# A Modular Compiler for Network Programming Languages

Hao Li
Xi'an Jiaotong University

Peng Zhang
Xi'an Jiaotong University

Guangda Sun
Xi'an Jiaotong University

Chengchen Hu
Xilinx Labs Asia Pacific

Danfeng Shan
Xi'an Jiaotong University

Tian Pan
Beijing University of Posts and Telecommunications

Qiang Fu
Victoria University of Wellington

## ABSTRACT

Network programming languages (NPLs) empower operators to program network data planes (NDPs) with unprecedented efficiency. Currently, various NPLs and NDPs coexist and no one can prevail over others in the short future. Such diversity is raising many problems including: (1) programs written with different NPLs can hardly interoperate in the same network, (2) most NPLs are bound to specific NDPs, hindering their independent evolution, and (3) compilation techniques cannot be readily reused, resulting in much wasteful work. These problems are mostly owing to the lack of modularity in the compilers, where the missing part is an intermediate representation (IR) for NPLs. To this end, we propose *Network Transaction Automaton (NTA)*, a highly-expressive and language-independent IR, and show it can express semantics of 7 mainstream NPLs. Then, we design *CODER*, a modular compiler based on NTA, which currently supports 2 NPLs and 3 NDPs. Experiments with real and synthetic network programs show CODER is efficient and scalable.

## CCS CONCEPTS

• **Networks** → **Programming interfaces**.

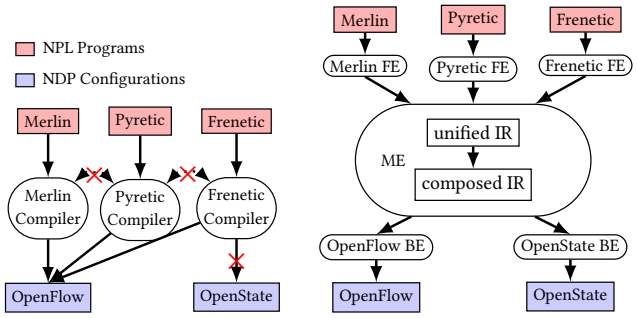
## KEYWORDS

Network Programming Language, Intermediate Representation, Software Defined Networks

**ACM Reference Format:**
Hao Li, Peng Zhang, Guangda Sun, Chengchen Hu, Danfeng Shan, Tian Pan, and Qiang Fu. 2020. A Modular Compiler for Network Programming Languages. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386367.3432063


*CoNEXT '20, December 1–4, 2020, Barcelona, Spain*

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3432063




(a) The monolithic compiler (b) CODER architecture

**Figure 1: CODER modularizes the compilation into three stages: front end (FE), middle end (ME), and back end (BE).**

## 1 INTRODUCTION

With the advance of Software Defined Networking (SDN), many languages (Frenetic [21], Pyretic [32], *etc*) have been proposed for programming computer networks. These languages, which we refer to as *network programming languages (NPLs)*, offer operators with an unprecedented way to program *network data planes (NDPs)*. Different from general-purpose languages shipped with controllers (*e.g.*, Java in Floodlight [6] and Python in Ryu [5]), NPLs provide high-level constructs that can greatly facilitate composing complex functions like path selection, monitoring, QoS, *etc*.

Multiple NPLs and NDPs coexist in modern networks. Recent surveys [30, 43] report more than 15 NPLs including Pyretic [32], Merlin [42], SNAP [10], PGA [35], and more than 10 NDPs including OpenState [15], NetConf [1], P4 [16]. We believe such diversity in both NPLs and NDPs will persist in the short future, due to the following reasons.

*First, each of NPLs and NDPs offer different sets of features.* For example, Merlin can specify a routing path with waypoints [42], while SNAP can realize a stateful end-to-end monitoring function [10]. These two NPLs are designed for fulfilling different management demands in the first place, and cannot be simply replaced with one of them. Another example could be POF [41] and OpenState [15], where the former extends the match fields of OpenFlow and the latter supports stateful operations. These two NDPs also cannot be replaced with each other.

*Second, deploying a unified NPL/NDP can be quite risky and costly.* As currently there is not a "perfect" NPL/NDP that can prevail over others, deploying a unified NPL/NDP is risky: it is very likely we

need to update it very frequently. Moreover, even recent NDPs like P4 [16] claim that they outperform the OpenFlow-related ones in almost all perspectives (programmability, flexibility, forwarding performance, *etc*), the high cost still obstructs their broad deployment in the Internet and data centers. Such cost includes not only the much higher price of the devices, but also the cost for training the operators, and the potential risks of introducing new vulnerabilities and bugs.

The long-term coexistence of multiple NPLs and NDPs means the operators may need to deploy cross-language programs in the single network, or port programs into heterogeneous data planes. Unfortunately, existing NPL compilation systems are monolithic and offer neither of these features. In the following, we first elaborate the problems resulting from the de facto monolithic compilers, then propose our approach with key contributions highlighted.

## 1.1 Problems of Monolithic Compilation

Existing NPL compilers translate programs all the way to a specific NDP, as shown in Figure 1a. Such monolithic approach can raise many problems when handling the coexistence of multiple NPLs and NDPs.

**Cross-language programs cannot interoperate.** Due to the diversity of NPLs, the operator may have to run cross-language programs for having all their merits, which is realized by the *program composition* technique. The correct compositions require to retain complete semantics from all programs, which however is only achievable in single-language programs [10, 12, 32, 35, 44], because none of them can be aware of others' semantics. For cross-language programs, the only possible way is to merge the NDP configurations (*e.g.*, OpenFlow rules) that are compiled individually from their own NPL compilers. However, this cannot be achieved in a safe way due to the rule conflicts. Consider two simple programs, one sets a waypoint $B$, and the other wants to count the packet in an end-to-end way. The individual compilers may interpret these two intents to two paths, $A \rightarrow B \rightarrow D$, and $A \rightarrow C \rightarrow D$, respectively. These two paths raise a rule conflicts in $A$, and cannot be merged or overwritten directly, because $B$ is a waypoint of the first program, and $C$ *could* be the counting switch of the second program.

CoVisor [27] addresses this problem by assuming all the rules are either (1) compatible, *e.g.*, a forwarding rule and a counting rule can naturally operate on the same traffic, or (2) manually prioritized , *e.g.*, a forwarding rule from a firewall program can overwrite another forwarding rule from a routing program. However, most programs would generate the forwarding rules, which can be incompatible for the same traffic. Moreover, even the operators can manually prioritize all the programs, the overwriting operation can only provide limited composition ability, *e.g.*, it cannot generate a possible new solution like $A \rightarrow B \rightarrow C \rightarrow D$. Finally, CoVisor will fail on merging different NDP configurations.

**NPLs and NDPs cannot independently evolve.** As current NPL compilers compile the program all the way down to a specific NDP, it is costly for an NPL compiler to support every NDP, especially a new one. Similarly, NDPs are also evolving for serving complex operations: *e.g.*, fine-grained flow control, stateful operations. However, existing NPLs barely support the newly designed NDPs, because of the out-of-date abstractions they rely on, *e.g.*, many NPLs [21, 39, 40] are built upon the NetCore abstractions [31], which does not support stateful operation. This close binding between NPLs and NDPs greatly hinders their independent evolution.

**Compilation modules cannot be reused.** Since each NPL compiler only concerns its own high-level constructs and semantics, the compilation techniques they employ are not reusable for other NPLs. For example, FatTire focuses on finding the backup rules on the topology, thus a breadth-first searching is used [39]; while SNAP must solve the variable placement, which conducts a jointly decision problem of mixed integer liner program (MILP) [10]. As a result, the NPL designer has to implement the full compilation process.

## 1.2 Our Approach and Contributions

To break the monolith above, we intuitively draw an analogy with the successful PC compiler, which also compiles the programs written in high-level languages (*e.g.*, C) into low-level instructions (*e.g.*, assembly). One critical missing part of the NPL compiler is that PC compilers firstly compile the source code to an intermediate representation (IR), before further translating it to target code. To this end, we propose *network COmpiler Design with intermediatE Representation (CODER)*, which introduces the IR concept into network compiler, and modularizes the compilation into three stages (Figure 1b): a set of *front ends* translate cross-language programs into a unified IR, a *middle end* conducts compositions, and a set of *back ends* translate the IR into various NDP configurations.

By decoupling the NPLs and NDPs, the aforementioned problems can be naturally addressed: (1) the programs are compiled into the IR that retains all intents, which can be composed and compiled into NDP configurations without causing any conflicts; (2) a new NPL only needs to implement a thin front end for supporting all NDPs, and a new NDP can implement a lightweight back end for supporting all NPLs; and (3) IR sets a unified playground of the compilation, so that most compilation techniques can be reused.

Based on this basic idea, we present the challenges of realizing CODER, and state our research contributions.

**Contribution 1: An expressive and unified IR (§3.1).** The key to make the compiler modular is an expressive IR that can fully cover the semantics of NPLs. However, The heterogeneous constructs employed in NPLs make this design difficult. For example, Merlin uses automaton to express the path waypoints but without any stateful semantics [42]; SNAP supports stateful operations using one-big-switch abstraction that has no internal path information [10]. These constructs are fundamentally different for serving various NPL features. Hence, it is not possible to create a proper IR by simply reusing, merging, or extending the existing heterogeneous representations.

CODER introduces *Network Transaction Automaton (NTA)*, a new automaton that can express the semantics of existing (and possibly future) NPLs. The key difference of NTA is that we incorporate network resources and state variables into its transitions. This enables NTA to express not only path constraints, but also resource constraints and stateful operations, in a fine-grained, hop-by-hop way (see §3.2).

**Contribution 2: Compositions without semantics loss (§3.3).** Even having an expressive IR, the composition operations on it

are still undefined. We notice that the conventional techniques *e.g.*, composition of deterministic finite automaton, forwarding diagram [10], policy graph [35], one-big-switch [27, 32], cannot be directly reused for this newly designed representation.

CODER designs a set of composition operators that respect the physical meanings of each element in the transition, so that NTAs can be composed without any semantics loss.

**Contribution 3: Efficient and scalable compilation (§3.4).** Considering the rich semantics the desired IR supports, conducting an optimization problem could be a package solution for compilation. However, creating and solving MILP could be very time-consuming, as the #constraints of MILP would exponentially grow with complexity of intents. What is worse, though many MILP solvers can leverage multiple CPUs [3, 7], they do not guarantee a performance boost, due to the large overhead of synchronization and timing [29, 38].

CODER applies a series of heuristics to greatly reduce the problem scale, and make the solving process highly parallel. We then formulate a much smaller MILP, which can be solved in moderate time even for a large network.

Our case study and evaluation show that NTA can express semantics of 7 mainstream NPLs, and can be compiled into 3 different NDPs (§4); the prototype of CODER is efficient in compilation: 4× faster than the state of the art (§5).

## 2 CODER OVERVIEW

In this section, we will first take a glance at the proposed IR in CODER, and then use a concrete example to walk through the whole compilation process.

### 2.1 A First Look at the IR

Our design of IR is based on the following two observations. First, we observe that the semantics of existing NPLs can be grouped into three classes: (P1) path control with waypoints, *e.g.*, traversing a firewall, (P2) path control with resource constraint, *e.g.*, bandwidth reservation, and (P3) stateful packet manipulation with variables persistent on NDP, *e.g.*, counting all SSH packets at some switch. Second, we observe that operators expect to specify the above semantics in the finest grain, *i.e.*, hop-by-hop. A representation combining above two features could provide an unprecedented expressiveness *e.g.*, traversing the firewall before counting the packets, reserving less bandwidth after traversing a load balancer, which cannot be realized by any existing NPL. This ensures the high compatibility for future NPLs.

With the above observations, we show how to design an IR for NPLs. Firstly, we note that the Deterministic Finite Automaton (DFA) is a good starting point, as it can easily represent the path control semantics of (P1): proceeding a transition corresponds to the action of *forwarding to a next hop*. Secondly, to express the resource constraints of (P2), we add resource consumption actions into DFA transitions: proceeding a transition will *consume the specified resources* at the current switch. Then, resource constraints like reserving an end-to-end bandwidth can be expressed by adding the bandwidth consumption into each DFA transition. Finally, for expressing the stateful operations in (P3), we embed variable operations, *i.e.*, *checking and updating variables*, into DFA transitions, so that the stateful manipulation can be assigned to specific switches. Putting the above together, to distinguish from traditional DFA, we refer to a transition in our new automaton as a *network transaction*, which is an atomic set of operations including forwarding to a next hop, consuming specified resources, and checking and updating variables. Then, we refer to our new automaton equipped with such transitions as *Network Transaction Automaton (NTA)*.

### 2.2 A Walk-through Example

Now we walk through the compilation process of CODER using two real programs written in Merlin and SNAP.

**Using NTA to express programs.** Figure 2a shows a Merlin policy [42], which specifies a waypoint $B$ for packets sent from $\texttt{ip}_1$ to $\texttt{ip}_2$, while consuming 100MB/s bandwidth along the above path. Figure 2c is the corresponding NTA for this policy. Note that this NTA binds to a certain packet class $\texttt{srcip=ip}_1\texttt{\&dstip=ip}_2$, implying the NTA only deals with packets sent from $\texttt{ip}_1$ to $\texttt{ip}_2$. Each transition in the NTA carries a three-tuple, in the order of next hop, resource consumption, and variable operation. This NTA has a start state (node 0) indicating the packet is entering the network, and an end state (node 3) indicating the packet has left the network. The loops on node 1 and 2 along with the transition $1 \rightarrow 2$ realize the waypointing and NTA explicitly uses `e` to denote the destination host in transition $2 \rightarrow 3$. `bw` is a 2-dimensional array indicating the available bandwidth of each link, The consumption $r_1$`: bw[`$h'$`][`$h$`]`$\triangleleft$`100MB/s` means that a bandwidth of 100MB/s is consumed on the link from the current switch $h'$ to the next hop $h$. Since $r_1$ appears in all transitions (except those connect to start and end nodes), this NTA reserves 100MB/s for the end-to-end connection.

Figure 2b shows a SNAP program [10], which counts the first 1000 packets sent from $\texttt{ip}_1$ to $\texttt{ip}_2$ to ensure the two hosts are properly connected. Instead of a single NTA, this program corresponds to a NTA group, as shown in Figure 2d and 2e, which maps to the two network transaction spaces, *i.e.*, counting+routing ($\texttt{g}_1$), and routing only ($\texttt{!g}_1$). Since compiling an NTA will generate *one* transaction sequence, we need to express all possible variable checking results using a group of NTAs. Note that this is an end-to-end counting program, so NTAs should consider all possible locations triggering the counting operation, *i.e.*, in middle of the network (transition $1 \rightarrow 2$), or at the last hop (transition $1 \rightarrow 3$).

**Composing programs at NTA.** Since NTA is language-agnostic and has the complete semantics from the programs, it is a sweet spot to compose cross-language programs without causing conflicts. CODER achieves the composition of two programs by product-intersecting their respective NTAs or NTA groups (see §3.3).

In our case, the composition of the two example programs is a new NTA group consisting of the intersection of Figure 2c and 2d, and the intersection of Figure 2c and 2e. The former is shown in Figure 2f, where `num_pkt` is checked and updated exactly once along the path traversing $B$. The other NTA is much the same with Figure 2f except ($\texttt{g}_1$,$\texttt{u}_1$) is replaced with $\texttt{!g}_1$. The composite semantics can be stated as "forwarding the packet class with 100MB/s bandwidth while traversing $B$, and counting the first 1000 of them".

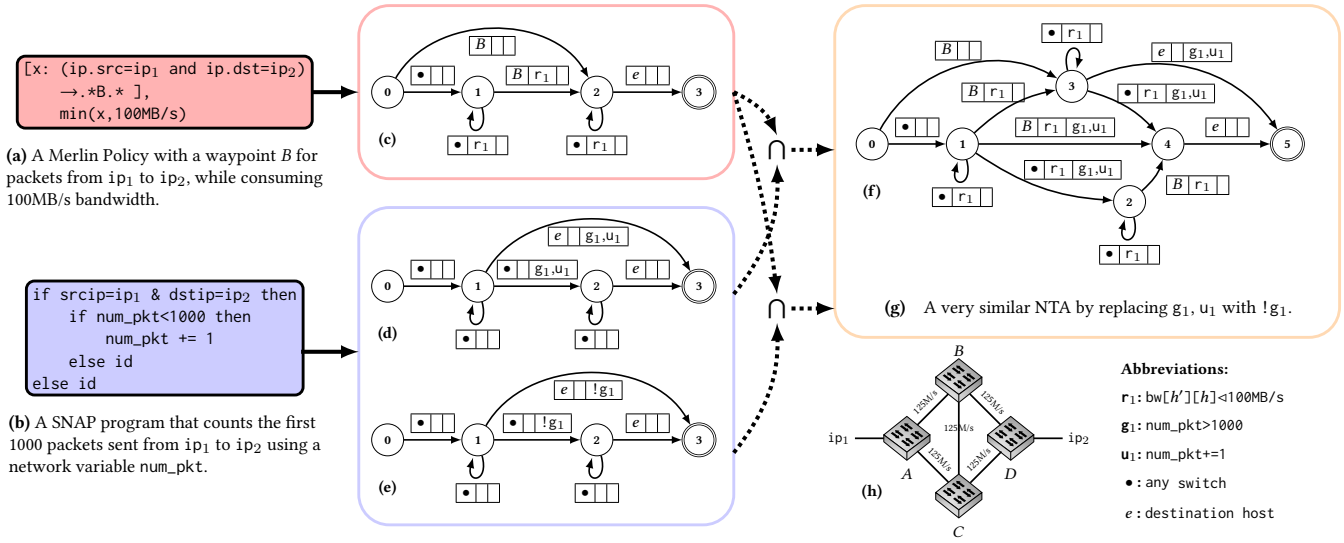



**Figure 2: The compilation process in CODER. (a)–(b): Two programs written in Merlin and SNAP; (c): NTA for Merlin policy; (d)–(e): NTA group for SNAP program; (f)–(g): Composed NTA group; (h): A simple network with 125MB/s bandwidth per link.**

**Compiling NTAs.** CODER compiles the NTAs in two steps: (1) generating a valid transaction sequence, and (2) mapping each transaction in the sequence into NDP configurations.

The first step is challenging: to find a valid transaction sequence, CODER needs to consider three types of constraints: (1) path constraints: the path must traverse the specified waypoints; (2) resource constraints: the resource consumption should not exceed the available resource, *e.g.*, path $A \rightarrow B \rightarrow D$ in Figure 2h is invalid for Figure 2c if another NTA consumes 50MB/s bandwidth on link $(A, B)$; (3) consistency constraints: the same variable should be operated at the same switch in order to avoid synchronization, *e.g.*, Figure 2d and 2e must check and update `num_pkt` at the same switch. Finding a feasible solution for above constraints might take long time with complex NTA and/or large topology. Fortunately, this step is NDP-independent, thus can be reused for all NDPs. In other words, the factual back-end contains only the second step, which is relatively straightforward and can be easily adopted for a new NDP.

To accelerate the first step, CODER applies a series of heuristics instead of directly creating a large MILP, *e.g.*, path selection and clustering. These heuristics can significantly reduce the problem scale, and more importantly, make the problem solvable in parallel.

## 3 DESIGN OF CODER

In this section, we detail the design of CODER. Specifically, we first formally define NTA (§3.1), and show how NTA can express various semantics (§3.2). Next, we present the middle end that composes multiple NTAs (§3.3), and the back end that efficiently enforces NTAs in the NDP (§3.4).

### 3.1 Network Transaction Automaton

**Network transaction.** A network transaction is a three-tuple, $\boxed{h}\boxed{r}\boxed{d}$, where $h$ is the next hop to be forwarded, $r$ is the consumption of the network resources, and $d$ is a stateful operation that first checks the variables against a set of predicates, namely *guard*, and then modifies the variables with a set of operations, namely *update*.

**Network transaction automaton.** A network transaction automaton (NTA) is defined as a 5-tuple $(\Sigma, Q, q_0, a, \mathcal{T})$, where $\Sigma$ is the set of all possible network transactions, $Q$ is the set of NTA nodes, $q_0 \in Q$ is the start node, $a \in Q$ is the end node, and $\mathcal{T}$ is the set of transitions. Each transition $t \in \mathcal{T}$ is a 3-tuple $(q, \sigma, q')$, where $q$ and $q'$ are the NTA nodes, and $\sigma \in \Sigma$ is the network transaction.

As an analog, NTA can be viewed as the "language" for specifying network transaction sequences that comply with the program intent, and the compilation of NTA is to produce one "sentence" under the constraints of network topology, network resources, and variable consistency.

**Elements in NTA.** NTA involves four major elements: the next hop, the resource, the variable, and the packet class.

The *next hop* represents the forwarding target(s). The operator can specify "any switch" with "dot", a specific switch if it is known to her, *e.g.*, switch $B$, or a kind of switches with a mnemonic, *e.g.*, *DPI*, which can be replaced by real switches according to the network configurations in the back end.

The *resource* represents the static constraints of the network, *e.g.*, link bandwidth `bw` in Figure 2c, which are shared by all NTAs. Resources are non-negative, so CODER must respect this nature through the compilation, by considering all consumptions on the same resource. The consumptions are enforced off-line using specific configurations, *e.g.*, meter action in switch for reserving bandwidth. In other words, the on-line processing is irrelevant to the resources.

The *variable* records the network states, which is persistent on the data plane switch [10], *e.g.*, `num_pkt` in Figure 2d and 2e. The guard and update of variables will be translated into matching fields and actions in data plane rules, respectively. There could be different network transaction spaces depending on the results of variable guard, so a group of NTAs will be generated to handle each of them, as shown in Figure 2d and 2e. In contrast of resources,

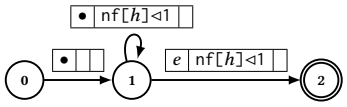



**Figure 3: The NTA that constrains the utilization of switches. `nf` is the resource array of each switch's flow entry capacity.**

CODER does not care about how to check and update the variable, but only where to perform it for consistency concerns.

The *packet class* binding to the NTA is a packet header filter. The filter must specify the source and destination IP addresses, because they determine the concrete entrance and exit in the network. Additionally, the filter must be assigned statically, *i.e.*, it can/will be checked at every switch along the routing path. On the other hand, the header filter that depends on a variable will be treated as a variable guard. For example, if `susp` is variable stored at a certain switch, `srcip=susp` is a variable guard, because the source IP address must be checked at the same switch with `susp`.

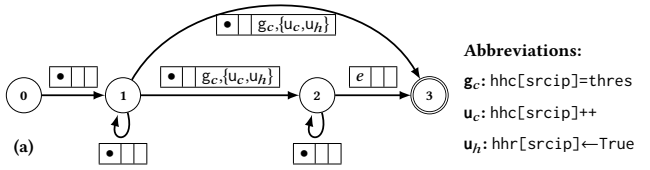
NTAs from the same group specify different transaction spaces for the same packet class, while NTAs from different groups should have orthogonal packet class; otherwise the overlapped part should be composed in the middle end.

## 3.2 Front End: Expressiveness of NTA

Due to the simplicity of NTA, the front end of CODER is quite thin and easy to implement (see §4.2). Instead, the major concern is whether the NTA is expressive enough to cover the NPL semantics. A recent survey classifies those semantics into three catalogs [43]: (1) *traffic engineering* that optimizes the routing paths, *e.g.*, waypointing [36], QoS [42], failure tolerance [39]; (2) *virtualization* that abstracts a much simpler virtual topology for the operators [9, 10]; and (3) *monitoring* that collects the telemetry data, *e.g.*, #packets traversing the network [32, 34]. In the following, we show NTA is capable for expressing these and even more complex semantics. We omit the binding packet class in all following examples.

**Traffic Engineering (TE)** includes waypointing, QoS, and failure tolerance. First, due to its DFA form, NTA naturally supports all path requirements compliant with regular grammar for waypointing. For QoS, NTA can express the constraints of network resources using consumptions, *e.g.*, Merlin's NTA in Figure 2c. For fault tolerance, NTA can tolerate $k$ link failures in two ways: (1) finding $k$ disjointed paths for each packet class by setting a mutex resource of each link; or (2) finding $C_n^k$ paths for each packet class ($n$ is #links), by taking out $k$ links from the topology, *i.e.*, initializing a zero resource for those links (see §4.1). Figure 3 illustrates another NTA for TE: each switch can install rules for at most 100 flows. This semantics can be used to balance the flow table utilization of switches. Specifically, `nf` is a map of resources, which represents the capacity of switches. NTA initializes each element of `nf` to be 100 and consumes it along the path, which constrains the number of installed flows.

**Virtualization (VT)** hides the low-level network details, so that programmers can install the micro-flow rules on a higher-level abstraction. There are typically two kinds of VT: one-to-many and many-to-one. For the first, since NTA nodes also have a one-to-many correspondence to the switches, NTA can directly use `.*` transitions to support such virtualization. For example, node 1 in




**(b)** A similar NTA by replacing ($g_h$,{$u_c$,$u_h$}) with (!$g_h$,$u_c$)

**Figure 4: The NTA group for *heavy-hitter-detection* semantics, which counts #flows sent from a certain IP, and tags it as a heavy hitter if the counter reaches a threshold.**

Figure 2d maps to the one-big-virtual-switch that performs the counting task. For the many-to-one VT, the mappings are explicitly specified by the operators [27], so NTA can leverage the devirtualized result from the native compilers, *i.e.*, network transaction at a certain switch, and maps an NTA node to that switch.

**Monitoring (MT)** records and updates network statistics, *e.g.*, #packets of a matching flow. NTA supports such stateful semantics using network variables. Consider a *heavy-hitter-detection* semantics which identifies the hosts establishing too many flows, this semantics maps to two transaction spaces, and is addressed by the two NTAs in Figure 4: Figure 4a counts ($u_c$) and tags the flow ($u_h$), by assuming the guard of threshold succeeds ($g_c$); Figure 4b handles the negative results (!$g_c$) which counts the flow only. Here, the NTAs bind to all pairwise packet classes with `tcp.flags=SYN`, which can be extracted by a programmable parser [16].

**Complex semantics.** Thanks to the hop-by-hop expressiveness, NTA can represent the combination of above semantics. For example, we could fix where to count the packets in Figure 2d by setting a waypoint. Moreover, we could concatenate Figure 2d with Figure 2c , which produces a new semantics: the first 1000 packets must traverse $B$ (see Figure 5 in §3.3). These semantics, where the stateful operation depends on path, or the path depends on stateful contexts (the value of `num_pkt`), cannot be expressed by either the one-big-switch abstraction [10] or regular expression [42].

## 3.3 Middle End: Modular Compositions

At the middle end, CODER can manipulate NTAs in a language-independent way. We currently focus on *program composition*, one of the most needed features. For simplicity of presentation, in the following we just consider how to compose two NTAs. Composition of two groups of NTAs can be viewed as a product composition of each NTA in the groups. Let the two NTAs be $n_1$ and $n_2$, and CODER aims to offer the following three types of program compositions.

- *Parallel composition (+)* produces an NTA that accepts $n_1$ and $n_2$ simultaneously. This is perhaps the most important type of composition, since it enables cross-language programs to manipulate the same traffic.
- *Sequential composition (>>)* produces an NTA that performs $n_1$ and $n_2$ sequentially. This composition can be used when one program is triggered by another, *e.g.*, *counting* the suspicious flows identified by a *firewall*.
- *Either-or composition (△)* produces an NTA that performs exactly one of $n_1$ and $n_2$. This is useful for load-balancing

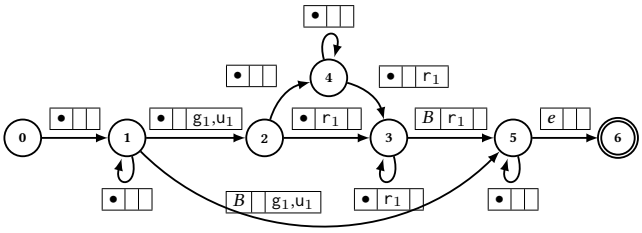



**Figure 5: Sequential composition (Figure 2d>>Figure 2c).**

traffic among identical network functions, *e.g.*, traversing one of multiple *firewalls*.

The theoretical basis of above composition is the operations on automaton, *e.g.*, concatenation and intersection. However, due to the resource and stateful elements carried in NTA transitions, directly applying the conventional operations will result in semantic loss. In the following, we customize three operations of NTA, which can be used to realize the above compositions without semantic loss.

**Intersection.** CODER adopts the Cartesian production for intersecting two NTAs, which realizes the parallel composition. In a nutshell, the node set of the intersected NTA is the product of the nodes in $n_1$ and $n_2$, *i.e.*, $Q_1 \times Q_2$. Next, for the new start node $(q_{1,0}, q_{2,0})$, CODER tries to produce a new transition by *merging* transitions starting from $q_{1,0}$ and $q_{2,0}$, and the process iterates for other nodes, as detailed below.

We say two transitions can be merged, if they carry the same next hop, or at least one of them has a next hop of "dot". The merged transition will carry the same or the non-dot next hop. For the stateful operations, the guard in the merged transition is the intersection of the original guards, *i.e.*, $g_3$=$g_1$&$g_2$, and the update is the union of the original updates, *i.e.*, $u_3$=$u_1$∪$u_2$. For resource consumptions on the different resources, we retain both of them; for those consuming the same resources, we use the largest consumption to overwrite others. For example, consider two NTAs reserving different bandwidth for the same packet class, say 10MB/s and 20MB/s, respectively. The parallel composed NTA should not consume 30MB/s, because 20MB/s bandwidth has already satisfied the semantics of both original NTAs. This principle can be expanded to other resources, *e.g.*, switch flow entries.

**Concatenation.** The sequential composition can be viewed as the concatenation of two NTAs. The major difference between concatenating two NTAs and concatenating DFAs is that the start node and end node in NTA map to the network states that the packets are *not* inside the network. As a result, the concatenated NTA should not include the end node of the left operand and the start node of the right operand.

In detail, for concatenating $n_1$ and $n_2$, CODER removes the end node $a_1$ in $n_1$ and the start node $q_{2,0}$ in $n_2$. Next, for all transitions pointing to $a_1$, CODER replaces $e$ in the next hop field with (•), and product-merges them with the transitions starting from $q_{2,0}$. Since the modified transitions must have a dot next hop, the merging is ensured to be successful.

For example, when concatenating Figure 2d ($n_1$) with Figure 2c ($n_2$), node 3 in $n_1$ and node 0 in $n_2$ will be removed. Transition 1⟶3 in $n_1$ will be modified as [•|g1,u1], and then be merged with transition 0⟶2 in $n_2$, producing a new transition [B|g1,u1] from node 1 in $n_1$ to node 2 in $n_2$. By product merging all the end transitions in $n_1$ with the start transitions in $n_2$, we obtain a concatenated NTA shown in Figure 5. Note that this process may produce a non-deterministic NTA, and we can reduce it using conventional technique.

**Symmetric difference.** This operation maps to the either-or composition, which is quite simple, as we just merge the start nodes and the end nodes of two NTAs, and reduce the composed NTA if necessary.

## 3.4 Back End: Enforce NTA into Data Plane

The back end of CODER translates the set of NTAs into rules that can be installed in the NDP. Typically, such translation can be formulated as the *multi-commodity flow problem* (MCFP), which is to find a proper set of links that can form a complete path for each flow, while satisfying the bandwidth constraints. Many approximation and specialized algorithms are proposed to handle this well-known NP-complete problem [17, 18], which, however, cannot be directly adopted by the back end of CODER. The major reason is that the MCFP must be solved on the graph generated by the *product* of NTA and the topology (see Step 1 in the following), which is usually quite huge. As a result, solving the MCFP, even for a small topology, can take unacceptable time. In this section, we propose a set of heuristics to conquer the complexity of this process, making the back end scalable for the complex NTA and/or large topologies.

Let $N$ be the set of all NTAs, in the following we show how the back end generates the NDP rules in four steps and reacts to the network changes.

**Step 1: Constructing path graph.** This step aims to construct a *path graph* $\mathcal{G}_u$ for each $n_u \in N$, A path graph is a digraph where each path corresponds to a sequence of network transactions that respect the forwarding requirements in $n_u$, *e.g.*, the source/destination and the waypoints. Specifically, the construction takes places in two stages.

*(1) Transforming NTA into NFA.* First, we denote the transition [h|r|d] as $h_{rd}$. In this way, we can obtain a DFA where transitions are triggered only by $h$. Then, we further transform this DFA into a Nondeterministic Finite Automaton (NFA) $\mathcal{M}_u$, by expanding all possible $h$. For example, the left part of Figure 6 shows the NFA for the NTA in Figure 2f, where the • (dot) transitions are expanded to all switches in the physical network. To save space, we introduce some shorthand (*e.g.*, $A_1$ is the shorthand for $A_{r_1}$). Note that transition 0⟶1 and 0⟶3 in Figure 2f are specialized to $A_N$, since in this stage we have known that ip1 connects to $A$.

*(2) Mapping the NFA to physical network.* Let $L$ denote the set of switches in physical network, and $Q_u$ denote the set of nodes in $\mathcal{M}_u$. Then, the set of nodes in $\mathcal{G}_u$ is the Cartesian product $L \times Q_u$, and each node is a pair $(x, y)$ where $x$ is a physical switch and $y$ is a transition in the NFA. Let $T$ be an operator that extracts $h$ from a transition in $\mathcal{M}_u$, *i.e.*, $T(h_{rd}) = h$. Then, there is an edge from $(a, q)$ to $(b, q')$ in $\mathcal{G}_u$ iff: (1) $(T(a), T(b))$ is a link in $L$, and (2) $(q, q')$ is a valid transition of $\mathcal{M}_u$ when processing $b$. It is easy to verify that paths in $\mathcal{G}_u$ are real paths in physical network, which satisfy the waypoint constraints. In addition, $\mathcal{G}_u$ retains the resource and variable information of $n_u$, since each node in $\mathcal{G}_u$ can map back to a transition in $n_u$.

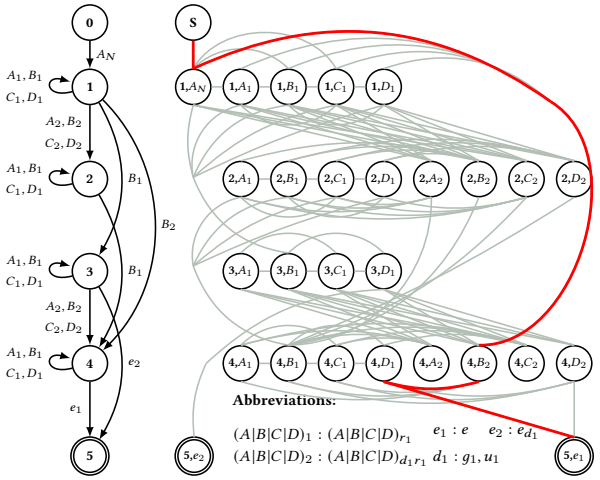



**Figure 6: The path graph constructed by the NTA in Figure 2f and the topology in Figure 2h (directed arrows are omitted). The thick path illustrates a path candidate.**

The right side of Figure 6 is the path graph for the example, and the thick line $S \rightarrow A_N \rightarrow B_2 \rightarrow D_1 \rightarrow e_1$ shows a candidate path: $A \rightarrow B \rightarrow D$ with the counting action performed at $B$.

**Step 2: Reducing problem size in parallel.** As mentioned earlier, having all path graphs, a straightforward method is to conduct an MILP that respectively captures the MCFP for each $\mathcal{G}_u$, with constraints on basic routing, resource invariant, and variable consistency. However, this MILP could be too large to be solved within acceptable time, *e.g.*, over 30 minutes for solving Figure 6 (4 switches). CODER applies a series of heuristics that can fast prune the infeasible solutions, before feeding them to MILP. These heuristics can be easily parallelized, so that can benefit from the multiple CPUs.

*(1) Generating path candidates.* CODER enumerates all simple paths (*i.e.*, no loops) for each path graph in parallel. We can use A* heuristic to further cut off the paths with too many hops that are not usable in practice. Each path graph will select exactly one path to form a feasible solution.

*(2) Analyzing variable consistency.* The dependent variables should be placed at the same switches; otherwise, multiple switches will have to synchronize the values of dependent varibles through the packets or the centralized controller, both of which would incur large overhead in runtime. Instead of directly feeding all paths to MILP and finding a combination that respects the varible consistency, CODER prunes the invalid variable placements on the candidate paths beforehand.

Considering two path graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, one is the for the counting case shown in Figure 6, and the other is the non-counting path graph. $\mathcal{G}_2$ is very similar to $\mathcal{G}_1$ by replacing $(A|B|C|D)_2$ with $(A|B|C|D)_3$, which stands for $(A|B|C|D)_{d_2 r_1}$ ($d_2$ =!$g_1$). We can easily find two path candidates from $\mathcal{G}_1$, $p_1 : A_N \rightarrow B_2 \rightarrow D_1$ and $p_2 : A_N \rightarrow B_1 \rightarrow C_2 \rightarrow D_1$. Similarly, we have two candidates from $\mathcal{G}_2$, $p_3 : A_N \rightarrow B_3 \rightarrow D_1$ and $p_4 : A_N \rightarrow B_1 \rightarrow C_3 \rightarrow D_1$.

Straightforward MILP will put these four paths together, and try to find out a feasible combination. However, it is obvious that both $p_1$ and $p_3$ put the variable in $B$, while $p_2$ and $p_4$ check the variable guard in $C$. In other words, it is no way that $p_1$ and $p_4$ can work out a solution, due to the broken consistency. Hence, CODER clusters these paths by the positions they put the variables, which generates a set of path clusters, each of which ensures the variable consistency. CODER applies a divide-and-conquer strategy for this process, *i.e.*, recursively merging the sub-clusters, which can be easily parallelized. Moreover, CODER can cut off the paths from the same path graph in the cluster; SOL reports that 5 paths for each packet class is sufficient for respecting only resource requirements [25].

**Table 1: Variables involved in MILP.**

| Variable | Description |
|---|---|
| $\mathcal{G}_u$ | path graphs |
| $p_i, p_j$ | paths in a path cluster |
| $Q_{iu}$ | 1 if $p_i$ is from $\mathcal{G}_u$, 0 otherwise |
| $C_k$ | the capacity of network resource $k$ |
| $R_{ik}$ | the consumption on resource $k$ if $p_i$ selected |
| $O_{inma}$ | 1 if $p_i$ matches $m$ (packet class+variable guard) and performs $a$ (forwarding+variable update) at switch $n$, 0 otherwise |
| $H_i$ | 1 if $p_i$ is selected, 0 otherwise |

**Table 2: Constraints of the MILP.**

| **Basic Routing** | **Distinguishable Operations** |
|---|---|
| $\sum_i Q_{iu} H_i = 1$ | $\forall n \in$ all switches. $\forall i, j . \forall m . \forall a .$ |
| **Resource Invariants** | if $H_i = H_j = 1$ |
| $\forall k . \sum_i R_{ik} H_i \leq C_k$ | $O_{inma} = O_{jnma}$ |

We can respectively create one MILP for each path cluster, then solve these MILPs in parallel, and shut the whole process as soon as one of them finds a feasible solution.

**Step 3: Creating and solving MILP.** An MILP for a path cluster takes two inputs: the path cluster $P$, and the resource capacity $C = \{C_1, \ldots, C_k\}$ for $k$ types of resources. It outputs a set of 0–1 variable $H_i$ that indicates whether $p_i \in P$ is selected. With the variables defined in Table 1, we explain the constraints on this problem shown in Table 2.

*(1) Basic routing.* To ensure the connectivity for each packet class, we select exactly one path from each path graph.

*(2) Resource invariant.* Each path $p_i \in P$ can consume a set of the network resources. Since the network resources are immutable and shared by all programs, we can calculate the total impact for $k$ types of resources by accumulating the consumptions along $p_i$, denoted as $R_i = \{R_{i1}, \ldots, R_{ik}\}$. The constraint is to ensure that, after applying $R_i$ for each selected $p_i$, each element in $C$ is non-negative.

*(3) Distinguishable operations.* Considering another path candidate $p_5$ from $\mathcal{G}_2$, $A_N \rightarrow C_1 \rightarrow B_3 \rightarrow D_1$, $p_1$ and $p_5$ comply with the variable consistency (checking `num_pkt` at $B$), but will confuse $A$, as it does not know where to forward the packet ($B$ or $C$). We call it an "indistinguishable case", and eliminate such cases by adding the following constraints: the selected paths must ensure that each switch $n$ takes the same operations $a$ (*i.e.*, forwarding+variable update) for the same matching field $m$ (*i.e.*, packet class+variable guard).

**Step 4: Generating NDP configuration.** The final step in the back end is to enforce the selected paths, *i.e.*, network transaction

sequences, into the NDP. To be specific, two kinds of configurations will be generated:

*(1) Switch rules.* Most NDPs support the MatchAction rule, where Match is the matching fields, mapping to packet class+variable guard, and Action is to manipulate the packets and network-wide variables with forwarding+variable update. All the four elements are easy to obtain from the network transaction sequences.

*(2) Configurations consuming network resources.* We pre-define a set of configuration templates to enforce the resource consumption. For example, for bandwidth, we can define a meter action in the switch, or interpret it into `tc` policies in the hosts. Some resources are naturally consumed, *e.g.*, the flow table entries are consumed by installing the rules.

This step is the only data-plane-dependent process in the back end that should be modified when porting programs to a new NDP. Actually, the modification is quite trivial, since given the concrete routing path and state mappings, this module only needs to handle the syntaxes of the target instruction.

**Reacting to network changes.** A full recompilation for for each network change (a link failure, a new NTA, *etc*) is time-consuming. CODER uses simple heuristics to mitigate such overhead.

First, CODER collects the impacted packet classes from the network changes, *e.g.*, a link failure. Next, CODER re-compiles their NTAs with following heuristics: (1) remove the network resource consumed by the non-impacted NTAs; (2) only solve the path cluster that has the same variable placement with the non-impacted NTAs. If the network resources are relatively sufficient, these heuristics can significantly cut the overhead of obtaining new solution (see §5.2); otherwise, CODER will fully re-compile all the NTAs.

## 4 CODER IN ACTION

In this section, we demonstrate the expressiveness of NTA for 7 mainstream NPLs (§4.1). Then we present some key designs in CODER that make it practical (§4.2). We finally discuss another application, *i.e.*, network verification, upon CODER (§4.3).

### 4.1 Expressiveness for Diverse NPLs

**Real programs from Merlin and SNAP.** We collect 14 programs previously implemented by Merlin [4] and SNAP [2], as listed in Table 3. The front ends of CODER can successfully translate all these programs into NTAs, meaning that NTA can indeed cover the semantics of these two languages.

**Pyretic [32]** generalizes the abstractions from NetCore [31], and offers a *virtual topology* construct. As discussed in §3.2, NTA can express such semantics by using `.*` transitions to connect the nodes that map to the virtual switches.

**FlowLog [34]** offers a SQL-like query syntax to manipulate packets using the states stored in the controller. We successfully express this semantics by mapping the database table/entry used by FlowLog into network variables in NTA, so that such manipulation can take place in the NDP.

**FatTire [39]** introduces a fault-tolerance semantics that can tolerate $k$ link failures. As described in §3.2, there are two realizations for this semantics, and we adopt the first one, *i.e.*, finding $k+1$ disjointed paths. Specifically, CODER places a variable `fail` in the ingress switch, ranging from 0 (primary path) to $k$ ($k$th backup path). Then CODER generates $k+1$ NTAs for each value of `fail`. Each transition of the NTA consumes a mutex resource allocated for the corresponding link. Compiling those NTAs will result in $k+1$ disjointed paths, and the controller can switch to $i$th backup path by setting `fail` to $i$.

**Table 3: Real programs from Merlin (1-4) and SNAP (5-14)**

| | |
|---|---|
| 1 | defense (two sequential firewalls with bandwidth reservation) |
| 2 | iot (security policy for IoT device) |
| 3 | min (reserve minimum bandwidth) |
| 4 | isolation (isolate two flows) |
| 5 | many-ip-domains (count #domain per IP) |
| 6 | many-domain-ips (count #IP per domain) |
| 7 | stateful-firewall (only establish certain connections) |
| 8 | DNS-tunnel-detect (detect DNS tunneling) |
| 9 | ftp-monitoring (count the FTP data traffic) |
| 10 | heavy-hitter-detection (as shown in Figure 4) |
| 11 | selective-packet-dropping (drop B frames in MPEG streams) |
| 12 | sample-small (sample small flow) |
| 13 | sample-medium (sample medium flow) |
| 14 | sample-large (sample large flow) |

**NetKAT [9]** provides a sound and complete set of semantics including *path selection* and *virtual topology*, which have already been covered by NTA. Thus, NetKAT programs can be readily translated into NTA.

**PGA [35]** allows operators to draw the policy graphs for different applications separately, and automatically compose them. Since there are only path constraints, it is easy to translate a policy graph into an NTA.

We summarize the features of above NPLs in Table 4, and find that no NPL supports all features. This possibly confirms the necessity of running cross-language programs in the same network. We also present the corresponding NTA for a snippet of each NPL, and conclude that NTA can express all semantics of those NPLs to aggregate their respective features.

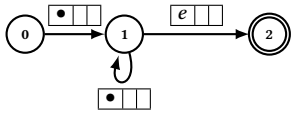
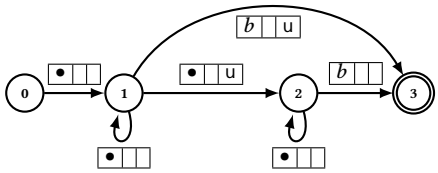
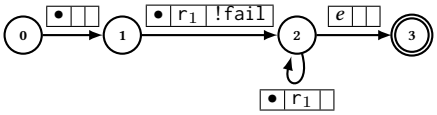
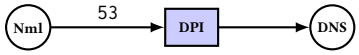
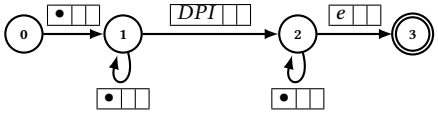
### 4.2 Prototype Implementation

We implement CODER with ~3K lines of code (LOC) in Python/-Cython, including two front ends for Merlin and SNAP, a middle end that supports three types of composition, and two back ends for OpenState [15] and NetASM [33]. We use the Gurobi optimizer [7] to solve the MILP shown in Table 2. We highlight the key implementations in CODER in the following.

**APIs.** CODER poses a set of APIs to manipulate NTA for NPL designers and compiler users. The following defines an NTA with two transitions in Figure 2f, *i.e.*, node 1 loop, transition 1⟶4. It then composes and compiles the NTA.

```
nta, (n0,n1,n2,n3,n4,n5) = NTA.with_states(6)
nta[n1].on(None, None, any_switch, bw*100).to(n1)
       .on(np>1000, np<<np+1, B, bw*100).to(n4)
# parallel composition
new_nta = old_nta + nta
# compile NTA on the given topology and resource
problem = new_nta @ topo @ resource
solution = problem.solve()
rules = solution.gen("OpenFlow")
```

**Table 4: Features, snippets and NTAs for NPLs**

| NPL | *TE* | *VT* | *MT* | *CP* | Snippets | Corresponding NTA |
|---|---|---|---|---|---|---|
| Pyretic | | √ | √ | √ | `(match(dstip='10.0.0.1') » fwd(6))`<br>route traffic with dstip 10.0.0.1 to virtual port 6 |  <br>PC: `srcip=any&dstip=10.0.0.1` |
| Flowlog | | | √ | √ | `ON packet_in(p) WHERE p.nwPort = 23:`<br>`INSERT (p.nwSrc) INTO blacklist;`<br>block sender's IP if its TCP port is 23. |  <br>PC: `srcip=any&dstip=any&port=23`<br>`u`: `blacklist[srcip]←True` `b`: a black hole (drop) |
| FatTire | √ | | | √ | `tpDst = 22 ⇒ [.*] with 1`<br>specify a tolerance level for secure traffic. |  <br>The other NTA is very similar by replacing `!fail` with `fail`<br>PC: `srcip=any&dstip=any&dstport=22`<br>$r_1$: `mutex[srcip][dstip][h']` $\lhd$ `[h]1` |
| NetKAT | √ | √ | | √ | `(if (dstip='10.0.0.1') then pt←6)`<br>route traffic with dstip 10.0.0.1 to virtual port 6 | same with the NTA of Pyretic snippet |
| PGA | √ | | | √ |  <br>route Nml's DNS traffic to DNS traversing DPI |  <br>PC: `srcip=Nml&dstip=any&dstport=53` |
| Merlin | √ | | | | see Figure 2a | see Figure 2c |
| SNAP | | √ | √ | √ | see Figure 2b | see Figure 2d |

**Abbreviations:** *TE*: *traffic engineering*, *VT*: *virtual topology*, *MT*: *monitoring*, *CP*: *composition*, *PC*: *packet class*

**Parallel acceleration.** Our prototype maximally parallelizes the back end, including path graph construction (parallel for all NTAs), path generation (parallel for all path graphs), path clustering (divide and conquer for all paths), and MILP creation and solving (parallel for all path clusters).

**Development efforts.** For supporting a new NPL, we report that (1) the font end for Merlin takes ~50 LOC, while the native compiler of Merlin has ~6K LOC; (2) the front end for SNAP takes ~80 LOC, while the native compiler of SNAP's compiler has ~5K LOC. These results show that with CODER, NPL developers can focus on the design of syntaxes, instead of how to enforce them.

For porting programs into another NDP, one can simply modify the last module of the back end, as the generated transaction sequences can be reused. This process is quite lightweight, *e.g.*, producing OpenState rules takes ~100 LOC.

**Reference design: SNAP's front end.** The SNAP compiler firstly translates its programs into the extended forwarding diagram (xFDD). Thus, it could be a sweet spot to translate xFDD, instead of the original SNAP program, into NTA.

An xFDD is defined as either a branch $(t?d1 : d2)$ ($t$ is a test, $d_1/d_2$ is xFDD), or an action set. In brief, xFDD processes a packet from the root node to a leaf node, denoted as "if $t_1\&...\&t_m$ then $as$", where $t_i$ is the test (or its inverter) along the above path, and $as$ is the action set in the leaf node. We split the xFDD by the tests on packet class, and view the rest tests as a set of guards, and the actions as a set of updates.

Next, we decide the order of triggering the guard and update functions, which is related to the variable dependencies in xFDD [10]. To be specific, xFDD poses three types of dependencies between two variables $s$ and $t$: (1) $(s,t) \in tied$, if $s$ and $t$ must be placed in the same switch, (2) $(s,t) \in deps$, if $s$ must be operated before $t$, and (3) $s$ and $t$ are not dependent, if they are in different strongly connected components in xFDD. The front end handles these dependencies as follows: if $(s,t) \in tied$, the operations on $s$ and $t$

should be carried by the same transition; if $(s, t) \in deps$, the operations on $s$ should precede these operations for $t$; if $s$ and $t$ have no dependency, they should be put into separated NTAs, and be parallel composed afterwards.

Specifically, a sequence of NTA nodes are linearly connected, and each transition between them carries a set of variables that belongs to *tied*, in the order inferred from *deps*. Next, the front end adds a "dot" loop for each node, and finalizes the NTA by appending the start and end node.

**Reference design: OpenState's back end.** OpenState tracks the user-defined states by proposing the eXtended Finite State Machine (XFSM) [15]. For each incoming packet, the OpenState switch will first look up the current variable for this packet in the state table, then trigger an XFSM transition in the XFSM table, and finally update the state table accordingly.

We refer a row in XFSM table as an OpenState rule, which consists of four columns: (C1) a state provided as a user-defined label, (C2) an event expressed as an OpenFlow match, (C3) a list of OpenFlow actions, and (C4) a next-state label. It is straightforward for mapping a network transaction to an OpenState rule: variable guard→C1, packet class→C2, forwarding to next hop→C3, and variable update→C4.

### 4.3 Network Verification upon CODER

In §3.3, we present the most desired transformation in the middle end, *i.e.*, composition. In fact, given the complete semantics maintained in NTA, the middle end can conduct more optimizations and applications. Here we consider the network verification as another typical example.

Traditionally, the network verifier checks the correctness and consistency of and between the control plane (*e.g.*, OSPF and BGP configuration) [13, 20, 23] and data plane (*e.g.*, access control list, forwarding table) [46, 47]. In the context of NPL and NDP, the data plane verification is to check whether the current NDP configuration complies with the high-level intent, and the control plane verification is to check whether the NPL programs can be executed without violating the network invariants. CODER can help realizing both of those verification needs. Specifically, CODER inputs the NPL programs, and tries to compose and compile them according to the network configurations, *e.g.*, topologies and resources. Then, we can verify the following network properties.

**Do NDP configurations comply with the high-level intents?** Given the NDP configurations, the operators want to ensure they are consistent with the high-level intents. Previously this demands the operators to provide a set of unified policy representations for all intents [47], which is a considerable burden. On the other hand, the NTA in CODER naturally reveals the high-level intent, and can be easily verified against the data plane configurations. Recall that NTA is the "language" of the legal sequences of network transactions. As a result, the verification process is to transform the NDP configurations into a sequence of network transactions, and then match it with the NTA. If the sequence is accepted, then the configurations comply with the high-level intent.

**Do NPL programs violate the network invariants?** After writing the NPL programs, the operators may be concerned by the correctness of the programs, *i.e.*, will they violate the network invariants like reachability? This concern is unnecessary when there is only one program, as most NPLs do not allow the operators to specify an intent with a loop or blackhole. However, when there are multiple programs, potentially written in different NPLs, the conflicts may lead to a violation of the network invariant. With CODER, the operators can verify the invariants by simply compiling the composite NTAs of all programs. And if CODER can find a feasible solution, it means that the programs are compatible and the network invariants can be assured.

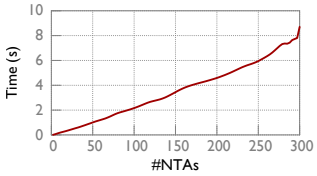



**Figure 7: The time cost of the parallel composition vs. #composed NTAs.**

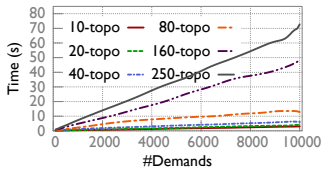



**Figure 8: The compilation time vs. different topologies and #demands (`dns-defense`).**

## 5 PERFORMANCE EVALUATION

This section evaluates the performance of CODER. For front ends, we observe the translation for each program in Table 3 finishes within 100ms, which is fast enough for common usage. As such, we only test the performance of the middle and back end, *i.e.*, composing NTAs and producing transaction sequences. Experiments are performed on a machine with dual 20-core Intel Xeon 2.2GHz CPUs and 192GB memory.

### 5.1 Middle End

Since there are only a small number of real programs available to us, we synthesize more NTAs to test the composition performance in the middle end. We observe that the NTAs for real programs are relatively small: for programs in Table 3, each NTA has ~4 nodes and ~7 transitions on average. Thus, the number of nodes and transitions in our synthesized NTAs are set to 4–10 and 5–15, respectively. Since sequential and either-or compositions only operate the start and end nodes, their running time is independent of NTA sizes, and very fast (within 1ms for two large NTAs). Figure 7 only reports the running time for parallel composition of NTAs, with the number of NTAs varying from 2 to 300.

### 5.2 Back End

When evaluating the back end, we are interested in answering the following two questions: (1) will the running time scale with the network size, #packet classes, and NTA complexity? (2) can CODER efficiently react to network changes?

**Settings.** Our experiments use the topologies from SNAP [2], as shown in the left part of Table 5. Here, "demand" refers to the end-to-end bandwidth requirement for a packet class. Therefore, in the following we will use #demands and #packet classes interchangeably. For NTA, we use `dns-defense`, the parallel composition of `defense` and `DNS-tunnel-detect` in Table 3, which specifies two random waypoints, consumptions on one resource, and three variables. We apply `dns-defense` to 10% of all packet classes in each

**Table 5: Compiling `dns-defense` on collected topologies**

| topo. | #sw | #edges | #demands | P1 | P2 | P3 | P4 | total |
|---|---|---|---|---|---|---|---|---|
| Stanford | 26 | 92 | 20736 | 0.44s | 0.40s | 0.73s | 24.82s | 26.39s |
| Berkeley | 25 | 96 | 34225 | 3.37s | 1.38s | 1.57s | 36.13s | 42.45s |
| Purdue | 98 | 232 | 24336 | 1.54s | 0.33s | 1.36s | 37.79s | 41.02s |
| AS 1755 | 87 | 322 | 3600 | 0.59s | 0.54s | 1.10s | 8.96s | 11.19s |
| AS 1221 | 104 | 302 | 5184 | 0.79s | 1.19s | 1.17s | 11.95s | 15.10s |
| AS 6461 | 138 | 744 | 9216 | 2.79s | 27.34s | 8.79s | 26.07s | 64.89s |
| AS 3257 | 161 | 656 | 12544 | 4.03s | 21.70s | 12.24s | 46.82s | 84.79s |

***P1:*** *path graphs construction,* ***P2:*** *path candidates generation,* ***P3:*** *path clustering,* ***P4:*** *MILP creation and solving*

topology, and just assure the basic routing and the bandwidth reservation for other packet classes. Applying the policy to more packet classes is possible while it is uncommon to route all packet classes through a single switch (for variable consistency), and there may not be a solution. By using 10% of packet classes, we can guarantee that there is at least one solution for the selected waypoints and packet classes. We apply an adaptive threshold in the path generation process, by only collecting paths with no more than $stp+3$ hops for each path graph, where $stp$ is the length of shortest path. In each path cluster, we randomly select 10 paths from the same path graph to reduce the MILP size (we used experiments to validate that 10 paths suffice for obtaining a solution, and a similar phenomenon is reported in SOL [25]).

**Breakdown of compilation time.** Table 5 reports a breakdown of compilation time of `dns-defense`. where we can see the total compilation time is within 100s for all topologies. As a comparison, SNAP consumes 380s for compiling `dns-tunnel-detect` (*i.e.*, `dns-defense` without waypoints) on AS 3257. In addition, due to the parallelism in the back end, the speed of each phase grows linearly with the #cores, which is not possible for pure MILP approaches [10, 42]. CODER will get faster if more packet classes are involved in the stateful policies. This is because more stateful packet classes will lead to more and much smaller path sets, which actually accelerates the problem solving.

**The impact of topology size and #packet classes.** The running time of back end will be impacted by the #packet classes which determines the #path graphs, and the topology size which determines the size of each path graph. We synthesize 5 topologies with 10–250 switches using IGen [37], and randomly generate 100–10K packet classes. Figure 8 shows the compilation time, where we can see the time grows linearly with the #packet classes but exponentially with the topology size. Again we argue that the parallel feature in CODER's back end can largely tame the overhead explosion; deploying a 10-server cluster can bring a reduction of one magnitude of the compilation time. Besides, we can apply other heuristic parameter to control the problem size, *e.g.*, the cutoff threshold in path generation and clustering. Moreover, the full-compilation only happens in bootstrap stage, and we can apply the proposed heuristics when handling continuous network changes (see the last experiment).

**The impact of NTA complexity.** As the real programs in Table 3 are relatively small, we synthesize a large NTA by either-or composing `dns-defense` with itself. Since we deliberately choose not

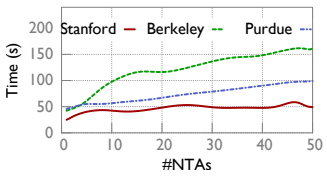



**Figure 9: The compilation time vs. #composed NTAs on campus topologies.**

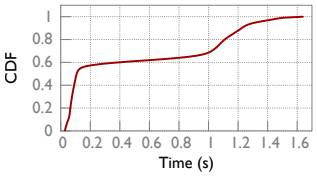



**Figure 10: CDF of reacting to a link failure (`dns-defense` on IGen 80-switch topology).**

to merge equivalent nodes, the size of path graph grows exponentially. Figure 9 shows a quite stable compilation time for the campus topologies. The reason is that in each path cluster, we cut off the paths from the same path graph to a fixed number of 10, so the MILP size is stable no matter how complex the path graph is. Note that in practice, compositions will not significantly enlarge the path graph since equivalent nodes can be merged. As a reference, the path graph created by parallel composing all programs in Table 3 on Stanford topology has 8K nodes and 42K edges, while the 50-either-or-composed `dns-defense` produces a path graph with 12K nodes and 60K edges on Stanford topology, which can be solved within 50s. This means CODER can compile NTAs of real programs in moderate time.

**Reacting to network changes.** In cases of network updates like link failures, CODER only needs to re-compile the NTA for impacted packet classes. Thus, the recompilation overhead depends only on #impacted packet classes. Here, we use the NTA of `dns-defense` on the IGen 80-switch topology with 10K demands, and after computing a solution, we randomly fail one of the links. We observe that one link failure can impact 93 packet classes on average, and Figure 10 reports that CODER can re-compile the NTA within 1.6s for all cases. This means that CODER can fast react to network changes with incremental compilation.

## 6 LIMITATIONS AND DISCUSSION

**Completeness and correctness** NTA aims to be full-expressive for all NPLs, which is extremely difficult, if not impossible, to achieve. For example, currently NTA cannot express a few complex semantics, *e.g.*, multi-casting, bandwidth negotiation [42], latency minimization [24]. We would explore the enhancement of NTA's expressiveness in our future work, and here we discuss one typical semantics, multi-casting, in detail. Currently we restrict NTA to be deterministic, *i.e.*, the packet can only be processed by one switch at the same time. As a result, NTA cannot express multi-casting actions, since they would activate multiple transitions for the same packet. Surely we can transform the non-deterministic NTA into a deterministic one, but such process would break the mapping between NTA nodes and the physical switches.

On the other hand, the correctness of the compilation process in CODER can be easily verified, as the composition handles the physical meanings of the NTA transitions, and the back-end compilation is well described with an MILP.

**The optimality of compilation.** The different compilation techniques used in the compiler may impact the optimality of the output solution, *e.g.*, the average number of hops for each flow, the total number of data plane rules, the usable bandwidth of each link. In

general, the heuristics used in CODER narrow the solution space for better compilation speed, thus might lower the optimality of the final output. However, we argue that if those metrics are critical to the operators, it could be easy to add them into the constraints/objectives of the MILP, which will ensure the generated solution can meet the requirements. In sum, this is a trade-off between the compilation speed and optimality of the solution, and CODER actually offers such flexibility to the developers and operators.

**Per-packet stateful processing.** The variable NTA employs is persistent at the NDP switch, while there exists the per-packet variable that traverses the network with the packet, *e.g.*, FlowTags [19]. NTA can support such variables by loosing the constraint of placing them; only variables for the same packet class should be placed at the same switch.

**Too many variable guards.** The #guard results determine #NTAs in an NTA group. Due to the product-composition effect, the #NTAs could quickly grow to an unacceptable number if too many variable guards are involved. However, we note that multiple NTAs are necessary, only if the routing paths or resource consumptions vary to the guard results; otherwise, we need not create the second NTA, but just reuse the variable placement in the first one. For example, Figure 2d and 2e correspond to the same routing path, so we can just create and compile the former one, and place $!g_1$ at the same switch with $g_1$. Currently, most stateful semantics are in an end-to-end way, which means they will not create extra NTAs. In this way, we believe the risk of explosive #NTAs is minor.

## 7 RELATED WORK

**NPLs** use formal representations to carry their semantics. NetKAT [9, 22] is the first to use regular expressions for describing the end-to-end network behaviors, which is based on Kleene algebra with Tests (KAT) and make the network semantics sound and complete. Merlin [42] also express the path requirements with regular grammar, and extends them with bandwidth requirements. Kinetic [28] uses Finite State Machine to capture the dynamic semantics of network. SNAP [10], based on the one-big-switch model, uses the xFDD to express the stateful end-to-end network behaviors. While these representations are expressive for specific programming purposes, the proposed NTA draws most of their merits, making it suitable for serving as the IR for NPLs.

Specifically, NTA can fully cover a broad range of semantics in previous NPLs, including forwarding the packets [21, 32], reserving the resource [39, 42], and stateful operations [10]. Apart from the above static semantics, there also exists the dynamic semantics that negotiate the packet behaviors and/or network resource in the runtime, *e.g.*, bandwidth adjusting [42], latency minimization [26]. Currently, NTA cannot express those dynamic semantics (§6).

**Program interoperation**, *i.e.*, running cross-language programs in a single network, is a highly desired feature, which previously is only achievable for the programs written in the same NPL [10, 12, 32, 35, 44]. CoVisor tries to compose the NDP rules compiled from programs written with different NPLs [27]. However, due to semantics loss in NDP rules, it can only merge the actions from compatible rules (*e.g.*, a forwarding rule and a counting rule), or overwrite others with the highest-priority rule. Unlike the above approaches, CODER composes the programs at the IR level. Since NTAs retain the original intents of the programs, the composition will not lose any of the semantics. There are approaches to compose the dynamic semantics [11, 24], and CODER currently does not support this feature as NTA cannot express those semantics.

**Extended automata** are proposed for different purposes, *e.g.*, timed automaton for modeling real time systems [14], and cost automaton for advanced routing [8]. The proposed NTA extends DFA into from another perspective, *i.e.*, mapping the nodes and transitions into switches and actions, which makes it expressive for networking scenarios.

**Target-independent compilation** in previous works mainly focuses on NDP [16, 33, 45], which starts from the low level (*e.g.*, OpenFlow rule) to produce an even lower-level representation (*e.g.*, TCAM entry), while CODER aims to set a middle representation between the high level (NPL) and low level (NDP). SOL [25] proposes a set of APIs to realize varied optimization objectives to unify the compilation process from the language level, which does not support other NPLs, making it actually another NPL. In contrast, the front ends in CODER are lightweight, so diverse NPLs can be easily translated into the unified NTAs, where the compilation modules are reused.

## 8 CONCLUSION

This paper motivated the need to modularize the compiler of network programming languages with Intermediate Representation (IR). We proposed such an IR based on NTA, and designed a modular compiler named CODER. We prototyped CODER and evaluated it with real and synthetic programs on various networks. The results showed that CODER is both efficient and scalable in compiling network programs. We note that although CODER might not be the silver bullet for serving all the NPLs and NDPs, there still exists lots of value in looking for a proper IR to modularize the compiler, where CODER with NTA can be seen as the first step.

## ACKNOWLEDGMENTS

We thank the anonymous CoNEXT reviewers and shepherd for their valuable feedback. This work is supported by NSFC (No. 61702407, 61772412, 61902307, 61672425 and 61702049), and Fundamental Research Funds for the Central Universities. Peng Zhang is the corresponding author.

## REFERENCES

[1] 2011. RFC 6241: Network Configuration Protocol. https://tools.ietf.org/html/rfc6241/.
[2] 2016. SNAP - Stateful Network-Wide Abstractions for Packet Processing. https://bit.ly/2GF472Y.
[3] 2017. CPLEX Optimizer. https://ibm.co/2N2v6s6.
[4] 2017. Merlin. https://bit.ly/2TlaDOk.
[5] 2017. Ryu OpenFlow Controller. https://bit.ly/2TedVCF.
[6] 2018. Floodlight OpenFlow Controller. https://bit.ly/2Riemyh.
[7] 2018. Gurobi optimizer. https://bit.ly/1Kc3Hxc.
[8] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. 2013. Regular functions and cost register automata. In *ACM/IEEE LICS*.
[9] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices* (2014).
[10] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *ACM SIGCOMM*.

[11] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C. Mogul. 2014. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In *ACM CoNEXT*.

[12] A. Bairley and G. G. Xie. 2016. Orchestrating network control functions via comprehensive trade-off exploration. In *IEEE NFV-SDN*.

[13] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*.

[14] Johan Bengtsson and Wang Yi. 2004. Timed automata: Semantics, algorithms and tools. In *Lectures on concurrency and petri nets*. Springer, 87–124.

[15] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. 2014. OpenState: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 44–51.

[16] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[17] Andrei Z Broder, Alan M Frieze, and Eli Upfal. 1999. Static and dynamic path selection on expander graphs: a random walk approach. *Random Structures & Algorithms* 14, 1 (1999), 87–109.

[18] Amit Chakrabarti, Chandra Chekuri, Anupam Gupta, and Amit Kumar. 2007. Approximation algorithms for the unsplittable flow problem. *Algorithmica* 47, 1 (2007), 53–78.

[19] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2013. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *ACM SIGCOMM*.

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *USENIX NSDI*.

[21] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *ACM SIGPLAN ICFP*.

[22] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A coalgebraic decision procedure for NetKAT. In *ACM POPL*.

[23] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *ACM SIGCOMM*.

[24] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. 2018. Intent-driven Composition of Resource-management SDN Applications. In *ACM CoNEXT*.

[25] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. 2016. Simplifying software-defined network optimization using SOL. In *USENIX NSDI*.

[26] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *USENIX NSDI*.

[27] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX NSDI*.

[28] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable Dynamic Network Control. In *USENIX NSDI*.

[29] Thorsten Koch, Ted Ralphs, and Yuji Shinano. 2012. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research* 76, 1 (2012), 67–93.

[30] Zohaib Latif, Kashif Sharif, Fan Li, Md Monjurul Karim, and Yu Wang. 2019. A Comprehensive Survey of Interface Protocols for Software Defined Networks. arXiv:1902.07913 [quant-ph]

[31] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A Compiler and Run-time System for Network Programming Languages. In *ACM POPL*.

[32] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. 2013. Composing Software Defined Networks. In *USENIX NSDI*.

[33] Shahbaz Muhammad and Feamster Nick. 2015. The Case for an Intermediate Representation for Programmable Data Planes. In *ACM SOSR*.

[34] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-defined Networks. In *USENIX NSDI*.

[35] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *ACM SIGCOMM*.

[36] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *ACM SIGCOMM*.

[37] B. Quoitin, V. Van den Schrieck, P. Francois, and O. Bonaventure. 2009. IGen: Generation of router-level Internet topologies through network design heuristics. In *International Teletraffic Congress*.

[38] Ted Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. 2018. *Parallel Solvers for Mixed Integer Linear Optimization*. Springer International Publishing, 283–336.

[39] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *ACM SIGCOMM HotSDN*.

[40] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. In *ACM SIGCOMM*.

[41] Haoyu Song. 2013. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *ACM SIGCOMM HotSDN*.

[42] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *ACM CoNEXT*.

[43] C. Trois, M. D. Didonet Del Fabro, L. C. E. de Bona, and M. Martinello. 2016. A Survey on SDN Programming Languages: Towards a Taxonomy. *IEEE Communications Surveys Tutorials* 18, 4 (2016), 2687–2712.

[44] Wen Wang, Wenbo He, and Jinshu Su. 2016. Redactor: Reconcile network control with declarative control programs In SDN. In *IEEE ICNP*.

[45] Minlan Yu, Andreas Wundsam, and Muruganantham Raju. 2014. NOSIX: A lightweight portability layer for the SDN OS. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 28–35.

[46] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *USENIX NSDI*.

[47] Wenxuan Zhou, Jason Croft, Bingzhe Liu, Elaine Ang, and Matthew Caesar. 2018. Automatically Correcting Networks with NEAt. In *USENIX NSDI*.