NetGen: Synthesizing Data-plane Configurations for Network Policies

Shambwaditya Saha Santhosh Prabhu P. Madhusudan
University of Illinois at Urbana-Champaign, USA.
{ssaha6,prabhum2,madhu}@illinois.edu

ABSTRACT

Networks are hard to maintain. When the physical network changes or when policies, most importantly security properties change, it is hard to change the network while maintaining all other existing policies. We study the problem of *network change synthesis*, where given a current network and a desired change for it expressed as a high-level policy, we automate the process of synthesizing changes in the data-plane configuration so that the policy is met. We develop a new language that allows the user to express desired reroutings and, given such a policy and a current network, we design a novel synthesis engine based on abstraction and constraint-solving that can find (minimal) changes to the current network that satisfies the policy. We report on a preliminary implementation of our technique that shows that we can effectively and efficiently synthesize changes in large networks.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

Keywords

Software defined network; data-plane syntheis; network management; programming languages; constraint-solving

1. INTRODUCTION

Computer networks require constant management, thanks to the critical nature of the traffic they carry, constant occurrence of failures, and never-ending changes to networking policies. Making changes to a network, though unavoidable, is a complicated task for network operators, due to the possibility of unforeseen ways in which updates may interact, potentially causing connectivity problems, or violation of policies. The problem is exacerbated by the distributed nature of networks, which makes it difficult to make the correct changes, and when there are problems, to detect and debug them. Often, network operators make changes that appear to be right, only to discover later that they were not. Worse still, faulty configurations sometimes go unnoticed until they get exploited, which results in huge losses to the organizations involved, both in terms of money and in terms of credibility [1, 10].

Though Software Defined Networks allow network administrators to program the network in a more intuitive manner, it still takes considerable effort in deciding how to enforce a desired policy. There have been attempts [6, 11] to simplify this process, by providing higher level abstractions that are useful in specifying what the administrator wishes to enforce in the network.

In this paper, we propose NetGen, a platform that combines the power of formal methods and synthesis techniques with the convenience of SDNs to enable automatic updates to network configurations, with respect to desired administrative requirements. NetGen provides a specification format that identifies packets and paths that need modification, where paths and modifications are given using regular expressions (inspired by existing work [15, 8]). NetGen uses packet abstractions, creates models of the current network for each packet class, and uses a logical constraint solver to come up with a new dataplane configuration that satisfies the specified requirement, while leaving its behavior on other packets unchanged.

For instance, consider the network fragment illustrated in Figure 1. F1 and F2 are firewalls, and switches A, B and C are configured to forward all HTTP traffic to F1. We may wish to now have F2 (and not F1) handle HTTP traffic that originates from the IP address x.y.z.w, for load balancing or other reasons. However, we may have other constraints to satisfy, such as:

1. in the new dataplane state, packets are still delivered to their original destinations
2. the rules at F1 and F2 should not be changed.

We can state this requirement using the following NetGen specification, and then synthesize the new network using the technique and tool provided in this paper:

```
match(TP_SRC_PORT=80) & match(IP_SRC=x.y.z.w),{A,B,C}:
   .* F1 .* => (N-F1)* F2 (N-F1)* od NM:{F1,F2}
```

The first line identifies a set of packets and a set of sources ($\{A, B, C\}$) and is followed by a path descriptor(`.* F1 .*`): these identify the packets from the sources following certain

SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA

ACM 978-1-4503-3451-8/15/06 ...$15.00
DOI: http://dx.doi.org/10.1145/2774993.2775006.

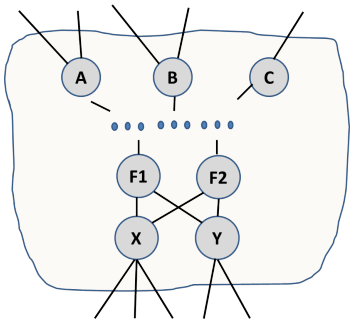

Figure 1: A fragment of a network

paths in the network that need to be rerouted. The path descriptor following ⇒ specifies the new route they should take (they shouldn't go through F1 but must go through F2 and reach their original destination *od*). Finally, the set `NM` gives the nodes whose next-hop relation cannot be changed. Semantics of specifications are explained further in Section 2.

Given this specification, and the current state of the network, NetGen can automatically determine what the new dataplane state should be (using a minimal number of changes to the current network), so that the identified traffic now passes through `F2`.

Though there have been previous attempts to synthesize network configurations for specific requirements [15, 12], to the best of our knowledge no work has yet studied the problem of synthesizing new configurations by making changes to an existing one. This makes NetGen unique in this growing space of synthesis tools for networks [15, 12, 17]. In general, synthesis tools face the problem of *underspecification*, wherein the operator would find it difficult to lay down the complete specification of the entire network precisely in order to create an acceptable network. However, we believe that synthesizing only delta-changes to the current network by specifying the change that is desired is more practical, allowing the operator to keep the rest of the design intact. We also note that using NetGen doesn't preclude any manual changes that the operator may wish to the make to the network, like installing static routes or updating firewall rules.

The key contributions of this paper are:

1. A language based for expressing the desired changes in an existing network.
2. A scalable synthesis engine for finding network changes based on abstractions and constraint-solvers that generates new dataplane states, that are different from the current network only in terms of the specified change, and that is guaranteed to be correct.

2. THE SPECIFICATION LANGUAGE

The language for specifying updates in NetGen is based on regular expressions, similar to notations used in existing work [8, 15]. Figure 2 illustrates the definition of the language. A specification is of the form
`ts, S: old_path` ⇒ `new_path NM:R` consists of five major components.

- A traffic spec `ts` describing the packets to which the change is to be applied. We assume that packets are not modified by the network.

```
spec ::= traffic, S: path => path[od] [NM:R]

traffic ts ::= true | match(header=val) |
match(header,prefix) | (ts & ts) | (ts | ts)
| (not ts)

path p ::= n | .  | p + p | p p | p*
```

where $S, R \subseteq N$ and $n \in N$.

Figure 2: Specification Language.

- A source-set `S` specifying the devices starting from where route changes need to be made.
- A current-path spec `old_path` defining the paths that need to be altered.
- A target-path spec `new_path` specifying the new paths that packets satisfying `ts` starting from `S` that take paths `old_path` must take in the new network.
- Optionally, a specification of a set of nodes `R` that should not be modified (`NM`) at all, in computing the new dataplane state.

For instance, consider the following rerouting spec:
All HTTP traffic entering through ingress router R1 should reach server S1.

This can be specified in the language as follows:

```
match(TP_SRC_PORT=80), {R1}:  .* => .* S1
```

The above specification is quite intuitive. The traffic spec (`match(TP_SRC_PORT=80)`) specifies our intention of changing the behavior of HTTP traffic. `{R1}` informs NetGen that only paths starting from router R1 need to change. `.*` enforces that all existing paths are of interest, and `.* S1` specifies the nature of the final paths to be synthesized. Since no nodes are specified as non-mutable, NetGen assumes that the forwarding table of any forwarding element may be changed.

To make the language more flexible and useful, we allow for the use of a special symbol `od`, which stands for *original destination*. To see how the symbol is useful, let us consider a scenario where for maintenance, we wish to redirect all traffic passing through a switch, say S1. The specification for this change can be written as follows:

```
true, U: .* => (N-S1)* od
```

Here, `U` denotes the set of all ingress points into the network and `N` denotes all nodes. The use of `od` ensures that the traffic is delivered in the new network to the destination to which it is being delivered in the current network. A list of example requirements and their specifications are mentioned in Table 1.

3. THE SYNTHESIS ALGORITHM

A *network topology* is a three-tuple $Top = (N, \rightarrow, External)$, where N is a finite set of nodes, $\rightarrow \subseteq N \times N$ is a binary relation that gives the connectivity of nodes in the underlying network topology, and $External \subseteq N$ identifies the set of external nodes, which are nodes representing the world, and which we have no control over. We assume there are no outgoing edges from external nodes; further, let us assume a special external node *drop*, which is connected to all non-external nodes and has no outgoing edges, and where forwarding to this node models the dropping of a packet.

Let us fix a finite set of packet (headers) $\mathbb{P}$; this set essentially constitutes all valid packet headers.

Requirement	Specification
Traffic to IP x.y.z.w entering at A is dropped	`match(IP_DST=x.y.z.w), {A}: .* => .* Drop`
Traffic to IP x.y.z.w entering through nodes in S passes through a traffic monitor	`match(IP_DST=x.y.z.w), S: .* => .* Monitor .* od`
No traffic entering through nodes in S goes through S1	`true, S: .* S1 .* => (N-S1)* od`
HTTP traffic entering through A, B and C should go through F2 instead of F1	`match(TP_SRC_PORT=80), {A,B,C}: .* F1 .* => (N-F1)* F2 (N-F1)* od NM:{F1,F2}`
All traffic passing the traffic monitor and then the firewall should traverse firewall first and then the traffic monitor	`true, U: .* FW .* Monitor .* => .* Monitor .* FW .* od NM:{FW}`

Table 1: Example rerouting specifications.

A network over a network topology *Top* and packets $\mathbb{P}$ is a structure $\mathcal{N} = (N, Rules)$, where $Rules : N \times Packets \longrightarrow N$, a mapping that maps pairs of nodes and packets to the next-hop node (dropped packets are sent to the node *drop*). *Rules* are usually compactly encoded using predicates on header attributes of packets, and are installed at individual nodes in look-up tables.

A network synthesis problem over the topology *Top* and the set of packets $\mathbb{P}$ is specified by a pair $(\mathcal{N}, \alpha)$, where $\mathcal{N}$ is the current network and α is the desired specification of the new network we want to synthesize by making a small number of changes to the current network. Let us fix a specification $\alpha =$ "$ts, S : old\_path \Rightarrow new\_path\ NM : R$" for the rest of the paper.

Our synthesis algorithm proceeds in several phases:

Phase I: Abstract the network using packet equivalence classes; construct the abstract network restricted to the packet classes mentioned in the spec ts.

Phase II: Convert specifications *old_path* and *new_path* to automata $\mathcal{A}_{old}$ and $\mathcal{A}_{new}$, identifying and replacing *od* labels.

Phase III: Identify further immutable nodes in the network from the abstract network and the traffic specification old_{path}.

Phase IV: Reduce synthesis to a constraint satisfaction problem, and solve it using SMT solvers.

Phase V: Extract the network update from the satisfying model.

In Phase 1, we will identify packet-classes and abstract the network according to these. Phases II-V can in fact be performed independently for each packet-class, as the synthesis for one packet-class is completely independent of another. This makes our solution highly parallelizable. Notice however that synthesis *cannot* be separately done for each source node mentioned in S. To see why, consider a scenario where there are two sources s_1 and s_2 in S, and the routes of a packet p starting at these nodes eventually merge at a node n before proceeding to their destination in the network. Now, assume that we want to reroute these packets to a firewall F, but this firewall is accessible in the topology only after they cross the node n. In this case, the rerouting specialized to either source is *not realizable* since it demands that the path taken by the packet from the other source takes the same path.

We now give details of the various phases.

I. Network Abstraction using Packet Equivalence

Though there are a large number of packets, they can be partitioned into equivalence classes, where two packets in the same equivalence class are forwarded by every node in precisely the same way. The idea of using equivalence classes of packets is not new (for instance, Veriflow [4] uses equivalence classes of packets to verify correctness of flows in networks).

Let us fix an equivalence relation over packets that satisfies the following condition (which we will ensure during construction of equivalence classes):

($*$) For any packet-class pc, any two packets p, p' in pc,

- *every* node in the network forwards both packets to the same node as the next-hop, or drops both of them, and
- either both p, p' satisfy the traffic description ts given in the specification or neither do.

We can then *abstract* the network into one that forwards packet-classes, instead of packets. The resulting abstract networks is: $\mathcal{AN} = (N, ARules)$, where the abstract rules $ARules : N \times PacketClasses \to N$ is given by $ARules(n, pc) = n'$ if there exists (equivalently, for every) packet $p \in pc$, $Rules(n, p) = n'$, for every packet-class pc and every $n \in N$.

Notice that the traffic description ts in the specification is the union of a finite set of packet-classes, which we shall refer to as the *relevant* packet classes, and denote them as the set *SpecPC*. We now restrict the abstract network to only contain forwarding edges for these relevant packet-classes in *SpecPC*. We will henceforth work only with this reduced abstract network, reformulating the synthesis problem as a synthesis problem on this abstract network. Note that this is not a limitation in any way–since the packet equivalence classes are indistinguishable by all nodes in the current network and by the specification, we can always work with this level of granularity. In fact, the *coarser* this granularity is, the better the synthesized solution would be, as it would make minimal changes to the original network.

For each packet-class $pc \in SpecPC$, we can construct the abstract network for each packet-class $pc \in SpecPC$ and apply the next phases of synthesis to each such network independently. We will assume that the current network doesn't have a cycle on any packet; hence this network will be acyclic. We identify the sources (nodes with no incoming edges) and sinks (nodes with no outgoing edges) in each such graph.

II: Specification routing to Automata

We convert both *old_path* into finite automaton $\mathcal{A}_{old}$ over the alphabet N and *new_path* into a finite automaton $\mathcal{A}_{new}$ over the alphabet $N \cup \{od\}$). Furthermore, for every packet class pc, we examine the abstract network on it: and, for any source $s \in S$, we find the original destination that this packet gets forwarded to in the network. Let us denote this by $od(s, pc)$; we will use this later in the SMT formulation.

III: Identify further immutable nodes

The specification demands a rerouting of packets from the sources S mentioned in the specification, but also demands that for other sources and packets, the network remain the *same*. Our synthesis algorithm will only reroute packets that satisfy the packet description given in the specification. However, for packets mentioned in the specification and for sources *not* in S, we need to ensure that the routing of these packets in the network are unaffected by our network change. Furthermore, for a packet mentioned in the specification and for a source in S, we must leave this routing unaffected if it is *not* in the current path specification *old_path*.

We achieve this by taking the path taken by packet classes mentioned in the specification (ts) from sources not in S, and mark all these nodes as *immutable* nodes. More formally, we identify, for every packet-class pc, a set $Imm(pc)$ which contains these immutable nodes. Also, for every packet class pc mentioned in the specification (ts) and for every source $s \in S$, we take the path taken by the packet from this source in the current abstract network, and check if the path conforms to the regular expression *old_path*, by checking acceptance by $\mathcal{A}_{old}$. If the path does not conform, then we add these nodes to the immutable set for pc.

We also add the immutable set R in the specification (`NM:R`) and the external nodes as immutable nodes for every packet class. Our SMT constraints will ensure that the immutability of these nodes is respected.

IV: Modeling synthesis using constraints

We now describe the core of our technique, which is to reduce the synthesis problem to a constraint-satisfaction problem over a decidable logical theory, namely the quantifier-free theory of *uninterpreted functions*. These constraints can be solved effectively by the emerging class of SMT (Satisfiability Modulo Theories) solvers [3], which furthermore can return a *model* satisfying the constraints, which then lead us to the synthesized network. Though network, rules, packets, etc. can be described by variables ranging over a *bounded* domain, and hence can be already expressed as a SAT (Boolean satisfiability) problem, we choose the theory of uninterpreted functions in order to encode *reachability* relations in the network using *recursive definitions*, including the complex reachability constraints expressed by the regular expressions in the specification.

Given the abstract network $\mathcal{AN} = (N, ARules)$, we model the nodes as natural numbers in the range $\{0, 1, \ldots, |N|\}$ (with 0 corresponding to the *drop*-node), the packet-classes also as natural numbers $\{1, \ldots, |SpecPC|\}$, and the abstract rules as a function $R : [1, |N|] \times [1, |SpecPC|] \to [0, |N|]$.

Modeling delta-changes to the data-plane:

We model the fact that we want a small change to the current network by setting a budget $k \in \mathbb{N}$, and allowing k next-hop routing changes to the current network; k is incremented if the constraints are unsatisfiable, till we can construct a network (we can stop when k reaches the total number of nodes in the network and this would signal an unrealizable specification). More precisely, we allow for k changes, where each change involves a single node and how it forwards a packet-class that flows into it. We model this using k triples $\langle n_i, pc_i, n_i' \rangle$, $i = 1, \ldots k$, where these reflect the fact that n_i sends any incoming packet in the packet class pc_i to n_i' instead of where it is sent in the current network. The constraints we add hence are, for each packet-class $pc \in SpecPC$:

$$\bigwedge_{i \in [1,k]} \left(\bigwedge_{m \in Imm(pc)} (n_i \neq m) \quad \wedge \bigvee_{n,n' : n \to n'} (n_i = n \wedge n_i' = n') \right)$$

which constrains the changes to not affect immutable nodes and to conform to the network topology *Top*.

Modeling the rerouting spec on the new network:

Finally, we model the fact that the routing in the new network for the relevant packet classes mentioned in ts from the sources S that satisfy the path specification *old_path* conform to the new routing specification *new_path*. In order to express this, we will impose constraints that demand a *witness* that *every* such routing path in the new network be accepted by $\mathcal{A}_{new}$. Let us remove the *od* element from this automaton by replacing it with an arbitrary node; we will handle constraints regarding original desitination separately.

We first *reverse* the language of $\mathcal{A}_{new}$ and *determinize* this automaton to get a deterministic automaton $\mathcal{B}_{new}$ for the reversal language. Note that for any destination node and any packet class, the set of all nodes that forward this packet class to that destination forms a *tree*, with the destination as the root. Our aim is to label this tree using states of the automaton $\mathcal{B}_{new}$, so that it gives a *uniform* witness that *all* paths from any source to any destination on any packet class satisfies the specification.

Let $\mathcal{B}_{new} = (Q, q_0, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta : Q \times N \to Q$ is the transition function. Let us assume that states are natural numbers in the range $[1, |Q|]$, with $q_0 = 1$. We introduce an uninterpreted function $\rho : N \times SpecPC \to Q$ and impose the following constraints: for every sink node n in the network and for each pc, we have the constraint:

$$\rho(n, pc) = \delta(q_0, n)$$

and for every non-sink node n in the network for pc we have:

$$\bigwedge_{i=1}^{k} \left[(n = n_i \wedge pc = pc_i) \Rightarrow \rho(n, pc) = \delta(\rho(n_i', pc), n) \right]$$

$$\wedge \bigwedge_{i=1}^{k} \left[(n \neq n_i \vee pc \neq pc_i) \right] \Rightarrow \rho(n, pc) = \delta(\rho(R(n, pc), pc), n)$$

The above constraints essentially "run" the automaton $\mathcal{B}_{new}$ on the new network, using the new next-hop relations given by the tuples $\langle n_i, pc_i, n_i' \rangle$, and falling back on the current network (R) if a node's routing hasn't been modified.

Also, for each source node $s \in S$ in the specification (and those only) and packet-class pc, to ensure that the path in the current network from s met the path condition *old_path*, we impose the following constraint:

$$\bigvee_{q_j \in F} \rho(s, pc) = q_j$$

The salient aspect of the above formulation is that reachability constraints in the network are captured using uninterpreted functions that are recursively defined; this technique is motivated by similar techniques in verifying programs that do delta-changes to data-structures in the program verification literature [9]. Furthermore, these functions also capture the *witness* in terms of state-labeling of the nodes that prove the specification is satisfied; since automata have only local constraints between states, they can be captured by writing down one constraint per node. Consequently, the entire

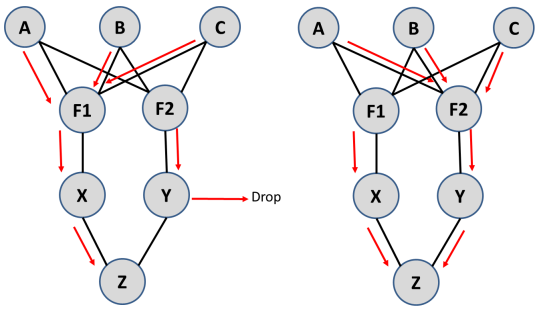

Figure 3: Update synthesized by NegGen (The original configuration is on the left, and the synthesized one on the right).

encoding is *linear* in the size of the abstract network (as opposed to quadratic-sized constraints if we captured reachability in r steps, for each r in the range $[1, n]$, as would be required in a SAT encoding).

Finally, we need to ensure that there are no *cycles* in the new network that is being synthesized (in fact, even the correctness of the various conditions above depend on this property). We ensure cycle-freedom by assigning a *rank*, which is a natural number to every node, where all sink nodes get rank 0 and every node gets a rank *greater* than the rank of its successor node in the network. We introduce an uninterpreted function $\pi : N \times SpecPC \rightarrow \mathbb{N}$ and impose the following constraints, for each packet-class pc:

$$\pi(n, pc) = 0 \text{ for any sink node } n$$

and for every non-sink node n:

$$\bigwedge_{i=1}^{k} \left[(n = n_i \wedge pc = pc_i) \Rightarrow \pi(n, pc) > \pi(n_i', pc)\right]$$

$$\wedge \quad \bigwedge_{i=1}^{k} \left[(n \neq n_i \vee pc \neq pc_i)\right] \Rightarrow \pi(n, pc) > \pi(R(n, pc), pc)$$

Notice again that the use of uninterpreted functions and arithmetic helps us capture the above property of acyclicity succinctly.

Finally, we define another recursive function *dest* which is constrained to map each node and packet-class to the final sink node that it is forwarded to. Its definition is similar to the above definitions, and we skip it. If the specification demands that the original destination of packet-classes are maintained (has the optional *od*), then we demand that the destination node for all sources s in the specification is the original destination $od(s, pc)$.

V: Extracting the updates to the network

We use SMT solvers for solving our constraints, which returns to us a minimal change in the current network state (in terms of changes of next-hop relation on packet-classes) that results in a new network state that satisfies the new policy on the routes given by the specification. This change, modeled using the variables $\langle n_i, pc_i, n_i' \rangle$, $i \in [1, k]$, and these changes can be readily incorporated. We expect NetGen to be running as an SDN application, and the updates can be inserted into the network by the controller. However, while migrating from the current network to the new network, care should be taken to do the migration carefully to ensure that packets that get through during the change are handled correctly. This is a problem that has been well studied in recent research [13, 14, 5].

Spec.	# Classes Modified	# Changes	Time
Drop Traffic	10	10	1.1 sec
Pass Traffic through Monitor	10	53	8.6 sec
Do not use a specified switch	245	1481	253 sec

Table 2: Evaluation.

4. IMPLEMENTATION AND EVALUATION

Implementation: We have implemented the core synthesis formulation that forms the primary component of the NetGen framework. The first major challenge in deploying NetGen on real-world SDN is to have an accurate knowledge of the current dataplane state, with equivalence classes defined that satisfy the conditions listed in Section 3. In our experiments with real-world AS dataplane states, we used the equivalence classes defined by the Veriflow dataplane verifier [4]. By tweaking Veriflow slightly, we can get an abstraction that computes an equivalence class that satisfies both our requirements (that packets in any class are indistinguishable by both the nodes in the network as well as the specification). Note that while doing synthesis online, we can maintain up-to-date information about the network state, in terms of this abstraction, by simply ensuring that every change made to the dataplane is reported to the synthesis engine. Hence, we need not compute the abstraction from scratch each time we do synthesis.

The synthesis engine itself uses the uninterpreted linear arithmetic solver in Z3, from Microsoft Research[3]. The synthesis proceeds by synthesizing minimal changes to the network (by increasing the budget k till synthesis succeeds) for each packet-class in the specification independently.

Experiments: All experiments were performed on a system with an Intel Core i7 2.2 GHz processor and 4GB main memory running 64-bit Ubuntu 14.04 OS.

We evaluated the correctness of the synthesis formulation by synthesizing the update specifications listed in Table 1 over hand-created topologies. Figure 3 illustrates how the fourth specification from Table 1 was synthesized over an 8 node network. The traffic specification is exactly captured by the equivalence class of packet pc. It can be seen that as mandated by the spec, packets in pc starting from nodes A, B and C now pass through $F2$ instead of $F1$, and the specified packets still reach the original destinations. Moreover, the synthesis engine picks the most reasonable set of changes to obtain the new dataplane state, due to the *minimal* k constraint.

To test the scalability of our technique, we also synthesized variants of the first three specs over a snapshot of the Rocketfuel [16] topology AS 1755, consisting of 172 nodes with 5 million forwarding entries overall, generated through an OSPF simulation. Table 2 describes the time taken for synthesis. The numbers suggest that our technique scales well to networks of larger size.

5. RELATED WORK

Recent years have seen a lot of interest in the use of synthesis techniques for networks. Merlin [15] allowed specification of path and bandwidth constraints in a regular expression based language, and synthesized network configurations to meet the constraints. FatTire [12] enabled synthesis for fault tolerance specifications expressed similarly. Alloy [7] is another tool that falls into the network configuration synthesis category. NetGen is different from these tools in that the administrator is only required to specify a change to the existing network configuration, rather than an entirely new network configuration. Work on automatic firewall fixing by Chen et al. [2] is similar in spirit to Netgen, but applies only to firewall policies, with respect to correcting specific kind of errors. In comparison, NetGen is more flexible, and widely applicable. NetGen, being a dataplane synthesis tool, is also different from control plane synthesis tools like NetEgg [17], which synthesizes SDN programs from example behaviors.

Synthesis (and more generally formal methods) has also been used in the context of change management, for migrating from one network configuration to another [13, 18]. NetGen is orthogonal to these techniques. NetGen is targeted at synthesis of network configurations themselves. Change management techniques can be then used to migrate the network to the configurations synthesized by NetGen.

6. FUTURE WORK

The primary future direction we see is a full-fledged implementation of our technique as an SDN application. We see our tool as being part of a larger suite of formal tools that make it convenient to manage the network in a provably correct manner. We would also like to see an extension of our technique to a larger class of specifications, like QoS constraints; our general technique of using SMT solvers is easily extensible as these constraints can be expressed in decidable SMT logics.

7. ACKNOWLEDGEMENTS

We thank Matthew Caesar, Philip Brighten Godfrey, and Boon Thau Loo for discussions, and thank Ahmed Khurshid for providing us with sample networks. This work was partilly supported by NSF Expeditions in Computing ExCAPE Award #1138994.

8. REFERENCES

[1] Abrams, R. Target Puts Data Breach Costs at $148 Million, and Forecasts Profit Drop. `http://tinyurl.com/l36payc`, August 2014.

[2] Chen, F., Liu, A. X., Hwang, J., and Xie, T. First Step Towards Automatic Correction of Firewall Policy Faults. *ACM Trans. Auton. Adapt. Syst. 7*, 2 (July 2012), 27:1–27:24.

[3] De Moura, L., and Bjørner, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 337–340.

[4] 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.

[5] McClurg, J., Foster, N., and Cerný, P. Efficient synthesis of network updates. *CoRR abs/1403.5843* (2014).

[6] 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), pp. 1–14.

[7] Narain, S. Network Configuration Management via Model Finding. In *Proceedings of the 19th Conference on Large Installation System Administration Conference - Volume 19* (2005), pp. 15–15.

[8] Narayana, S., Rexford, J., and Walker, D. Compiling Path Queries in Software-defined Networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (2014), pp. 181–186.

[9] Pek, E., and Madhusudan, P. Explicit and symbolic techniques for fast and scalable points-to analysis. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014* (2014), pp. 1–6.

[10] Rainone, C. Time warner cable says outage largely resolved. `http://tinyurl.com/m6y4qul`, August 2014.

[11] Reich, J., Monsanto, C., Foster, N., Rexford, J., and Walker, D. Modular SDN Programming with Pyretic. *USENIX ;login 38*, 5 (October 2013).

[12] 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.

[13] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), pp. 323–334.

[14] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. Consistent Updates for Software-defined Networks: Change You Can Believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), pp. 7:1–7:6.

[15] 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.

[16] Spring, N., Mahajan, R., Wetherall, D., and Anderson, T. Measuring isp topologies with rocketfuel. In *In Proc. ACM SIGCOMM* (2002), pp. 133–145.

[17] Yuan, Y., Alur, R., and Loo, B. T. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (2014), pp. 20:1–20:7.

[18] Zhou, W., Jin, D., Croft, J., Caesar, M., and Godfrey, P. B. Enforcing customizable consistency properties in software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (May 2015), pp. 73–85.