SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2020, Vol. 63 172102:1–172102:17
https://doi.org/10.1007/s11432-019-2644-8

From model to implementation: a network algorithm programming language

Jian WANG[1,2], Jie AN[3], Mingshuai CHEN[1,2], Naijun ZHAN[1,2], Lulin WANG[4], Miaomiao ZHANG[3] & Ting GAN[5*]

[1]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100190*, China;*
[2]*University of Chinese Academy of Sciences, Beijing* 100049*, China;*
[3]*School of Software Engineering, Tongji University, Shanghai* 201804*, China;*
[4]*Network Integration Technology Research Department, Huawei Technologies Co., Ltd., Shenzhen* 518129*, China;*
[5]*School of Computer Science, Wuhan University, Wuhan* 430072*, China*

Received 25 January 2019/Revised 22 April 2019/Accepted 22 July 2019/Published online 1 June 2020

Abstract Software-defined networking (SDN) is a revolutionary technology that facilitates network management and enables programmatically efficient network configuration, thereby improving network performance and flexibility. However, as the application programming interfaces (APIs) of SDN are low-level or functionality-restricted, SDN programmers cannot easily keep pace with the ever-changing devices, topologies, and demands of SDN. By deriving motivation from industry practice, we define a novel network algorithm programming language (NAPL) that enhances the SDN framework with a rapid programming flow from topology-based network models to C++ implementations, thus bridging the gap between the limited capability of existing SDN APIs and the reality of practical network management. In contrast to several state-of-the-art languages, NAPL provides a range of critical high-level network programming features: (1) topology-based network modeling and visualization; (2) fast abstraction and expansion of network devices and constraints; (3) a declarative paradigm for the fast design of forwarding policies; (4) a built-in library for complex algorithm implementation; (5) full compatibility with C++ programming; and (6) user-friendly debugging support when compiling NAPL into highly readable C++ codes. The expressiveness and performance of NAPL are demonstrated in various industrial scenarios originating from practical network management.

Keywords software-defined networking (SDN), network algorithm programming language (NAPL), network abstraction

Citation Wang J, An J, Chen M S, et al. From model to implementation: a network algorithm programming language. Sci China Inf Sci, 2020, 63(7): 172102, https://doi.org/10.1007/s11432-019-2644-8

1 Introduction

The Internet has rapidly grown over the last few decades. Because of increasingly sophisticated policies and the proliferation of heterogeneous devices, underlying networks have faced a wide range of management challenges. Meanwhile, the distributive infrastructure of traditional networks statically combines the control and data flow, compelling network administrators to express their policies through complicated and frustrating interfaces. To alleviate such difficulties, researchers have developed software-defined networking (SDN) [1, 2], an emerging network architecture that decouples the routing process (control plane) logic from the forwarding process of network packets (data plane). The SDN architecture centralizes the control of data path elements independently of the network technologies used to connect

* Corresponding author (email: ganting@whu.edu.cn)

these devices that can originate from different vendors. The centralized control embeds all the intelligence and maintains a network-wide view of the data path elements and their connecting links. This centralized up-to-date view makes the controller competent to perform network management functions while allowing easy modifications to the networking functions through the centralized control plane. On this centralized controller, administrators can implement various standard network algorithms such as shortest-path routing and traffic monitoring, along with more sophisticated algorithms such as load balancing and maximum flow algorithms. For comprehensive surveys on SDN, readers are referred to [3–5] and references therein.

Although SDN has greatly enhanced network programmability[1)], a wide gap persists between the limited capability of SDN application programming interfaces (APIs) and the reality of practical network management. As observed in [6–8], existing SDN APIs are either low-level or limited in functionality, significantly increasing the cost of pre-development and implementation processes. Following the SDN programmers' urgent quests to reduce the efforts involved in implementing network algorithms, several SDN programming languages, such as NetCore [6], Frenetic [7], Flog [9], and Merlin [10], have emerged in recent years. These programming languages offer convenient programming interfaces and can be readily implemented in the prevalent SDN architecture.

The aforementioned problems can be solved by using most of the available SDN programming languages, but a challenging problem remains: both the requirements specified by network operators and properties of network devices vary frequently. Accordingly, the existing SDN programming languages cannot fulfill practical demands such as adding new properties of devices. Therefore, they cannot readily adapt to the rapid emergence of (or changes in) devices, topologies, and demands in the underlying network. For example, suppose the network operator intends to add a fresh attribute named "affinity" (a measure of fitness to perform potential services) to every existing router in the network. To incorporate this attribute, developers usually modify the language itself rather than the code of the routing algorithms, owing to a lack of abstraction and expansion mechanisms. In the aforementioned programming languages, although a device with emerging properties can be abstracted as a new class inherited from an existing class, abstracting multiple ever-emerging network devices often complicates the inheritance relationship in the network program, that is, when these new classes are used in different network algorithm modules, the code loses cohesion, hampering maintainability and reusability, as argued in [11].

In this paper, we present a network algorithm programming language (NAPL), a novel SDN programming language dedicated to fast abstraction and expansion of network devices and constraints. NAPL adopts the mechanism commonly used in the design of dynamic programming languages such as Python; that is, the attributes encoding the properties of network objects can be added, removed, or modified by simple statements during runtime. This mechanism frees the programmers from the complex inheritance relationships among classes and consequently enhances the maintainability and reusability of the code. In addition, because the grammar of NAPL follows the declarative programming paradigm, the programmer can focus on the logic of the network routing policies while ignoring the underlying implementation details. This facilitates the fast design and prototyping of network routing policies. For usability in industrial scenarios, we complete NAPL by providing a rapid programming flow from topology-based network models to maintainable C++ implementations. In brief, NAPL provides the following collection of critical high-level network programming features: (1) topology-based network modeling and visualization; (2) fast abstraction and expansion of network devices and constraints; (3) a declarative paradigm for fast forwarding-policy design; (4) a built-in library for complex algorithm implementation; (5) easy embedding of C++ libraries and code fragments (ensuring full compatibility with existing C++ implementations that are extensively deployed in the network); and (6) user-friendly debugging support during compilation into highly readable C++ codes. These features are not achievable by SDN switches (APIs) alone, or by existing SDN programming languages. NAPL also supports additional functionalities such as memory management.

As shown in Figure 1, the programming framework of NAPL can be logically decomposed into four

1) The dynamic control, change, and management of network behavior by software implemented through open interfaces is a tremendous advance from relying on closed boxes and proprietarily defined interfaces.

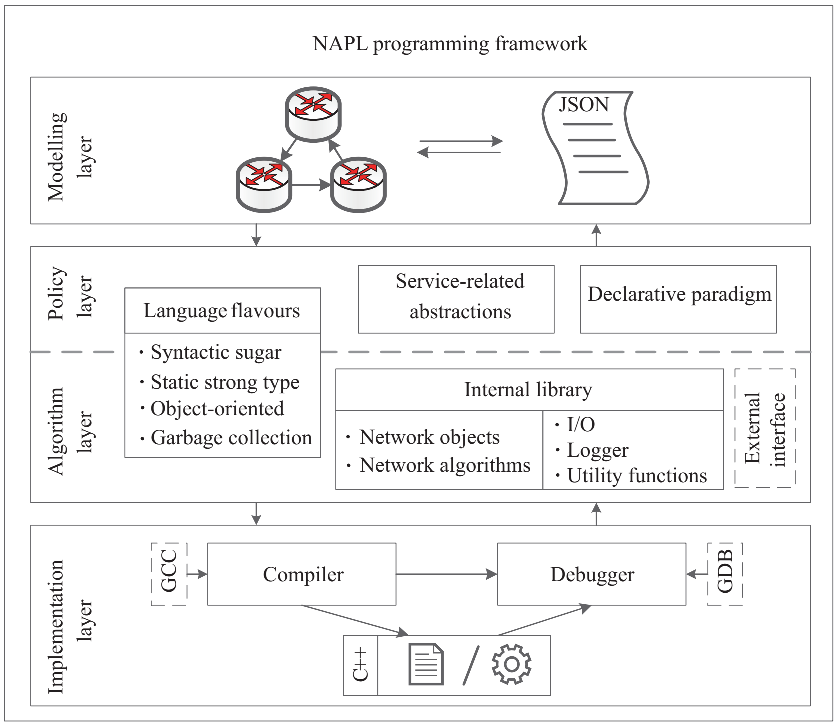


Figure 1 (Color online) Overview of the NAPL programming framework.

layers: the modeling layer, policy layer, algorithm layer, and implementation layer. The modeling layer equips NAPL with the abovementioned programming features; that is, with the topology-based network model (nodes and links) stemming from dynamical abstractions of the network equipment. The policy layer encodes a series of service-related abstractions (such as services, paths, and routing demand) on the abstracted network topology. This layer offers a declarative programming paradigm that enables succinct descriptions of service requirements and the fast design of forwarding policies. The algorithm layer, which shares sufficient syntactic sugar with the policy layer, facilitates the fast development of complex network routing algorithms, e.g., the shortest-path routing constrained by time delays and hop counts. Further, it encapsulates an internal library of standard and practical routing algorithms, such as constrained shortest path first (CSPF) [12] and disjoint protected constrained shortest path (CSPDP). Moreover, NAPL supports the free embedding of C++ code fragments and third-party libraries such as Boost[2] and LEMON[3]. With these features, NAPL is fully compatible with the existing C++ implementations extensively deployed in networks. The implementation layer embeds a compiler that converts NAPL programs into highly readable C++ codes for better maintenance and a debugger that provides user-friendly debugging of NAPL implementations.

The major contribution of this work is an NAPL with a layered structure that facilitates the fast design and implementation of network algorithms. The proposed NAPL will tremendously accelerate the prototyping of network-oriented software processes. Moreover, the expressiveness and performance of NAPL extend beyond the spectrum of state-of-the-art languages. Later, these advantages will be demonstrated in various industrial scenarios originating from practical network management.

Related work. Several network programming languages have been proposed in recent years. Based on their network operational granularity, existing languages can be divided into two categories: languages that control packet forwarding and those that operate on the network topology. Languages in the former category include Frenetic [7], Pyretic [8], NetKat [13], Nettle [14], NetCore [6], Flog [9], FlowLog [15], Maple [16], and P4 [17]. Packet-forwarding control can be applied to prevailing open-source SDN interfaces such as OpenFlow [2] and NOX [18] in a simple manner. For instance, Frenetic provides a

2) https://www.boost.org/.
3) https://lemon.cs.elte.hu/.

declarative query language for classifying and aggregating network traffic as well as a functional library for describing packet-forwarding policies. Flog is an event-driven logic programming language that generates a packet-forwarding policy after the occurrence of a network event. P4 is a strawman proposal for the future evolution of OpenFlow toward protocol independence, target independence, and reconfigurability. Although package-based languages have variable features, they cannot easily define and implement algorithms within the global framework of the underlying network topology. Moreover, programmers using these languages cannot specify attributes or impose constraints on the network objects. This deficiency is critically restrictive in networks with ever-changing devices, topologies, and requirements.

Topology-based network programming languages, such as FML [19] and Merlin [10], ease the design of the routing policy. As the first SDN programming language, FML provides a rule-based idiom for forwarding policies. FML inherits the logical programming paradigm of Datalog [20], wherein users can specify the maximum values of latency, jitter, and bandwidth. Merlin provides a declarative language framework for expressing high-level policies by using regular expressions and arithmetic formulas to define forwarding paths and bandwidth constraints. Nevertheless, these languages prohibit the fast abstraction and expansion of network constraints and objects. Furthermore, they do not support the constraints on many frequently used network attributes, such as costs, delays, and hop counts. Because such constraints pertain to network services, they differ from throughput constraints and other constraints imposed on single devices. For instance, the path-selection problem in Merlin is essentially a multi-commodity flow problem [10], which cannot solve constraints on network services. Moreover, the aforementioned languages cannot actively modify the network topology. In real networks, network elements (e.g., devices and links) are dynamically added and removed, especially when devices fail or the network topology changes.

The hierarchical programming framework in NAPL carries the declarative programming paradigm in [6, 7, 10, 15], but can express more complex routing constraints than those of [10, 19]. It also possesses exclusive features such as fast adaptation to network changes, a built-in library for complex algorithm implementations, full compatibility with C++ programming, and user-friendly debugging support during compilation into highly maintainable C++ codes.

Article structure. The remainder of this paper is organized as follows. Section 2 gives an overview of the NAPL programming framework and presents an illustrative example. The overview is followed by consecutive elaborations on the modeling layer (Section 3), policy layer (Section 4), algorithm layer (Section 5), and implementation layer (Section 6). Section 7 presents empirical results on the expressiveness and performance of NAPL in various industrial scenarios. The paper concludes with Section 8.

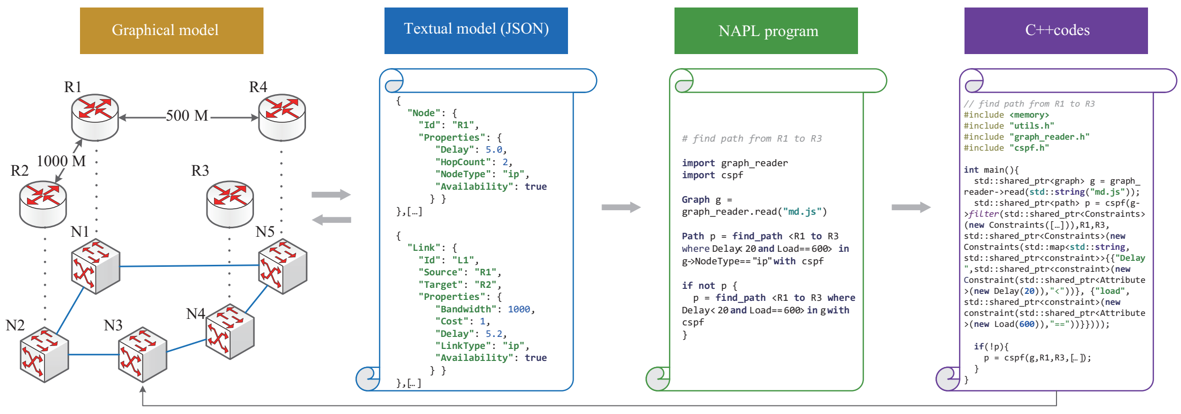
2 NAPL programming framework

The hierarchical philosophy of NAPL streamlines the software process of network management from topology-based network models to maintainable C++ implementations. This section gives an overview of the layered structure of NAPL and demonstrates its modeling, abstraction, programming, and compilation techniques by an illustrative example.

2.1 Overview of NAPL

Figure 1 depicts the hierarchical framework of network algorithm development in NAPL, which consists of four logical layers that hierarchically oversee the modeling, policy design, algorithm design, and implementation. To maintain a topology-based network model, the modeling layer abstracts the underlying network into nodes (devices such as routers, switches, and modems) and links (connections between hosts or nodes; e.g., optical fiber cables) that constitute a directed graph. Various attributes of the network are straightforwardly captured by being specified as properties in the graph. This layer supports both a graphical representation (based on the Dotty module of Graphviz [21]) and a textual alternative (based on the JSON format[4)]) of the graph model, which are dynamically synchronized during the alteration of either of the two representations.

4) https://www.json.org/.



```
{
  "Node": {
    "Id": "R1",
    "Properties": {
      "Delay": 5.0,
      "HopCount": 2,
      "NodeType": "ip",
      "Availability": true
    } }
},[…]

{
  "Link": {
    "Id": "L1",
    "Source": "R1",
    "Target": "R2",
    "Properties": {
      "Bandwidth": 1000,
      "Cost": 1,
      "Delay": 5.2,
      "LinkType": "ip",
      "Availability": true
    } }
},[…]
```

```
# find path from R1 to R3

import graph_reader
import cspf

Graph g =
graph_reader.read("md.js")

Path p = find_path <R1 to R3
where Delay< 20 and Load==600> in
g->NodeType=="ip" with cspf

if not p {
  p = find_path <R1 to R3 where
Delay< 20 and Load==600> in g with
cspf
}
```

```
// find path from R1 to R3
#include <memory>
#include "utils.h"
#include "graph_reader.h"
#include "cspf.h"

int main(){
  std::shared_ptr<graph> g = graph_
reader->read(std::string("md.js"));
  std::shared_ptr<path> p = cspf(g-
>filter(std::shared_ptr<Constraints>
(new Constraints([…])),R1,R3,
std::shared_ptr<Constraints>(new
Constraints(std::map<std::string,
std::shared_ptr<constraint>>{{"Delay
",std::shared_ptr<constraint>(new
Constraint(std::shared_ptr<Attribute
>(new Delay(20)),"<"))}, {"load",
std::shared_ptr<constraint>(new
constraint(std::shared_ptr<Attribute
>(new Load(600)),"=="))}})));

  if(!p){
    p = cspf(g,R1,R3,[…]);
  }
}
```

Figure 2 (Color online) Routing example illustrating the programming flow in NAPL. The network model (graphical or textual) covers an IP layer (with nodes R1–R4) and optical layer (with nodes N1–N5). The notation [. . .] is used in the JSON script and the C++ codes to abbreviate similar code snippets for ease of presentation. The arrow at the bottom indicates the potential queries and alterations of the network model in future executions of the C++ program.

Based on the abstracted network topology, the language kernel (composed of the policy and algorithm layers) is a key component of the NAPL framework. It recognizes abundant syntactic sugar for defining general data structures and incorporates object-oriented mechanisms while supporting additional features such as garbage collection, as those in high-level languages. In particular, the policy layer encodes a series of service-related abstractions (service, path, routing demand, etc.) while enabling succinct descriptions of service requirements and fast design of forwarding policies through its declarative programming paradigm. The algorithm layer facilitates the fast development of complex network routing algorithms, such as shortest-path routing under time delay and hop count constraints. The internal library of the algorithm layer serves as the runtime environment of NAPL. The built-in library implements all the network objects that facilitate abstractions in the above layers, along with various standard routing algorithms and practical ones originating from industrial scenarios. Additional ingredients, such as a logger, utility functions, and I/O interfacing, are provided for user-friendly interactions with programmers. Moreover, NAPL supports the free embedding of C++ code fragments and external libraries such as Boost and LEMON, rendering it fully compatible with the existing C++ implementations that are extensively deployed in modern networks.

As in the implementation layer, NAPL operates a customized compiler that compiles NAPL programs into highly maintainable C++ codes or executables and a debugger that enables user-friendly debugging directly on the NAPL programs. To maximize the benefits of existing C++ compilers and debuggers, GCC and GDB developed in the GNU project[5] are invoked. The underlying interactions are implicit and therefore transparent to users.

2.2 Illustrative example

Figure 2 illustrates the programming flow in NAPL. The presented example spans all the logical layers introduced above. The network fragment covers an IP layer (with nodes R1–R4) and optical layer (with nodes N1–N5) (see graphical model in Figure 2). In both the graphical and textural models, the interesting attributes of the devices and links (delay, hop counts, and bandwidth) are simultaneously and intuitively encoded as properties in JSON format (as`md.js` files, for instance).

Suppose a new service is issued to run in the network. The intended path of the incoming service is R1 to R3 with a delay tolerance up to 20 milliseconds, and the service must carry a load of 600 MB. Moreover, the path should be preferentially searched in the IP layer, and if no such route is detected, the underlying optical layer should be searched. This instruction is readily expressed by the following slice of the NAPL program (which operates across the policy and algorithm layers), where the declarative statement `find_path` describes the routing demand in the user-provided routing algorithm `cspf`, which

5) https://www.gnu.org/.

can be specified in either NAPL or C++.

```
# Find path from R1 to R3
import graph_reader
import cspf
Graph g = graph_reader.read("md.js")
Path p = find_path <R1 to R3 where Delay < 20 and Load == 600> in g->NodeType == "ip"
         with cspf
if not p {
  p = find_path <R1 to R3 where Delay < 20 and Load == 600> in g with cspf
}
```

By simply dropping the statements `import cspf` and `with cspf`, the user may alternatively resort to the built-in CSPF algorithm provided in NAPL's internal library. Moreover, an initially buggy NAPL program can be adjusted using the NAPL debugger, then compiled to a maintainable C++ implementation and further to an executable file that interacts with the underlying network model. In the present case, the optimal deployment path of the incoming service was detected as R1 $\rightarrow$ R2 $\rightarrow$ N2 $\rightarrow$ N3 $\rightarrow$ N4 $\rightarrow$ R3.

Note that even this rather simple scenario reveals the high-level abstraction capability of NAPL, which facilitates the fast and flexible design of network algorithms.

3 Modeling layer

The NAPL modeling language aligns the graphical representation and a textual alternative in an intermediate structure called an attributed directed graph (ADG), which captures the dynamically changing network topology and attributes of the devices and links therein.

Definition 1 (Attributed directed graph)**.** Let Λ be a finite set of attributes. An attributed directed graph over Λ takes the form $\mathcal{G} = \langle V, E, L\rangle$, where V is a finite (non-empty) set of vertices, $E \subseteq V \times V$ is a set of ordered vertex pairs called edges, and $L \subseteq (V \cup E) \times 2^{\Lambda}$ is a labeling function that associates each vertex and edge with a set of attributes.

The finite set of attributes in a network model is formulated as $\Lambda = \{\langle \alpha_k, \beta_k\rangle \mid k \in \mathbb{N}_{>0}\}$, where the kth α denotes the exclusive name of attribute k and the kth β is its corresponding value. The values can be (for example) natural numbers, Booleans, real numbers, or string identifiers. It is worth highlighting that the network topology and associated attributes carried by the underlying ADG can both be easily and arbitrarily extended or modified, enabling fast reactions and flexible adaptations to the ever-changing network devices and requirements.

Several service-related concepts can be easily defined on top of an ADG. For example, a path is a finite consecutive sequence of distinct vertices, $p = v_0v_1 \dots v_k$, such that for all $0 \leqslant i \leqslant k-1$, $(v_i, v_{i+1}) \in E$; a demand (defined later) encodes certain routing constraints over the ADG; and a service, as an abstracted network service (application), encloses a group of demands, priority enabling preemption, load specifying the required bandwidth, and a set of paths on which the service is running (empty before the service is deployed).

3.1 Textual representation

The textual representation shown in Figure 2 consists of two sections encoding the vertices and edges as nodes and links, respectively, in the universal JSON format. Each entity in the representation has a unique identifier Id, while each link has additional fields called source and target that expand the network topology. All primitive attributes of the underlying devices and links are encapsulated as properties. Some of the properties frequently used in network modeling are listed in Table 1.

3.2 Graphical representation

NAPL offers both textual and graphical representations of network models. The graphical alternative, which is dynamically synchronized with the textual representation, enables intuitive interactions between

Table 1 Commonly used network properties

Name	Type	Description
NodeType	String	Type of the node (“ip”/“optical”)
LinkType	String	Type of the link (“ip”/“optical”)
Bandwidth	Float	Bandwidth of the link
HopCount	Integer	Number of hops through the device
Cost	Float	Cost of the network device or link
Delay	Float	Delay of the network device or link
Availability	Boolean	Availability of the device or link

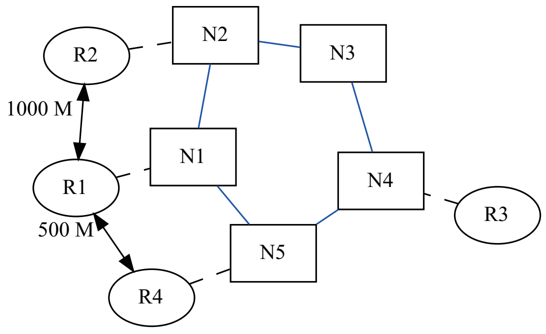


Figure 3 (Color online) Dotty graph depicting the model in Figure 2. Most of the attributes are specified implicitly.

the programmer and network model. For visualization, we selected the Dotty module of Graphviz (Figure 3) because its programmable interactive feature inherently supports on-the-fly topology alteration and attribute update.

4 Policy layer

The policy layer of the NAPL framework provides a collection of constructs by which users can specify the intended network behavior at a high level of abstraction. In this section, we give a comprehensive account of the syntax and interpretation of the NAPL language, which facilitates the fast generation of network routing policies for the network operator providing the network services.

The NAPL syntax encodes two types of grammar: the general grammar of general-purpose languages and a network grammar dedicated to typical network algorithm statements.

4.1 General grammar

The general grammar of NAPL supports a variety of syntactic sugars (e.g., containers and assertions). For example, the syntax for building and iterating over a list is

```
list<int> s = [1, 2, 3, 4]
list<int> ss = [2 * i for int i in s if i % 2 == 0].
```

The general grammar can also incorporate object-oriented programming features such as inheritance, encapsulation, and polymorphism:

```
class A{
  public int a = 0
  def public getA() -> int{return this.a}
  def public setA(int a){this.a = a}
}
class B:A{
  def public getA() -> int{return this.a + 2}
}
A a = new B()
print(a.getA()) # The output is 2
```

Upon ignoring the limitation of finite memory, NAPL is Turing-complete in the sense that its expressiveness equals that of any general-purpose language. Accordingly, NAPL can describe and solve a wide range of algorithmic problems in network programming.

4.2 Network grammar

Network grammar enables the high-level abstraction of forwarding policies and complex network algorithms. The syntax of a network-related statement S is inductively defined in the following Backus-Naur form (BNF):

$$\begin{array}{lr|lr}
S ::= S_{\mathcal{G}} \mid S_{\Lambda} \mid S_{\mathcal{P}} & & \mathrm{objs} ::= \mathrm{obj} \mid \mathrm{objs}, \mathrm{objs} & \\
S_{\mathcal{G}} ::= \mathcal{G} \sim \mathrm{objs} & \text{(residual graph)} & \mathrm{obj} ::= n \mid l \mid s & \text{(node, link, or service)} \\
\qquad \mid \mathcal{G} \to \mathrm{cons} & \text{(max. constrained subgraph)} & \mathrm{cons} ::= \mathrm{con} \mid \texttt{not}\ \mathrm{cons} \mid \mathrm{cons} \diamond \mathrm{cons} & (\diamond \in \{\texttt{and}, \texttt{or}\}) \\
S_{\Lambda} ::= \mathrm{obj} \leftarrow \langle \alpha, \beta \rangle & \text{(add fresh attribute)} & \mathrm{con} ::= \alpha \rhd \beta & (\rhd \in \{==, <, >, <=, >=\}) \\
\qquad \mid \mathrm{obj} \to \alpha = \beta & \text{(update attribute)} & \mathrm{dmds} ::= \mathrm{dmd} \mid \mathrm{dmds}, \mathrm{dmds} & \\
S_{\mathcal{P}} ::= \texttt{find\_path}\ \langle \mathrm{dmds} \rangle\ \texttt{in}\ \mathcal{G}\ [\texttt{with}\ f] & & \mathrm{dmd} ::= n \to n\ [\texttt{min}\ \omega]\ [\texttt{where}\ \mathrm{cons}] & \text{(demand)}
\end{array}$$

Here $\mathcal{G}$ is an ADG, $\langle \alpha, \beta \rangle \in \Lambda$ is the key-value pair denoting an attribute, f represents a function handle to a certain user-specified routing algorithm, and ω is a function handle that calculates the weight of a given path as the objective value to be optimized over all path candidates. The optional components in the syntax are enclosed in $[\cdot]$.

Being tailored to network manipulations, the NAPL syntax exploits a declarative programming paradigm built over highly abstracted components, including objs (a collection of network objects), cons (Boolean combinations of attribute constraints), and dmds (a group of constrained demands optimized with respect to a certain weight measure). These components will be demonstrated with an example in the subsequent section. The following subsections elaborate on several important network-related statements, which substantially facilitate the convenient design of network routing policies and deliver prompt reactions to dynamically changing network devices and demands.

4.2.1 *Attribute statements* (S_{Λ})

The attribute statement encloses two production rules: obj $\leftarrow \langle \alpha, \beta \rangle$ for adding a fresh attribute and obj $\to \alpha = \beta$ for updating an existing attribute. The former inserts an attribute $\langle \alpha, \beta \rangle$ into a specified object obj. This operation is legal only if obj does not contain an attribute α. The latter changes the value of attribute α in obj to β only if attribute α already exists in obj.

The abstracted ADG network modeling enables straightforward NAPL operations, such as retrieving, updating, and appending on the attributes encoded in the network objects. This capability greatly assists practical network management, and is particularly useful in device upgrade and routing customization. For example, the network operator can distinguish the fitness of a service on different nodes and links by simply adding an "Affinity" attribute to all network objects and by constraining this attribute in the `find_path` statement.

4.2.2 *Graph statements* ($S_{\mathcal{G}}$)

The graph statement includes two production rules. The rule $\mathcal{G} \sim$ objs obtains the residual graph after removing objects objs from the underlying graph $\mathcal{G}$. Note that when a node n in objs is removed, all the links connected to n are removed simultaneously. The other rule, $\mathcal{G} \to$ cons, returns the maximal subgraph of the network components subjected to a given group of constraints cons. For instance, the graph generated by the statement `g -> NodeType == "ip"` is the residual graph after removing all network objects in `g`, whose `NodeType` attribute (if it exists) is not valued as `"ip"`.

The graph statements tackle frequently used logical (virtual) restrictions on ADGs. In particular, $\mathcal{G} \sim$ objs removes specified network components from the underlying graph and $\mathcal{G} \to$ cons obtains the maximum subgraph satisfying the given constraints. In most cases, only an intended part of the network needs to be explored rather than the whole network. Therefore, besides improving the flexibility of the routing procedure, these statements simplify the design of the routing algorithms. The general grammar of NAPL also enables physical modifications of the network models. As these modifications are trivial, they are omitted here.

4.2.3 *Routing statements* ($S_{\mathcal{P}}$)

The routing statement `find_path` $\langle$dmds$\rangle$ `in` $\mathcal{G}$ [`with` f] returns one or more paths (depending on f) in $\mathcal{G}$ that fulfill dmds. The path(s) is (are) found by an optionally specified routing algorithm f.

As most network manipulations can be formulated as a path-selection problem addressed at the centralized controller, routing functionality is a core statement in almost all SDN programming languages. The declarative `find_path` function provided by NAPL describes a domain-specific requirement in the form of demands, i.e., dmds, specifying the properties of the target path. This declaration frees the programmer from otherwise highly intertwined network models and implementation details while retaining control of the routing approach by a routing algorithm f if desired and specified by the user. The default routing algorithm written in NAPL's internal library implements CSPF using breadth-first search. Furthermore, the `find_path` statement competently solves the multi-service path-selection problem (see the case study in Section 7) by allowing routing under multiple demands, each with multiple constraints that may interfere mutually.

5 Algorithm layer

Rapidly emerging new algorithms and mutable requirements have necessitated the detailed control of network algorithms. Unlike the policy layer, which requires a high level of abstraction for the fast design of forwarding policies, the algorithm layer needs sufficient flexibility for the implementation of complex algorithms. For this purpose, developers can access the built-in library and third-party libraries imported through an external interface.

5.1 Internal library

The internal library, which serves as the runtime environment of NAPL, mainly comprises the network object encoding, a handful of standard and practical routing algorithms, and several additional ingredients such as a logger, utility functions, and I/O interfacing. The internal library is seamlessly linked to the C++ codes compiled from NAPL programs, which are further compiled into executable files by the GCC.

5.1.1 *Network objects*

The network objects in NAPL (ADGs, demands, and services) are abstractions of the network devices, topologies, and service-related constructs. These abstractions are naturally declared in the library as C++ objects taken from corresponding classes with previously mentioned patterns. Obviously, such abstractions and encapsulations allow the developer to design forwarding policies in the policy layer without bothering about the low-level details.

5.1.2 *Network algorithms*

Recall that the `find_path` statement of NAPL requests an algorithmic function definition. Without loss of generality, consider a routing statement with a single demand between a source node `A` and target node `B`:

```
find_path <A -> B min func_weight where cons> in g with func_route
```

that generates a function call of the form:

```
func_route(g, new Demand(A, B, func_weight , cons)).
```

The function `func_route` should either be predefined by the user, or called from the internal library. The default routing algorithms in the internal library are CSPF, CSPDP, and point to multiple points (P2MP). CSPF is a fundamental extension of the shortest-path algorithm that seeks the optimal path fulfilling a set of network constraints. The CSPDP algorithm improves the robustness of networks, especially against device failures. This frequently used strategy searches for an active path and a completely separate standby path that diverts the network flow when failures occur on the main path. Meanwhile, the P2MP algorithm is a superior option for finding multiple paths sharing the same source node while minimizing the number of links.

5.1.3 *Utility functions*

The internal library also provides a family of generic algorithms and utility functions that facilitate the NAPL programming process. The generic algorithms perform basic functions such as sorting and pattern-based selecting over containers. The utility functions involve syntactic sugars such as `len()` for obtaining the length and `range()` for generating an integer sequence.

5.1.4 *I/O and logger*

NAPL dedicates an I/O package not only to general input/output methods (such as the console and file I/Os), but also to the I/O interfacing of textual and graphical network models. Furthermore, the logging package exports logs at different levels, generating debugging logs, warning logs, error logs, regular information logs, and a special log depicting the state of the underlying ADG. The I/O and logger usages are demonstrated in the following slice of NAPL code:

```
GraphReader reader = new GraphReader()
Logger logger = new Logger("out_path")
DottyWriter writer = new DottyWriter("out.dot")
Graph g = reader.read("md.js")
logger.debug(g)  # Log the ADG info. for debugging
writer.write(g)  # Save the ADG into a Dotty format
```

5.2 External interface

As mentioned previously, NAPL supports the free embedding of raw C++ code fragments and precompiled libraries such as Boost[2)] and LEMON[3)]. Therefore, it is fully compatible with existing C++ implementations, which are particularly common in SDN controllers. For example, the code fragment

```
import lemon/list_graph
import lemon/dijkstra
# [...] NAPL definitions of g, len, src, tgt
CPP{
  Dijkstra<Graph, LengthMap> dijkstra(g, len);
  dijkstra.run(src);
  std::shared_ptr<std::vector<int>> path(new std::vector<int>());
  # Store the shortest path
  for (Node v=tgt; v!=src; v=dijkstra.predNode(v)){
    path->push_back(g.id(v));
  }
  path->push_back(src);
}CPP
# Get the results back to NAPL
for int i in 1: path.size() + 1 {
  print(path[path.size() - i]);
}
```

illustrates how C++ code can be intuitively embedded in NAPL using the `CPP{}CPP` identifiers by reusing the existing Dijkstra's algorithm written in LEMON, a well-known precompiled C++ library. Note that the interaction between NAPL and C++ is indeed straightforward because the shared variables can be used in a sequential manner.

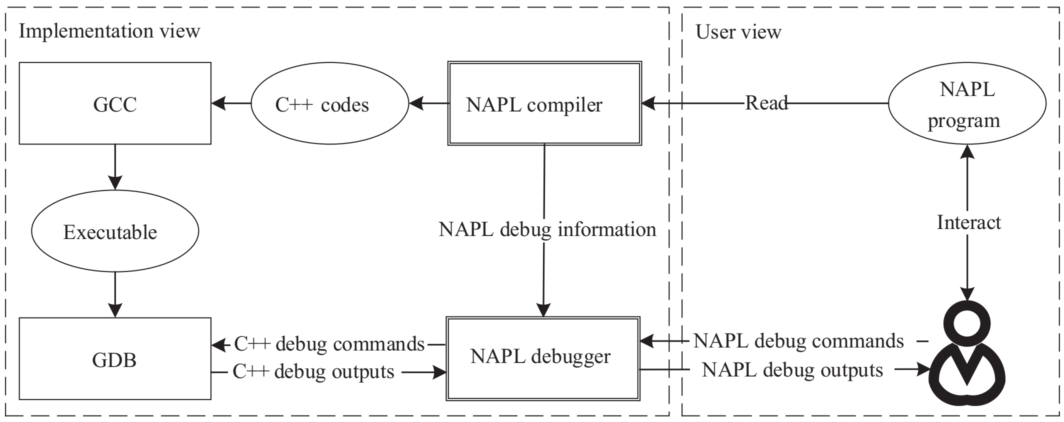


Figure 4 User interface for the NAPL implementation schemes.

6 Implementation layer

The implementation layer handles the fundamental techniques that support the entire NAPL hierarchy. As depicted in Figure 4, the user interacts with the implementation schemes by composing NAPL programs for compiling and exchanging debug commands/outputs for debugging. The NAPL compiler and debugger cooperate with GCC and GDB to produce intermediates such as maintainable C++ codes and executable files, which are also readily accessible by the user. This section focuses on the design details of the implementation layer, especially of the compiler and debugger.

6.1 Compiler design

The compiler is the most crucial part of the compile-edit-debug cycle. The main tasks of the NAPL compiler are to embed the NAPL grammars and semantics and thereafter convert the NAPL program into readable C++ codes. Instead of constructing the NAPL compiler from scratch, which is complicated and error-prone, we employ the parser generator PLY[6)], which implements the lex and yacc [22] parsing tools. PLY uses reasonably efficient LR-parsing and supports empty productions, precedence rules, error recovery, and ambiguous grammars, making it particularly suitable for rapidly constructing compilers dedicated to domain-specific languages. The PLY library was customized to the NAPL grammar by introducing type inference for static strong typing, and by tailoring the exception-handling procedure for more intuitive construction. The NAPL grammar is encoded by the customized PLY and transferred to the NAPL compiler.

Following the high-level abstraction of network objects, the NAPL compiler produces well-structured C++ programs while preserving almost all the names, types, and annotations of the original NAPL program. Therefore, the generated C++ codes are easily maintainable and can be further compiled, together with the internal library of NAPL, into executable modules by GCC. Moreover, the debugging mode of the NAPL compiler provides adequate debugging information, primarily, the mapping relation from the program statements of NAPL to those of the derived C++ programs.

6.2 Debugger design

The debugger in the NAPL framework is dedicated to the direct, user-friendly debugging of NAPL programs. Unfortunately, most SDN programming languages lack this feature. Indeed, debugging the generated C++ codes and executables is trivial, but mapping the C++ debug outputs back to the NAPL level is difficult and burdensome. Being motivated by this problem, we tailored an NAPL debugger that tracks the dynamic behavior of the NAPL program.

Because a debugger must understand the runtime context of the program being debugged, we incorporate GDB in a shell that maps the NAPL debug commands/outputs to the C++ debug commands/out-

6) http://www.dabeaz.com/ply/.

puts. As shown in Figure 4, the debugger checks the validity of a debug command from the user. If the command is valid, it is converted to the corresponding GDB command based on the debug information fed by the compiler; otherwise, it is discarded. A GDB instance then assumes control the debug command and returns outputs that can be fetched and converted back to the NAPL level by the debugging shell. The user interacts only with the NAPL program without observing the underlying communications.

We have implemented several useful debug commands, including (but not limited to) list, break, break-info, run, continue, next, and print. A registration mechanism enables convenient registration and implementation of new debug commands in NAPL.

6.3 Additional flavors

6.3.1 *Duck typing*

NAPL allows arbitrary definitions of attributes over various network objects. This flexibility indicates a dynamic feature in a strongly statically typed language. It is achieved by introducing the duck typing idea into a nominative-type system, by which SDN programmers can concisely abstract new network devices. In brief, one can simply define a subclass that inherits `sdn_object` for a new network object with default attributes and a subclass that inherits `Attribute` (while overriding the "getter/setter") if a fresh attribute is required.

6.3.2 *Garbage collection*

The lack of automatic memory management is a recognized barrier to the fast development of C++ codes, as programmers must manually cope with frequent memory allocation/deallocation. Therefore, in the C++ codes compiled from NAPL, we incorporate a garbage collection technique that performs reference-counting by a so-called smart pointer using the `std::shared_ptr` class. This feature enables automatically shared memory management of dynamically allocated objects.

7 Evaluation

This section demonstrates the expressiveness and performance of NAPL in different benchmarking scenarios originating from practical network management at Huawei Technologies Co., Ltd. in Guangdong, China. All experiments were performed on a 2.7-GHz Intel Core-i7-6820HQ processor with 16 GB RAM, running 64-bit Deepin 15.5.

7.1 Expressiveness of NAPL

In three case studies, we demonstrate that NAPL can adequately express demands, implement algorithms, and concisely handle network mutations. All these tasks are difficult to execute with state-of-the-art SDN programming languages.

7.1.1 *Multi-constrained shortest path first*

Description. In network traffic engineering, CSPF is one of the most commonly used methods for computing the optimal network route fulfilling a set of constraints. To achieve fine-grained traffic engineering, multiple constraints must be simultaneously considered, including (but not limited to) constraints on bandwidth, delay, and hop count. However, introducing multiple constraints inevitably complicates the algorithm and renders it inextricable for the service provider (aka, the network operator). Accordingly, the hierarchical framework of NAPL logically separates the policies and algorithms, meaning that SDN developers (or equipment suppliers) can write algorithms in the algorithm layer while the service provider designs the routing strategy concisely in the policy layer.

Specification. Consider a routing demand that seeks a path from node `A` to `B` under the following constraints:

- The service holds a load of 600 MB;

• The overall delay tolerance is up to 40.0;

• The overall hop count should be less than or equal to 10.

The optimization objective is to minimize the trade-off between total cost and delay with weights of 0.4 and 0.6, respectively.

NAPL implementation. The above objective can be achieved by declaring the routing demand as

```
import graph_reader
import obj_weight
Graph g = graph_reader.read("md.js")
Path p = find_path <A to B min obj_weight where Load == 600 and Delay < 40.0 and
         HopCount <= 10> in g
```

and implementing a `obj_weight` function that captures the weighted objective to be optimized (given below).

```
def obj_weight(Path p)->float{
  float weight = 0
  for Node n in p.get_nodes(){
    weight += 0.4 * n->Cost + 0.6 * n->Delay
  }
  for Link l in p.get_links(){
      weight += 0.4 * l->Cost + 0.6 * n->Delay
  }
  return weight
}
```

This case demonstrates the cardinal scenario of NAPL network routing in the presence of multiple coexisting constraints and an optimization objective.

7.1.2 *CSPDP with service preemption*

Description. In practical network traffic engineering, demands are scheduled sequentially on a first-come-first-serve basis, and preemption is permitted to enhance the effectiveness of the network. In the CSPDP scenario, where the active path and standby alternative may be subjected to distinctly different constraints under different quality of service requirements, squeezing all the constraints into a single demand is implausible. To solve this problem, NAPL filters out (through the residual graph computation) the deployed services that cannot be preempted ahead of routing under multiple parallel demands.

Specification. Suppose several services currently running in the network are prioritized from 1 to 3, and we wish to schedule an active and a standby path for a freshly incoming service under the following constraints:

• The new demand can take only the paths of services with priorities less than 2;

• The acceptable delay of the active path is up to 20.0;

• The active and standby paths of the fresh service each carry a 600-MB load.

NAPL implementation. CSPDP with service preemption in NAPL is implemented by the following program:

```
import graph_reader service_reader
import cspdp
Graph g = graph_reader.read("md.js")
list<Service> services = service_reader.read("deployed_services.js")
list<Service> services = [s for s in services if s.priority >= 2]
# Rule out non-preemptible deployed services
for s in service_list{
  g = g ~ s
}
# Route multiple paths w.r.t. multiple demands
Path p = find_path <A to B where Load == 600 and Delay < 20.0, A to B where Load==600>
         in g with cspdp
```

Note that the above program is simplified by the syntactic sugar embedded in NAPL for building and iterating over lists, as previously discussed for the general grammar.

7.1.3 *Fast adaptation to attribute extensions*

Description. Because of the omnipresence of rapid device changes in networks, fast abstraction and expansion of the underlying network and algorithms is critically important. In the conventional development flow of network algorithms, a device with emerging attributes is usually abstracted as a new class inherited from an existing one. However, this abstraction often complicates the inheritance relationship in the network program; consequently, when these classes are used in different network algorithm modules, the code loses cohesion and cannot be maintained and reused, as argued in [11]. In contrast, NAPL users can dynamically append attributes to an existing network object without altering any class definitions, which unifies the network object abstraction and preserves the maintainability and reusability of the code.

Specification. Assume that the CSPF routing presented in Subsection 7.1.1 has been implemented in the network and the operator intends to specify an "Affinity" attribute to each network element. The new attribute characterizes the fitness of each element to potential services during the CSPF-based routing.

NAPL implementation. The above specification can be trivially accomplished by assigning a binary-valued attribute "Affinity" to all network objects, e.g., `<Affinity,01>` for a node and `<Affinity,11>` link. The attribute can be assigned through either the S_Λ statement or the model file. A clause interpreting an XOR gate (taken from C++) is attached over this attribute, and a binary filter is applied to the original constraints in the `find_path` statement. Note that this adaptation is minimal because no class definitions or algorithmic implementations are modified; accordingly, the NAPL program retains its maintainability and reusability despite the ever-changing attributes in the network.

7.2 Performance of NAPL

In this subsection, we evaluate the performance of NAPL in terms of the runtime overhead and the size of code using a set of 37 NAPL benchmark programs (cases) in five categories (Table 2): general grammar, external interfacing, network objects, network statements, and network algorithms. As the category names suggest, these cases cover different aspects of NAPL; for instance, the general grammar category consists of 11 code fragments written in the general grammar of NAPL. Note that the network algorithms category comes from real world tasks while the other categories thoroughly cover the grammatical features of NAPL. Table 2 shows the performance evaluations of the abovementioned benchmarks. The numerical values are explained in subsequent subsections.

7.2.1 *Runtime overhead*

The NAPL program is compiled in C++ prior to execution, and therefore the runtime overhead should not far exceed that of a program encoded directly in C++. Therefore, we compared the runtimes of C++ codes converted automatically from NAPL and those written by well-trained network programmers. Both sets of codes accomplished the same functionality. As depicted in Figure 5, although the time consumptions largely depended on the scales of the various cases, the difference between the runtimes of both codes was almost zero for each given case. This result confirms that NAPL programming incurs insignificant runtime overhead, and thus its efficiency is comparable with that of C++ programming.

7.2.2 *Size of code*

To demonstrate that NAPL is capable of expressing complex manipulations with less code while maintaining high readability of the generated C++ program, we compare the sizes of the NAPL program and derived C++ program in terms of lines of code (Figure 6(a)) and number of bytes of code (Figure 6(b)), respectively. Note that although the internal libraries and user-implemented algorithms were both implemented in C++, only the sizes of the user-implemented programs were compared between NAPL and C++, because the functions of the built-in libraries can be invoked directly in NAPL, not in C++. Both the lines and bytes of the general grammar and external interfacing codes in NAPL were similar to those generated in C++, confirming that the compilation preserves the structure and readability of C++ code. However, the codes of network objects, network statements, and network algorithms were considerably

Table 2 Performance evaluation of NAPL with a set of benchmark NAPL programs

Category	ID	Name	Lines of code		Bytes of code		Time (ms)	
			NAPL	C++	NAPL	C++	NAPL	C++
General grammar	1	Basic type	60	72	1357	1784	2	2
	2	Complex type	94	116	2219	2763	2	2
	3	Data structure	98	154	1941	3438	78	74
	4	Expression	57	79	1050	1819	23	22
	5	Function	60	72	872	1271	22	21
	6	I/O	24	99	830	2886	20	19
	7	Statement	71	83	1448	1715	12	12
	8	Class definition	42	54	753	1197	18	17
	9	Polymorphism	40	52	452	914	13	12
	10	Attribute	81	93	2078	2625	20	19
	11	Comprehensive examples	211	331	4166	6792	2	2
External interfacing	12	C++ code plug-in	28	40	652	1062	2	2
	13	Dynamic library calls	6	18	328	586	3	3
	14	Third-party library calls	52	68	1594	1886	3	3
	15	Boost	67	84	2555	3006	6	6
	16	Lemon	67	85	1872	2161	6	6
Network objects	17	Custom attributes	41	768	900	16533	14	13
	18	Custom demands	24	254	548	6205	16	15
	19	Graph	28	844	816	24376	26	25
	20	Link	43	817	779	17802	29	28
	21	Node	42	769	704	17105	21	20
	22	Path	16	1146	354	25518	21	20
	23	Service	25	378	690	8635	26	24
	24	Network	21	1289	508	33290	3	3
Network statements	25	Network statements	67	1920	3368	62648	20	19
	26	Routing statements	72	1792	3469	57172	23	22
	27	Network object operations	23	1508	609	40155	28	27
	28	Others	21	1856	515	47295	12	11
	29	Weight	67	1197	1409	27213	16	15
	30	Statements in network algorithms	49	1997	3014	62640	4	4
	31	Attributes of network objects	50	1535	1306	44032	5	5
Network algorithms	32	CSPF	14	1671	485	51745	30	29
	33	CSPDP	14	2170	492	62675	65	62
	34	IP + optical	19	1893	586	56709	14	13
	35	Optical	111	1430	3294	39366	14	13
	36	P2MP	157	2347	4404	68280	76	72
	37	Recovery	22	1914	734	56277	11	10

smaller than the C++ codes, indicating the high abstraction power of NAPL. This improvement was particularly notable in the network-related cases (ID. 17–37), in which NAPL reduced the (weighted averaged) line numbers and byte sizes by 32 and 28 times respectively, compared with those of C++.

8 Conclusion

This paper presents the design and implementation of NAPL, which is a novel network algorithm programming language that enriches the SDN framework by enabling a rapid programming flow from topology-based network models to C++ implementations. To overcome the critical problems of programming SDN controllers, NAPL introduces high-level abstraction, flexible service routing, and convenient built-in libraries. With these tools, users can easily modify, maintain, and reuse existing implementations.

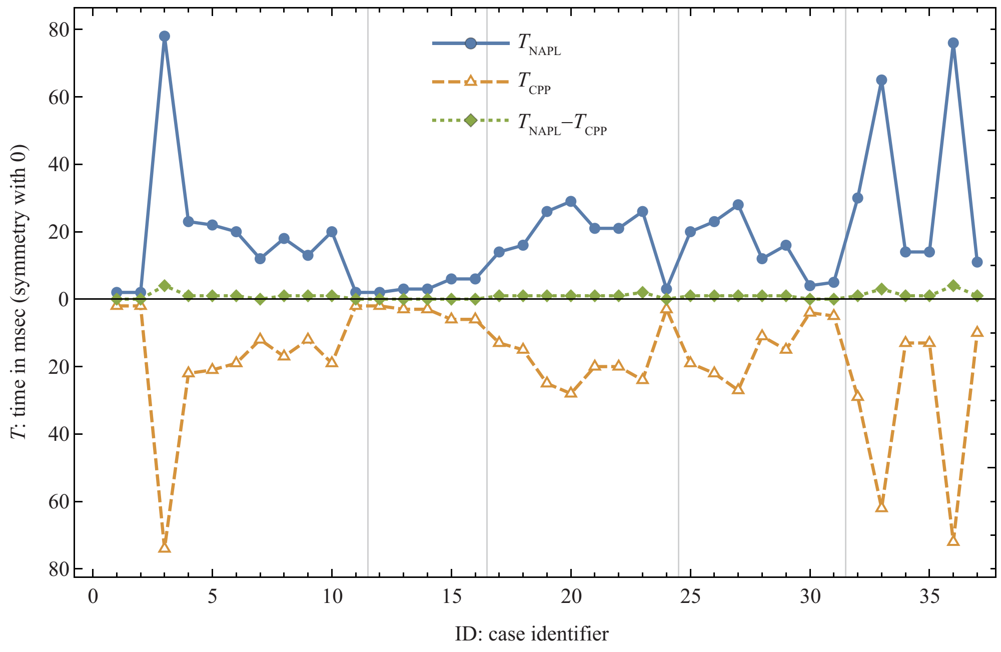
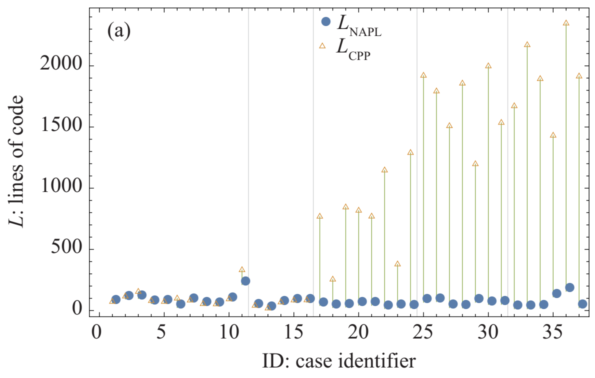


Figure 5 (Color online) Runtimes of the C++ codes compiled from NAPL ($T_{\rm NAPL}$) and those written by experienced programmers ($T_{\rm CPP}$), as well as the overhead in between ($T_{\rm NAPL} - T_{\rm CPP}$). The vertical lines delineate the different case categories.



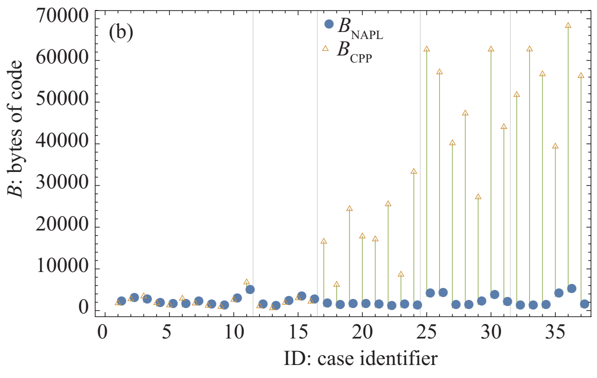


Figure 6 (Color online) Number of lines (a) and bytes (b) in NAPL code ($L_{\rm NAPL}$) and the derived C++ ($L_{\rm CPP}$). The blue lines highlight the differences between the two codes.

The high expressiveness, negligible overhead, and promising savings of programming efforts in NAPL are demonstrated in industrial benchmarking scenarios.

In future studies, we plan to automate the synthesis of NAPL programs in the policy layer for a given network model and specification, which can provide a promising correct-by-construction methodology. We are also pursuing formal verification techniques, which will ensure that the underlying network configuration meets the high-level requirements of the operator.

Acknowledgements This work was supported partly by Huawei (Grant No. YBN2016030056). Jian WANG, Jie AN, Mingshuai CHEN, and Naijun ZHAN were supported partly by National Natural Science Foundation of China (NSFC) (Grant Nos. 61625206, 61732001). Jie AN and Miaomiao ZHANG were supported partly by National Natural Science Foundation of China (NSFC) (Grant No. 61472279). The authors would like to thank the anonymous reviewers for their insightful comments and suggestions raised in the reviewing process.

References

1 McKeown N. Software-defined networking. In: Proceedings of Keynote Talk at the 28th Conference on Computer Communications, Valencia, 2009

2 McKeown N, Anderson T, Balakrishnan H, et al. Openflow: enabling innovation in campus networks. SIGCOMM Comput Commun Rev, 2008, 38: 69–74

3 Kreutz D, Ramos F M V, Verissimo P E, et al. Software-defined networking: a comprehensive survey. Proc IEEE, 2015, 103: 14–76

4 Nunes B A A, Mendonca M, Nguyen X N, et al. A survey of software-defined networking: past, present, and future of programmable networks. IEEE Commun Surv Tut, 2014, 16: 1617–1634

5 Hakiri A, Gokhale A, Berthou P, et al. Software-defined networking: challenges and research opportunities for future Internet. Comput Netw, 2014, 75: 453–471

6 Monsanto C, Foster N, Harrison R, et al. A compiler and run-time system for network programming languages. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Philadelphia, 2012. 217–230

7 Foster N, Harrison R, Freedman M J, et al. Frenetic: a network programming language. In: Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, Tokyo, 2011. 279–291

8 Monsanto C, Reich J, Foster N, et al. Composing software-defined networks. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, Lombard, 2013

9 Katta N P, Rexford J, Walker D. Logic programming for software-defined networks. In: Proceeding of Workshop Cross Model Design Validation, Copenhagen, 2012

10 Soule R, Basu S, Kleinberg R, et al. Managing the network with Merlin. In: Proceeding of the 12th ACM Workshop on Hot Topics in Networks, College Park, 2013

11 Hunt J. Inheritance considered harmful! In: Guide to the Unified Process featuring UML, Java and Design Patterns. Berlin: Springer, 2003

12 Ziegelmann M. Constrained Shortest Paths and Related Problems — Constrained Network Optimization. Saarbrücken: VDM Verlag, 2007

13 Anderson C J, Foster N, Guha A, et al. Netkat: semantic foundations for networks. SIGPLAN Not, 2014, 49: 113–126

14 Voellmy A, Hudak P. Nettle: taking the sting out of programming network routers. In: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages, Austin, 2011. 235–249

15 Nelson T, Guha A, Dougherty D J, et al. A balance of power: expressive, analyzable controller programming. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, 2013. 79–84

16 Voellmy A, Wang J, Yang Y R, et al. Maple: simplifying SDN programming using algorithmic policies. In: Proceedings of ACM SIGCOMM 2013 Conference, Hong Kong, 2013. 87–98

17 Bosshart P, Varghese G, Walker D, et al. P4: programming protocol-independent packet processors. SIGCOMM Comput Commun Rev, 2014, 44: 87–95

18 Gude N, Koponen T, Pettit J, et al. NOX: towards an operating system for networks. SIGCOMM Comput Commun Rev, 2008, 38: 105–110

19 Hinrichs T L, Gude N, Casado M, et al. Practical declarative network management. In: Proceedings of the 1st ACM SIGCOMM Workshop on Research on Enterprise Networking, New York, 2009

20 Huang S S, Green T J, Loo B T. Datalog and emerging applications: an interactive tutorial. In: Proceedings of ACM SIGMOD International Conference on Management of Data, Athens, 2011. 1213–1216

21 Gansner E R, North S C. An open graph visualization system and its applications to software engineering. Softw-Pract Exper, 2000, 30: 1203–1233

22 Levine J R, Mason T, Brown D. Lex & Yacc. 2nd ed. Sebastopol: O'Reilly, 1992