Research on Network Policy Combination and Conflict Detection in SDN

Bohan He, Ligang Dong(✉), Tijie Xu, Shuocheng Fei, Huafei Zhang, and Weiming Wang

School of Information and Electronic Engineering, Zhejiang Gongshang University, No. 18, Xuezheng Street, Xiasha University Town, Hangzhou 310018, China
donglg@zjgsu.edu.cn

Abstract. Since the current SDN southbound interface level is low and programming situation is complex, it requires a high-level abstract programming language to simplify programming. First, this paper improves the NetCore programming language to generate NetCore-M language, so that it can support deployment of multi-policies combination including packet drop action. This paper describes in detail the syntax, semanteme, and implementation of NetCore-M language. Secondly, this paper describes the network policy conflict systematically and solves it. Finally, this paper shows that the modified multi-policies combination algorithm can effectively detect and prompt policies conflicts based on the implementation of the Pyretic project.

Keywords: Policy combination · Conflict detection · SDN · Pyretic

1 Introduction

Compared with the traditional network [1–3], Software Defined Network (SDN) [6] is a new type of network architecture whose goal is to simplify network control and management with the programmability of the network leading innovation. Despite SDN uses open, standard interfaces such as ForCES [4], OpenFlow [5] to replace the private configuration commands of different equipment suppliers to simplify the network configuration task. A high-level programming language for SDN is very necessary.

There are several kinds of high-level programming languages for SDN, such as Frenetic [12], Pyretic [9], NetCore [7], Procera [13]. NetCore is a programming policy language based on Frenetic. Our study is based on NetCore.

NetCore policy combination algorithm only takes the forwarding operation into consideration. The conflict between policies hasn't been solved yet. Therefore, this paper modifies NetCore language to support forwarding and packet drop, and then proposes NetCore-M policy combination algorithm to achieve the conflict detection of policies combination in order to make the algorithm adapt to more complex programming environment.

The main part of this paper is divided into seven sections: The Sect. 2 analyzes the related research of network programming language in SDN. The Sect. 3 introduces the

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2017
S. Guo et al. (Eds.): TridentCom 2016, LNICST 177, pp. 24–34, 2017.
DOI: 10.1007/978-3-319-49580-4_3

improved NetCore-M programming language, syntax and forms. The Sect. 4 introduces the policy conflict problems in policy combination algorithm. In the Sect. 5, we give a verification experiment to show the results of the policy combination and the policy conflict. The Sect. 6 summarizes the paper.

2 Related Work

Researchers have developed a number of high-level network programming languages for SDN to hide the complexity of SDN programming (Table 1).

Table 1. High-level network programming languages

Languages	Controllers	Actions	The Operation model
Frenetic	NOX	Forwarding	Parallel model
Pyretic	POX	Forwarding, Packet Drop	Parallel model, Serial model
Kinetic	POX	Forwarding, Packet Drop	Parallel model, Serial model
NetCore	NOX	Forwarding, Packet Drop	Parallel model, Serial model

Frenetic is a policy language based on the Ocaml [14] programming language. Frenetic languages can be classified into two sub-language, one is network policy management library which process packet forwarding based on FRP [15] and the other is a declarative SQL language for the network status inquiry.

Pyretic language uses the policy as a function and packets as input and output variables. Packets can be processed in the form of the parallel or sequential combination. Later versions of Pyretic is Kinetic [8] which supports combinations of several consecutive service functions in series and parallel connections. It achieves the function of simple static service chains.

NetCore is a policy language developed on the basis of Frenetic with more expressive syntax than that of Frenetic. Besides, NetCore can use arbitrary functions to process packets with more flexibilities. In addition, NetCore contains a minimalist inquiry formula language which can be used to analyze the flow.

These four languages have a common feature which is transforming a few abstract high-level policies into numerous and complicated OpenFlow [10] commands with the cooperation of the NOX/POX controllers.

3 NetCore-M Programming Language

This paper modifies the NetCore policy combination algorithm and, adds the action of packet drop and detects the policy conflict. It also proposes the policy conflict detection mechanism and the policy option scheme based on the priority compromise policy options.

3.1 Formal Syntax and Semantics of NetCore-M Programming Language

In this section, we will modify the NetCore language [7] as NetCore-M, to describe the policy services including language syntax, semanteme, and the description of the achievement.

We continue to use the basic syntax and semanteme [7] of NetCore and extend the packet drop action D to the original syntax of the action set A, so that the policy can support packet drop. Thus the following syntactic definition is added.

$$\text{Drop action } \ d$$
$$\text{Packet drop set } \ D ::= \{d\}$$
$$\text{Action set } \ A ::= D \mid S$$

Fig. 1. The improved formal syntax

NetCore-M contains two parts including predicate and policy. The predicate describes a set of packets that policy is interested in, and the policy specifies the way to handle packet sets. Figure 1 shows the improved formal syntax of predicate and policy.

Two types of action sets can't work together in the same packet, so the current policy contains two basic forms, namely, e → S and e → D. When the packet is matching predicate policy in the policy, the packet will implement the attached action.

3.2 The Description of Policy Semanteme

Policy is a priority list composed of priority, mode and action list [11]. The Policy Compiler is the core component of network policy service. Policy combination and policy conflict detection will be implemented in the Policy Compiler.

The classification table $\vec{r}$ is composed of sequence rules r$(r_1, \ldots, r_i, \ldots, r_n)$. Switches process packets based on the information provided by the rules. Each rule consists of a mode Z and an action list α. The order of the rule in the sequence represents the priority while the priority of the rule is lower than the rules on the left side and higher than the rules on the right side.

Functions of rule model are that if the packet p can match the z model of the rule r_i, packet will implement action α according to the description of rule.

The operation semanteme of the policy compiler and switch is shown in action list which can be expressed as the three cases in the Table 2.

Table 2. The actions of rules

Symbol	Meaning
S	Forward packets to each switch of set S
Ω	Forward packets to controllers
__(Empty)	Drop packets

We will describe operation semanteme of the compiler and the switch by the molecular machine [16] as same as NetCore used to. The definition of the relevant symbols of the molecular machine is shown in Fig. 2.

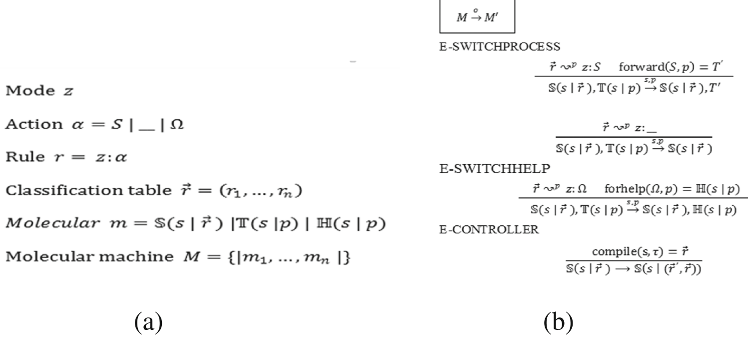

Fig. 2. The molecular machine, (a) & (b)

As shown in Fig. 2(b), the operational semanteme is given in the form of derivation rules. The switch molecules $\mathbb{S}(s|\vec{r})$ in the figure contain switch s in classification table r. The transport molecules $\mathbb{T}(s|p)$ represent packet p on the way to switch s; the assistant molecules $\mathbb{H}(s|p)$ indicate switch s send a requests to the controller for help on how to process the packet p.

E-SWITCHPROCESS is utilized when matching rules of packets have no "sent to the controller" action $\mathbb{T}(s|p)\text{forward}(S,p)$. The molecular machine will remove and then it will determine whether use the function according to the rules of the specific action list. If matching rules of the packet contain "sent to the controller" action, then E-SWITCHHELP is utilized and a help request of switch structure is sent to the controller. In this process, the molecular machine will remove processed transport molecules, and then use function $\text{forhelp}(\Omega,p)$ to generate assistant molecules.

The derivation rule E-CONTROLLER describes the way controllers use compiler to compile policy classification table and the means to issue and update switch.

3.3 Compilation Process of Forwarding Policy Services

The compilation process of the policies can be divided into two steps, namely, the policy intermediate form and the classification tablet of policy intermediate form. The previous step can be further subdivided into the following steps:

(1) Detection and resolution of policy combination.
(2) Detection and resolution of predicate combination.
(3) Predicate compiles to predicate intermediate form.
(4) The combination of predicate intermediate form.
(5) Predicate intermediate form compiles to policy intermediate form.

(6) Policy conflict detection.
(7) The combination of policy intermediate form.

The whole procedure is carried out in sequence, and the result of the last step is the input of the next step.

We define the intermediate form of syntax as follows in order to discuss policy conflicts in an easier way.

Boolean value $b ::= \text{True} \mid \text{False}$

Switch level matching mode $z ::= (h_1 : \overrightarrow{w}) \wedge \ldots \wedge (h_n : \overrightarrow{w})$

Predicate intermediate form $\pi ::= \langle e : z : b \rangle$

Policy intermediate form $\rho :: = \langle e : z : A \rangle$

Predicate intermediate form contains three values: sequence predicate e, sequence mode z (i.e., regular mode), and Boolean value b.

The sequence predicate e and the sequence model z have different expression form, but they contain the same semanteme. The sequence predicate is patterned with header h and vector $\vec{w}$.

In the same tuple of intermediate forms, sequence model z is representation of bit vector of predicate sequences e. When the model z_1 can match the packet set is a subset of sets which z_2 matches, it is denoted as $z_1 \sqsubseteq z_2$. We can give a slight extension of the symbol $\sqsubseteq$ which makes $p \sqsubseteq z_2$ mean that packet p can match the mode z.

Boolean value b represents the way to process packet set defined by the ordinary predicates. Policy intermediate form $\langle e : z : A \rangle$ is similar to predicate intermediate form, among them, A represents the action set.

3.4 Compilation and Combination Algorithm of Predicate and Policy

Figure 3(a) shows the formal description of predicate compilation and combination algorithm.

$$\mathrm{T}(s, e) = \vec{\pi}$$

$$\mathrm{T}(s, h : \vec{w}) = \langle (h : \vec{w}) : \mathrm{O}(h : \vec{w}) : \text{True} \rangle, \quad \langle * : \top : \text{False} \rangle.$$

$$\mathrm{T}(s, \text{switch } s') = \begin{cases} \langle * : \top : \text{True} \rangle, & if\ s = s' \\ \langle * : \top : \text{False} \rangle, & if\ s \neq s' \end{cases}$$

$$\mathrm{T}(s, \neg e) = \prod_i \langle e_i : z_i : \neg b_i \rangle, \quad (\mathrm{T}(s, e))_i = \langle e_i : z_i : b_i \rangle.$$

$$\mathrm{T}(s, e \cap e') = \prod_i \prod_j \langle e_i \cap e_j' : z_i \sqcap z_j' : b_i \wedge b_j' \rangle, \quad (\mathrm{T}(s, e))_i = \langle e_i : z_i : b_i \rangle, \ (\mathrm{T}(s, e'))_j = \langle e'_j : z'_j : b'_j \rangle.$$

$$\mathrm{T}(s, e \cup e') = \prod_i \prod_j \langle e_i \cap e_j' : z_i \sqcap z_j' : b_i \vee b_j' \rangle, \quad (\mathrm{T}(s, e))_i = \langle e_i : z_i : b_i \rangle, \ (\mathrm{T}(s, e'))_j = \langle e'_j : z'_j : b'_j \rangle.$$

(a)

$$\mathrm{T}(s, \tau) = \vec{\rho}$$

$$\mathrm{T}(s, e \rightarrow A) = \prod_i \begin{cases} \langle e_i : z_i : A \rangle, & if\ b_i = \text{True} \\ \langle e_i : z_i : \emptyset \rangle, & if\ b_i = \text{False} \end{cases}, \quad (\mathrm{T}(s, e))_i = \langle e_i : z_i : b_i \rangle.$$

$$\mathrm{T}(s, \tau \cup \tau') = \prod_i \prod_j \langle e_i \cap e_j' : z_i \sqcap z_j' : A_i \cup A_j' \rangle, \quad (\mathrm{T}(s, \tau))_i = \langle e_i : z_i : A_i \rangle, \ (\mathrm{T}(s, \tau'))_j = \langle e'_j : z'_j : A'_j \rangle.$$

$$\mathrm{C}(s, \tau) = \vec{r}$$

$$\mathrm{C}(s, \tau) = \prod_i \begin{cases} z_i : S, & \text{if } A_i = S \\ z_i : _, & \text{if } A_i = D \\ z_i : \Omega, & \text{if } A_i = \emptyset \end{cases}, \quad \mathrm{T}(s, \tau) = \vec{\rho}, \quad (\vec{\rho})_i = \langle e_i : z_i : A_i \rangle.$$

(b)

Fig. 3. The compilation and combination algorithm

$T(s,e)=\prod_{i} e_i : z_i : b_i$ indicates the sequence of predicate intermediate form is compiled by original predicate e. Among the sequences, the i-th tuple is marked as $(T(s,e))_i = e_i : z_i : b_i$. The first equation in Fig. 3(a) shows that the compiler will compile original predicate $h : \vec{w}$ into a sequence of intermediate form tuple containing with two predicates. The first tuple in the sequence $\langle (h : \vec{w}) : \mathrm{O}(h : \vec{w}) : \mathrm{True}\rangle$ contains model which is generated by compilation oracle.

For intersection operation of predicates, predicates should be compiled in advance. And we have tuple members of the predicate e and tuple members of $e^{'}$ intersection combination operation. All of the operation results such as $\langle e_i \cap e_j^{'} : z_i \sqcap z_j^{'} : b_i \wedge b_j^{'}\rangle$ constitute of $e \cap e^{'}$. If a packet match z_i which comes from e, as well as z_j' which comes from e', and finally the packet will match the sequence model is $z_i \sqcap z_j'$.

For the predicate "and" operation $e \cup e'$, the compilation process is similar to the compilation process of $e \cap e'$. What we should focus on is the combination of sequence predicates and sequence model also remains intersection operation.

For the predicate "not" operation ¬e, the compilation results is the negation of Boolean values of the intermediate form.

Figure 3(b) resents a description of policy compilation and combination algorithm.

Function T(s,τ) describes the process of policy compiling to the policy intermediate form, and $\mathrm{C}(s,\tau)$ corresponds to the process of policy intermediate form generating classification table.

If we want to compile a basic policy $e \rightarrow A$, firstly the compiler need to compile predicate e to generate predicates intermediate sequence, and then add actions for each predicate intermediate tuple in order to generate policy intermediate forms tuples which additional actions are determined by the values b_i of $\langle e_i : z_i : b_i\rangle$. There are two kinds of situations, if b_i is true, the additional action is A, as $\langle e_i : z_i : A\rangle$; if b_i is false, the additional action is $\emptyset$, as $\langle e_i : z_i : \emptyset\rangle$. There is different between the action of predicate intermediate tuples and the action of the classification table, so it requires function $\mathrm{C}(s,\tau)$ for further conversion.

For policy combination $\tau \cup \tau'$, the compilation process is similar to it of predicate "and" operation. The difference is the operation $b_i \wedge b_j^{'}$ of the Boolean value is replaced by action set operation $A_i \cup A_j^{'}$.

Because of policy conflicts, we must conduct conflict detection after packets processing action is added to predicate intermediate form and before the action combine into policy intermediate form. The specific issues of policy conflict will be introduced in Sect. 4.

4 Policy Conflicts

This section gives further discussions in the policy conflicts.

We divided policy action into two categories including the set of forwarding and set of packet drop. A packet will never implement packet drop and forwarding operations at the same time. Therefore, policy conflicts can be defined as following:

Define 1 (policy conflict). There is intersection in the packet sets of different policy predicate definitions and the actions of forwarding and packet drop exist in the intersection.

We obtained five cases which are shown in Fig. 4.

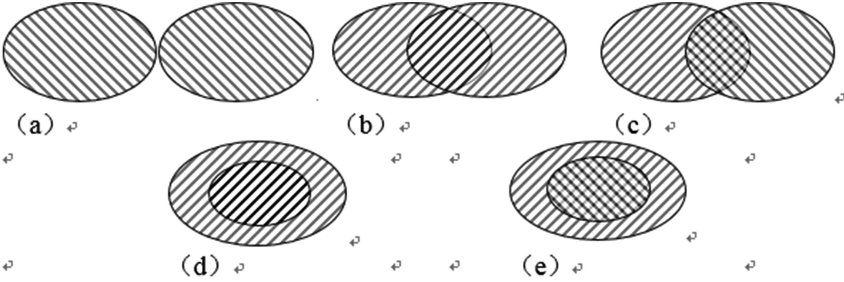

Fig. 4. Relationship between policies

As shown in Fig. 4(c) and (e) describe the existence of policy conflicts, Fig. 4(a) (b) and (d) describe cases of no conflict.

As mentioned above, if there is a conflict, you can choose the appropriate conflict policy considering the functions of the policy and the scope of the intersection in order to implement the maximization of the semantics of the policy.

If the conflict in case of Fig. 4(c) occurs, it indicates that the conflict occurs in the local scope of the two policies. It is the time we further analyze the influence of the scope of the conflict to compromise policy. If the scope of conflicts have little impact on compromise policy, we can make compromise policy valid outside the scope of the conflicts. If the scope of conflicts have much impact on compromise policy, then the compromise policy must be completely removed.

If a conflict in Fig. 4(e) occurs, it indicates there is a comprehensive conflict policy. At this point, if the local conflict policy is chosen as a compromise policy, we do further analysis by the above method. If comprehensive conflict policy is chosen as a compromise policy, it can be completely removed.

The method is to set the conflict scope set C during the policy combination process and make the operation under the following conditions:

if $e_i \cap e_j^{'} \neq \emptyset$ and $A_i \cup A_j^{'} = \{S, D\}$ then C $\cup \left(e_i \cap e_j^{'}\right) \overset{assign}{\rightarrow}$ C
if C $= \emptyset$, no conflict
if C $= e$, completely conflict in τ
if C $= e^{'}$, completely conflict in $\tau^{'}$
if C$\subseteq e$, partly conflict in τ

$$\text{if C} \subseteq e^{'}, \text{partly conflict in } \tau^{'}$$

Obviously, the necessary and sufficient conditions for conflicts in form can be expressed as $C \neq \emptyset$.

If the local conflicts compromise policy is required to be valid outside the scope of the conflict in the process of policy combination, we can replace D or S to $\emptyset$ in accordance with the priority policy. Therefore, we get the following forms.

$$A_i \cup A_j' = \begin{cases} A_i, & if\ \ C \neq \emptyset\ \text{且} P(A_i) > P\left(A_j'\right) \\ A_j', & if\ \ C \neq \emptyset\ \text{且} P(A_i) < P\left(A_j'\right) \end{cases}$$

Among them, $P(A_i)$ represents the priority of the corresponding actions attached in policies.

5 Experiments of Policy Combination Algorithm

Pyretic project and NetCore project share similarities in contents, Therefore, this section chooses the Pyretic project as the experimental platform to test the policy combination algorithm.

5.1 Experimental Environment

In order to test the effects of policy combination, this study builds an OpenFlow (OpenFlow version 1.1. 0) & SDN network test platform based on Mininet and POX controller, and the test platform runs under Linux.

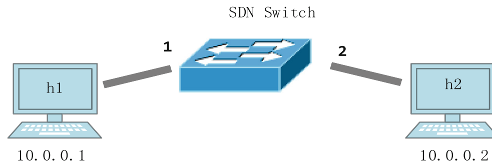

Fig. 5. The experimental topology

At the beginning of the experiment, firstly we need to implement the shell script / pyretic/mininet.sh to start up Mininet and build the network topology as shown in Fig. 5.

The topology uses two Mininet simulation hosts (h1 and h2) as well as an OpenFlow switch, and IP of the two hosts are 10.0.0.1 and 10.0.0.2, respectively.

Then Pyretic project is inputted into integrated development environment PyCharm.

In the condition without any policy applications, host h1 and host h2 are connected physically but they are logically disconnected. Set the host h1 and h2 connected with the policies as follow, Fig. 6.

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *

link = (match(dstip='10.0.0.1')>>fwd(1)) + (match(dstip='10.0.0.2')>>fwd(2))

def main():
    return link
```

Fig. 6. The policy of connection

The function of this policy is to forward the packet of destination IP address 10.0.0.1 to interface 1 and to forward the packet of destination IP address 10.0.0.2 to interface 2. After the application of the policy, the result was shown in Fig. 7. It shows that the two hosts have been successfully connected physically and logically so that the experimental environment has been successfully completed.

```
From 10.0.0.2 icmp_seq=97 Destination Host Unreachable
From 10.0.0.2 icmp_seq=98 Destination Host Unreachable
From 10.0.0.2 icmp_seq=99 Destination Host Unreachable
64 bytes from 10.0.0.1: icmp_req=100 ttl=64 time=0.191 ms
64 bytes from 10.0.0.1: icmp_req=101 ttl=64 time=0.243 ms
64 bytes from 10.0.0.1: icmp_req=102 ttl=64 time=0.038 ms
64 bytes from 10.0.0.1: icmp_req=103 ttl=64 time=0.041 ms
```

Fig. 7. Connected policy application results

5.2 The Experiment of Policy Conflict

The result of policy conflict as following Fig. 8.

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *

conflict = (match(dstip='10.0.0.1')>>fwd(1)) + (match(dstip='10.0.0.2')>>fwd(2)) + (match(dstip='10.0.0.1')>>drop)

def main():
    return conflict
```

Fig. 8. Policy conflict sample

The policy adds actions of forwarding and packet dropping to packets of destination IP address 10.0.0.1. According to the definition of conflict in Sect. 4, the modified policy generates to policy conflicts. After the policy is implemented, the result was shown in Fig. 9.

```
64 bytes from 10.0.0.1: icmp_req=34 ttl=64 time=0.033 ms
64 bytes from 10.0.0.1: icmp_req=35 ttl=64 time=0.227 ms
64 bytes from 10.0.0.1: icmp_req=36 ttl=64 time=0.037 ms
64 bytes from 10.0.0.1: icmp_req=37 ttl=64 time=0.136 ms
```

Fig. 9. The result of conflict policy application results

The results indicates that the function of sub-policy “match (dstip = ‘10.0.0.1’) >> drop” was not achieved and has no prompt.

According to the discussion of the policy conflict in Sect. 4, policy conflicts can not be resolved but can be detected. Therefore, this study adds the detection of policy conflict and prompt mechanism to the policy combination algorithm in Pyretic projects. After the policy is utilized in the modified Pyretic project, the results was shown in Fig. 10.

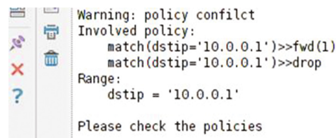

Fig. 10. Policy conflict prompt

5.3 Summary

This part carries out a policy conflict test. The policy conflict test shows that the modified policy combination algorithm can effectively detect the conflict as well as prompt.

6 The Summary and Prospect

This paper improves the original NetCore programming language. The design of policy combination algorithm in this paper mainly refers to the policy combination algorithm in the operation of NetCore system. And on this basis, we added the action of packet drop to switches. We propose the detection and resolution project after occurrences of the policy conflict. Finally in order to verify whether the policy combination algorithm can effectively detect and prompt policy conflict, we implement relevant experiences to testify the effect of the algorithm based on the Pyretic project.

This paper studies and analyzes on the policy management of the controller, but due to the limited time and proficiency, the following aspects need to be improved.

(1) We neglect the analysis and comparison of the algorithm performance which will be completed after the work of forwarding management subsystem is finished.
(2) The algorithm need to be experienced several complex situations and to find the defects of it. We will finish these tasks in the future.

References

1. Ballani, H., Francis, P.: CONMan: a step towards network manageability. J. ACM SIGCOMM Comput. Commun. Rev. **37**(4), 205–216 (2007)
2. Chen, X., Mao, Y., Mao, Z.M., et al.: Declarative configuration management for complex and dynamic networks. In: International Conference. ACM, pp. 1–12 (2010)
3. Loo, B.T., Hellerstein, J.M., Stoica, I., et al.: Declarative routing: extensible routing with declarative queries. J. ACM SIGCOMM Comput. Commun. Rev. **35**(4), 289–300 (2005)
4. Doria, A., Salim, J.H., Haas, R., et al.: Forwarding and Control Element Separation (ForCES) Protocol Specification. In: IETF RFC 5810 (Proposed Standard) (2010)
5. Mckeown, N., Andemon, T., Balakrishnan, H., et al.: OpenFlow: enabling innovation in campus networks. J. ACM SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
6. Yi, Z., Yiqiang, H., Xiaofeng, H.: Characteristics, development and future of SDN. J. Telecommun. Sci. **29**(9), 102–107 (2013). (in Chinese)
7. Monsanto, C., Foster, N., Harrison, R., et al.: A compiler and run-time system for network programming languages. J. ACM SIGPLAN Not. **47**(1), 217–230 (2012)
8. Kim, H., Reich, J., Gupta, A., et al.: Kinetic: verifiable dynamic network control. In: USENIX NSDI 2015 (2015)
9. Reich, J., Monsanto, C., Foster, N., et al.: Modular SDN programming with Pyretic. USENIX; login **38**(5), 128–134 (2013)
10. OpenFlow Switch Specification Version 1.0. 0. OpenFlow Switch Consortium (2009)
11. Jin, X., Rexford, J., Walker, D.: Incremental update for a compositional SDN hypervisor. In: Third Workshop on Hot Topics in Software Defined Networking. ACM, pp. 187–192 (2014)
12. Foster, N., Harrison, R., Freedman, M.J., et al.: Frenetic: a network programming language. ACM SIGPLAN Not. **46**(9), 279–291 (2011). ACM
13. Voellmy, A., Kim, H., Feamster, N.: Procera: a language for high-level reactive network control. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks. ACM, pp. 43–48 (2012)
14. Hickey, J.: Introduction to the Objective Caml programming language. Verfügbar unter. http://docs.happycoders.org/html/dev/ocaml/index.php
15. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. ACM, pp. 51–64 (2002)
16. Berry, G., Boudol, G.: The chemical abstract machine. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 81–94 (1989)