dot2net: A labeled graph approach for template-based configuration of emulation networks

Satoru Kobayashi
Okayama University
sat@okayama-u.ac.jp

Ryusei Shiiba
Sokendai
siiba@nii.ac.jp

Ryosuke Miura
NICT
r-miura@nict.go.jp

Shinsuke Miwa
NICT
danna@nict.go.jp

Toshiyuki Miyachi
NICT
miyachi@nict.go.jp

Kensuke Fukuda
NII/Sokendai
kensuke@nii.ac.jp

***Abstract*—Network emulation is an effective approach to ensure sustainable and reliable network services by verifying the correctness and fault tolerance of them. However, deploying and modifying emulation networks with existing platforms is time-consuming and prone to cause configuration errors because existing emulation platforms do not provide a suitable method for scalable network configuration. To overcome this problem, we propose the design and implementation of dot2net, a template-based platform for simple, scalable, and expressive configuration of emulation networks. The key idea is to separate network configuration into network topology as a labeled graph and label definitions as config template blocks. We evaluate the performance and efficiency of config file generation and show that dot2net is particularly effective at scaling the network topologies. We also demonstrate the expressiveness of dot2net for complicated networks and advanced technologies with test emulation networks of FRR, a widely used router software.**

***Index Terms*—Configuration management, Emulation network, Topology graph, Config template**

I. Introduction

The digital twin is a valuable concept that is a digital duplicate of a real-world object available for digital uses such as analysis and verification [1]. In the context of network management, network emulation is a suitable choice to achieve this digital twin [2]. It is helpful for operators to deploy emulation networks corresponding to the production networks in virtual environments because they enable operators to verify networks more dynamically, such as modifying configurations and injecting failures [3], [4].

However, constructing an emulation network is a time-consuming task [5]. Usually, we cannot use production network configuration directly for the emulation network because we often need to use different parameters (such as IP addresses) in a virtual environment. Also, configuring emulation networks often takes time and effort, even for simple changes [6]. Suppose we want to add one more network connection to an emulation network. In that case, we need to add a bunch of config lines similar to other devices or interfaces (we usually try "copy and paste" to make it, which sometimes causes errors due to a lack of consideration of parameter changes). The "copy-and-pasted" configuration descriptions increase network changes and debugging efforts. Especially on emulation networks, we usually change configurations repeatedly for verification and improvement. Therefore, it is essential to provide an intuitive way to change the configurations of emulation networks in some abstract way.

A generally available way to abstract network configuration is to use config templates [7]. A config template is an incomplete configuration description including some variable specifiers. By specifying variables for templates, we can obtain configuration files for multiple devices with similar roles. The advantage of the config template is its generality: It simply generates configuration strings with variables, so the method is not dependent on network protocols or configuration formats. However, existing config templating platforms are not reasonable for network configuration. Templating platforms are typically per-device, which is awkward for describing network interfaces that span multiple functions or protocols. The config templates for these platforms will be complicated with control syntax macros like Figure 1(a). In response to this issue, we need a more reasonable configuration platform for emulation networks with finer-grained config template blocks like Figure 1(b).

Our goal is to provide a simple, scalable, and expressive way to configure emulation networks using a template-based approach. To this end, we propose dot2net, a new configuration platform for emulation networks. The key idea is to separate network topology as a labeled graph and label definitions as config template blocks. The labels describe what kind of functions (i.e., config templates) the network objects correspond to, as shown in Figure 2. With this design, we can make simple changes to the network structure by modifying only the topology graph, which is visible and intuitive enough to reduce human error in the configuration. For an intuitive description of complicated networks with this platform, we establish five design principles on dot2net: (1) explicit separation of network topology and configuration, (2) declarative style of definition description, (3) no control syntax in config templates, (4) minimum manual parameter assignment, and (5) robustness for complicated networks (discussed in detail in § III-A). To satisfy these design principles, we highlight two challenging issues in the design: automated IP address assignment, and relative parameter reference from config template blocks.

Based on these design principles, we implement dot2net, a configuration platform for Docker-based emulation networks. Dot2net is publicly available as open-source software in GitHub [8]. In our evaluation with dot2net, we demonstrate that

978-3-903176-59-1

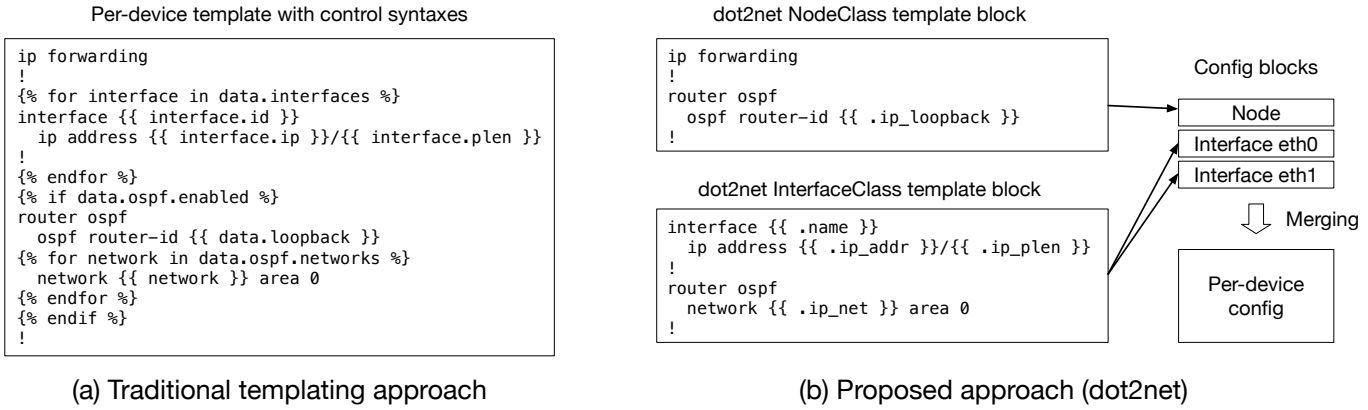

Fig. 1. Difference of templating methods in traditional and proposed approaches.

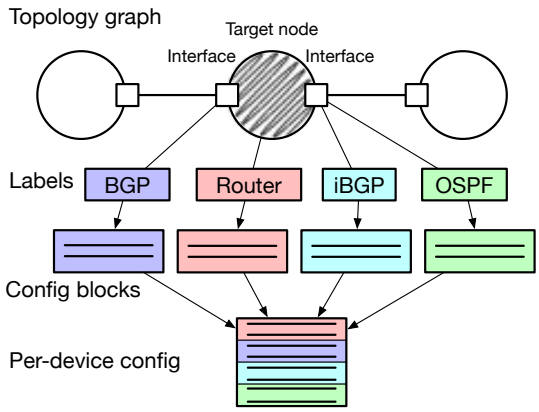

Fig. 2. Graph labels to describe config template block correspondence.

dot2net can generate a set of configuration files for large-scale networks of CLOS topology with 1,000 nodes in less than 3.5 seconds with 2.65GB memory consumption at most. We also confirm that dot2net reduces config file size by $\approx$ 10% on the network topology extension. We evaluate the design principle achievement of dot2net with case studies of network scenarios derived from test emulation networks of FRR, a widely-used router software [9].

The contribution of this paper is twofold. (1) We propose a design of dot2net (§ III) that satisfies the five design principles coming from the emulation network configuration nature (§ II). (2) We implement dot2net (§ IV) and evaluate it in terms of performance, configuration efficiency, and the expressiveness for complicated networks (§ V, § VI).

II. Background and Related Work

A. Templating network configuration

Config templates are widely used to describe configurations, not only in networks. Recent configuration management tools such as Terraform [10] and Ansible [11] provide templating functionality. However, it is still difficult to describe network configurations with these tools. The difficulty is mainly due to the variable specification in the templates. In the IaC tools, we usually specify a set of parameters corresponding to a device and embed it into a template. Thus, it is not easy to embed parameters of other devices (e.g., neighbor IP addresses). Even if the input parameters have network structure information, we need to use control syntax macros (e.g., "for" and "if") in the template to specify multiple parameters as shown in Figure 1 (A). The macros will make the template description more complicated for continuous maintenance. Furthermore, the existing template functions are not suitable for describing complicated network structures. If a network is under a simple tree structure, its parameters can be easily described by a parameter list. In contrast, for a more complicated structure such as CLOS topology [12] , we need to handle combinations of parameters (e.g., network interface pairs of connections). The number of combinations is enormous on large-scale emulation networks for manual specification.

One reason for these difficulties in config templating is that the templates are basically prepared per device. This design is suitable for microservices (the primary use case of IaC tools) because a device corresponds to a smaller service and devices are connected with a relatively simple network. In contrast, in infrastructure networks (the primary use case of emulation networks), devices (i.e., routers) have multiple components (i.e., network interfaces) and functions (i.e., protocols). In that situation, the appropriate granularity of the configuration description should be components or functions, not devices. If these granular template blocks were linked together based on network topology, as in Figure 1 (B), the description of template blocks would be more intuitive and sophisticated.

B. Configuration for network emulation

Configuration in emulation networks has two aspects: Structural configuration and functional configuration. Structural configuration describes the information required for virtualization, such as node devices and links. Emulation networks can be deployed in a variety of virtual environments. Recently, containers such as Docker have become a popular approach that is lighter and easier to scale [13]. The virtual networks

for these environments are deployed using Linux bridges and network namespaces on the host machines, which are not as easy to configure as physical networks. For this reason, there are multiple platforms for structural configuration of emulation networks such as Netkit [14], Mininet [15], Kathará [16], TiNET [17], and Containerlab [18].

Functional configuration provides software settings on the virtual devices. The configuration format is usually similar to that of physical networks because we use the same or similar software on emulation networks. Functional configuration includes network device behavior such as IP address assignment and routing control, which is more difficult in existing tools because we need to care about parameter assignments [19]. However, the existing platforms described above do not support operators to describe functional configuration.

In past literature, there are several approaches for efficient functional configuration, such as PRESTO [7], AutoNetkit [5], [20], Propane [21], and HolistIX [22]. We compare our proposed method with these methods in § V-E.

III. System design

A. Design principles and challenging issues

For easier configuration of emulation networks, we establish the following design principles on dot2net.

Principle 1: Explicit separation of network topology and configuration. Traditionally, network configuration is described per device, which makes it difficult to recognize the potential network topology in the network config files. This problem can lead to failures due to gaps between the intended topology and the described configuration. Separating network topology and configuration will make it easier to intuitively configure network devices.

Principle 2: Declarative style of definition description. There are two definition description styles for IaC tools: declarative style (e.g., Terraform [10]) and procedural style (e.g., Ansible [11]). Dot2net is declarative style that is more user-friendly because it focuses on what is to be configured instead of how it is to be configured [23]. In addition, procedural style is not reasonable for verifying the description in other tools because the description has no concrete structure.

Principle 3: No control syntax in config templates. Control syntax in templates is a powerful tool for improving parameter expressiveness. However, it makes the template description more complicated and likely to cause failures. We hypothesize that the control syntax can be hidden by subdividing config templates into blocks of appropriate abstraction and combining them according to the network topology.

Principle 4: Minimum manual parameter assignment. Emulation networks are usually configured independently with different parameters from existing production networks to avoid the risk of negative impacts on the services. In other words, the parameter assignment strategy is usually not the primary focus. We design an automatic parameter assignment mechanism to reduce parameter description errors in network configuration.

Principle 5: Robustness for complicated networks. Emulation networks are sometimes used for experimentation or preliminary verification of advanced network technologies. It is important for dot2net to accept these advanced technologies. In this paper, we focus mainly on the technologies used in test emulation networks (shown in § V-A) such as IPv4/IPv6 dual stack and overlay networks.

The key idea of dot2net is separating network configuration into network topology as a labeled graph and label definitions as config template blocks. This idea comes from Principle 1 and intends to satisfy Principles 2, 3, and 5: graphs and template blocks can both be described in a declarative style, labels in the topology graph will replace the control syntax of config templates, and the labels will also represent devices/interfaces in a variety of different roles.

Towards satisfying these design principles, we additionally need to consider the following two challenging issues:

Issue 1: Automated IP address assignment. To achieve both Principles 4 and 5, we need a way to assign parameters that accepts complicated networks. For assigning most parameters (such as interface names and AS numbers), we just need to generate integer numbers (sometimes with some headers or footers) in a given rule. The most challenging part is IP address assignment, which requires consideration of network segments [24]. Automated IP address assignment can be more complicated with advanced network technologies such as IPv4/IPv6 dual stack and overlay networks.

Issue 2: Relative parameter reference from config template blocks. By removing the control syntax in config template blocks (Principle 3), we face two potential limitations of parameter specification in config templates. First, the parameters can only be specified relatively. Without “for” and “if” control syntax, it is difficult to find related parameters by iterating over objects ad hoc. Second, the parameters must be mapped to the objects on a one-to-one basis because the templates cannot iterate parameter lists without “for” control syntax. These limitations prevent us from describing complicated networks (Principle 5) with dot2net.

B. System overview

Dot2net takes two input components, a labeled graph of network topology and the label definitions of config template blocks. The output is a set of generated configuration files. We first explain the two input components and then describe the process flow of the conversion.

One input data is a labeled graph that represents the network topology. The graph is described in DOT language [25], a declarative data description language for graphs. As shown in Figure 2, the topology graph corresponds to the layer-2 topology structure. The nodes in the graph correspond to network node devices such as switches, routers, and servers. Network interfaces in the devices are represented as edge ends between nodes, which means the edge corresponds to a network connection. These network objects (i.e., nodes, interfaces, and connections) have labels (one object can have multiple labels) in the attributes of nodes or edges.

The other input data is the label definitions that describe the detailed roles or settings of the network objects. In the

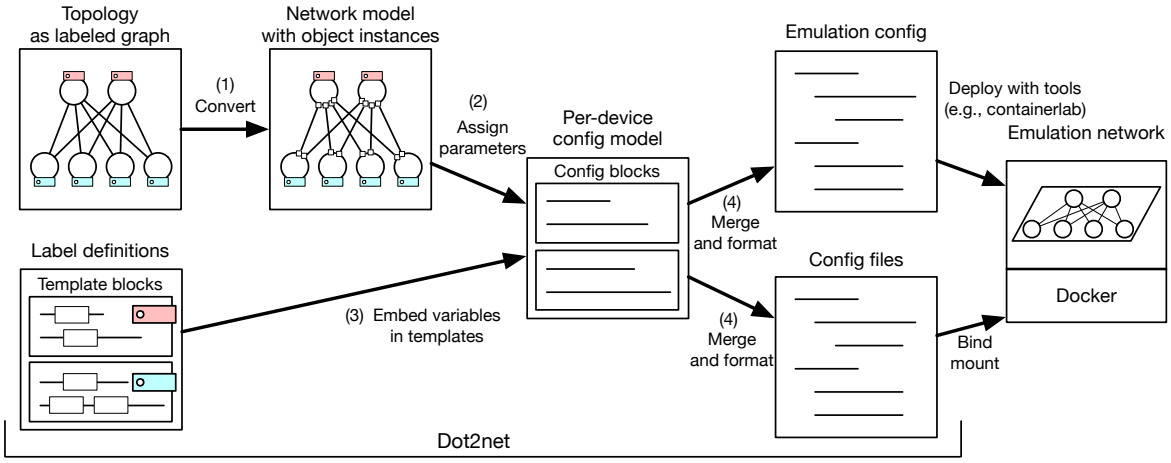

Fig. 3. Overview of dot2net processing flow.

TABLE I
DOT2NET OBJECT CLASSES

Name	Parent	Instance	Labels	Template
NodeClass	-	Node	✓	✓
InterfaceClass	-	Interface	✓	✓
ConnectionClass	-	Interface	✓	✓
GroupClass	-	Group	✓	
NeighborClass	InterfaceClass	Neighbor		✓
MemberClass	any[1]	Member		✓

definitions, a label represents a class (i.e., the objects with labels are class instances). A class definition includes config template blocks and required flags for address assignments and namespaces. There are six classes in dot2net currently, as shown in Table I, and we focus on the three capital classes with orange color due to page limitations. The definitions are described in some declarative structured data formats, such as YAML.

Figure 3 shows the overview of dot2net. There are four steps of the config generation:

1) Generate an internal network model consisting of network objects from the input network topology graph.
2) Assign parameters in the namespace of each network object.
3) Generate config blocks corresponding to the network objects from the input config template blocks and parameter namespaces.
4) Generate per-device config files by merging (including ordering and formatting) the config blocks.

Through the four steps of config file generation, dot2net has three issues for complicated networks: Generating config description, parameter assignment, and parameter reference from templates. The first one is easily achieved with the labeled graph approach. Most advanced network technologies are closed to a protocol layer. The protocol layer in the configuration description can be represented with labels in the topology graph by assigning labels to all objects that belong to the layer. In the remainder of this section, we discuss the other two issues.

[1]MemberClass depends on any of classes with template blocks.

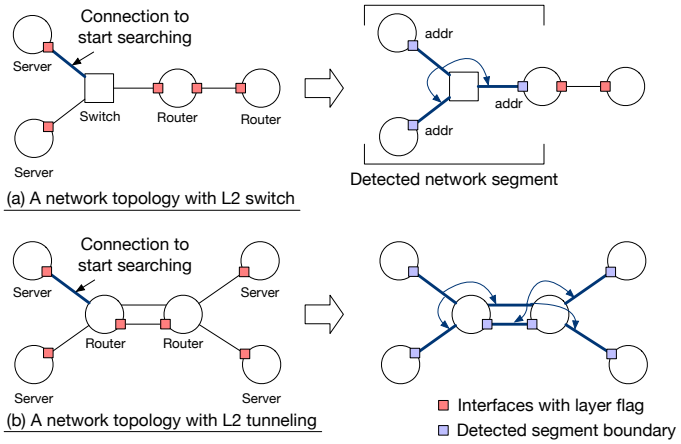

Fig. 4. Example of network segment search.

C. Automated parameter assignments

To support complicated networks, we can intuitively separate the input topology graph into multiple layers. However, this approach is likely to cause errors due to mismatched layers. If we want to add a device to the layered topology graph, we have to add it for all related layers, which sometimes causes errors due to missing edits.

To avoid this problem, we introduce layer flags, which specify the layers to which network objects (especially interfaces) belong. These flags are included in the label definitions, so they are indirectly assigned to network interfaces with labels on topology graphs. Each layer has an IP address pool, and dot2net assigns IP addresses in the pool to the flagged interfaces.

The main part of the IP address assignment algorithm is searching network segments in each layer. Figure 4(a) shows an example of network segmenting procedure. The search starts from one of the connections. Dot2net then checks its two

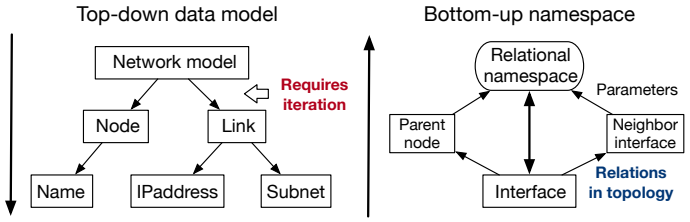

Fig. 5. Bottom-up structure of dot2net namespace.

end interfaces. If the checked interface is aware of the layer (i.e., with layer flag, red boxes in Figure 4), dot2net adds the interface to the network segment boundary interfaces (blue boxes). Otherwise, dot2net will check connections from all the other unchecked interfaces of the same device in the same way. If there are no adjacent unchecked interfaces after repeating this check, the network segment boundary interfaces form a network segment. After that, dot2net restarts the search from other unchecked connections to find other network segments until all connections are checked. We can determine all the network segments in the layer with this procedure.

With the detected network segments, we can easily assign IP addresses with the following two steps. First, dot2net cuts out IP subnets of the specified prefix length from the IP address pool (specified as IP policies for each layer) and assigns them to the network segments. For example, if an IP policy has an address pool of "10.0.0.0/16" and the specified prefix length is 24, dot2net will assign IP subnets of "10.0.0.0/24", "10.0.1.0/24", and so on. Next, dot2net assigns IP addresses from the IP subnets to the segment boundary interfaces.

In this algorithm, IP addresses are assigned independently in each layer. The layer flags enable us to automate address assignments of network topologies with advanced network technologies. Figure 4(b) shows an example network topology with L2 tunneling technology (such as L2TP or VXLAN). This figure shows a customer layer where interfaces belong to customer network (bridged with L2 tunneling) with red boxes. In this example, we assume that there are two logically-different connections between two routers: one is of the provider network (no flags in the figure, but addressed in another layer), and the other is of the customer network. On the customer layer, dot2net considers all the interfaces belong to the same network segment because the provider network connection can bridge the L2 connectivity without any segment boundary. In this way, the algorithm can handle remote network segments that are relayed with other networks.

D. Parameter namespaces

Due to the limitations explained in § III-A Issue 2, it is difficult to specify parameters without control syntax in the existing top-down data structures used in Figure 1(a). To address this issue, we introduce a namespace with a bottom-up structure (shown in Figure 5). The bottom-up approach enables parameter specifications by names of relationships in the topology instead of object iterations with control syntax.

```
r1 {{ .name }} = r1
r1.net0 {{ .ip_addr }} = 10.0.0.1  # Param A
r1.net0 {{ .ip_net }} = 10.0.0.0/24
r1.net0 {{ .ip_plen }} = 24
r1.net0 {{ .name }} = net0
r1.net0 {{ .node_name }} = r1
r1.net0 {{ .opp_ip_addr }} = 10.0.0.2
r1.net0 {{ .opp_ip_net }} = 10.0.0.0/24
r1.net0 {{ .opp_ip_plen }} = 24
r1.net0 {{ .opp_name }} = net0
r1.net0 {{ .opp_node_name }} = r2
r2 {{ .name }} = r2
r2.net0 {{ .ip_addr }} = 10.0.0.2
r2.net0 {{ .ip_net }} = 10.0.0.0/24
r2.net0 {{ .ip_plen }} = 24
r2.net0 {{ .name }} = net0
r2.net0 {{ .node_name }} = r2
r2.net0 {{ .opp_ip_addr }} = 10.0.0.1  # Param B
r2.net0 {{ .opp_ip_net }} = 10.0.0.0/24
r2.net0 {{ .opp_ip_plen }} = 24
r2.net0 {{ .opp_name }} = net0
r2.net0 {{ .opp_node_name }} = r1
...
```

Fig. 6. Example of parameters in the namespaces.

A namespace in dot2net is an associative array. An object instance, such as a node or an interface, has one namespace. It contains automatically generated parameters with names for specification. The config template blocks embed parameters into the namespace.

In network configuration using templates, we usually need to specify the parameters of the device and its neighbors. In Layer 2, the interface neighbor means an opposite interface of its connection (i.e., there is at most one Layer 2 neighbor for each interface). Since the network topology basically represents the Layer 2 topology structure, the Layer 2 neighbor interface is intuitively available on dot2net.

In Layer 3 or overlays, an interface can have multiple neighbor interfaces. To handle these neighbor parameters on a one-to-one basis, we introduce a subclass named NeighborClass of the interface classes (listed in Table I). A NeighborClass describes configuration template blocks for the parent interface (not the neighbor interface) that requires parameters of one of the neighbor interfaces (i.e., an interface has the same number of NeighborClass instances as the neighbor interfaces). A NeighborClass instance has the same namespace as the parent interface and has the parameters of the corresponding neighbor interface (whose name additionally has a header for neighbors).

Figure 6 lists available parameters in an example namespace. This table means, for instance, that if there is an InterfaceClass label for r1.net0 (i.e., the net0 interface of router r1), the InterfaceClass config template block will replace a parameter specifier "{{.ip_addr}}" with "10.0.0.1" for the interface (Param A). The same parameter can also be used with the specifier "{{.opp_ip_addr}}" by r2.net0 which is the opposite of r1.net0 in the network topology (Param B). This way, we can specify parameters in related devices with relative names.

Dot2net implementation has some other modules to extend the namespaces for more complicated parameter specifications (omitted due to page constraints in this paper).

```
digraph {
 r1[class="router"];
 r2[class="router"];
 r3[class="router"];

 r1->r2[dir=none];
 r2->r3[dir=none];
}
```

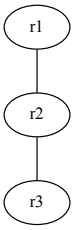

Fig. 7. Example topology graph (a DOT file and its visualization).

IV. Implementation

Dot2net is implemented in Go language and published as an open-source software [8]. Dot2net is designed mainly for Docker-based emulation networks.

A. Topology graph

The network topology graph is described as a multigraph because node pairs have multiple connections for redundancy techniques such as link aggregation. Figure 7 demonstrates an example topology graph description for the scenario in DOT format. There are three defined nodes, r1, r2, and r3, with class attributes "router", which means these nodes belong to the NodeClass named "router". If required, we can also specify multiple labels by separating them with ";" in the class attribute strings. Also, the two connections do not have any class label specification. If defined, the interfaces belong to "default" classes.

B. Label definitions

The label definitions are described in YAML format. Figure 8 demonstrates a part of an example YAML file of label definitions. It defines two object classes: NodeClass "router" and InterfaceClass "default". The "default" InterfaceClass will be applied to all the interfaces without labels (e.g., two connections in Figure 7). These class definitions include both structural configuation (e.g., container images) and functional configuration (i.e., config template blocks).

Dot2net currently has six object classes listed in Table I. In addition to the three classes (NodeClass, InterfaceClass, and NeighborClass) described in § III, there are three classes designed for flexible description of config templates. ConnectionClass is a label definition corresponding to edge labels (not head or tail labels) on network topology graphs. ConnectionClass basically means that the described config templates are added to both end interfaces of the connection. ConnectionClass also has a flag for describing connectivity in overlay layers. GroupClass is a label definition corresponding to subgraphs on network topology graphs. GroupClass can be used to describe groups of nodes that share the same parameters, such as AS numbers and OSPF areas. MemberClass is a subclass similar to NeighborClass; it is used to refer to parameters of member objects of some object class.

For practical purposes, some classes can be defined as virtual classes. The virtual classes do not have config template blocks

```
nodeclass:
  - name: router
    primary: true
    tinet:
      image: quay.io/frrouting/frr:8.5.0
    clab:
      kind: linux
      image: quay.io/frrouting/frr:8.5.0
    config:
      - file: daemons
        sourcefile: ./daemons
      - file: vtysh.conf
        sourcefile: ./vtysh.conf
      - file: frr.conf
        template:
          - "ip forwarding"
          - "!"
          - "router ospf"
          - " ospf router-id {{.ip_loopback}}"
          - "!"

interfaceclass:
  - name: default
    primary: true
    ipaware: [ip]
    config:
      - file: frr.conf
        template:
          - "interface {{.name}}"
          - " ip address {{.ip_addr}}/{{.ip_plen}}"
          - "!"
          - "router ospf"
          - " network {{.ip_net}} area 0"
          - "!"
```

Fig. 8. An excerpt of example label definitions (YAML).

but they have parameter namespaces and will obtain assigned parameters. The parameters can be referred to as relative parameters from other non-virtual objects, so they will help describe configurations related to out-of-emulation parameters (an example is provided in § V-D2).

Dot2net is designed with automated parameter assignment, but it also supports manual parameter assignment. The manual parameters are specified in graph attributes similar to class labels. If some IP addresses are specified manually, they are reserved and not used in automated address assignment. This means that we can manually assign IP addresses for important objects and let others be assigned automatically.

C. Output

As the main target of dot2net is emulation networks, the standard output of dot2net is configuration files for emulation network platforms (including both structural configuration and functional configuration). Dot2net currently supports two Docker-based emulation network platforms: Containerlab [18] and TiNET [17]. With these tools, we can automate emulation network deployment and configuration from dot2net input.

V. Evaluation and Case Studies

A. Network topology scenarios

We use six network topology scenarios in dot2net for evaluation. Table II lists the scenarios and their scale. Five of them are from FRR topotests [2], a suite of topology tests on mininet. The tests are described in Python scripts, so we generate equivalent

TABLE II
NETWORK TOPOLOGY SCENARIOS

Network scenario		Base topology	
Source	Name	Nodes	Links
FRR topotests	rip_topo1	9 (+1)	8 (+1)
FRR topotests	ospf_topo1	10	9
FRR topotests	ospf6_topo1	10 (+3)	9 (+4)
FRR topotests	bgp_features	15	15
FRR topotests	bgp_evpn_vxlan_topo1	5	9
TiNET examples	basic_clos	14	16

TABLE III
LARGE-SCALE NETWORK TOPOLOGIES BASED ON BASIC_CLOS

Name	Nodes				Links
	Tier 1	Tier 2	Tier 3	Total	
Clos100	64	32	4	100	2,176
Clos200	128	64	8	200	8,704
Clos300	192	96	12	300	19,584
Clos400	256	128	16	400	34,816
Clos500	320	160	20	500	54,400
Clos600	384	192	24	600	78,336
Clos700	448	224	28	700	106,624
Clos800	512	256	32	800	139,264
Clos900	576	288	36	900	176,256
Clos1000	640	320	40	1,000	217,600

configuration files for Containerlab and TiNET using dot2net. There is another scenario "basic_clos" from TiNET examples [3]. This scenario is based on scalable CLOS topology, so we use this scenario mainly for scalability evaluation.

We use dot2net version v0.2.5 for the following evaluation.

B. Performance

We first evaluate the performance (processing time) of dot2net for large-scale network topologies. As the large-scale network topologies, we extend the "basic_clos" scenario with ten topologies (listed in Table III). The "basic_clos" is a 3-tier CLOS network , and we change the number of nodes in each tier of the topologies. For the experiments, we use a computer with an Ubuntu 18.04 server (x86_64) equipped with Intel(R) Xeon(R) Gold 6258R CPU 2.70GHz and 128GB memory.

Figure 9 shows the processing time to generate configuration files for these topologies. The processing time depends on the number of links in the network topology because the internal model generation and the address assignments are the major part of the processing. The processing time is at most 3.5 seconds for Clos1000 which is acceptable because the deployment of the emulation network will take much longer. In addition, the maximum memory consumption is 2.65GB for Clos1000, so we can use dot2net with general-performance computers on this scale.

C. Configuration efficiency

We next evaluate how dot2net reduces config file size by decreasing the "copy-and-pasted" description of configuration with dot2net config template blocks. Table IV lists the comparison of file size. In this table, we generate config files for

[2]https://github.com/FRRouting/frr/tree/master/tests/topotests
[3]https://github.com/tinynetwork/tinet/tree/master/examples

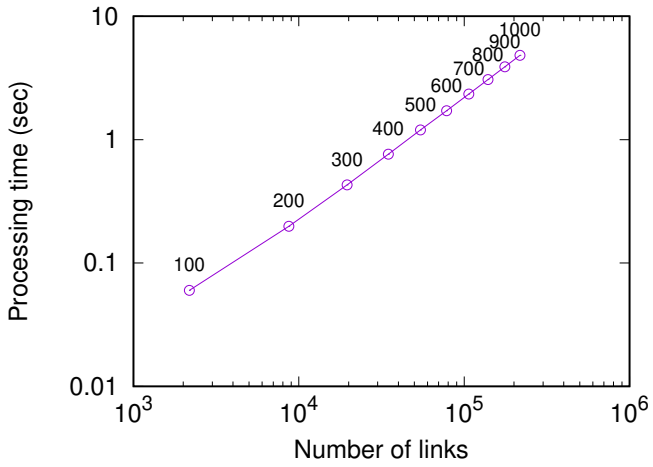

Fig. 9. Processing time of dot2net for large-scale CLOS topologies in log-log-scale plot. Each plot is labeled with the total number of nodes.

Containerlab and TiNET with dot2net and compare their file size (bytes) with the dot2net input. Dot2net reduces the file size by one-half to one-third in each scenario.

In addition, we extend the network topologies (shown in Table II) by adding one router and one connection ("+1" in Table IV) to emphasize the efficiency of dot2net in extending network topologies. In dot2net, expanding the topology only increases the size of the topology (DOT). The difference in config file size is less than 10% in each scenario. Therefore, dot2net effectively shrinks the input for configuring emulation networks with config template blocks.

D. Case study

In this section, we discuss the robustness of dot2net for complicated networks along with the two challenging issues introduced in § III-A.

1) IP addressing issue: The first issue is automated IP address assignment for complicated networks. We demonstrate two network scenarios: an IP dual stack and a VXLAN network.

IP dual stack: The "ospf_topo1" scenario describes a network routed with OSPFv2 for IPv4 and OSPFv3 for IPv6 in parallel. Figure 10 shows the assigned IP addresses. We can see both IPv4 and IPv6 addresses and subnets in the graph.

VXLAN: The "bgp_evpn_vxlan_topo1" scenario depends on VXLAN, one of the overlay network techniques. We can describe this scenario with the idea of layer flags explained in § III-C and Figure 4(b). With two layer flags, one for the provider network and the other for the customer network (bridged with VXLAN), the automated assignment algorithm successfully determines IP addresses with appropriate subnets.

2) Namespace issue: The second issue is relative parameter reference from config template blocks. This feature allows us to describe some complicated scenarios.

BGP neighbor: The "bgp_features" scenario is a BGP-based network that includes non-BGP routers (routed with OSPF) and L2 switches. In this network, a BGP router requires parameters from neighboring BGP routers. Since there are

TABLE IV
COMPARISON OF CONFIGURATION FILE SIZE (BYTES)

		dot2net						Containerlab		TiNET	
Scenario	(Expansion)	Topology	(diff)	Config	(diff)	Total	(diff)		(diff)		(diff)
rip_topo1		536		3,128		3,664		7,140		7,509	
rip_topo1	(+1)	591	(+55)	3,128	(±0)	3,719	(+55)	8,929	(+1,789)	9,323	(+1,814)
ospf_topo1		539		2,962		3,501		11,547		11,906	
ospf_topo1	(+1)	582	(+43)	2,962	(±0)	3,544	(+43)	13,708	(+2,161)	14,085	(+2,179)
ospf6_topo1		982		2,813		3,795		9,872		10,265	
ospf6_topo1	(+1)	1,026	(+44)	2,813	(±0)	3,839	(+44)	11,762	(+1,890)	12,180	(+1,915)
bgp_features		1,037		6,234		7,271		19,119		19,815	
bgp_features	(+1)	1,122	(+85)	6,234	(±0)	7,356	(+85)	21,999	(+2,880)	22,713	(+2,898)
bgp_evpn_vxlan_topo1		777		4,171		4,948		13,787		14,088	
bgp_evpn_vxlan_topo1	(+1)	862	(+85)	4,171	(±0)	5,033	(+85)	15,024	(+1,237)	15,345	(+1,257)
basic_clos		858		1,275		2,133		9,300		10,275	
basic_clos	(+1)	909	(+51)	1,275	(±0)	2,184	(+51)	9,932	(+632)	10,970	(+695)

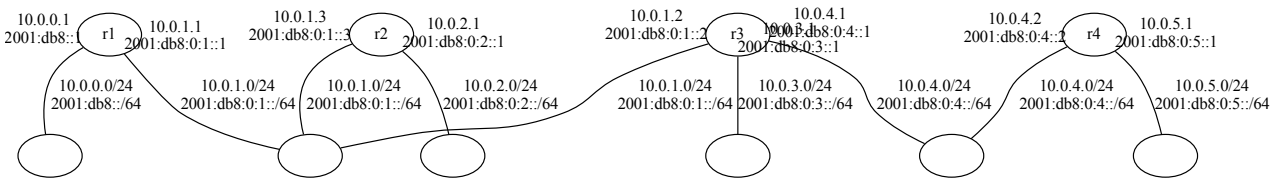

Fig. 10. Address assignments for an IPv4/IPv6 dual stack network. Nodes without labels in the graph are L2 switches.

L2 switches between BGP routers, the neighbor parameters cannot be referred to as L2 opposite parameters. By defining NeighborClasses of the BGP routers in a BGP layer, we can specify neighbor parameters for the config blocks in the Neighbor Classes.

Stub network: The "rip_topo1" and "ospf6_topo1" scenarios involve mixed routing policies: dynamic routing and static routing. In addition, the static routing policy includes routes for out-of-emulation (stub) nodes and subnets. To describe these network scenarios, we use virtual objects (explained in § IV) in their topology graphs. If there are virtual nodes and connections, dot2net can assign IP addresses to their interfaces and IP subnets to the virtual connections, which can be referred from other objects. Therefore, we can describe config blocks with stub parameters by virtual objects and relative references in namespaces.

E. Comparison with existing methods

Finally, we qualitatively compare dot2net to similar existing approaches for functional configuration. Table V describes the comparison in three aspects: Simplicity, scalability (ease of expansion), and expressiveness of the configuration description. We can confirm that dot2net is the only method that satisfy all of the requirements.

Especially on simplicity, Figure 1 compares config templates of (A) AutoNetkit and (B) dot2net. It is clear that the config template blocks can be simpler if not using control syntax macros.

VI. DISCUSSION

A. Application

Although dot2net is specifically designed for Docker-based emulation networks, it is also applicable to networks in other environments because dot2net essentially just generates config files from template blocks and the topology graph. For example, we can distribute generated config files to physical network devices using existing tools.

Dot2net can even generate files other than config files such as test scripts and specification documents. Dot2net is versatile because of its template-based approach, so the applicability is worth discussing further.

B. Limitation

As the dot2net implementation provides a way to specify parameters manually, there are no non-descriptive configurations on dot2net in theory (if inefficient ways are allowed). However, there are several limitations to efficient configuration.

Since dot2net depends on the network topology to specify parameters, it is difficult to list parameters that is not related to the topology. This limitation is problematic, for example, when describing Access Control Lists (ACLs). It requires service-oriented information to describe ACLs, so we need different approaches for them.

It is also difficult to describe topology information that does not fit graph description languages. For example, we cannot intuitively describe source routing policies (such as segment routing) in dot2net.

The one-to-one basis of parameter specification, which comes from the principle of not using control syntax in config

TABLE V
QUALITATIVE COMPARISON TO EXISTING CONFIGURATION AUTOMATION METHODS

Method	Approach	Description simplicity	Scalability	Expressiveness
PRESTO [7]	Extended template blocks	**Complicated due to extended control syntax**	Scalable	VPN, VoIP, etc.
AutoNetkit [5], [20]	Scripted model + templates	**Complicated due to procedural style and control syntax**	Limited support of parameter assignment	**Limited to BGP and OSPF**
Propane [21]	Product graph + domain specific language	Sophisticated but not intuitive	Scalable	**Limited to BGP**
HolistIX [22]	Topology diagram	Simple and intuitive	**Requires manual parameters**	**Only for IX network**
dot2net	Labeled graph + template blocks	Simple and intuitive	Easily scalable as § V-B, § V-C	Expressive as § V-D

template, can cause another limitation. Essentially, dot2net creates a NeighborClass instance for each neighbor interface as explained in § III-D. If there is a config block that require parameters from multiple instances, we cannot describe it in a straightforward way with dot2net. Fortunately, network configuration is usually object-oriented, so the dispersing parameter reference is a rare case in emulation network configuration.

VII. CONCLUSION

In this paper, we proposed the design and implementation of a template-based configuration platform named dot2net for simple, scalable, and expressive configuration description. The key idea is to separate network topology as a labeled graph and label definitions as config template blocks. Dot2net supports automated assignment of IP parameters and relative parameter specification with namespace, which accepts complicated networks such as IPv4/IPv6 dual-stack networks and overlay networks. We confirmed that dot2net is scalable enough to generate a 1,000 node CLOS network in less than 3.5 seconds. We also showed that dot2net reduces the increase in config file size to less than 10% as the network topology expands. Finally, we demonstrated that dot2net can represent complicated networks with case studies of test/example network scenarios.

In future work, we will perform a demonstration experiment of dot2net on testbed networks to improve the applicability to more practical use cases. In particular, the expressiveness of dot2net can be improved with additional classes and namespace extensions. We also consider the way to generate dot2net input from config files of existing networks to support environment migration and expansion of existing networks.

ACKNOWLEDGEMENTS

This work is supported by JSPS KAKENHI Grant Number JP22K17886.

REFERENCES

[1] A. Fuller, Z. Fan, C. Day, and C. Barlow, "Digital twin: Enabling technologies, challenges and open research," *IEEE Access*, vol. 8, pp. 108 952–108 971, 2020.
[2] H. X. Nguyen, R. Trestian, D. To, and M. Tatipamula, "Digital twin for 5G and beyond," *IEEE Communications Magazine*, vol. 59, no. 2, pp. 10–15, 2021.
[3] R. Alimi, Y. Wang, and Y. R. Yang, "Shadow configuration as a network management primitive," in *Proceedings of SIGCOMM '08*. ACM, 2008, p. 111–122.
[4] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully Emulating Large Production Networks," in *Proceedings of SOSP '17*, 2017, pp. 599–613.
[5] S. Knight, H. Nguyen, O. Maennel, I. Phillips, N. Falkner, R. Bush, and M. Roughan, "An automated system for emulated network experimentation," in *Proceedings of CoNEXT '13*. ACM, 2013, pp. 235–246.
[6] R. Emiliano and M. Antunes, "Automatic network configuration in virtualized environment using gns3," in *Proceedings of ICCSE '15*, 2015, pp. 25–30.
[7] W. Enck, T. Moyer, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, Y.-W. E. Sung, S. Rao, and W. Aiello, "Configuration management at massive scale: system design and experience," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 323–335, 2009.
[8] S. Kobayashi, "dot2net," https://github.com/cpflat/dot2net/, 2023.
[9] FRRouting Project, "Frrouting," https://frrouting.org/, 2017.
[10] HashiCorp, "Terraform," https://www.terraform.io/, 2014.
[11] Ansible Inc., "Ansible," https://www.ansible.com/, 2012.
[12] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for Routing in Large-Scale Data Centers," https://datatracker.ietf.org/doc/html/rfc7938, 2016.
[13] S. Peach, B. Irwin, and R. van Heerden, "An overview of linux container based network emulation," in *Proceedings of ECCWS '16*, 2016, pp. 253–259.
[14] M. Pizzonia and M. Rimondini, "Netkit: network emulation for education," *Software: Practice and Experience*, vol. 46, no. 2, pp. 133–165, 2016.
[15] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of HotNets '10*. ACM, 2010, pp. 1–6.
[16] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, "Kathará: A container-based framework for implementing network function virtualization and software defined networks," in *Proceedings of NOMS '18*. IEEE, 2018, pp. 1–9.
[17] tinynetwork, "tinet," https://github.com/tinynetwork/tinet/, 2019.
[18] Nokia, "Containerlab," https://containerlab.dev/, 2021.
[19] S. M. Bellovin and R. Bush, "Configuration management and security," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, p. 268–274, 2009.
[20] H. Nguyen, M. Roughan, S. Knight, N. Falkner, O. Maennel, and R. Bush, "How to Build Complex, Large-Scale Emulated Networks," in *Proceedings of TridentCom '11*, vol. 46. Springer, 2011, pp. 1–16.
[21] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations," in *Proceedings of SIGCOMM '16*. ACM, 2016, pp. 328–341.
[22] C. Visser, S. Yamamoto, T. Tomine, Y. Sekiya, and M. Bruyere, "HolistIX: A zero-touch approach for IXPs," in *Proceedings of NetPA '21*. IFIP, 2021, pp. 1–7.
[23] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys*, vol. 47, no. 4, 2015.
[24] D. Caldwell, A. Gilbert, J. Gottlieb, A. Greenberg, G. Hjalmtysson, and J. Rexford, "The cutting edge of ip router configuration," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, p. 21–26, 2004.
[25] E. Gansner, E. Koutsofios, and S. North, "Drawing graphs with dot," Technical report, AT&T Research, 2006.