Composing Middlebox and Traffic Engineering Policies in SDNs

Yiyang Chang*, Gustavo Petri†, Sanjay Rao*, Tiark Rompf*
*Purdue University, †LIAFA – Université Paris Diderot

***Abstract*—Middleboxes present new requirements that need to be integrated with traffic engineering applications that are already complex and consider myriad factors (e.g., routing, QoS, load-balancing). While it is possible to revisit traffic engineering algorithms to explicitly integrate middleboxes, such an approach is not compositional. Existing efforts at compositional SDN application development do not apply since they support application composition *after* static packet-forwarding policies are generated by application modules. Consequently, routes computed by a traffic engineering module cannot be influenced by the constraints imposed by a module that specifies middlebox traversal requirements. In this paper, we explore an alternate approach where application composition is done *prior to* the generation of packet-forwarding policies. Each application is written as a logic program, and provides a set of *requirements* that must be respected by a synthesized solution. A constraint solving engine iterates over these requirements to search the solution space and find a solution respecting all the requirements. We illustrate our approach with a concrete case study, and implement it using the Z3 SMT solver [1]. Our initial results point to the promise of the approach.**

I. Introduction

With the extensive deployment of middleboxes in enterprise and ISP networks, there is much recent interest in revisiting traffic engineering to take the presence of such middleboxes into account [2], [3], [4]. For instance, operators may often need to specify policies which require that traffic of a particular type traverse a given sequence of middleboxes.

One approach is to revisit existing traffic engineering algorithms to explicitly account for new requirements posed by middleboxes (e.g., [3]). This approach requires developing custom algorithms for every combination of traffic engineering and middlebox requirement. However, consider that middlebox requirements must be considered as part of network design, along with many other considerations such as routing, QoS, and load-balancing. Further, large networks are often managed by multiple teams of operators, and may need to support applications developed by many different vendors [5], [6]. Developing custom algorithms to consider middlebox requirements may not be viable as the number of applications, and potential combination of application requirements grow.

Our goal in this paper is to facilitate a compositional approach to integrating middlebox requirements in traffic engineering and SDN applications. Our work is motivated by the following typical enterprise scenario. Consider two IT staff members (say Alice and Bob), who belong to two different IT teams that manage the network of an enterprise. Alice is in charge of the routing module, and Bob is responsible for network security. Working separately, Alice implements a shortest-path algorithm, while Bob enforces a policy that requires all network traffic to traverse an intrusion detection system (IDS). Could these modules be easily composed without Alice and Bob being explicitly aware of their respective implementations?

There have been some initial efforts at tackling the problem of compositionality in SDNs, notably the Frenetic project [7], [8]. However, the scenario above cannot be supported with the Frenetic family of languages, and its associated sequential and parallel composition operators [7], [8]. This is because path computations are first done in a general purpose language such as Python (guided by traffic engineering goals), and composition is done *subsequently* using computed paths. The routing module would still produce packet-forwarding policies corresponding to the shortest path, which may not traverse an IDS. It is unclear how to then compose the outcome with the security module to ensure that the path traverses an IDS. While it is possible to modify the code in a general purpose language (e.g., Python) to generate shortest paths that traverse an IDS prior to the generation of the packet-forwarding policies, this approach is not modular.

In this paper, we investigate an alternate approach where compositionality is supported *prior to* the generation of packet forwarding policies. In our approach, each application is written as a logic program, and provides a set of *requirements* that must be respected by a synthesized solution. A constraint solving engine iterates over these requirements to search the solution space and find a solution respecting all the requirements.

A key issue is generating the constraints associated with each application such that they can be combined together in a correct fashion. As a concrete case study, we consider routing in the context of the IDS requirement. We show that the traditional approach of expressing the shortest path requirement leads to a formulation that does not compose correctly with the IDS requirement. We propose an alternate *walk-based* formulation for the shortest path requirement which has the attractive property that the formulation can be easily composed with the IDS requirements.

We discuss how our approach may be generalized to consider a richer set of traffic engineering objectives such as picking paths that minimize link utilizations, and routing using multiple diverse paths.

We present preliminary performance results based on the Microsoft Z3 SMT solver [1]. Our focus is on an offline phase

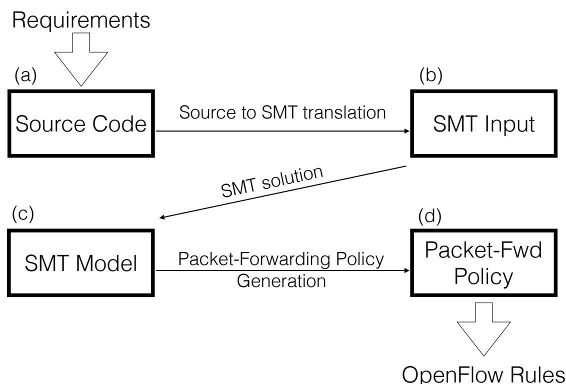

Fig. 1. Framework

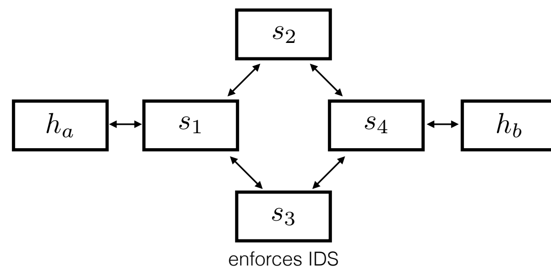

Fig. 2. IDS Example

for traffic engineering that involves a selection of paths that correctly meet the middlebox requirements [4], [9]. Our results are promising, while also pointing to important questions that we hope to address as part of our ongoing research.

II. From requirements to rules

Figure 1 presents the overall framework that we envision for composing application requirements. The framework consists of the following components:

(a) **Specifying requirements:** Requirements are expressed in a high-level Domain Specific Language (DSL). It frees operators from directly expressing policy in a low-level constraint solving engine, which is tedious and error-prone, and also allows for modularity in that the underlying solver could be modified without changing how requirements are specified. The DSL includes primitives for network objects such as switches, routers, packets with their different fields, and the notions of links and paths. While our DSL could be embedded in many host languages, for concreteness, we consider a Prolog-like syntax where constraints are expressed as rules determining the necessary conditions under which a certain property should hold.

(b) **Solver engine:** To solve the aggregated constraints of different policies, any constraint solving engine could be used (not necessarily the constraint solving engine associated with Prolog). We focus on Satisfiability Modulo Theory (SMT) solvers, which are efficient constraint solving engines that are capable of finding a model for a set of constraints, usually expressed as a first order logical formula. More generally, other solver engines (e.g., based on Integer Linear Programming (ILP)), or other traditional optimization techniques could be used. We focus on SMT solvers since more complicated formulas with arbitrary Boolean combinations of linear inequalities are more naturally expressed with SMTs, but defer a detailed investigation of the trade-offs between solvers to future work. Hence, the first step of our tool is a translation from our Prolog-like source language, to an axiomatic encoding of the network and the policies to be implemented to the input language of the SMT solver, which is first order logic, with the addition of decidable theories such as arithmetic and uninterpreted functions.

(c) **Model synthesis:** Once the constraints are encoded into the SMT solver, the solver is invoked. The resulting model is regarded as defining the flow-level policies that have to be implemented in the network. When a model cannot be found, either because the set of constraints are contradictory with each other, or because the SMT solver cannot find a satisfiable solution within a reasonable amount of time, an error is reported back to the user. In Sec. V, we also discuss the support to specify soft requirements and how conflicts could be handled by relaxing such requirements.

(d) **Generation of packet forwarding policies:** The final step of our toolchain is to translate the model into switch rules. To simplify that task, we compile the model into a language to express packet-forwarding policies. We use Pyretic [7] for this purpose given its publicly available code base, and extensive usage.

III. Composing Requirements

We illustrate our approach by expanding on the example introduced in Sec. I. Consider a network with two hosts ($\{h_a, h_b\}$), and four switches ($\{s_1, s_2, s_3, s_4\}$) organized as depicted in Figure 2. Consider the case where only s_3 forces traffic through an IDS.

The first operator of our network is required to define the routing from host h_a to h_b. Unbeknownst to him/her, the second operator is tasked with ensuring that all routes go through a switch enforcing the passage through an IDS (s_3).

A question that arises in this environment is: how can the implementers of the two different procedures, the one for computing optimized routes, and the other for enforcing the IDS policy, write their procedures in a way that enables composing each other (or with any other policy written in the same framework) without knowing a priori that the operator will choose to compose them? Our work is motivated by this challenging question.

In our approach, each application provides *requirements*, i.e., a set of constraints that must be satisfied by any packet-forwarding policy finally deployed in the network. A constraint

solving engine then attempts to obtain a network policy that satisfies the requirements of each module. If a model satisfying the requirements can be found, this model can be translated into packet-forwarding policies to implement the solution.

To illustrate our proposed solution, let us consider a candidate implementation in our framework. Assume that the first operator defines a route as a relation between a pair of hosts and a path, in this case the list of switches. Consider a Prolog-like constraint query (fed to an SMT solver) to find a route X between hosts h_a and h_b defined by the clause

route(h_a,h_b,X)

A possible solution to the query would route traffic from h_a to h_b through the switches s_1, s_2 and s_4. Formally, the solution results in

X = [s_1, s_2, s_4]

While this path routes traffic correctly, it fails to enforce the IDS policy tasked to the second operator. To remedy this situation, the second operator could consider the following strengthening to the routing policy:

```
hasIDS([s3|X]).
hasIDS([S|X]) :- hasIDS(X).
routeIDS(H1,H2,X) :- route(H1,H2,X),
                     hasIDS(X).
```

The definition hasIDS above tests whether a route contains a switch enforcing an IDS or not (in our case must have s_3)[1]. Then, the clause routeIDS uses route to find a path X, but we add the constraint that the path found must contain an IDS. Thus, using routeIDS the operator is able to find a correct path through an IDS if there is one. In our example this is reflected in the query below:

routeIDS(h_a,h_b,X).
X = [s_1, s_3, s_4]

We next discuss in Sec. IV how to translate route and hasIDS into constraints in an SMT solver.

IV. Compositional Representation

Ideally, a network operator could specify multiple goals for the network independently, and rely on our runtime system to *compose* and solve the combination of constraints. We will use the case study of routing and IDS to show naively composing constraints may not always work, and present a more compositional formulation.

Naive composition may not work. Consider the following shortest-path formulation on a graph $G(V, E)$, with V the set of vertices of the graph, and E the set of edges. Indicator variable $x_{i,j}$ is 1 if link $(i,\ j)$ lies along the path, and 0 otherwise; s is the source node and d is the destination node.

$$\forall i,\ \neg x_{i,i} \quad (1)$$
$$(\exists i,\ x_{s,i}) \wedge (\exists i,\ x_{i,d}) \quad (2)$$
$$\forall i, j,\ x_{i,j} \wedge (j \neq d) \Rightarrow \exists k,\ x_{j,k} \quad (3)$$
$$\forall i, j, k,\ x_{i,j} \wedge (i \neq k) \Rightarrow \neg x_{k,j} \quad (4)$$
$$\forall i, j,\ x_{i,j} \wedge (i \neq s) \Rightarrow \exists k,\ x_{k,i} \quad (5)$$
$$\forall i, j, k,\ x_{i,j} \wedge (j \neq k) \Rightarrow \neg x_{i,k} \quad (6)$$

The first constraint requires there are no self-loop links chosen.[2] The second constraint requires a unit of flow to leave the source, and a unit of flow to reach the destination. Constraint 3 requires that for every node with an incoming flow (other than the destination) there must be an exit flow, and Constraint 4 states that at most one incoming flow is permitted for each node. Symmetric constraints are added for exit flows in Constraints 5 and 6.

The path length is the sum of all $x_{i,j}$ variables, minimizing which leads to the shortest path. To do so, we use the theory of arithmetic to constrain the number of $x_{i,j}$ variables with value 1 to a desired bound (M). We iteratively increase M until a solution is found (which corresponds to the shortest path) or until the bound reaches the number of nodes in the graph without finding a solution. In the latter case the constraints cannot be solved.

Now we would like to add a constraint imposing that the path must have a waypoint w (e.g., IDS) by simply adding the following constraint $\exists j, x_{w,j}$. However, this is *incorrect* because it could result in a solution that involves the shortest path from the source s to destination d, along with a *disconnected loop* that involves the waypoint w. Instead, we need a solution where the waypoint is actually traversed in the path.

Custom formulations may not always be viable. The issues above can be addressed by generating a custom formulation based on the observation that the optimal solution involves a combination of (i) the shortest path from the source to the IDS; and (ii) the shortest path from the IDS to the destination. However, given that there could be several SDN applications with a variety of requirements (e.g, QoS, robustness, security etc.), the potential combinations of constraints is high and writing custom formulations for every possible combination may not be viable. For instance, a completely different custom formulation may be required to capture requirements that multiple IDS's must be traversed where the order of traversal is unimportant, compared to one where the ordering is important. Thus, we are interested in a more generic formulation for the shortest path requirement which can easily support additional requirements added in a compositional fashion.

Compositional formulation. Here we give a logic-based formulation of route and hasIDS which is compositional and easy to reason about. Consider a graph $G(V, E)$, with V the

[1] We use the Prolog syntax [X|XS] to represent a list with head X and tail XS. Upper case arguments (X) are variables, and lower case names are constants.

[2] We use quantifiers that are restricted to the finite domain of the given graph.

set of vertices of the graph, ranged over by the metavariables $i \in V$, and E the set of edges encoded as a set of binary variables $e_{i,j}$ indicating whether there is an edge between the vertices i and j (i.e., $e_{i,j} \in E \iff e_{i,j} = 1$). The constraint imposed by `route` seeks to find a valid walk from a source node $s \in V$ to a destination node $d \in V$. Let M be the maximum permissible length of walk, counting the source and destination nodes. We add a set of M binary variables t_k where $k \leq M$ which allow us to encode the length of the walk. Then, a walk of size K will satisfy $t_k = 1$ for $k \in [1, K]$ and $t_k = 0$ otherwise. Finally, consider a set of binary variables $x_{i,k}$ representing the fact that the node i should be visited in the k-th step of the walk found if $x_{i,k} = 1$, and the node should not be visited otherwise.

Then, we can solve the `route` queries by solving the following set of constraints:

$$x_{s,1} \wedge \ t_1 \tag{7}$$

$$\forall i, k,\ x_{i,k} \wedge x_{j,k+1} \Rightarrow e_{i,j} \tag{8}$$

$$\forall k,\ t_k \wedge \neg t_{k+1} \Rightarrow x_{d,k} \tag{9}$$

$$\forall i, j, k, i \neq j \Rightarrow \neg x_{i,k} \vee \neg x_{j,k} \tag{10}$$

$$\forall i, k,\ x_{i,k} \Rightarrow t_k \tag{11}$$

$$\forall k,\ \neg t_k \Rightarrow \neg t_{k+1} \tag{12}$$

$$\exists k, x_{d,k} \wedge \neg t_{k+1} \tag{13}$$

Finding the minimum M which makes the query above satisfiable guarantees that the found path is minimal.

Constraint 7 states that the source node s is scheduled first. Constraint 8 requires that if node i is visited in step k and j is visited in step $k+1$ an edge must exist between nodes i and j in G. Constraint 9 ensures that the last node of the walk is the destination d. The requirement that $t_k \wedge \neg t_{k+1}$ encodes the fact that the walk has exactly k steps. Constraints 12 and 13 ensure that the destination node d exists in the path and eliminates trivial solutions. Constraint 10 requires that at most one node is visited in step k. Finally Constraint 11 establishes the relation between variable $x_{i,k}$ and t_k.

Composing middlebox requirements. Consider adding the requirement enforced by `hasIDS` that node w must be traversed. It suffices to add the constraint below:

$$\exists k, x_{w,k} \tag{14}$$

To ensure one of multiple IDS nodes in set W is traversed, we could instead add the following constraint:

$$\exists k, w \in W, x_{w,k} \tag{15}$$

To support the requirement that a set of IDS nodes must be traversed but the order of traversal is unimportant, we simply replicate Constraint 14 for each IDS node. Finally, the requirement IDS node w_1 must be traversed prior to w_2 could be expressed as:

$$\exists k_1, k_2, x_{w_1,k_1} \wedge x_{w_2,k_2} \wedge (k_1 < k_2) \tag{16}$$

Overall, this encoding exposes how composition through solving constraints can simplify the task of writing both modular and compositional applications.

V. Richer Composition Scenarios

In this section, we consider how middlebox requirements may be composed with more sophisticated traffic engineering goals than shortest path routing.

Bounding link utilization. A common goal in enterprise networks is to ensure no link is too heavily utilized. To limit the maximum traffic on a certain link $l_{i,j}$, we add a primitive predicate `util(i,j)` representing the link utilization of $l_{i,j}$, and a predicate `demand(`h_a`,`h_b`)` representing the traffic demand from host h_a to host h_b (d_{ab}). U is a bound on link utilization, ranging from 0 and 1. Then, the routing predicate may be combined with the constraints on utilization as follows:

```
demand(h_a,h_b) :- d_ab.
demand(h_a,h_c) :- d_ac. ...
routes_util([X,Y,...]) :-
      route(h_a,h_b,X), route(h_a,h_c,Y), ...
      util(s_1,s_2) < U, util(s_1,s_3) < U, ... .
```

To deploy middleboxes (e.g., IDS) in such a network, we simply change the predicate `route` to `routeIDS` (Sec. III). To implement this, we add the following constraints for the route predicate discussed in Sec. IV:

$$\exists k,\ x_{i,k} \wedge x_{j,k+1} \Rightarrow u_{i,j} = d \tag{17}$$

$$\forall k,\ \neg x_{i,k} \vee \neg x_{j,k+1} \Rightarrow u_{i,j} = 0$$

Here, variable $u_{i,j}$ denotes the total traffic carried by link $l_{i,j}$, and d is given by the `demand` predicate described above. To support multiple pairs of sources and destinations, we use $x_{i,k}^{a\to b}$ and $d^{a\to b}$ to respectively denote the route and traffic demand from host a to host b in Equation 17 above. We then add the following constraint to capture that the utilization of link $l_{i,j}$ (with capacity $c_{i,j}$) is bounded by constant U:[3]

$$\sum_{a,b} u_{i,j}^{a\to b} / c_{i,j} < U$$

Multiple paths. It may be desirable to compute multiple paths between each source and destination that meet the middlebox requirement. This could help both to (i) deal with failures by picking a backup path that was precomputed offline; and (ii) allow for traffic to be split across multiple paths. Given a user defined predicate `distinct_path(X,Y)` that indicates when two paths are sufficiently different (e.g., whenever `X` and `Y` share less than n switches), we can construct a query to obtain two different paths from host h_a to h_b that both satisfy the middlebox requirement as follows:

```
two_routes(h_a,h_b,X,Y) :- routeIDS(h_a,h_b,X),
                           routeIDS(h_a,h_b,Y),
                           distinct_path(X,Y).
```

The strategy can be extended to more routes given a sufficiently sophisticated `distinct_path` predicate.

Soft requirements to aid conflict resolution. An operator may wish to specify soft requirements when all requirements

[3]The quantifier ranges over a finite enumeration.

TABLE I
RUNTIME FINDING THE SHORTEST PATH AND THE SHORTEST PATH TRAVERSING ONE GIVEN CORE SWITCH AS A WAYPOINT ON DIFFERENT K-ARY FAT-TREES

K	No. of nodes	Shortest-path (s)	1-waypoint (s)
4	20	0.08526	0.3298
8	80	2.226	11.94
12	180	40.67	262.6
16	320	285.3	725.2
20	500	2037	3978
24	720	2452	2241

are not simultaneously satisfiable. To that end we add programming annotations in query definitions, whereby a constraint is made soft. In this case, a less constrained requirement is fed iteratively until a path is found. For example consider a request to find routes which have an IDS, while ensuring the utilization of a certain link does not exceed a bound U.

$$\texttt{routeIDS_U}(h_a, h_b, \texttt{X}, \texttt{Y}) \texttt{ :- } \texttt{routeIDS}(h_a, h_b, \texttt{X}), \texttt{@soft util}(s_1, s_2) < \texttt{U}.$$

Upon inability to solve the two constraints above, a second query would be made to the solver by dropping the `util` constraint which is `@soft`. This approach may be extended to define priorities among the constraints, and to relax (e.g., provide a higher utilization bound) rather than completely eliminate a constraint.

VI. Preliminary results

We have conducted initial experiments with our framework to assess its suitability for the offline phase of traffic engineering. It is typical in traffic engineering for an operator to pre-compute an initial set of candidate paths (or tunnels) in the offline phase. An online phase simply selects the appropriate path(s) from the pre-selected subset and determines how traffic must be split across the selected paths [9], [4], [10]. We envison that the offline phase may be augmented to select paths that meet the middlebox requirements.

We implemented the compositional walk-based formulation described in Sec. III using the Microsoft Z3 SMT solver [1]. Our evaluation is based on the Z3 Python API (version 4.4.0), and using different K-ary fat-tree topologies [11]. Table I presents the runtime when finding the shortest path from the left-most node to the right-most node in a fat-tree, with the constraints that the path must traverse given waypoints.

The performance is acceptable for moderate-sized topologies. While the runtime is higher for larger topologies, our implementation is a proof of concept, and there is a lot of room for performance improvement. In particular, to minimize the length of walk we use binary search to determine the smallest M (bound on walk length) for which Equations 7–13 are satisfiable. The running time could benefit with tighter bounds on M, and by exploiting Z3Opt [12], a newly added feature that equips Z3 with optimization primitives (e.g., minimize or maximize). We also discuss other avenues for performance enhancements in Sec. VIII.

VII. Related Work

Merlin [3] presents a language for controlling forwarding packets, including flow bandwidth requirements and middlebox traversal constraints. Policies specified in this language are then mapped by a compiler into specific optimization problems. DEFO [13] translates high-level goals for traffic engineering and service chaining into forwarding decisions in the context of segment routing. These works are not compositional in that the compiler designer must associate every combination of constraints in the original problem with a specific pre-determined optimization formulation. SOL [4] focuses on providing an interface to help users easily capture optimization formulations when developing SDN applications. In contrast to all these works, our focus is on mapping application requirements to solver constraints in a compositional manner.

There has been much interest in better programming models for networks, many based on logic programming (e.g., [14], [15], [16], [17], [18], [19]. The emphasis of our work is not on proposing a new DSL, but rather to show the viability of composing middlebox and routing policies by means of a solver.

Several works [20], [5], [21], [22] have considered different resolution of conflicts between multiple application modules competing for limited shared resources, where concurrent actions taken independently by these modules can violate a global safety invariant though individual application requirements are met. We focus on an orthogonal problem where the goal is to express application requirements as constraints that may be composed together, with infeasible solutions indicating the constraints cannot be simultaneously satisfied.

VIII. Conclusion and Open Issues

In this paper, we have explored how middlebox requirements may be incorporated in traffic engineering and SDN applications in a compositional manner. We have argued that doing so requires composition *prior to* the generation of packet-forwarding policies, in contrast to current approaches that perform composition *after* packet-forwarding policies are generated.

We have presented an initial approach for such composition. While we believe our framework is important and viable, our work is preliminary, and we expect several decisions to evolve. Some specific areas that we hope to investigate in the future include:

Generality. While we have shown the viability of our compositional approach in various common traffic engineering scenarios, considering a wider range of applications and requirements can guide the design of more network primitives. For instance, a tree-based primitive may be added for multicast applications.

Performance. While we believe a compositional approach is viable for offline traffic engineering tasks, more experience is needed with the performance of the approach. Although we have used SMT solvers for concreteness, our framework is more general, and it may be interesting to explore performance

trade-offs with alternative solving engines such as ILP solvers. Use of ILP solvers may also help support a richer set of optimization objectives. If the runtimes of off-the-shelf solvers are found to be unsatisfactory for complex scenarios, we will consider optimized decision procedures to aid the solver, especially for frequently occurring constraint combinations.

Source Language. Our source language is based on first order logic equipped with networking primitives, lists, and inductive relations and definitions. Our current choice for the input language has a Prolog-like syntax, but in the future we may consider a source level syntax more amenable to network operators such as a user defined syntax for relational operators. Further, we currently have included a limited number of domain specific primitives. As more applications are considered in our language we believe this set need to be expanded.

ACKNOWLEDGMENT

We thank the reviewers for their insightful feedback. This work was supported by Google Research Award.

REFERENCES

[1] "Microsoft Z3," https://github.com/Z3Prover/z3.

[2] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proc. of SOSR*, 2015.

[3] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in *Proc. of CoNEXT*, 2014.

[4] V. Heorhiadi, M. K. Reiter, and V. Sekar, "Simplifying software-defined network optimization using sol," in *Proc. of NSDI*, 2016.

[5] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner, "Corybantic: Towards the modular composition of sdn control programs," in *Proc. of HotNets*, 2013.

[6] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proc. of NSDI*, 2015.

[7] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. of NSDI*, 2013.

[8] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. of ICFP*, 2011.

[9] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proc. of ACM SIGCOMM*, 2013.

[10] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang, "R3: Resilient routing reconfiguration," in *Proc. of SIGCOMM*, 2010.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of ACM SIGCOMM*, 2008.

[12] "Microsoft Z3Opt," http://rise4fun.com/z3opt/tutorial/.

[13] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, "A declarative and expressive approach to control forwarding paths in carrier-grade networks," in *Proc. of ACM SIGCOMM*, 2015.

[14] S. Narain, "Network configuration management via model finding." in *Proc. of LISA*, 2005.

[15] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *Proc. of NDSS*, 2013.

[16] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *Proc. of HotSDN*, 2013.

[17] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proc. of WREN*, 2009.

[18] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. of NSDI*, 2014.

[19] N. P. Katta, J. Rexford, and D. Walker, "Logic programming for software-defined networks," in *Workshop on Cross-Model Design and Validation*, 2012.

[20] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A network-state management service," in *Proc. of ACM SIGCOMM*, 2014.

[21] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul, "Democratic resolution of resource conflicts between SDN control programs," in *Proc. of CoNEXT*, 2014.

[22] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of sdns," in *Proc. of ACM SIGCOMM*, 2013.