

Compiling Cross-Language Network Programs Into Hybrid Data Plane

Hao Li¹, Peng Zhang¹, Guangda Sun, Wanyue Cao, Chengchen Hu, *Member, IEEE*,
Danfeng Shan¹, *Member, IEEE*, Tian Pan¹, and Qiang Fu

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 programs show CODER can correctly compile those programs for real networks within moderate time.

Index Terms—Network programming language, intermediate representation, software defined networks, hybrid data plane.

I. INTRODUCTION

WITH the advance of Software Defined Networking (SDN), many languages (Frenetic [2], Pyretic [3], *etc*) have been proposed for programming computer networks. While the *network data plane (NDP)* protocols like OpenFlow and P4 program the *single switch*, these languages, referred as *network programming languages (NPLs)* aim to compose the *network-wide* behaviors, which offer operators

Manuscript received December 17, 2020; revised August 14, 2021 and November 23, 2021; accepted November 23, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor F. Dressler. This work was supported in part by the National Natural Science Foundation of China under Grant 62172323 and Grant 61902307 and in part by the Fundamental Research Funds for the Central Universities. The preliminary version of this paper is published in [1] [DOI: 10.1145/3386367.3432063]. (*Corresponding author: Peng Zhang.*)

Hao Li, Peng Zhang, Wanyue Cao, and Danfeng Shan are with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: hao.li@xjtu.edu.cn; p-zhang@xjtu.edu.cn; wanyuecao@stu.xjtu.edu.cn; dfsan@xjtu.edu.cn).

Guangda Sun was with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China. He is now with the Department of Computer Science, National University of Singapore, Singapore 119077 (e-mail: sung@comp.nus.edu.sg).

Chengchen Hu is with NIO Inc., Shanghai 201804, China (e-mail: huc@ieee.org).

Tian Pan is with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: pan@bupt.edu.cn).

Qiang Fu is with the School of Computing Technologies, RMIT University, Melbourne, VIC 3000, Australia (e-mail: qiang.fu@rmit.edu.au).

Digital Object Identifier 10.1109/TNET.2021.3132303

with an unprecedented way to program NDPs. Different from general-purpose languages shipped with controllers (*e.g.*, Java in Floodlight [4] 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 [6], [7] report more than 15 NPLs including Pyretic [3], Merlin [8], SNAP [9], PGA [10], and more than 10 NDPs including OpenState [11], NetConf [12], P4 [13]. 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 [8], while SNAP can realize a stateful monitoring function [9]. These two NPLs are designed for different management tasks in the first place, and cannot be simply replaced with one of them. Another example could be POF [14] and OpenState [11], 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 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 [13] 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, port programs into different data planes, or even more complex – run cross-language programs on a hybrid data plane that consists of heterogeneous devices. 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.

A. Problems of Monolithic Compilation

Existing NPL compilers translate programs all the way to a specific NDP, as shown in Fig. 1a. Such monolithic approach

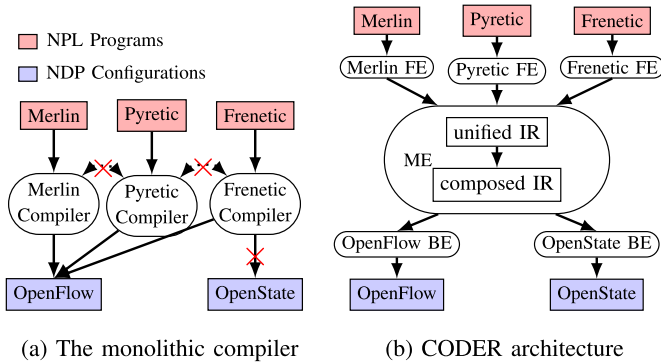


Fig. 1. CODER modularizes the compilation into three stages: front end (FE), middle end (ME), and back end (BE).

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 [3], [9], [10], [15], [16], 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. Their 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.

CoVisor [17] 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 [2], [18], [19] are built upon the NetCore abstractions [20], 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 [18]; while SNAP must solve the variable placement, which conducts a jointly decision problem of mixed integer liner program (MILP) [9]. As a result, the NPL designer has to implement the full compilation process.

Hybrid data plane cannot be managed uniformly. Similarly to the reason that cross-language programs could be deployed for drawing all of their merits, the network operators might deploy data plane devices that following different NDP standards in a single network [21]. This is quite reasonable, as the operators would incrementally deploy new devices in their networks, and expect they can work seamlessly with the existing ones, especially from the perspectives of management. However, since current NPL compilers only support a single specific NDP, none of NPLs can program a hybrid data plane.

B. 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 1): 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 existing compilation techniques can be reused, and new techniques like hybrid-data-plane support can be developed based on the unified IR.

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

Contribution 1: An expressive and unified IR (Section III-A). 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 [8]; SNAP supports stateful operations using one-big-switch abstraction that has no internal path information [9]. 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 Section III-B).

Contribution 2: Compositions without semantics loss (Section III-C). 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 [9], policy graph [10], one-big-switch [3], [17], 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: Practical and feasible compilation (Section III-D). A mature solution for compiling the rich semantics supported by the desired IR is to conduct an optimization problem. However, creating and solving MILP could be very time-consuming, as the number of constraints of MILP would exponentially grow with complexity of intents. What is worse, though many MILP solvers can leverage multiple CPUs [22], [23], they do not guarantee a performance boost, due to the large overhead of synchronization and timing [24], [25].

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. Note that mapping the complex intents into the large networks is an extremely difficult problem, and CODER's goal is not to fundamentally conquer such problem, instead, its focus is to make the compilation much more feasible (*i.e.*, can finish within moderate time) for real intents in real topology.

Contribution 4: Unified compilation for hybrid data plane (Section III-E). The major challenge for programming the hybrid data plane is the lack of the semantics of different NDPs, *i.e.*, what capabilities do they enable for processing the packets. For example, OpenFlow can read limited fields of packets, while P4 can extract arbitrary fields; the compiler has to know such capability difference to properly decompose the high-level intent into heterogeneous devices.

We propose a simple abstraction to describe the capability of different NDP standards. CODER then modifies its back end based on the abstraction, and compiles the IR code into the hybrid data plane, such that each device is capable for performing the actions assigned to it.

Our case study and evaluations show that NTA can express semantics of 7 mainstream NPLs, and can be compiled into 3 different NDPs (Section IV); and the compilation of CODER is 4× faster than the state of the art (Section V).

II. 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.

A. 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)*.

B. 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. Fig. 2a shows a Merlin policy [8], which specifies a waypoint *B* for packets sent from ip_1 to ip_2 , while consuming 100MB/s bandwidth along the above path. Fig. 2c is the corresponding NTA for this policy. Note that this NTA binds to a certain packet class $srcip=ip_1&dstip=ip_2$, implying the NTA only deals with packets sent from ip_1 to 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

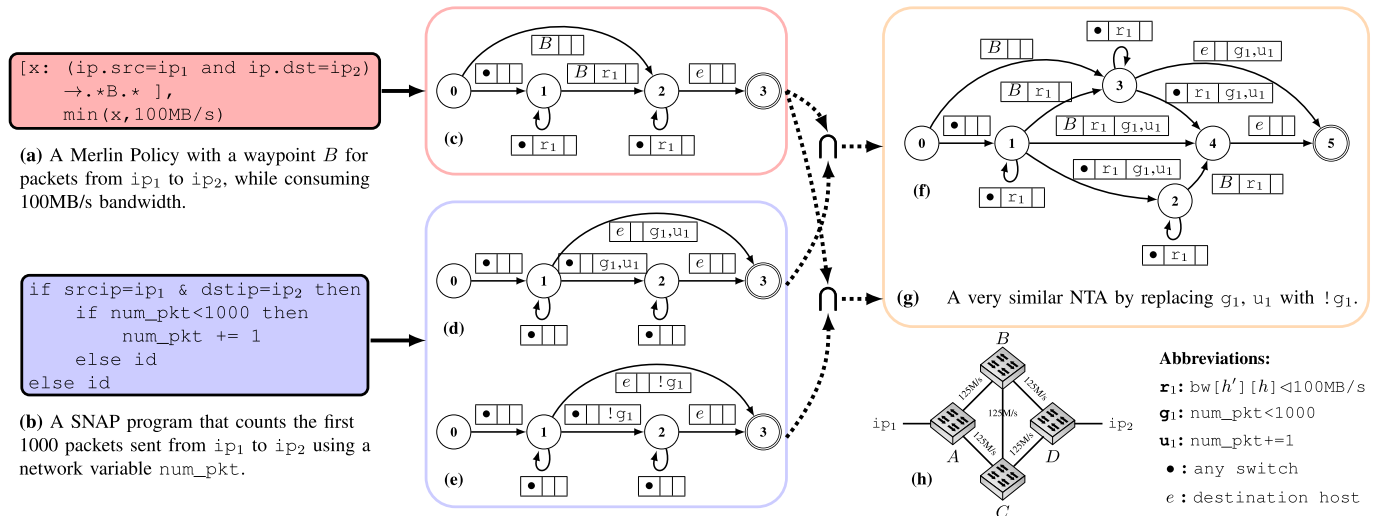


Fig. 2. The compilation process in CODER. (a)–(b): Two programs written in Merlin and SNAP; (c): NTA for Merlin policy; (d)–(e): NTA group for SNAP program; (f)–(g): Composed NTA group; (h): A network with 125MB/s bandwidth per link.

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 : $bw[h'] [h] < 100MB/s$ 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.

Fig. 2b shows a SNAP program [9], which counts the first 1000 packets sent from ip_1 to 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 Fig. 2d and 2e, which maps to the two network transaction spaces *i.e.*, counting+routing (g_1), and routing only ($!g_1$). 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 Section III-C).

In our case, the composition of the two example programs is a new NTA group consisting of the intersection of Fig. 2c and 2d, and the intersection of Fig. 2c and 2e. The former is shown in Fig. 2f, where `num_pkt` is checked and updated exactly once along the path traversing B . The other NTA is much the same with Fig. 2f except (g_1, u_1) is replaced with $!g_1$. 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”.

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 Fig. 2h is invalid for Fig. 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.*, Fig. 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 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 lightweight 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.

Supporting hybrid data plane. To enable interoperability in the hybrid data plane, a capability abstraction of different NDP standards is a necessity, which should describe the processing capabilities of each kind of NDP device, including which kind of packets can be processed (identified) by the devices, which sets of actions can be performed on the packets, *etc.*

Given the topology and the device capabilities, CODER applies more heuristics and constraints in its back end to generate a proper solution, which ensures that (1) all operations are assigned to capable devices *e.g.*, the counting operations can be put on either OpenFlow or P4 switches, while the stateful operations can only be carried by the latter; and (2) the device capabilities are maximally leveraged, *e.g.*, even OpenFlow switches may only identify partial fields of an arbitrary-defined

flow, we can still leverage their forwarding capabilities to meet the high-level intents (see Section III-E).

III. DESIGN OF CODER

In this section, we detail the design of CODER. Specifically, we first formally define NTA (Section III-A), and show how NTA can express various semantics (Section III-B). Next, we present the middle end that composes multiple NTAs (Section III-C), and the back end that efficiently enforces NTAs in the NDP (Section III-D). We finally discuss the capability framework for different NDPs, and adopt the back end to support the hybrid data plane (Section III-E).

A. Network Transaction Automaton

Network transaction. A network transaction is a three-tuple, $[h|r|d]$, 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 Σ 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, σ, 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 [9], *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, CODER does not care about how to check and update the variable, but only where to perform it for consistency concerns.

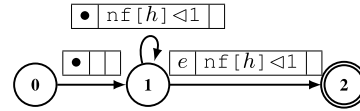


Fig. 3. The NTA that constrains the utilization of switches. nf is the resource array of each switch’s flow entry capacity.

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.

B. Front End: Expressiveness of NTA

Due to the simplicity of NTA, the front end of CODER is quite thin and easy to implement (see Section IV-B). 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 [6]: (1) *traffic engineering* that optimizes the routing paths, *e.g.*, waypointing [26], QoS [8], failure tolerance [18]; (2) *virtualization* that abstracts a much simpler virtual topology for the operators [9], [27]; and (3) *monitoring* that collects the telemetry data, *e.g.*, #packets traversing the network [3], [28]. 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 disjoint 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 Section IV-A). 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

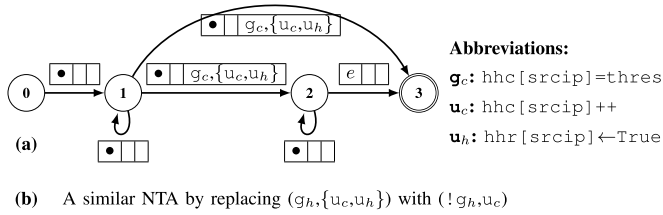


Fig. 4. The NTA group for *heavy-hitter-detection* semantics, which counts #flows sent from a certain IP, and tags it as a heavy hitter if the counter reaches a threshold.

have a one-to-many correspondence to the switches, NTA can directly use \cdot $*$ transitions to support such virtualization. For example, node 1 in 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 [17], 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 [13].

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 Section III-C). 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 [9] or regular expression [8].

C. 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* (\gg) 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*.

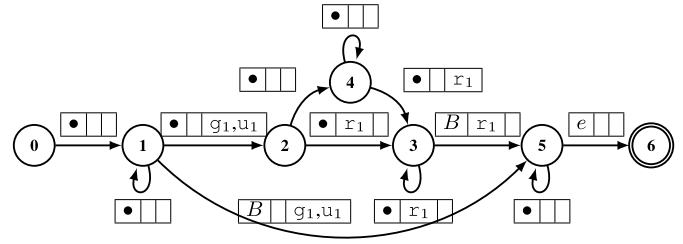


Fig. 5. Sequential composition (Fig. 2d \gg Fig. 2c).

- *Either-or composition* (Δ) produces an NTA that performs exactly one of n_1 and n_2 . This is useful for load-balancing 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.*, $u_3 = u_1 \cup u_2$. Note that usually u_1 and u_2 operate on different variables, because the network programs are independently written and should not rely on any shared variable. The middle end would abort and report to users when such cases detected. 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 (\bullet) , 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 Fig. 2d (n_1) with Fig. 2c (n_2), node 3 in n_1 and node 0 in n_2 will be removed. Transition $1 \rightarrow 3$ in n_1 will be modified as $\bullet \boxed{G_1, U_1}$, and then be merged with transition $0 \rightarrow 2$ in n_2 , producing a new transition $\boxed{B} \boxed{G_1, U_1}$ 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 Fig. 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.

D. 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 [29], 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 mitigate the complexity of this process, making the back end feasible for the real 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.

1) *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 $\boxed{h} \boxed{r} \boxed{d}$ 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 Fig. 6 shows the NFA for the NTA in Fig. 2f, where the \bullet (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 $0 \rightarrow 1$ and $0 \rightarrow 3$ in Fig. 2f are specialized to A_N , since in this stage we have known that i_{P_1} connects to A .

(2) *Mapping the NFA to physical network*. Let L denote the set of switches in physical network, and \mathcal{Q}_u denote the set of nodes in \mathcal{M}_u . Then, the set of nodes in \mathcal{G}_u is the Cartesian

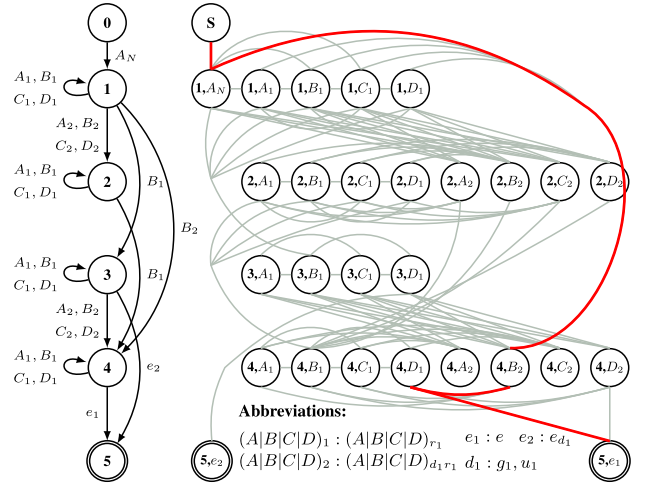


Fig. 6. The path graph constructed by the NTA in Fig. 2f and the topology in Fig. 2h (directed arrows are omitted). The thick path illustrates a path candidate.

product $L \times \mathcal{Q}_u$, 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 .

The right side of Fig. 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: $A \rightarrow B$ (counting) $\rightarrow D$.

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 Fig. 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 variables 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 variable 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 Fig. 6, and the other is

TABLE I
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 II
CONSTRAINTS OF THE MILP

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

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 [30].

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 $C = \{C_1, \dots, C_k\}$ 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 I, we explain the constraints on this problem shown in Table II.

(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 $R_i = \{R_{i1}, \dots, R_{ik}\}$. 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 each network change (*e.g.*, a link failure) 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 Section V-B); otherwise, CODER will fully re-compile all the NTAs.

E. Back End: Support Hybrid Data Plane

The back end introduced in Section III-D assumes that all switches in the data plane have the same capabilities, *i.e.*, they can perform all operations indicated by the NTAs. This assumption could be too strict in practice, as the real networks consist of heterogeneous devices. In this section, we extend the back end of CODER, which compiles NTAs to a hybrid data plane by respecting the different capabilities of the switches.

Capability abstraction for different NDP standards. To realize the wire-speed processing, data plane devices often follow a compact processing pipeline, where the de facto scheme for programmable switches is Parse+Match+Action. In the stage of Parse, the switch parses packets with fixed or arbitrary-defined protocols, and extracts the fields needed for the next stage. Next, the Match stage matches the fields with a (set of) table(s), which consists of user-defined rules. Finally, the Action stage performs the action of the matched rules. In the following, we analyze the processing capabilities in these stages, and propose a simple abstraction to capture them.

The Parse stage determines what fields can be extracted by the switch. For example, OpenFlow switches embed a fixed parser that can extract 44 fields from the packets [31]; while P4 switches can program the parser, which means they can identify arbitrary-defined flows. To this end, we use a set of 44 fields to depict the parsing ability of OpenFlow, and a set of infinite fields to depict the parsing ability of P4.

The capability in Match stage can be divided into two parts: what fields can be put in the table and how multiple tables can be organized. The first part is actually determined by the Parse stage, since the table must and only needs to support the fields extracted by the parser. For the second part, the multiple table layout is widely studied in previous literature, which takes input as the one logic table design, and outputs a physical layout with multiple tables [32]. That is, we can directly feed the logic rules (*i.e.*, the output of Step 3 of Section III-D), and leverage the existing techniques to generate the physical layout (*i.e.*, in Step 4 of Section III-D). In sum, we do not need to capture the capability of this stage.

A simple abstraction to capture the capability of Action stage is a set of the actions supported by the switches. For example, the action set of OpenFlow contains packet forwarding, and the action set of P4 further includes the packet reassemble. Besides those conventional actions, we notice that the recent proposed switches support the stateful operations. These operations read/write not only the packet and/or the Match table, but also a data structure storing states, *e.g.*, a state table or a register. One key feature of these data structures is the number of values they can store for a state. For example, a state table can only store a quite limited number of values (one value per entry, depending on the size of state table), while a register can store 2^n values (n is the width of the register). We view this number as another capability metric.

In sum, given an NDP standard, we describe its processing capability with a three-tuple (F, A, V) , which is a field set, an action set, and a number of values it can store for a state. As an example, the capability of OpenFlow 1.5 can be described as $(\{44 \text{ fields}\}, \{\text{forwarding}, \dots\}, 0)$, and the capability of P4 is $(\{\infty\}, \{\text{forwarding}, \text{reassemble}, \dots\}, 2^{32})$.

Compilation for hybrid data plane. Besides the constraints listed in Table II, CODER's back end needs to further consider two constraints: (1) the actions can only be performed on capable devices, and (2) the capabilities of each device should be maximally leveraged.

For the first constraint, CODER has to ensure that (1a) the operations assigned to switch s must be in $s.A$, and (1b) if we want to put a state at s , the number of possible values of the state must be less than $s.V$. To be specific, after constructing the path graph (Step 1 in Section III-D), CODER traverses the graph and removes all the nodes that do not satisfy (1a) and (1b). For example, node $(2, B_2)$ and $(4, B_2)$ in Fig. 6 mean that the counting variable will be put at switch B . However, assuming B is an OpenFlow switch, which does not support counting the packets with a variable, *i.e.*, $V = 0$, we can directly remove these two nodes. As such, Step 2 will not generate any path candidate that violates this constraint.

Satisfying the second constraint is more tricky. Intuitively, we should ensure that the packet class of NTAs can be identified by $s.F$. For example, if a packet class is `dstip=X&&tcp.flags=SYN`, we should route this packet class with P4 switches only, since other switches cannot extract `tcp.flags`. However, this scheme can waste the capability of other switches, as shown in the following.

Assume the above packet class is the only packet class specifying `dstip=X` in the whole network. Then, we can actually use OpenFlow switches to realize the correct forwarding, because `dstip` is enough to *distinguish* this packet class from others. The key insight from this example is that we do not need to ensure all the packet classes in s can be identified by $s.F$, instead, they must be *distinguishable* by $s.F$.

In fact, our original MILP shown in Table II has already considered the distinguishable constraint, which requires that if two packet classes are not distinguishable at switch s , *i.e.*, being viewed as the same packet class by s , the operations on them must be the same. For example, two packet classes `dstip=X` and `dstip=X&&tcp.flags=SYN` are not distinguishable at an OpenFlow switch, but that switch might also be useful, if we can perform the same operations (like forwarding to the same port) to these two packet classes.

IV. CODER IN ACTION

In this section, we demonstrate the expressiveness of NTA for 7 mainstream NPLs (Section IV-A). Then we present some key designs in CODER that make it practical (Section IV-B). We finally discuss another application, *i.e.*, network verification, upon CODER (Section IV-C).

A. Expressiveness for Diverse NPLs

Real programs from Merlin and SNAP. We collect 14 programs previously implemented by Merlin [33] and SNAP [34], as listed in Table III. 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 [3] generalizes the abstractions from NetCore [20], and offers a *virtual topology* construct. As discussed in Section III-B, NTA can express it by using $.*$ transitions to connect the nodes that map to the virtual switches.

FlowLog [28] 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 III
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 Fig. 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)

table/entry used by FlowLog into network variables in NTA, so that such manipulation can take place in the NDP.

FatTire [18] introduces a fault-tolerance semantics that can tolerate k link failures. As described in Section III-B, there are two realizations for this semantics, and we adopt the first one, *i.e.*, finding $k+1$ disjoint 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 $k+1$ disjoint paths.

NetKAT [27] 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 [10] 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 IV, 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.

B. Prototype Implementation

We implement CODER with ~ 3 K 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 three back ends for OpenFlow [31], OpenState [11] and NetASM [35]. We use the Gurobi optimizer [22] to solve the MILP shown in Table II. 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 Fig. 2f, *i.e.*, node 1 loop,

transition 1 \rightarrow 4. It then composes and compiles the NTA. The information of switch capabilities is embedded in `topo`.

```
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")
```

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 front end for Merlin takes ~ 50 LOC, while the native compiler of Merlin has ~ 6 K LOC; (2) the front end for SNAP takes ~ 80 LOC, while the native compiler of SNAP's compiler has ~ 5 K 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.

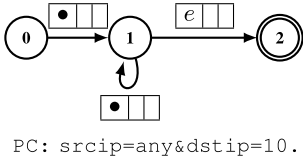
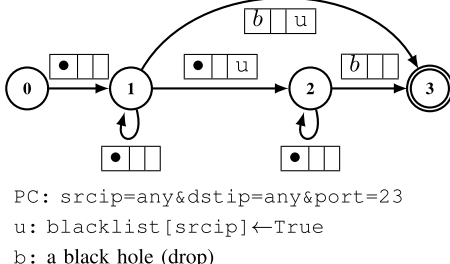
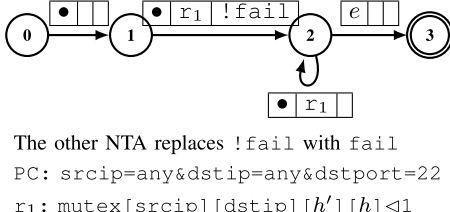
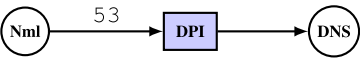
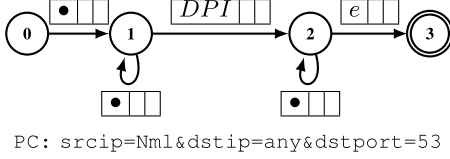
An xFDD is defined as either a branch ($t?d1 : d2$) (t is a test, $d1/d2$ is xFDD), or an action set. In brief, xFDD processes a packet from the root node to a leaf node, denoted as “if $t_1 \& \dots \& t_m$ then as ”, where t_i is the test (or its inverter) along the above path, and as is the action 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 [9]. 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 (XFMSM) [11]. For each incoming packet, the OpenState switch will first look up the current variable for this

TABLE IV
FEATURES, SNIPPETS AND NTAs FOR NPLs

NPL	TE	VT	MT	CP	Snippets	Corresponding NTA
Pyretic		✓	✓	✓	<pre>(match(dstip='10.0.0.1') >> fwd(6))</pre> <p>route traffic with dstip 10.0.0.1 to virtual port 6</p>	 <p>PC: srcip=any&dstip=10.0.0.1</p>
Flowlog			✓	✓	<pre>ON packet_in(p) WHERE p.nwPort = 23: INSERT (p.nwSrc) INTO blacklist;</pre> <p>block sender's IP if its TCP port is 23.</p>	 <p>PC: srcip=any&dstip=any&port=23 u: blacklist[srcip]←True b: a black hole (drop)</p>
FatTire	✓			✓	<pre>tpDst = 22 => [.*] with 1</pre> <p>specify a tolerance level for secure traffic.</p>	 <p>The other NTA replaces !fail with fail PC: srcip=any&dstip=any&dstport=22 r1: mutex[srcip][dstip][h'][h]<1</p>
NetKAT	✓	✓		✓	<pre>(if (dstip='10.0.0.1') then pt←6)</pre> <p>route traffic with dstip 10.0.0.1 to virtual port 6</p>	same with the NTA of Pyretic snippet
PGA	✓			✓	 <p>route Nml's DNS traffic to DNS traversing DPI</p>	 <p>PC: srcip=Nml&dstip=any&dstport=53</p>
Merlin	✓				see Fig. 2a	see Fig. 2c
SNAP		✓	✓	✓	see Fig. 2b	see Fig. 2d

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

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.

C. Network Verification Upon CODER

In Section III-C, 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.*, BGP configuration) [36] and data plane (*e.g.*, forwarding table) [37], [38]. 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 verification needs. Specifically, CODER inputs the NPL programs, and tries to compose and compile them according to the 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

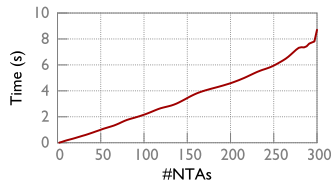


Fig. 7. The time cost of the parallel composition vs. #composed NTAs.

unified policy representations for all intents [38], 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? Operators may be concerned by the correctness of the NPL programs, *i.e.*, will they violate the 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.

V. PERFORMANCE EVALUATION

This section evaluates the performance of CODER. For front ends, we observe the translation for each program in Table III 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.

A. 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 III, 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). Fig. 7 only reports the running time for parallel composition of NTAs, with the number of NTAs varying from 2 to 300.

B. Back End

When evaluating the back end, we are interested in answering the following questions: (1) is the running time acceptable

TABLE V
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

for real network configurations? (2) can CODER efficiently react to network changes? (3) is the compiled solution (near-)optimal compared to existing approaches? (4) can CODER perform efficiently on hybrid data plane?

Settings. Our experiments use the topologies from SNAP [34], as shown in the left part of Table V. 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 III, which specifies two random waypoints, consumptions on one resource, and three variables. We apply `dns-defense` to 10% of all packet classes in each 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 [30]).

Breakdown of compilation time. Table V 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 [8], [9]. 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 [39], and randomly generate 100–10K packet classes. Fig. 8 shows

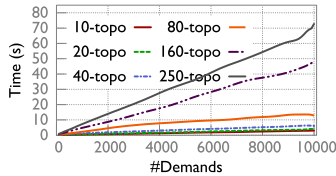


Fig. 8. The compilation time vs. different topologies and #demands (`dns-defense`).

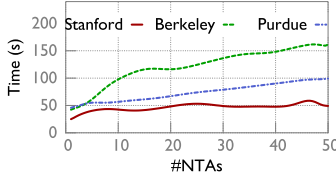


Fig. 9. The compilation time vs. #composed NTAs on campus topologies.

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 latter experiment).

The impact of NTA complexity. As the real programs in Table III are relatively small, we synthesize a large NTA by either-or composing `dns-defense` with itself. Since we deliberately choose not to merge equivalent nodes, the size of path graph grows exponentially. Fig. 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 III 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 Fig. 10 reports that CODER can re-compile the NTA within 1.6s for all cases.

Compilation optimality. We measure optimality by compiling `defense` and `dns-tunnel-detect` in Table III and comparing with Merlin and SNAP respectively. We mainly

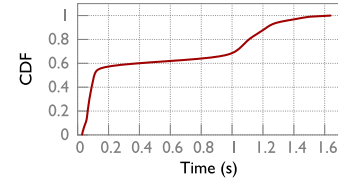


Fig. 10. CDF of reacting to a link failure (`dns-defense` on IGen 80-switch topology).

TABLE VI
AVERAGE NUMBER OF HOPS FOR EACH DEMAND

topo.	Merlin	CODER (+%)	SNAP	CODER (+%)
Stanford	6.98	0%	3.32	0%
Berkeley	6.04	0%	3.07	0%
Purdue	9.09	1.2%	4.08	0.9%
AS 1755	11.53	0%	5.99	1.1%
AS 1221	13.31	1.6%	6.12	1.7%
AS 6461	12.16	1.4%	5.23	1.6%
AS 3257	10.21	0.6%	5.75	0.8%

compare the average number of hops for each demand, because it shows the latency overhead in runtime, and also infers the resource overhead (*i.e.*, the number of generated rules). Other settings are consistent with Table V. Results in Table VI show that for most cases, CODER incurs no overhead compared with Merlin and SNAP. In a few cases, CODER might bring minor overhead. The reason is that in Step 4 of Section III-D, CODER would solve multiple MILPs in parallel, and it is possible that the first solved path is not the optimal (shortest) one. However, since we constrain the path candidates to be shorter than $stp + 3$ (stp is the length of the shortest path), the overhead is quite minor, *i.e.*, less than 2%.

Practicalness of heuristics. Mapping complex high-level intents into data plane is quite difficult, especially considering the rich semantics NTA supports. To this end, we synthesize complex NTA and large networks to stress-test the middle end and back end. The results show that: (1) the middle end would consume unacceptable time (*i.e.*, longer than 2 hours) if we compose 150 NTAs that each has hundreds of transitions and nodes; (2) the back end will break down (*i.e.*, do not compile within 2 hours) on a 800-node topology with 10,000 demands.

We argue that the real intents are not that complex, as mentioned in Section V-A, and in most cases, large topologies often come with symmetric feature, where many heuristics can be used to accelerate the compilation [40]. In sum, CODER does not try to fundamentally conquer the vast complexity, but aims to provide a feasible solution in most practical cases.

Performance on hybrid data plane. In this experiment, we pre-define two kinds of switches, P4 and OpenFlow, using the capability abstraction proposed in Section III-E. Then, we use the same NTA and topologies shown in Table V, and randomly specify some switches as the P4 switches, and others as the OpenFlow switches. Table VII shows that the compilation speed boosts with the decrement ratio of P4 switch. This is because we eliminate the impossible nodes from the path graph, such that the following step will generate

TABLE VII
dns-Defense ON HYBRID DATA PLANE

topo.	10% P4 switches	50% P4 switches	80% P4 switches	original
Stanford	7.97s	18.71s	26.28s	26.39s
Berkeley	12.44s	30.93s	42.69s	42.45s
Purdue	12.88s	31.13s	40.36s	41.02s
AS 1755	3.41s	8.72s	10.69s	11.19s
AS 1221	3.96s	10.92s	14.76s	15.10s
AS 6461	19.20s	47.17s	65.59s	64.89s
AS 3257	25.14s	58.90s	82.57s	84.79s

much less path candidates compared to the original version. Besides, the path elimination process is simple and fast, and we do not employ new constraints in MILP. As a result, lower ratio of P4 switches narrows the possibilities of variable placement, and hence the higher compilation speed.

VI. LIMITATIONS AND DISCUSSION

Semantics completeness. As mentioned in Section I, NTA focuses on the semantics of *network-wide* behaviors, *e.g.*, routing and endpoint actions like counting, while the *single-point* processing is out of the scope of NTA, *e.g.*, count-min sketch and bloom filter. NTA does not support the control plane program with dynamic semantics. For example, bandwidth negotiation [8], latency minimization [41] and vector-distance routing [42] require to probe the network and react to the network changes in the data plane, which cannot be expressed by NTA for now. We would explore the enhancement of NTA's expressiveness in our future work, and currently, NTA focuses on expressing the deterministic end-to-end semantics.

Hardware capability of NDP. CODER's NDP capability model only considers the processing ability, *i.e.*, whether an operation *is supported* by the NDP, while to compile P4 programs into a real P4 switch, one should also consider the capability of the hardware like the size of TCAM, the length of pipeline and the number of ALUs [43]. A simple solution would be like the adding rule capacity constraint in OpenFlow switches. However, modeling constraints for ALU and pipeline capacity is more difficult because it is not clear how the resource would be consumed if we put a specific P4 operation on a switch. We will explore how to involve the hardware capability into the NDP model in the future work.

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 [44]. 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, Fig. 2d and 2e correspond to the same routing path, so we

can just create and compile the former one, and place !g₁ at the same switch with g₁. 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.

VII. RELATED WORK

NPLs use formal representations to carry their semantics. NetKAT [27], [45] 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 [8] also express the path requirements with regular grammar, and extends them with bandwidth requirements. Kinetic [46] uses Finite State Machine to capture the dynamic semantics of network. SNAP [9], 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 [2], [3], reserving the resource [8], [18], and stateful operations [9]. Besides above static semantics, there also exists the dynamic semantics that negotiate the packet behaviors in the runtime, *e.g.*, bandwidth adjusting [8], latency minimization [42]. Currently, NTA cannot express those dynamic semantics (Section VI).

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 [3], [9], [10], [15], [16]. CoVisor tries to compose NDP rules compiled from the cross-language programs [17]. 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 [41], [47], and CODER currently does not support this feature as NTA cannot express those semantics.

Target-independent compilation in previous works mainly focuses on NDP [13], [35], 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 [30] 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.

Other IRs are also used in programmable networks, most of which, however, are for single-switch programming. P4 [13] has its IR called HLIR (high-level IR), which is the starting point of the program being compiled into different back end, like FPGA, Tofino, *etc.* NetASM [35] is an IR for bridging the

TABLE VIII
COMPARISON OF IRs IN PROGRAMMABLE NETWORKS

	Target	Basic Form of IR	Major Features
P4-HLIR [13]	Single-switch	Control-flow graph	U-NDP
μ P4 [48]	Single-switch	Control-flow graph	U-NDP, CP
NetASM [35]	Single-switch	Assembly-like instructions	U-NDP, PO
SNAP [9]	Network-wide	Decision diagrams	CP
Merlin [8]	Network-wide	Automaton	PO
FatTire [18]	Network-wide	Forwarding graph	PO
CODER	Network-wide	Automaton	U-NPL, U-NDP, CP, PO

U-NPL: Unifying NPLs, U-NDP: Unifying NDPs
CP: Composition, PO: Performance optimization

single-switch high-level language (P4, click) with the hardware implementation, which can further optimize the processing by re-scheduling the pipeline, eliminating the dead code, *etc.* μ P4 [48] proposes to write the P4 program in a modular way, and composes the program pieces with its IR, which can be further compiled into different forms of implementation.

IRs are also used in the network-wide programming frameworks, which, however, are not for unifying the NPLs or NDPs, instead, they are to facilitate the compilation or optimizations of its own NPLs. For example, FatTire [18] uses a forwarding graph as its IR, to fast locate the feasible backup paths for fault tolerance; Merlin [8] first transforms the program into an automaton, in order to decide the optimal forwarding paths; SNAP [9] uses xFDD as its IR, to merge program pieces written in SNAP.

Table VIII briefly summarizes IRs used in programmable networks. It can be seen that previous IRs either are for single-switch programming, or can only serve a certain NPL/NDP, while CODER with NTA is the first IR that unifies the network-wide NPLs and hybrid NDPs.

VIII. 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 practical and feasible 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.

REFERENCES

- [1] H. Li *et al.*, "A modular compiler for network programming languages," in *Proc. 16th Int. Conf. Emerg. Netw. Exp. Technol.*, Nov. 2020, pp. 198–210.
- [2] N. Foster *et al.*, "Frenetic: A network programming language," in *Proc. ACM SIGPLAN ICFP*, 2011, pp. 279–291.
- [3] C. Monsanto *et al.*, "Composing software defined networks," in *Proc. USENIX NSDI*, 2013, pp. 1–13.
- [4] (2018). *Floodlight OpenFlow Controller*. [Online]. Available: <https://bit.ly/2Riemyh>
- [5] (2017). *Ryu OpenFlow Controller*. [Online]. Available: <https://bit.ly/2TedVCF>

- [6] C. Trois, M. D. Del Fabro, L. C. E. de Bona, and M. Martinello, "A survey on SDN programming languages: Toward a taxonomy," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 4, pp. 2687–2712, 4th Quart., 2016.
- [7] Z. Latif, K. Sharif, F. Li, M. M. Karim, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," 2019, *arXiv:1902.07913*. [Online]. Available: <https://arxiv.org/abs/1902.07913>
- [8] R. Soulé *et al.*, "Merlin: A language for provisioning network resources," in *Proc. ACM CoNEXT*, 2014, pp. 213–226.
- [9] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *Proc. ACM SIGCOMM*, 2016, pp. 29–43.
- [10] C. Prakash *et al.*, "PGA: Using graphs to express and automatically reconcile network policies," in *Proc. ACM SIGCOMM*, 2015, pp. 29–42.
- [11] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.
- [12] *NETCONF*, document RFC 6241, 2011. [Online]. Available: <https://bit.ly/3nEpPoM>
- [13] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.
- [14] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. ACM HotSDN*, 2013, pp. 127–132.
- [15] W. Wang, W. He, and J. Su, "Redactor: Reconcile network control with declarative control programs in SDN," in *Proc. IEEE 24th Int. Conf. Neww. Protocols (ICNP)*, Nov. 2016, pp. 1–10.
- [16] A. Bairley and G. G. Xie, "Orchestrating network control functions via comprehensive trade-off exploration," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2016, pp. 114–120.
- [17] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proc. USENIX NSDI*, 2015, pp. 87–101.
- [18] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proc. ACM HotSDN*, 2013, pp. 109–114.
- [19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2012, pp. 1–12.
- [20] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2012, pp. 217–230.
- [21] R. Amin, M. Reisslein, and N. Shah, "Hybrid SDN networks: A survey of existing approaches," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 3259–3306, 4th Quart., 2018.
- [22] (2018). *Gurobi Optimizer*. [Online]. Available: <https://bit.ly/1Kc3Hxc>
- [23] (2017). *CPLEX Optimizer*. [Online]. Available: <https://ibm.co/2N2v6s6>
- [24] T. Ralphs, Y. Shinano, T. Berthold, and T. Koch, *Parallel Solvers for Mixed Integer Linear Optimization*. Springer, 2018, pp. 283–336. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-319-63516-3_8
- [25] T. Koch, T. Ralphs, and Y. Shinano, "Could we use a million cores to solve an integer program?" *Math. Methods Operations Res.*, vol. 76, no. 1, pp. 67–93, Aug. 2012.
- [26] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *Proc. ACM Conf. SIGCOMM*, Aug. 2013, pp. 27–38.
- [27] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 29, no. 1, pp. 113–126, 2014.
- [28] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. USENIX NSDI*, 2014, pp. 519–531.
- [29] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar, "Approximation algorithms for the unsplittable flow problem," *Algorithmica*, vol. 47, no. 1, pp. 53–78, 2007.
- [30] V. Heorhiadi, M. K. Reiter, and V. Sekar, "Simplifying software-defined network optimization using SOL," in *Proc. USENIX NSDI*, 2016, pp. 223–237.
- [31] (2015). *OpenFlow Switch Specification 1.5.1*. [Online]. Available: <https://bit.ly/1Wqi7N2>
- [32] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. ACM SIGCOMM*, 2013, pp. 1–12.
- [33] (2017). *Merlin*. [Online]. Available: <https://bit.ly/2TlaDOK>

- [34] (2016). *SNAP*. [Online]. Available: <https://bit.ly/2GF472Y>
- [35] S. Muhammad and F. Nick, "The case for an intermediate representation for programmable data planes," in *Proc. ACM SOSR*, 2015, pp. 1–6.
- [36] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 155–168.
- [37] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *Proc. USENIX NSDI*, 2020, pp. 241–255.
- [38] W. Zhou, J. Croft, B. Liu, E. Ang, and M. Caesar, "Automatically correcting networks with neat," in *Proc. USENIX NSDI*, 2018, pp. 595–608.
- [39] B. Quoitin, V. V. den Schrieck, P. Francois, and O. Bonaventure, "IGen: Generation of router-level internet topologies through network design heuristics," in *Proc. 21st Int. Telettraffice Congr.*, Sep. 2009, pp. 1–8.
- [40] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM SIGCOMM*, 2014, pp. 503–514.
- [41] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, "Intent-driven composition of resource-management SDN applications," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2018, pp. 86–97.
- [42] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tamma, and D. Walker, "Contra: A programmable system for performance-aware routing," in *Proc. USENIX NSDI*, 2020, pp. 701–721.
- [43] J. Sonchack, D. Loehr, J. Rexford, and D. Walker, "Lucid: A language for control in the data plane," in *Proc. ACM SIGCOMM*, 2021, pp. 731–747.
- [44] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proc. ACM SIGCOMM*, 2013, pp. 19–24.
- [45] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for NetKAT," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 2015, pp. 343–355.
- [46] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *Proc. USENIX NSDI*, 2015, pp. 59–72.
- [47] A. AuYoung *et al.*, "Democratic resolution of resource conflicts between SDN control programs," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2014, pp. 391–402.
- [48] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with $\mu P4$," in *Proc. ACM SIGCOMM*, 2020, pp. 329–343.

Hao Li received the Ph.D. degree in computer science from Xi'an Jiaotong University in 2016. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His main research interests include programmable networks and network measurement.

Peng Zhang received the Ph.D. degree in computer science from Tsinghua University in 2013. He was a Visiting Researcher at The Chinese University of Hong Kong and Yale University. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. He is also with the MOE Key Laboratory for Intelligent Networks and Network Security. His research interests include network verification and network security.

Guangda Sun received the bachelor's degree in computer science from Xi'an Jiaotong University in 2020. He is currently pursuing the Ph.D. degree with the National University of Singapore. His research interests include distributed systems in datacenter, programmable network hardware and hybrid network design, and network verification.

Wanyue Cao received the bachelor's degree in computer science from the Dalian University of Technology in 2019. She is currently pursuing the master's degree with Xi'an Jiaotong University. Her research interests include network management and configuration synthesis.

Chengchen Hu (Member, IEEE) is currently the Chief Expert and the Associate VP of Technology Planning at NIO Inc. Prior to joining NIO, he was a Principal Engineer and the Founding Director of Xilinx Labs Asia-Pacific, Singapore. Before his experience with Xilinx, he was a Professor and the Department Head of the Department of Computer Science and Technology, Xi'an Jiaotong University, China. His research theme is to monitor, diagnose, and manage networking and distributed computing through hardware optimized and software-defined systematical approaches. He was a recipient of the New Century Excellent Talents in University Award from the Ministry of Education, China, a fellowship from the European Research Consortium for Informatics and Mathematics (ERCIM), and a fellowship of Microsoft "Star-Track" Young Faculty.

Danfeng Shan (Member, IEEE) received the B.E. degree in computer science and technology from Xi'an Jiaotong University, China, in 2013, and the Ph.D. degree in computer science and technology from Tsinghua University, China, in 2018. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His research interests include data center networks and congestion control.

Tian Pan received the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, in 2015. He is currently an Associate Professor with the Beijing University of Posts and Telecommunications. His main research interests include data center networks, programmable data plane, and satellite networks.

Qiang Fu received the Ph.D. degree from The University of Queensland. He is currently a Senior Lecturer in cloud, systems, and security with RMIT University. His research interests are broadly in the areas of Internet and cloud-based systems including wireless and mobile systems. More recently, he has a focus on content delivery networks, data center design and analysis, cyber-physical systems and the IoT, virtualization, as well as network programmability.