An Intermediate Representation for Network Programming Languages

Hao Li
Xi'an Jiaotong University

Peng Zhang
Xi'an Jiaotong University

Guangda Sun
Xi'an Jiaotong University

Chengchen Hu
Xilinx Labs Asia Pacific

Danfeng Shan
Xi'an Jiaotong University

Tian Pan
Beijing University of Posts and Telecommunications

Qiang Fu
Victoria University of Wellington

ABSTRACT

Network programming languages (NPLs) empower operators to program network data planes (NDPs) with unprecedented efficiency. Currently, various NPLs and NDPs coexist and no one can prevail over others in the short future. Such diversity is raising many problems including: (1) programs written with different languages can hardly interoperate in the same network, and (2) most NPLs are bound to specific NDPs, hindering their independent evolution. 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 representation as the IR. We show that NTA can express semantics of 6 mainstream NPLs, and can be composed efficiently without any semantics loss.

ACM Reference Format:
Hao Li, Peng Zhang, Guangda Sun, Chengchen Hu, Danfeng Shan, Tian Pan, and Qiang Fu. 2020. An Intermediate Representation for Network Programming Languages. In *4th Asia-Pacific Workshop on Networking (APNet '20), August 3–4, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3411029.3411030

ACM ISBN 978-1-4503-8876-4/20/08…$15.00
https://doi.org/10.1145/3411029.3411030

1 INTRODUCTION

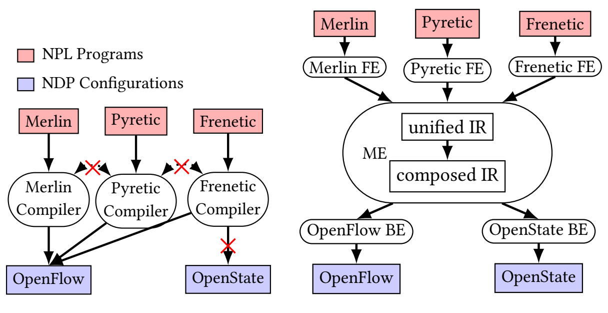
With the advance of SDN, many languages (*e.g.*, Pyretic [15]) have been proposed for programming computer networks. These languages, which we refer to as *network programming languages (NPLs)*, offer operators with an unprecedented way to program *network data planes (NDPs)*. Different from general-purpose languages shipped with controllers (*e.g.*, Java in Floodlight [1]), NPLs provide various high-level constructs that can greatly facilitate composing complex functions like path selection, monitoring, *etc*.

Multiple NPLs and NDPs coexist in modern networks. Recent surveys [13, 21] report more than 15 NPLs including Merlin [20], SNAP [3], *etc*, and more than 10 NDPs including OpenState [6], P4 [7], *etc*. We believe such diversity 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 [20], while SNAP can realize a stateful end-to-end monitoring function [3]. These two NPLs are designed for fulfilling different management demands in the first place, and cannot be simply replaced with one of them.

Second, deploying a unified NPL/NDP can be quite 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 [7] claim that they outperform OpenFlow in almost all perspectives (flexibility, forwarding performance, *etc*), the high cost, *e.g.*, the higher price of devices, the training cost for the operators, still obstructs their broad deployment in the Internet and data centers.

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



(a) The monolithic compiler (b) The monolithic compiler

Figure 1: Three stages in a modular compiler with IR: the front end (FE), the middle end (ME), and the back end (BE).

from the de facto monolithic compilers, then propose our approach with key contributions highlighted.

1.1 Problems of Monolithic Compilation

The monolithic compilers compile programs all the way to a specific NDP (Figure 1(a)), which can raise many problems when handling the coexistence of multiple NPLs and NDPs.

Cross-language programs cannot interoperate. Operators might run multiple programs by using *program composition*. The correct compositions require complete semantics from all programs, which however is only achievable in single-language programs [3, 15, 17]. For cross-language programs, the only possible way is to merge the NDP configurations compiled individually from their own NPL compilers. However, this cannot be achieved in a safe way due to the rule conflicts. Consider two programs, one sets a waypoint B, and the other counts the packet in an end-to-end way. The individual compilers may interpret these two intents to two paths, $A \rightarrow B \rightarrow D$, and $A \rightarrow C \rightarrow D$, respectively. These two paths raise a rule conflict in A, and cannot be merged or overwritten, because B is a waypoint of the first program, and C *could* be the counting switch of the second program.

CoVisor [11] 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, 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 [19] are built upon the NetCore abstractions [14], which does not support stateful operation. This close binding between NPLs and NDPs greatly hinders their independent evolution.

1.2 Our Approach and Contributions

To break the monolith, we intuitively analog to the successful PC compiler, which also compiles the high-level programs (*e.g.*, C program) 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 introduce the concept of IR into network compiler, and modularizes the compilation into three stages (Figure 1(b)): a set of *front ends* translate NPL programs into IR, a *middle end* conducts compositions, and a set of *back ends* translate the IR into various NDP configurations.

The decoupling of NPLs and NDPs naturally addresses the above problems: (1) the programs are compiled into the IR that retains all intents, which can be composed, and compiled into NDP configurations without any conflicts; and (2) a new NPL only needs to implement a thin front end for supporting all NDPs, and vice versa for a new NDP standard.

Based on this insight, we state our research contributions.

Contribution 1: An expressive and unified IR (§3.1). We introduce *Network Transaction Automaton (NTA)*, a new automaton that can express the semantics of existing (and possibly future) NPLs. The key difference of NTA is that we incorporate network resources and state variables into its transitions. This enables NTA to express not only path constraints, but also resource constraints and stateful operations, in a fine-grained, hop-by-hop way (see §3.2).

Contribution 2: Compositions without semantics loss (§3.3). We design the composition operator upon NTA that respect its physical meanings, so that NTAs can be composed without any semantics loss.

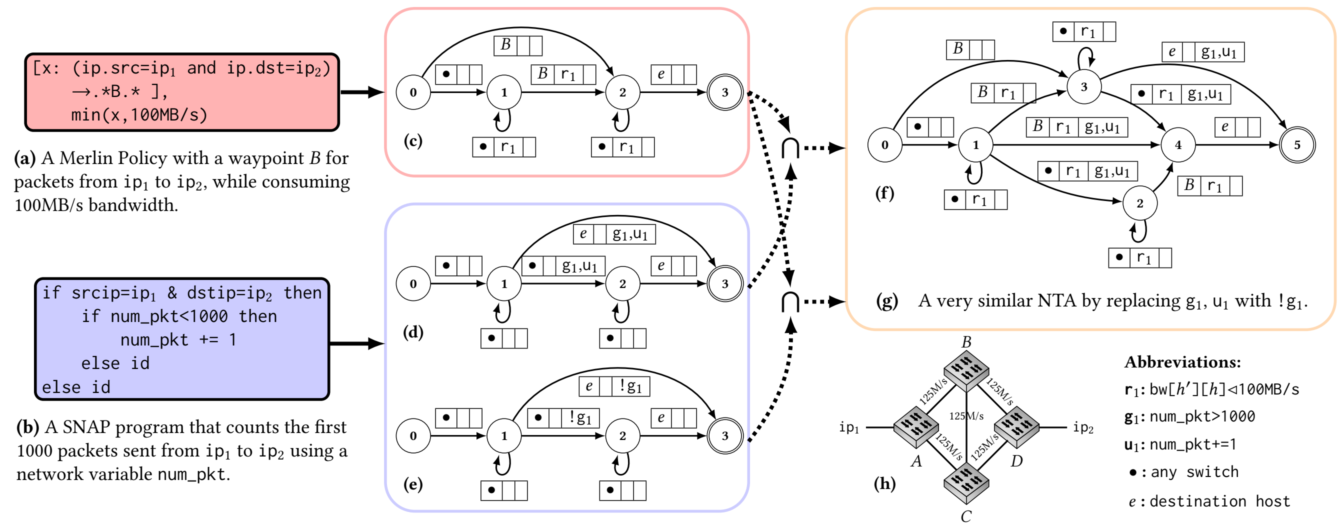
Our preliminary evaluation shows that NTA can express semantics of 6 mainstream NPLs, and can be composed efficiently and correctly without semantics loss (§4).

2 OVERVIEW

In this section, we will first take a glance at the proposed IR, *i.e.*, NTA, and then use a concrete example to present how IR modularizes the compilation process.

2.1 A First Look at NTA

We have two observations by revisiting the semantics of NPLs. First, NPL semantics 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



```
[x: (ip.src=ip₁ and ip.dst=ip₂)
    →.*B.* ],
    min(x,100MB/s)
```

(a) A Merlin Policy with a waypoint B for packets from ip$_1$ to ip$_2$, while consuming 100MB/s bandwidth.

```
if srcip=ip₁ & dstip=ip₂ then
    if num_pkt<1000 then
        num_pkt += 1
    else id
else id
```

(b) A SNAP program that counts the first 1000 packets sent from ip$_1$ to ip$_2$ using a network variable num_pkt.

(g) A very similar NTA by replacing g$_1$, u$_1$ with !g$_1$.

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

reservation, and (P3) stateful packet manipulation with variables persistent on NDP, *e.g.*, counting all SSH packets at some switch. Second, operators expect to specify the above semantics in the finest grain, *i.e.*, hop-by-hop.

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

2.2 A Walk-through Example

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

Using NTA to express programs. Figure 2a shows a Merlin policy [20], which specifies a waypoint B for packets sent from ip$_1$ to ip$_2$, while consuming 100MB/s bandwidth along the above path. Figure 2c is the corresponding NTA for packet class srcip=ip$_1$&dstip=ip$_2$. Each transition in the NTA carries a three-tuple, in the order of next hop, resource consumption, and variable operation. This NTA has a start state (node 0) indicating the packet is entering the network, and an end state (node 3) indicating the packet has left the network. The loops on node 1 and 2 along with the transition $1 \longrightarrow 2$ realize the waypointing and NTA explicitly uses e to denote the destination host in transition $2 \longrightarrow 3$. 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 connects to start and end nodes), this NTA reserves 100MB/s for the end-to-end connection.

Figure 2b shows a SNAP program [3], 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 Figure 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 the NTAs consider all possible locations triggering the counting operation, *i.e.*, in middle of the network ($1 \longrightarrow 2$), or at the last hop ($1 \longrightarrow 3$).

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.

In our case, the composition of the two example programs is a new NTA group consisting of the intersection of Figure 2c and 2d, and the intersection of Figure 2c and 2e. The former is shown in Figure 2f, where `num_pkt` is checked and updated exactly once along the path traversing B. The other NTA is much the same with Figure 2f except (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" (see §3.3 for details).

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

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

The compilation of NTA is outside the scope of this paper, and will be explored in our future work (see §5).

3 NETWORK TRANSACTION AUTOMATON

In this section, we detail the design of NTA. Specifically, we first formally define NTA (3.1), and show how NTA can express various semantics (3.2). We finally present the operations on NTA for modular compositions (3.3).

3.1 Definitions

Network transaction. A network transaction is a three-tuple, $\boxed{h \mid r \mid d}$, 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*.

NTA. An 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" of all 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.*, entries in the flow table, bandwidth `bw` in Figure 2c, which are shared by all NTAs and consumed by installing the switch rules or end host configurations.

The *variable* records the network states, which is persistent on the data plane switch [3], *e.g.*, `num_pkt` in SNAP's NTA. The guard and update of variables map to the matching fields and actions in data plane rules, respectively. There could be different network transaction spaces depending on the results of the guard, so a group of NTAs will be generated to handle each of them, as shown in Figure 2d and 2e.

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. 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 using composition operators.

3.2 Expressiveness of NTA

A recent survey classifies the semantics of NPLs into three catalogs [21]: (1) *traffic engineering* that optimizes the routing paths, *e.g.*, waypointing [18], QoS [20]; (2) *virtualization* that abstracts a much simpler virtual topology for the operators [2, 3]; and (3) *monitoring* that collects the telemetry data, *e.g.*, #packets [15, 16]. In the following, we show NTA is capable for expressing these and even more complex semantics.

Traffic Engineering (TE) includes waypointing and QoS. First, NTA naturally supports all path requirements compliant with regular grammar for waypointing. For example, for a sequence of waypoints $w_1, \ldots, w_n$, we can construct an NTA with a regular expression $.{*}w_1.{*}\ldots{*}w_n.{*}$. For QoS, NTA can express the constraints of network resources using consumptions, *e.g.*, Merlin's NTA in Figure 2c.

Virtualization (VT) hides the low-level network details, so that programmers can install the micro-flow rules on a higher-level abstraction. There are typically two kinds of VT: one-to-many and many-to-one. For the first, since NTA nodes also have a one-to-many correspondence to the switches, NTA can directly use `.*` transitions to support such virtualization. For example, node 1 in 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 [11], 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. For example, `num_pkt` in Figure 2b is used to record #packets that traverse certain switches. We can also use variables to record #active connections, if TCP SYN flag and FIN flag can be extracted by a programmable parser [7].

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 3 in §3.3). Such 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 [3] or regular expression [20].

3.3 Composition of NTAs

Let the two NTAs be n_1 and n_2, and we offer two types of composition on NTA: (1) parallel composition (+) that produces an NTA that accepts n_1 and n_2 simultaneously. which is perhaps the most important type of composition, as it enables cross-language programs to manipulate the same traffic; and (2) sequential composition (>>) that produces an NTA that performs n_1 and n_2 sequentially, which can be used when one program is triggered by another, *e.g.*, *counting* the suspicious flows identified by a *firewall.* Note that composition of two NTA groups can be viewed as a product composition of each NTA in the groups.

The theoretical basis of the composition is the operation on automaton, *e.g.*, intersection, concatenation. However, the resource and stateful elements carried in NTA transitions will be lost if directly applying the conventional operation. Hence, we customize those operations for composing NTAs.

Intersection. We adopt 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 $(q_{1,0}, q_{2,0})$, a new transition is produced 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=$u_1 \cup u_2$. 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. Figure 2f shows the intersection of Figure 2c and 2d.

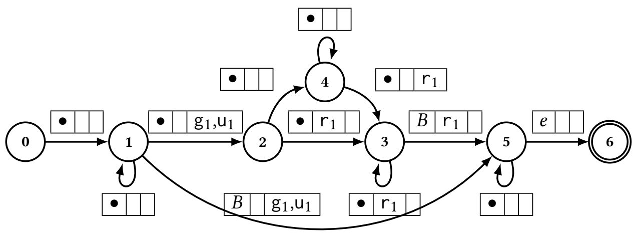


Figure 3: Sequential composition (Figure 2d>>Figure 2c).

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, we 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, we replaces e in the next hop field with (●), and product-merges them with the transitions starting from $q_{2,0}$. Since the modified transitions must have a dot next hop, the merging is ensured to be successful.

For example, when concatenating Figure 2d (n_1) with Figure 2c (n_2), node 3 in n_1 and node 0 in n_2 will be removed. Transition 1⟶3 in n_1 will be modified as $\boxed{\bullet \mid \; \mid g_1, u_1}$, and then be merged with transition 0⟶2 in n_2, producing a new transition $\boxed{B \mid \; \mid g_1, u_1}$ 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 Figure 3. Note that this process may produce a non-deterministic NTA, and we can reduce it using conventional technique.

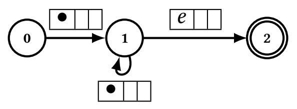
4 PRELIMINARY EVALUATION

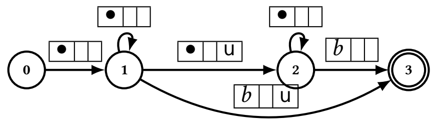
In this section, we evaluate (1) the expressiveness of NTA, and (2) the efficiency of composing many NTAs.

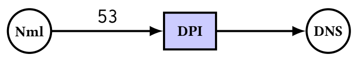
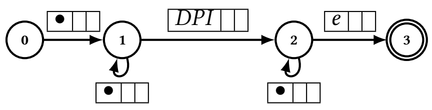
4.1 Expressiveness for Diverse NPLs

Besides Merlin and SNAP, we further investigate the expressiveness of NTA for other 4 NPLs.

Table 1: Features, snippets and NTAs for NPLs

NPL	*TE*	*VT*	*MT*	*CP*	Snippets	Corresponding NTA
Pyretic		√	√	√	`(match(dstip='10.0.0.1') >> fwd(6))` route traffic with dstip 10.0.0.1 to virtual port 6	 `PC: srcip=any&dstip=10.0.0.1`

Flowlog			√	√	`ON packet_in(p) WHERE p.nwPort = 23:`
					`INSERT (p.nwSrc) INTO blacklist;`
					block sender's IP if its TCP port is 23.
						
						`PC: srcip=any&dstip=any&port=23`
						`u: blacklist[srcip]←True` `b:` a black hole (drop)

NetKAT	√	√		√	`(if (dstip='10.0.0.1') then pt←6)` route traffic with dstip 10.0.0.1 to virtual port 6	same with the NTA of Pyretic snippet
PGA	√			√	 route `Nml`'s DNS traffic to DNS traversing DPI	 `PC: srcip=Nml&dstip=any&dstport=53`
Merlin	√				see Figure 2a	see Figure 2c
SNAP		√	√	√	see Figure 2b	see Figure 2d

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

Pyretic [15] generalizes the abstractions from NetCore [14], and offers a *virtual topology* construct. As discussed in §3.2, NTA can express such semantics by using `.*` transitions to connect the nodes that map to the virtual switches.

FlowLog [16] offers a SQL-like query syntax to manipulate packets using the states stored in the controller. We successfully express this semantics by mapping the database table/entry used by FlowLog into network variables in NTA, so that such manipulation can take place in the NDP.

NetKAT [2] 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 [17] uses policy graphs to specify the source, destination, and waypoints of the flows, which can be directly translated into an NTA that has no resources and variables.

We summarize above NPLs in Table 1, and find that no NPL supports all features. This partially confirms the necessity of running cross-language programs in the same network. We also present the NTA for a snippet of each NPL, and conclude that NTA can express all semantics of those NPLs.

4.2 Composition Performance

We synthesize NTAs to test the composition performance. We observe that the NTAs for real programs are relatively small: for programs in listed in SNAP's appendix [4], 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. We note that the sequential composition only operates the start and end nodes, so its running time is independent of NTA sizes, and very fast (within 1ms for two large NTAs). Figure 4 reports the running time for parallel composition, with the number of NTAs varying from 2 to 300, which shows to be moderate.

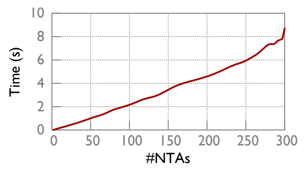


Figure 4: The time cost of the parallel composition.

5 SUMMARY AND FUTURE WORK

This paper motivated the need to modularize the compiler of NPLs with IR. We proposed such an IR, *i.e.*, NTA, and showed it to be expressive for 6 mainstream NPLs. We also proposed two composition operators of NTA and showed it can efficiently compose hundreds of NTA. While this work sheds the light of realizing a complete modular compiler, there still exists many challenges and future work, as outlined below.

Semantics completeness. NPL semantics is constantly evolving, from forwarding the packets [15], to reserving the resource [20], to stateful operations [3]. NTA can fully cover those static semantics. Besides, there also exists the dynamic semantics that negotiate the packet behaviors in the runtime, *e.g.*, bandwidth negotiation [20], latency minimization [10]. And there are approaches to compose the dynamic semantics [5, 8]. Currently NTA cannot express or compose such dynamic semantics, but we argue that they are realizable by further extending the semantics of transitions in NTA.

Middle end optimizations. Previously, running multiple programs in a single network is only achievable for the programs written in the same NPL [3, 15, 17], or between the compatible/prioritized NDP rules [11]. NTA currently supports the parallel and sequential composition in the middle end. In the future, it is possible to investigate other optimizations upon NTA, *e.g.*, the network verification [22].

Compilation of NTA. As discussed in §2.2, conducting an MILP could be a packaged solution for compilation. However, creating and solving MILP for NTA could be very time-consuming, as the #constraints of MILP would exponentially grow with complexity of intents, *e.g.*, the path and resource constraints, the consistency of variable placements. As a reference, SNAP takes more than 300s to solve a stateful intent (without path constraints) on a 160-switch topology [3]. This becomes more challenging if the incremental compilation is demanded for minor changes in network (*e.g.*, a link failure). To this end, we could leverage the heuristics introduced in SOL [9], which reduces the size of MILP by pre-eliminating the impossible solutions. Another direction is to parallelize the MILP solving, which might be infeasible for a single large MILP [12], but should be effective if we properly partition the problem and create many smaller MILPs instead.

ACKNOWLEDGMENTS

The authors thank the anonymous APNet reviewers for their valuable feedback. This work is supported by NSFC (No. 61702407, 61772412, 61702049 and 61902307), and in part by the Fundamental Research Funds for the Central Universities, InternetNZ, Computing and Networking Innovation Lab, Technology Innovation Department, Cloud BU, Huawei and MoE-CMCC Artifical Intelligence Project under Grant MCM20190701. The work of Peng Zhang was supported by the K. C. Wong Education Foundation.

REFERENCES

[1] 2018. Floodlight OpenFlow Controller. https://bit.ly/2Riemyh. (2018).

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices* (2014).

[3] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *ACM SIGCOMM*.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. *SNAP: Stateful Network-Wide Abstractions for Packet Processing.* Technical Report.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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