YouTube Video Streaming [4]. This was a sophisticated approach to rate limit the traffic. However, many services simply place ACL rules to control bandwidth. If traffic increases by a certain limit, the ACL rule is configured to simply drop packet[5]. Suppose there is a video call which is limited to 1 Mbps because some router is configured with an ACL to limit the QoS of the calls. If this is an unintended behavior, such a bug could be verified by reading all configurations of routers and keeping track of system throughput as the packet moves from one router to the next. If the throughput obtained at the destination is lesser than expected, the operator would know that there is a rate-limiting bug caused at some router in the path to the destination.

Moreover, there are also known theoretical link delays. It might not be the best idea to send, for example, video streaming traffic via links that cause much more delay than the other links. Hence, while configuring routes, network operator should also take into account such link delays.

We developed a tool that generalizes reachability to account for quantitative analysis. Our tool gives the network operator the overall picture of reachability from point A in the network to point B along with total delay and throughput that each packet would incur. We consider all paths from point A to point B and this makes it easy for operators to choose the best path appropriate to their requirements to steer the network traffic.

We considered two approaches to solve this problem. The first idea was the modify the packet state to $<$*header, quantity*$>$ in Network-Optimized Datalog (NoD)[7] where quantity is link delay or throughput of the system. However, NoD uses Difference of Cubes (DoC) data structure internally to represent packet states. To support the addition of quantity into packet state, we would then have to modify this data structure. Hence, NoD cannot be directly used with trivial modifications to meet our requirements.

Our next approach was based on the idea described in *Atomic Predicate Verifier*[6] paper. We decided to build our design over this because of its simplicity. This approach also allows us to modularize quantitative analysis so that each different type of quantity (e.g. latency, bandwidth, hop counts, etc) doesn't require different verification methods. Also, using this approach, we believe that quantitative network analysis can be done at the same speed and scale as Atomic Predicate Verifier if the same atomic predicate calculation algorithms are used.

However, our algorithms are quite different from the ones used in Atomic Predicate Verifier. For example, we do not make use of Binary Decision Diagram (BDD) data structures in our algorithms, but rather chose to go with Header Space Algebra to keep the work simple. We also change the representation of the reachability tree described in [6] to keep track of total delay and throughput at each port (or node) in a tree.

To summarize, this report makes the following two major contributions:

- *Using Header Space Algebra to calculate atomic predicate set*: We design a algorithm to calculate atomic predicate sets using Header Space Algebra instead of using BDDs. We also evaluate the scalability of our algorithm.
- *Modular Quantitative Analysis Approach*: We devise the algorithm that generalizes reachability tree model from the Atomic Predicate Verifier paper to account for any quantity type. However, we only use throughput and link

delays as an example. We again evaluate the scalability of our algorithm.

The outline of this report is as follows: In section 2, we discuss the Atomic Predicates Verifier paper. In section 3, we describe the implementation of our system. We talk about how we use these atomic predicates to compute reachability and quality of service in the network. Next in section 4, we go over our evaluation of the implementation by showing run-time statistics in a variety of synthetic networks. The limitations of our tool is explained in section 5 and we conclude the report in section 6.

2 ATOMIC PREDICATES VERIFIER

Our work is based on ideas from Atomic Predicates(AP) Verifier:

In a network, the data plane is determined by the forwarding rules and ACLs. These act as the filters for the flowing packets. These filters can be parsed and viewed as predicates that guard the input and output of ports in a network box. Each such predicate would allow a packet to pass through its guard based on whether it evaluates to true or false depending on the header fields of a packet. The set of packets that can travel from port s to port d through a sequence of packet filters can be obtained by computing the conjunction of predicates in the sequence or by the intersection of the corresponding packet sets. The intersection and union of packets is computationally expensive because it is operated on multi-dimensional sets. The paper presents a very fast algorithm to compute a set of atomic predicates given a set of predicates, which is minimal and unique. This is then used to find reachability within the network.

3 IMPLEMENTATION

3.1 Atomic Predicates

As mentioned before, we use Header Space Algebra to calculate atomic predicates set for a given forwarding rule list. Our algorithm is very naive and not as sophisticated as the one used AP Verifier. We discuss its performance later in the section 4.

3.1.1 Header Space Algebra. The first step was to implement Header Space Algebra. For this, we had to code header space set operations such as complement, intersection, and difference. Each operation uses the algorithms described in [2]. We use the word *prefix* for wildcard expressions representing packet headers. We summarize the set operations below:

- **Intersection:** Intersection is governed by the following table which shows single bit intersection rules:

b	b'	Result
1/0	1/0	1/0
1/0	0/1	z
1/0	x	1/0
x	x	x

  In the table, x means the bit could either be 1 or 0 and z means the intersection is empty. The intersection of prefixes is then applied bit-by-bit as per this table. If the result contains a z, the intersection of all bits is empty.

- **Complement:** The complement of prefix p - the union of prefixes that no intersect with p - is computed as follows:

```
result = []
for bit b in prefix $p$ do
    if b is not x then
        result.append(x..xb'x..x)
    end if
end for
return result
```

  Note that b' above is simply the complement of b. That is, if b is 1, then b' is 0 and vice-versa. Also, the union of prefixes is simply represented as prefixes in a list. For example, $1xx \cup 0xx$ is represented simply as [1xx, 0xx]. The algorithm basically finds all non-intersecting prefixes by replacing 0 or 1 in prefix p with its complement. This easily follows because just one non-intersecting bit will form a disjoint prefix. For example, (10xx)' = [0xxx, x1xx].

- **Difference:** The difference operation can be calculated using intersection and complement. $A - B = A \cap B'$.

3.1.2 Determining Atomic Predicates. Once the Header Space Algebra implementation was done, the next step was to design a algorithm for calculating atomic predicate set. Since our atomic predicate set $\{P_1, P_2, .., P_k\}$ will basically consist of prefixes, we define its properties as follows:

(1) $P_i \neq z, \forall i \in \{1, 2, ..., k\}$
(2) $P_1 \cup P_2 \cup ... \cup P_k = X$
(3) $P_i \cap P_j = z, if i \neq j$
(4) Each prefix in our prefix list is equal to the union of a subset of atomic predicates
(5) k is *minimum* number such that the set $\{P_1, P_2, ..., P_k\}$ satisfies above four properties.

Property (1) basically says that we shouldn't have empty prefixes in our atomic predicate set. Property (2) states that together all the prefixes should represent universe (X = xxxx..). Property (3) basically states that any two prefixes from the atomic predicate set should be disjoint. Property (4) simply means that each prefix in our given input list of prefixes to calculate their atomic predicates is equal to union of subset of atomic predicates. Property (5) guarantees that the calculated predicate set is minimal and unique.

We now show how to calculate a list of atomic predicates (consisting of prefixes) from given list of prefixes as an input. For an example, let our input prefix list be **L** = [10x, 1x, 110x].

Step 1: Sort the prefixes in increasing order of their length.

Sorting **L**, we get [1x, 10x, 110x].

Step 2: For each prefix *P*, get {P, P'} pair where P' is complement of P.

$1x \longrightarrow \{1x, 0x\}$

$10x \longrightarrow \{10x, [0xx, x1x]\}$

$110x \longrightarrow \{110x, [0xxx, x0xx, xx1x]\}$

Step 3: Find atomic predicates set for each pair obtained above. From the above example, it is apparent that P' will often be a union of prefixes. We need a way to break down this union into a list of disjoint prefixes. This is where difference operation comes into the picture. In case P' is not a union of prefixes, we do not make any changes to {P, P'} pair in this step.

$\{1x, 0x\} \longrightarrow \{1x, 0x\}$

$\{10x, [0xx, x1x]\} \longrightarrow \{10x, [0xx, x1x{-}0xx]\} \longrightarrow \{10x, 0xx, 11x\}$

$\{110x, [0xx, x0xx, xx1x]\} \longrightarrow \{110x, [0xxx, x0xx - 0xx, xx1x - x0xx - 0xx]\} \longrightarrow \{110x, 0xxx, 10xx, 111x\}$

Step 4: For each prefix, we now have the atomic predicates set. We now intersect atomic predicates set for the first prefix with that of the second prefix, the result with atomic predicates set of the third prefix and so on to get set of atomic predicates for all prefixes.

Iteration 1:
$\{1x, 0x\} \cap \{10x, 0xx, 11x\} = \{10x, 11x, 0x\}$

Iteration 2:
$\{10x, 11x, 0x\} \cap \{110x, 0xxx, 10xx, 111x\} = \{\mathbf{10x, 110x, 111x, 0x}\}$

This resulting set of atomic predicates can then represent all forwarding rules in the network at any ports.

Consider the network shown in figure 1. R1 and R2 are the two routers. S represents source while D represents destination. C is some management controller that we don't care about. P1, P2, P3, P4 P5, and P6 are ports. We are interested in calculating reachability from S to D while also doing quantitative analysis. The values on the links represent <throughput, delay >for that particular link. Now that we have atomic predicates for these forwarding rule prefixes, we represent them using the set of integers. Let 10x, 110x, 111x and 0x atomic predicates be represented by integers 0, 1, 2, and 3 respectively. The network now looks as shown in figure 2. **S(F)** simply means set of integers of corresponding atomic predicates describing forwarding rule(s) at that particular port. For example, the forwarding rule at P3 is 1**. This means it consists of atomic predicates 110x, 111x and 10x. As discussed above, these are represented using 1, 2 and 0 respectively. Hence, S(F) for P3 will be [0, 1, 2].

3.2 Network Model

Theoretically, we now have a network topology with forwarding rules represented as the set of integers that correspond to their respective atomic predicates. The next step is to model this so as to carry out reachability algorithms.

We initially started by considering a graph with network boxes as nodes and links between boxes as edges. However, this design was too complicated. The links don't really exist between boxes

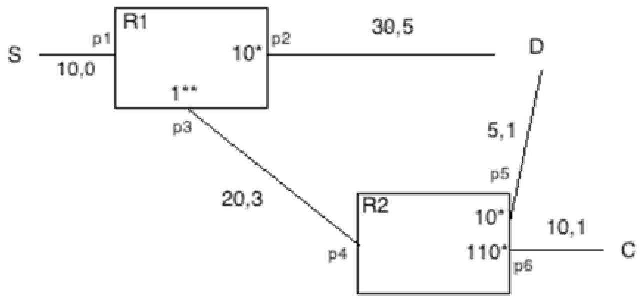


Figure 1: Network topology

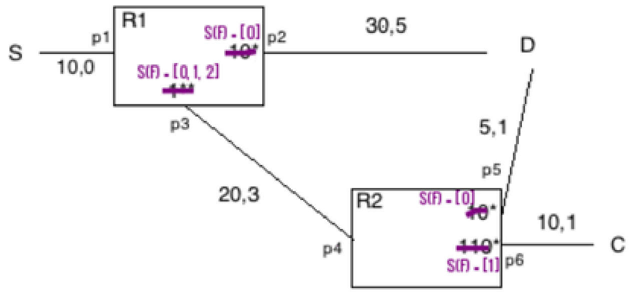


Figure 2: Network topology with forwarding rules represented as set of integers

but between ports. Also while computing reachability tree, we care about a hop from one port to another port, not from one box to another. Hence, we changed our design and represented ports in the networks as nodes and links between them as edges. Destinations were also represented as nodes and links between ports and destination were again represented as edges. The class stubs look as follows:

class Destination
- has id
- has input

class Port
- has id
- has list of forwarding rules represented as S(F)
- has list of next ports (i.e. ports it is directly connected to)
- has input

class Network
- has ports (nodes)
- has map to represent edges

To model the network, we perform following steps:

- Note that all nodes in the graph should have unique ID. In our example, we can assign following IDs to nodes: $P1 \longrightarrow 1$, $P2 \longrightarrow 2$, $P3 \longrightarrow 3$, $P4 \longrightarrow 4$, $P5 \longrightarrow 5$, $P6 \longrightarrow 6$, $D \longrightarrow 7$, and $C \longrightarrow 8$.

- For each port (i.e. node), we set the forwarding rules: $P3.fr \longrightarrow [0, 1, 2], P2.fr \longrightarrow [0], P5.fr \longrightarrow [0]$, and $P6.fr \longrightarrow [1]$. Note that P1 and P4 doesn't have forwarding rules.

- For each port (i.e. node), we set the list of next ports. Note that each next port entry will also carry information about throughput and link delay. If the current port and next port belongs to same box, delay is set to 0 and throughput remains same. We now have: $P1.next \longrightarrow \{P2 : \{delay : 0, throughput : 10\}, P3 : \{delay : 0, throughput : 10\}\}$, $P3.next \longrightarrow \{P4 : \{delay : 3, throughput : 20\}\}$, $P4.next \longrightarrow \{P6 : \{delay : 0, throughput : 20\}, P5 : \{delay : 0, throughput : 20\}\}$, $P2.next \longrightarrow \{DestinationD : \{delay : 5, throughput : 30\}\}$, $P5.next \longrightarrow \{DestinationD : \{delay : 1, throughput : 5\}\}$, and $P6.next \longrightarrow \{DestinationC : \{delay : 1, throughput : 10\}\}$

In code, it looks as follows:

```
D = Destination(7)
C = Destination(8)
P6 = Port(6, fr=[1], nxt=C:"delay": 1, "throughput": 10)
P5 = Port(5, fr=[0], nxt=D:"delay": 1, "throughput": 5)
P2 = Port(2, fr=[0], nxt=D:"delay": 5, "throughput": 30)
P4 = Port(4, nxt=P6:"delay": 0, "throughput": 20, P5:"delay": 0,
"throughput": 20)
P3 = Port(3, fr=[0, 1, 2], nxt=P4:"delay": 3, "throughput": 20)
P1 = Port(1, nxt=P2:"delay": 0, "throughput": 10, P3:"delay": 0,
"throughput": 10)
```

Input for port and destination is something we set while computing reachability. We will discuss that in next section. Using these as nodes, we then create a graph as follows:

```
network = Network([P1, P2, P3, P4, P5, P6, D, C])
network.add_edge(P1, P2)
network.add_edge(P1, P3)
network.add_edge(P3, P4)
network.add_edge(P4, p5)
network.add_edge(P4, P6)
network.add_edge(P2, D)
network.add_edge(P5, D)
network.add_edge(P6, C)
```

3.3 Computing Quality of Service

The two performance variables that StateKeeper accounts for are link delay and throughput. At each state, StateKeeper will tell the user the link delay and throughput at the current traversed route. We calculate link delay by summing the delays at each link traversed thus far. Throughput is found by selecting the minimum value of the links traversed thus far.

3.4 Computing Reachability Using Depth-First Search

We now have a network model - a graph of ports as nodes and links between ports as edges. Each port (i.e. node) also carries information such as ID, set of forwarding rules represented with integers corresponding to atomic predicates and set of ports it is directly connected to along with the link delay and throughput information as discussed in the last section.

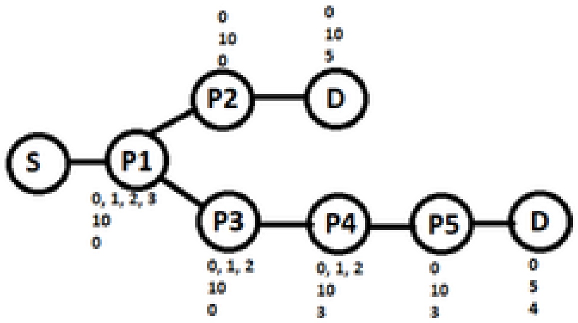


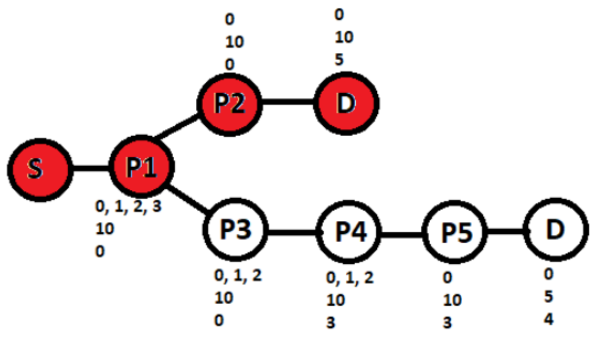
Figure 3: Reachability tree from source S to destination D

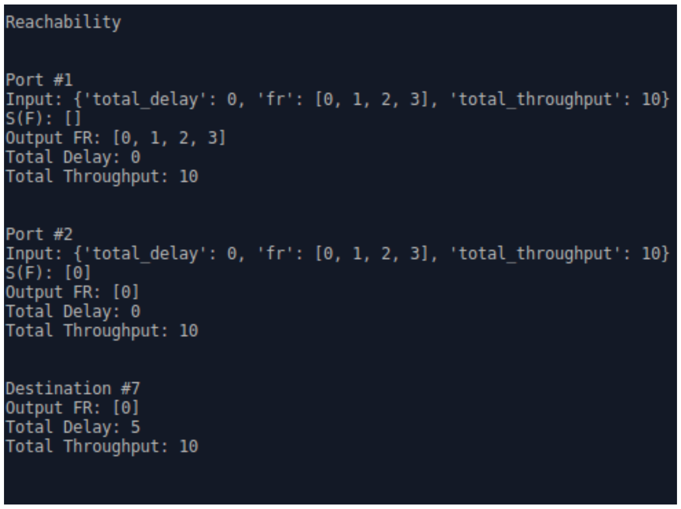
The tool uses depth-first search to find routes to a destination. A reachability tree is shown in figure 3. Using DFS, we get two routes to destination D from source S: S, P1, P2, D and S, P1, P3, P4, P5, D. Each node in a tree is node with set of integers representing atomic predicates for forwarding, throughput of the system observed so far and total delay incurred so far. For example, [0, 1, 2, 3] represents set of all packets injected into the network from source S. Note from figure 1 that the link delay from S and P1 is 0 while the throughput is 10. All these values are shown at node P1 in reachability tree. P1 then intersects the set with its forwarding rule, computes the overall delay to reach P2 and calculates the throughput of the system when it reaches P2 and sets input of P2 to these values. This basically means that all packets matching [0, 1, 2, 3] are forwarded to P2. The node P2 has S(F) = [0], throughput 10 and total delay so far 0. Since S(F) is non-empty, it means that P2 filters out all packets except the one matching atomic predicate represented by integer 0. Hence, it sets input of D to S(F) = 0, throughput 10 and total delay so far as 5.

Similarly, all the packets matching [0, 1, 2, 3] move from P1 to P3. P3 filters out all packets except the ones matching [0, 1, 2] and forwards them to P4. P4 doesn't have any filtering rules so forwards packets matching [0, 1, 2] to P5. P5 filters out all packets except the one matching atomic predicate represented by integer 0. On this path, the throughput of the system obtained is 5 and total delay incurred is 4.

From reachability tree, we can see that the only packets that could reach D via both paths are the ones matching atomic predicate represented by integer 0 which is 10x.

StateKeeper's output corresponding to highlighted branches is shown in figures 4 and 5.





```
Reachability


Port #1
Input: {'total_delay': 0, 'fr': [0, 1, 2, 3], 'total_throughput': 10}
S(F): []
Output FR: [0, 1, 2, 3]
Total Delay: 0
Total Throughput: 10


Port #2
Input: {'total_delay': 0, 'fr': [0, 1, 2, 3], 'total_throughput': 10}
S(F): [0]
Output FR: [0]
Total Delay: 0
Total Throughput: 10


Destination #7
Output FR: [0]
Total Delay: 5
Total Throughput: 10
```

Figure 4: Depth First Search output from source S to D - S, P1, P2, D

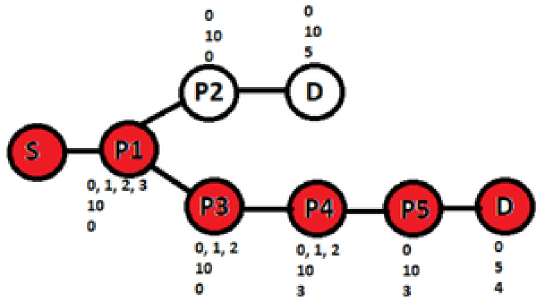
4 EVALUATION

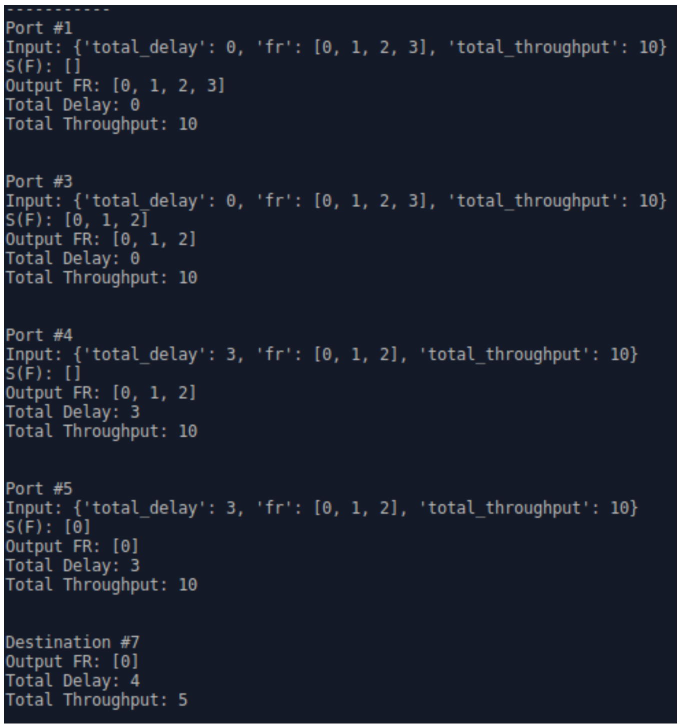
We used synthetic data for our evaluations. Forwarding rules, link delay, and throughput are chosen arbitrarily. We also do not have any notion of units for quantitative data.

4.1 Scaling to Forwarding Rules

We generate a list of N random prefixes of length 10 which represent forwarding rules in a network. For each such list, we calculate atomic predicates set and compute the runtime of our code. The algorithm to generate a list of N random prefixes is as follows:

```
result = []
while N >0 do
    digits = random_int(1..10)
    prefix = ""
    i = 0
    while i <digits do
        val = random_int(0, 1)
        prefix += val
        i = i + 1
    end while
    prefix = prefix + "x" * (10 - digits)
    result.append(prefix)
    N = N - 1
return result
```





```
-----------
Port #1
Input: {'total_delay': 0, 'fr': [0, 1, 2, 3], 'total_throughput': 10}
S(F): []
Output FR: [0, 1, 2, 3]
Total Delay: 0
Total Throughput: 10


Port #3
Input: {'total_delay': 0, 'fr': [0, 1, 2, 3], 'total_throughput': 10}
S(F): [0, 1, 2]
Output FR: [0, 1, 2]
Total Delay: 0
Total Throughput: 10


Port #4
Input: {'total_delay': 3, 'fr': [0, 1, 2], 'total_throughput': 10}
S(F): []
Output FR: [0, 1, 2]
Total Delay: 3
Total Throughput: 10


Port #5
Input: {'total_delay': 3, 'fr': [0, 1, 2], 'total_throughput': 10}
S(F): [0]
Output FR: [0]
Total Delay: 3
Total Throughput: 10


Destination #7
Output FR: [0]
Total Delay: 4
Total Throughput: 5
```

Figure 5: Depth First Search output from source S to D - S, P1, P3, P4, P5, D

The logic is simple. Since we want prefix of length 10, we choose a random integer k between 1 and 10, both inclusive. We then generate random string consisting with a total of k 0s and 1s followed by $(10 - k)$ x. Hence, if k is 2, the prefix generated would be one of 00xxxxxxxx, 01xxxxxxxx, 10xxxxxxx and 11xxxxxxxx. We repeat this N times to get a list of N prefixes.

Once we have a list of N prefixes, the next step in evaluation is to calculate atomic predicates set. We first compare the size of our input prefix list with the size of their atomic predicates set as shown in table 1. When the input prefix list size is small, it requires more number of atomic predicates to cover the entire universe. But for large prefix list sizes, the probability of a large number of prefixes being redundant is very high. Hence, the size of atomic predicates set we get is significantly lower than the size of the input list. Even in real networks, there are large amounts of redundancy in forwarding rules and so the number of atomic predicates obtained will be quite small. For example, there are only 494 atomic predicates for a total of 757,170 forwarding rules in Stanford Network[6].

Input Prefixes	Atomic Predicates
10	43
100	142
200	225
300	280
500	366
700	408
1000	450

Table 1: Input prefix list size vs AP set Size

We then evaluate the scalability of our algorithm by plotting the running time against the total number of input prefixes (forwarding rules in a real network). We can see from figure 6 that running time of our algorithm scales parabolically to the number of input prefixes. It takes about a minute to calculate atomic predicates set for a list of 1500 prefixes (of length 10), indicating that our naive algorithm is extremely slow when compared to the algorithm used by Atomic Predicate Verifier. This raises following two questions that could be covered in future work:

- Can we achieve efficiency by optimizing our algorithm and using Header Space Algebra?
- Is Header Space Algebra the root cause of this inefficiency when compared to algorithms using BDDs?

Including the set operations (complement and intersection), our current atomic predicate calculation algorithm has six nested for loops which depend on the size of input list and length of the prefix. Below we suggest some possible optimizations that could be included in future work:

(1) Encode each bit in prefix using two bits: $0 \longrightarrow 01$, $1 \longrightarrow 10$, $x \longrightarrow 11$ and $z \longrightarrow 00$. Intersection will then simply be an AND operation of encoded prefixes.
(2) Use trie structure to check for any intersections of the calculated predicate with existing predicates in the result list.
(3) Complement right now is represented by a list of non-disjoint prefixes. For example (10x)' = [0xx, x1x]. During calculation of atomic predicates for this prefix, we then have to take the difference between x1x and 0xx. Instead, these steps could be merged into a single loop. This would reduce three nested loops into one.

4.2 Scaling to Large Networks

We generate a linked list of N ports to represent a model of a linear network consisting of N/2 network boxes (each box has two ports - in and out). Since our DFS algorithm also involves set intersection of forwarding rule predicates, we keep these same at all ports in all test runs instead of randomizing so that we have fair comparisons. The algorithm to generate a model of N ports of a linear network is as follows:

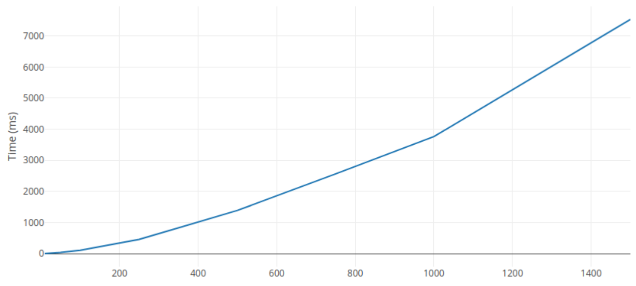


Figure 6: Atomic predicates set calculation algorithm scales parabolically to the number of forwarding rules (input prefixes)

```
forwarding_rules = list of forwarding rules
nodes = []
destination = Destination(N)
nxt = {destination: {delay: random_int, throughput: random_int}}
nodes.insert(0, destination)
for i in (N-1...0) do
    port = Port(id = i, forwarding_rules, next=nxt)
    nxt = {port: {delay: random_int, throughput: random_int}}
    nodes.insert(0, port) end for
network = Network(nodes)
for i in (1..N-1) do
    network.add_edge(nodes[i], nodes[i+1])
end for
    return network
```

The idea is to do a reverse build. We first a create destination node, then create (N-1)th node and set the destination as its next, then create (N-2)th node and set (N-1)th node as its next and so on. We will finally have a linked list that starts from the source node and goes up until destination.

We then evaluate the scalability of our algorithm by plotting the running time against the total number of ports. We can see from figure 7 that running time of our algorithm scales linearly to the number of ports in our network topology. This isn't a bad result. However, this was a very naive test. More complex efficiency and performance tests could be performed in future work. We propose a couple as follows:

(1) The scalability in this report was tested only in models of linear network topology. The DFS algorithm is all about branches and similar tests should be performed in a forest network topology where branching results. However, achieving fairness here is quite challenging. We list the challenges as follows:
    - Manually creating models for network with hundreds and thousands of nodes is impossible.
    - Randomly generating network topologies might not give good results. For example, say we have network 1 with 20 nodes and another network 2 with 10 nodes. In network 1, one has 5 possible ways to go from source to destination while in network 2, there is only a single way. Hence, the runtime of the algorithm for network 1 will come out to be larger than that for network 2

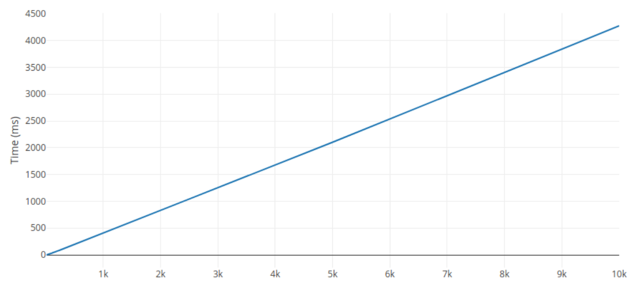


Figure 7: DFS algorithm scales linearly to the number of ports in linear network topology

which might lead to false conclusions (more the nodes, lesser the runtime!). To achieve fairness, we should thus factor in the number of possible branches while evaluating the results.

(2) As discussed before, DFS algorithm involves set intersection of forwarding rule predicates. Scalability test should also be done against the number of forwarding rules.

5 LIMITATIONS

As discussed in the last section, StateKeeper is currently facing performance and scalability issues. Realistically, it is still a toy, not a tool. It would take away an entire day for StateKeeper just to calculate atomic predicates set in a network as large as Stanford which isn't even too large. Hence, it is far from meeting performance standards of real networks.

As a result, StateKeeper is static and cannot take liveness into account. Changing any rules will involve the recalculation of atomic predicates which is extremely slow. Also, StateKeeper's reachability tree only indicates the expected (theoretical) values of throughput and delay. It cannot account for random jitter of traffic causing delays and throughput go up or down in real-time.

Another limitation of StateKeeper is that it has not been implemented to account for access control lists(ACL). Therefore it cannot account for dropped packets at ports because of ACL rules.

Once the performance is achieved after optimizations, StateKeeper can steal ideas from AP Verifier to do real-time analysis and also account for ACLs. AP Verifier's real-time analysis works by placing its verifier in between a software-defined network(SDN) controller and its middle boxes. It then can intercept network changes and verify its compliance. To account for ACLs, AP Verifier generates a separate set of atomic predicates specifically for ACLs. This is because forwarding rules and ACLs have different characteristics and locality properties.

6 CONCLUSION

Our paper presents StateKeeper: a network quantitative analysis and verification tool developed to analyze the issues of network reachability and quality of service. StateKeeper was built with the belief that analyzing throughput and delay can be just as important as verifying reachability failures. It works by generating a set of atomic predicates using header space algebra that represents the network forwarding rules. The atomic predicates can be used to compute reachability efficiently. While traversing the network checking for reachability, StateKeeper also keeps track of the accumulated delay and minimal throughput at each port. This gives the user key information on the progressions of different paths in the network.

REFERENCES

[1] N. Feamster, et al., "Detecting BGP Configuration Faults with Static Analysis.", NSDI, 2005
[2] P. Kazemian, et al., "Header Space Analysis: Static Checking for Networks.", NSDI, 2012
[3] H. Mai, et al., "Debugging the Data Plane with Anteater.", SIGCOMM, 2011
[4] M. Ghobadi, et al., "Rate Limiting YouTube Video Streaming.", USENIX, 2012
[5] D. Davis, "Control unwanted traffic on your Cisco Router with CAR.", TechRepublic, 2008, https://www.techrepublic.com/blog/data-center/control-unwanted-traffic-on-your-cisco-router-with-car-96521/
[6] H. Yang, et al., "Real-Time Verification of Network Properties Using Atomic Predicates.", in IEEE/ACM Transactions on Networking, vol. 24, no. 2, pp. 887-900, 2016
[7] N. Lopes, et al., "Checking Beliefs in Dynamic Networks.", NSDI, 2015