# OpenFlow rules interactions: definition and detection

Roberto Bifulco, Fabian Schneider
NEC Laboratories Europe
{firstname.lastname}@neclab.eu

***Abstract*—Software Defined Networking (SDN) is a promising architecture for computer networks that allows the development of complex and revolutionary applications, without breaking the backward compatibility with legacy networks. Programmability of the *control-plane* is one of the most interesting features of SDN, since it provides a higher degree of flexibility in network management: network operations are driven by ad-hoc written programs that substitute the classical combination of firewalls, routers and switches configurations performed in traditional networks. A successful SDN implementation is provided by the OpenFlow standard, which defines a rule-based programming model for the network. The development process of OpenFlow applications is currently a low-level, error prone programming exercise, mainly performed manually in both the implementation and verification phases. In this paper we provide a first formal classification of OpenFlow rules interactions into a single OpenFlow switch, and an algorithm to detect such interactions in order to aid the OpenFlow applications development. Moreover, we briefly present a performance evaluation of our prototype and how it has been used in a real-word application.**

## I. Introduction

Software Defined Networking (SDN) introduces a new architecture that separates the network *control-plane* from the network *data-plane*. Network equipment, i.e., network switches, is not involved in control-plane operations anymore: the control-plane is implemented in a separated device, usually called *Controller*, which communicates with network switches through well-established interfaces and protocols, introducing greater flexibility in the network, e.g., allowing the programmability of the control-plane. Programmability is actually one of the most important features of SDN: the control-plane behavior can be defined writing "network programs" that manage a set of switches, providing rich network applications and features. Programming the network, instead of configuring it, on the one hand gives a powerful tool to develop revolutionary applications, but on the other, it also increases the difficulty of providing efficient and reliable networks: the flexibility of a program, like in the case of classical computer software, can lead to errors that are even more complex to handle, given the ineherently distributed and asynchronous nature of a network. In some way, SDN is bringing into computer networks the same shift that in past decades electronic devices had from special purpose machines, to general purpose ones. With SDN, the current special purpose network can become a general purpose, hence programmable, network. Programmability, in a new context like computer networks, calls for new programming models, tools and languages. In this sense, to continue with the general purpose computer metaphor, SDN is still in the "machine language" phase, where the programming languages are rudimentary, strictly connected to the hardware, and the main part of network programming is still performed manually, without the aid of any tool. In last years, OpenFlow [8] has been a successful SDN implementation. The Open Networking Foudation [1] (ONF), that is responsible for the OpenFlow specification, currently involves a number of academic and industrial partners, making OpenFlow a promising technology, with a good number of industrial players already implementing the specification in their products. OpenFlow uses a flow abstraction to describe the network behavior, with switches configuration described through the use of flow forwarding rules: control-plane programs are written in a traditional high-level programming language, e.g., Python or Java, to provide a set of entries to be installed at different switches' flow tables (FT) as output. In this paper we present a set of definitions to characterize the interactions of entries installed in an OpenFlow switch, and an algorithm to automatically detect such interactions, with the aim of aiding the development and debugging of OpenFlow network applications. The paper is organized as follows. In section II we present related work. Section III introduces the OpenFlow switches programming model, while in section IV we present our contribution on the OpenFlow rules interaction definitions and in section V we introduce the algorithm we designed to detect such interactions with its evaluation. In section VI some possible application scenarios and a concrete use case are presented. In section VII we conclude and present future work.

## II. Related work

OpenFlow network programming is a problem addressed in some other works. Frenetic [4] is a high-level language based on the functional programming paradigm, that provides the programmer with an omniscient, centralized view of the network. A run-time system, linked to the language, "translates" the high-level instructions in a set of low-level packet processing rules, and manages them interacting with network equipments. NetCore [7] is an evolution of Frenetic, that extends the high-level language and provides some improvement in the compilation algorithms and run-time system, trying to

[1] https://www.opennetworking.org/

speed up the network performance. To test the correctness of OpenFlow applications, NICE was proposed in [3]. NICE is a tool for automatic OpenFlow applications testing, that combines model checking and concolic execution to explore the state space of OpenFlow programs written for the NOX [5] controller platform. FlowChecker [1] uses manually built binary decision diagrams to encode OpenFlow rules and then applies model checking in order to detect OpenFlow switches misconfigurations. Header space analysis [6] is a technique that can be applied to OpenFlow networks to check if a given property, e.g., reachability between two nodes, is provided by the network: it requires the modeling, either performed manually or automatically, of transform functions for the box composing the network, in order to map the transformation to which a packet undergo when traversing such a network.

The definitions and the algorithms we present in this paper are targeted at identifying and automatically detect OpenFlow table entries interaction in a single switch, hence, our work can be seen as a basis for new OpenFlow tools targeted at applications development and debugging, that, differently from, e.g., [1], requires no representation conversion of the flow table entries (FTE). Moreover, our work aims at providing a semantic interpretation of the interactions among FTEs in the switch, in order to help the developer in discovering bugs or checking the behavior of its application at runtime. Differently from the reported verification techniques our approach is limited to a single switch, nevertheless, our tool points at the exact flow table entries in the switch that are interacting among them and provides a clear definition of the discovered interactions.

## III. Rule-based programming

OpenFlow-enabled Switches (OFS) behavior is configured using FTEs, installed by means of the OpenFlow (OF) protocol [8]. A FTE is defined by the *match set*, that defines to which network flows the entry is applied, the *action set*, that defines the elaborations and the forwarding decision that must be applied to the matched flows, a *priority*, to relatively order among them the entries installed in a switch's FT, and an *expiration time*, specified through the use of timeouts. According to the OF specification [9], only the highest priority FTE that matches a packet is applied to that packet.

Because of this definition, the switch behavior is dictated by the combination of all the installed FTEs. In fact, looking at a single FTE does not suffice to understand the behavior of the switch in respect to the flow identified by the match set of such FTE, since other FTEs, with higher priorities, can introduce a different behavior.

The problem is usually raised up in the process of developing an OF application, that is, basically the process of defining when, where and what FTEs have to be installed at managed OFSes. This issue can be even more problematic if we are trying to extend an already developed OpenFlow application, or if we are combining several applications at the same time.

## IV. Interactions definition

To characterize the behavior of an OFS, we define FTEs interactions extending the work presented in [2]. An interaction is a particular relation between two FTEs. An interaction may be expected, i.e., the network programmer is aware that FTEs are interacting, or it may be raised by unexpected relations between FTEs, e.g., as result of a programming error. To define the possible interactions that can occur between two FTEs, we firstly define the relations that can be in place among match sets and among action sets. Then, based on such relations combination, we define FTEs interaction types.

### A. Match sets relations

A match set is composed by a number of match fields. Typical match fields are *l2 source [destination] address*, *l2 protocol type*, *l3 protocol type*, *l3 source [destination] address*, etc. All match fields can have a wildcard as value, that means *any value*. Some match fields can have partially wildcarded values, e.g., an l3 address can be associated with a bitmask to specify which bits are wildcard. Because of the presence of wildcards, there are four different relations among two match fields of the same type. The relation between match field $f_0$ and match field $f_1$ can be one of the following: *disjoint*, when match fields have different values ($f_0 \neq f_1$); *equal*, if $f_0$ value is the same of $f_1$ ($f_0 = f_1$); *subset*, if $f_0$ value is a subset of the value of $f_1$ ($f_0 \subset f_1$), e.g, $f_0$ has a defined value, while $f_1$ value is a wildcard; *superset*, when $f_0$ value is a superset of the value of $f_1$ ($f_0 \supset f_1$), e.g., $f_0$ value is an IP address in the form *192.168.0.0/16*, while $f_1$ value is *192.168.1.0/24*.

Using the defined match fields relations, we are now able to define the relations between two match sets. These relations can actually be described as classical relations among sets, anyway, we provide a formal definition of them based on the mutual relations of the match fields composing the match sets. We adopt this formal definition since it is the basis of our algorithm implementation. The relation between match set $M_0$ and match set $M_1$ can be one of the following:

*Disjoint*: $M_0$ and $M_1$ are *disjoint* if every field *i* in $M_0$ is *disjoint* with the correspondent field in $M_1$ ($M_0 \neq M_1$);

$$M_0 \neq M_1 \text{ if } \forall \text{ i } f_i^0 \neq f_i^1,\ f_i^0 \in M_0 \wedge f_i^1 \in M_1$$

*Exactly matching*: $M_0$ and $M_1$ are *exactly matching* if every field *i* in $M_0$ is *equal* to the correspondent field in $M_1$ ($M_0 = M_1$);

$$M_0 = M_1 \text{ if } \forall \text{ i } f_i^0 = f_i^1,\ f_i^0 \in M_0 \wedge f_i^1 \in M_1$$

*Subset*: $M_0$ is a *subset* of $M_1$ if one field *j* of $M_0$ is *subset* of the correspondent field of $M_1$ and any other field *i* in $M_0$ is *equal* or *subset* of the correspondent field in $M_1$ ($M_0 \subset M_1$);

$$M_0 \subset M_1 \text{ if } (\exists j | f_j^0 \subset f_j^1) \wedge [\forall \text{ i } (f_i^0 = f_i^1) \vee (f_i^0 \subset f_i^1)]\ ,\ i \neq j \wedge f_i^0, f_j^0 \in M_0 \wedge f_i^1, f_j^1 \in M_1$$

*Superset*: $M_0$ is a *superset* of $M_1$ if one field *j* of $M_0$ is *superset* of the correspondent field of $M_1$ and any other field *i* in $M_0$ is *equal* or *superset* of the correspondent field in $M_1$ ($M_0 \supset M_1$);

$$M_0 \supset M_1 \text{ if } (\exists j | f_j^0 \supset f_j^1) \wedge [\forall \text{ i } (f_i^0 = f_i^1) \vee (f_i^0 \supset f_i^1)] \ , \\ i \neq j \wedge f_i^0, f_j^0 \in M_0 \wedge f_i^1, f_j^1 \in M_1$$

*Correlated*: $M_0$ is *correlated* with $M_1$ at least one field *j* of $M_0$ is *superset* of the correspondent field of $M_1$ and any other field *i* in $M_0$ is *equal* or *subset* of the correspondent field in $M_1$ ($M_0 \sim M_1$);

$$M_0 \sim M_1 \text{ if } \exists j | f_j^0 \supset f_j^1 \wedge \exists i | f_i^0 \subset f_i^1 \ , \\ i \neq j \wedge f_i^0, f_j^0 \in M_0 \wedge f_i^1, f_j^1 \in M_1$$

### B. Action sets relations

An action set contains zero or more actions. An action has a type and a value, typical action types are *forward to port X*, *rewrite network source [destination] address*, *pop [push] VLAN tag*, etc. An action can be *equal*, *related* or *disjoint* in respect to another action. An action $a_0$ is *equal* to an action $a_1$ ($a_0 = a_1$) only if they have the same types and values, if the types are equal but values are different, the actions are *related* ($a_0 \sim a_1$). An action $a_0$ is disjoint from action $a_1$ ($a_0 \neq a_1$), if their types are different. Depending on the relations of the contained actions, the relation between action set $A_0$ and action set $A_1$ can be one of the following:

*Disjoint*: $A_0$ is *disjoint* from $A_1$ if for any action in $A_0$, such an action is disjoint from any action in $A_1$ ($A_0 \neq A_1$);

$$A_0 \neq A_1 \text{ if } \forall \text{ i,j } a_i^0 \neq a_j^1, \ a_i^0 \in A_0 \wedge a_j^1 \in A_1$$

*Related*: $A_0$ is *related* to $A_1$ if there is at least one action from $A_0$ that is related to an action of $A_1$ ($A_0 \sim A_1$);

$$A_0 \sim A_1 \text{ if } \exists i, j | a_i^0 \sim a_j^1, \ a_i^0 \in A_0 \wedge a_j^1 \in A_1$$

*Subset*: $A_0$ is a *subset* of $A_1$ if all the actions contained in $A_0$ are equal to actions contained in $A_1$, and $A_1$ contains more action than $A_0$ ($A_0 \subset A_1$);

$$A_0 \subset A_1 \text{ if } \forall \text{ i,j } a_i^0 = a_j^1 \wedge |A_0| < |A_1|, \ a_i^0 \in A_0 \wedge a_j^1 \in A_1$$

*Superset*: $A_0$ is a *superset* of $A_1$ if all the actions contained in $A_0$ are equal to actions contained in $A_1$, and $A_0$ contains more action than $A_1$ ($A_0 \supset A_1$);

$$A_0 \supset A_1 \text{ if } \forall \text{ i,j } a_i^0 = a_j^1 \wedge |A_0| > |A_1|, \ a_i^0 \in A_0 \wedge a_j^1 \in A_1$$

*Equal*: $A_0$ is *equal* to $A_1$ if all the actions contained in $A_0$ are equal to actions contained in $A_1$, and $A_1$ and $A_0$ contains the same number of actions ($A_0 = A_1$);

$$A_0 = A_1 \text{ if } \forall \text{ i,j } a_i^0 = a_j^1 \wedge |A_0| = |A_1|, \ a_i^0 \in A_0 \wedge a_j^1 \in A_1$$

### C. Interaction types

To define interactions between two FTEs we look at entries' priorities, match sets relations and action sets relations. Considering a FTE $R_x$, with match set $M_x$ and action set $A_x$, and a FTE $R_y$, with match set $M_y$ and action set $A_y$, assuming that $R_x$ priority is always smaller than $R_y$ priority (If it is not the case we will explicitly point it out), The interaction between $R_x$ and $R_y$ can be of one of the types listed in table I. Here we provide a brief description of them:

TABLE I: OpenFlow Rules interactions

| MATCH SET | ACTION SET | PRIORITY |
|---|---|---|
| Duplication | | |
| $M_x = M_y$ | $A_x = A_y$ | $prio(R_x) = prio(R_y)$ |
| Redundancy | | |
| $M_x \subset M_y$ | $A_x = A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \supset M_y$ | $A_x = A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \sim M_y$ | $A_x = A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x = M_y$ | $A_x = A_y$ | $prio(R_x) < prio(R_y)$ |
| Generalization | | |
| $M_x \supset M_y$ | $A_x \neq A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \supset M_y$ | $A_x \sim A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \supset M_y$ | $A_x \subset A_y$ | $prio(R_x) < prio(R_y)$ |
| Shadowing | | |
| $M_x \subset M_y$ | $A_x \neq A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \subset M_y$ | $A_x \sim A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \subset M_y$ | $A_x \supset A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x = M_y$ | $A_x \neq A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x = M_y$ | $A_x \sim A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x = M_y$ | $A_x \supset A_y$ | $prio(R_x) < prio(R_y)$ |
| Correlation | | |
| $M_x \sim M_y$ | $A_x \neq A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \sim M_y$ | $A_x \sim A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \sim M_y$ | $A_x \subset A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \sim M_y$ | $A_x \supset A_y$ | $prio(R_x) < prio(R_y)$ |
| Inclusion | | |
| $M_x = M_y$ | $A_x \subset A_y$ | $prio(R_x) < prio(R_y)$ |
| $M_x \subset M_y$ | $A_x \subset A_y$ | $prio(R_x) < prio(R_y)$ |
| Extension | | |
| $M_x \supset M_y$ | $A_x \supset A_y$ | $prio(R_x) < prio(R_y)$ |

***Duplication***: assuming that the priorities of two entries are equal, they are duplicated if they have the same match and action sets.

***Redundancy***: redundant FTEs have the same effect on the subset of flows matched by both entries, hence, in some conditions (e.g., no interactions with third rules), depending on the entries priorities, one of the entries could be deleted without affecting the datapath behavior, or the entries could be aggregated.

***Generalization***: entries have different actions, but $R_x$ matches a superset of the flows matched by $R_y$. So, action set $A_y$ will be applied to flows matched by $M_x \cap M_y$, while to the flows matched by $M_x - M_y$ the action set $A_x$ will be applied.

***Shadowing***: if $R_x$ is shadowed by $R_y$, then $R_x$ is never applied, since all the flows are matched by $R_y$ before that $R_x$ is examined.

***Correlation***: the FTEs have different match sets, but the intersection of these match sets is not void, so, to flows that are in the intersection only higher priority entry's action set ($A_y$) will be applied. Note that this interaction is different from the *shadowing* interaction, since for some flows the lower priority entry ($R_x$) is still applied.

***Inclusion***: *inclusion* interaction is similar to *shadowing*. The lower priority entry is never applied "as is", but its actions are still applied in combination with the actions of another entry (of higher priority). I.e., $R_x$ is never applied, but, since the

**Algorithm 1** $matchset\_relation(R_x, R_y)$

```
relation ← undetermined
field_relations ← compare_fields(R_x, R_y)

for field in match_fields do

    if field_relations[field] = equal then
        if relation = undetermined then
            relation ← exact
        end if

    else if field_relations[field] = superset then

        if relation = subset or relation = correlated then
            relation ← correlated
        else if relation ≠ disjoint then
            relation ← superset
        end if

    else if field_relations[field] = subset then

        if relation = superset or relation = correlated then
            relation ← correlated
        else if relation ≠ disjoint then
            relation ← subset
        end if

    else
        relation ← disjoint
    end if

end for

return relation
```

**Algorithm 2** $interactions\_detection(R_x, R_y)$

```
interaction ← None

ms_relation ← matchset_relation(R_x, R_y)
as_relation ← actionset_relation(R_x, R_y)

if priority(R_x) = priority(R_y) and ms_relation = exact and as_relation = equal
then
    interaction ← duplication

else if ms_relation ≠ disjoint then

    if ms_relation = correlated then
        if as_relation = equal then
            interaction ← redundancy
        else
            interaction ← correlation
        end if

    else if ms_relation = superset then
        if as_relation = equal then
            interaction ← redundancy
        else if as_relation = superset then
            interaction ← extension
        else
            interaction ← generalization
        end if

    else if ms_relation = exact then
        if as_relation = equal then
            interaction ← redundancy
        else if as_relation = subset then
            interaction ← inclusion
        else
            interaction ← shadowing
        end if

    else if ms_relation = subset then
        if as_relation = equal then
            interaction ← redundancy
        else if as_relation = subset then
            interaction ← inclusion
        else
            interaction ← shadowing
        end if

    end if
end if

return interaction
```

action set $A_x$ is a subset of the action set $A_y$, the actions of $A_x$ are still applied, but only in combination with the actions of $A_y$.

***Extension***: *extension* interaction is similar to *generalization*. An entry with lower priority is extending the action set applied by another entry, adding more actions. Only to the flows matched by $M_x - M_y$ the extended actions are applied.

## V. Interactions detection

To detect the interactions presented in previous sections, we designed an *interactions detection algorithm* (IDA). The algorithm takes two entries, $R_x$ and $R_y$, assuming $prio(R_x) \leq prio(R_y)$, and detects the interaction generated by the composition of the entries. We assume that $R_x$ and $R_y$ are data structures containing all the information we need regarding a FTE, i.e., match set, action set and priority. The algorithm uses two auxiliary algorithms to find match sets and action sets relations, respectively, these algorithms are represented by the functions $matchset\_relation(R_x, R_y)$ and $actionset\_relation(R_x, R_y)$.

Algorithm $matchset\_relation(R_x, R_y)$ finds the relation between match sets, by firstly comparing match fields one by one, using the $compare\_fields(R_x, R_y)$ function. Then it cycles among the fields' relations to evaluate the match sets relation, by applying a state machine where each state corresponds to one of the match sets relation (plus "undetermined") and transitions corresponds to match fields relations [2].

Algorithm $actionset\_relation(R_x, R_y)$ is not presented in these pages, since its implementation is simpler and can be easily derived by the definitions of action sets relations presented in previous sections.

Algorithm $interaction\_detection(R_x, R_y)$ uses the just defined functions to apply the interaction types definitions in order to find if the FTEs generate an interaction and, in that case, which type of interaction. We implemented a prototype of our algorithm in Python, integrating it into the NOX controller platform [5]. We performed a first evaluation of our implementation using randomly generated FTEs sets. We generated FTE sets defining three parameters: (i) number of entries in the set, (ii) number of non-wildcard match fields, (iii) probability of generating the same value for match fields belonging to different entries. This last parameter provides a mean to set the number of interactions in the FTEs set, e.g., a low probability corresponds to fewer interactions in the set. Action sets were also generated randomly, selecting from 1 to 3 actions per action set. For each FTEs set, we run the algorithm several times to extract a mean of the running times. The testbed machine is an Ubuntu Linux virtual machine, equipped with 2 GB of RAM and running on a dedicated cpu-core of an Intel CPU E7600 @ 3.06GHz. The execution times are shown in figures 1 and 2: our first implementation is providing a linear increasing of the execution time with the growing of the FTEs set dimension. Moreover, the greater is the number of wildcard match fields, the lower is the execution time. Interestingly, the current implementation provides better performance when the number of interactions in the rule set is bigger (Figure 1).

[2]We are using a "foreach" notation in this *for* cycle, putting in the $field$ variable, one by one, any value found in the $match\_fields$ variable, that contains a list of the all possible fields name

## VI. Use cases

The algorithm and the interactions definition we provided in this paper can be used as a basis for OpenFlow network

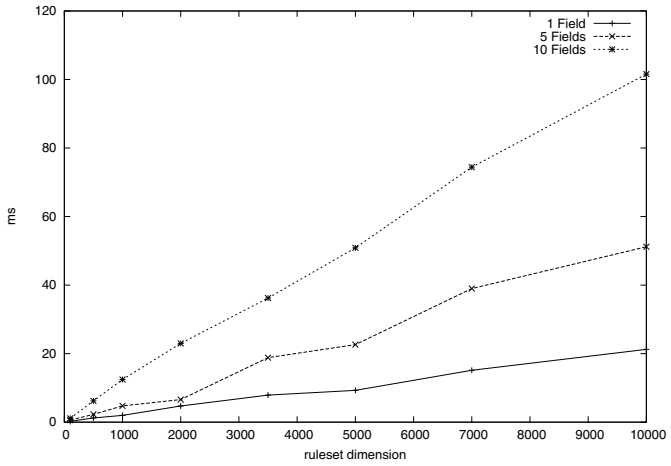


Fig. 1: Interactions detection algorithm performance with low match sets overlapping values probability. Entries in FTEs set have from 1 to 10 non-widlcard fields in the corresponding match sets.

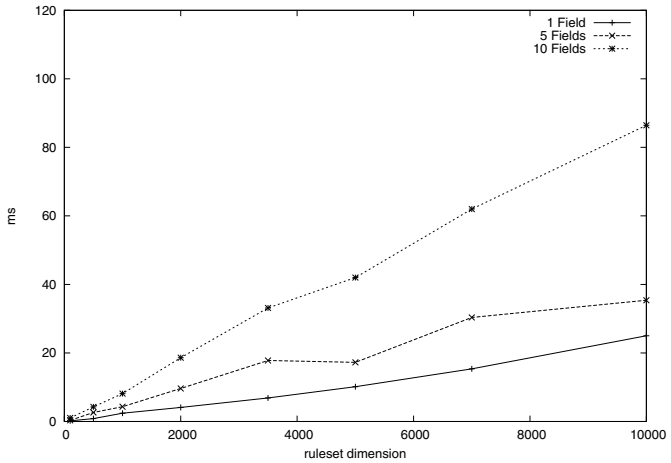


Fig. 2: Interactions detection algorithm performance with high match sets overlapping values probability. Entries in FTEs set have from 1 to 10 non-widlcard fields in the corresponding match sets.

development methodologies and for tools targeted at aiding OpenFlow network programming, analysis, switches management and optimization, etc.. In this section we briefly describe some possible applications, then, we present a concrete use case. During the development of an OpenFlow application, the IDA can be used as debug tool to verify the interactions among entries installed in a switch. In the simplest cases, it can point out entries duplications and redundancies, reducing any overhead in the developed application, or it can detect unexpected FTEs interactions that would lead to the wrong handling of some traffic flows. IDA could be also integrated in advanced controller platforms as a mean to analyze FTEs in order to provide some forms of automation in FTEs management. E.g., rejecting duplicated entries, reordering entries' priorities to avoid shadowing, FTE splitting to avoid redundancy and correlation, etc.. An advanced controller would require more work on the semantic part of the entries management, but the interactions detection algorithm is the enabling technology to analyze FTEs interactions. A different application would be the use of IDA to compare a FTE against a set of FTEs that is used as an admission control policy. Specifying the allowed interactions with the given policy FTEs set, a FTE can be checked to be admitted or not. Complex policies can specify which operations are allowed on which flows, using properly specified FTEs set and allowed interactions. As last application example, the IDA can be used to optimize the FTEs installed in a switch. FTEs can be checked against other FTEs to find interactions, and, in case, they can be rewritten to split or aggregate them for a better use of the switch hardware resources (E.g., some switches have multiple flow tables with different properties). Clearly, we presented only a few examples of possible IDA exploitations. In the following part of this section we briefly introduce a concrete application for the development of a real OpenFlow network application.

### A. Extending an OpenFlow application

Follow-Me Cloud (FMC) is a technology, developed at NEC Laboratories Europe, that provides mobility features in a TCP/IP network for both users and services, maintaining all the ongoing network connections active. FMC is applied to a TCP/IP network in which L2 access networks are connected to a "core" network, that provides connectivity among them, through OpenFlow-enabled switches (OFS). To provide mobility to a mobile node (MN) that is changing its access network from an "home" to a "foreign" network, FMC requires that a new IP address, belonging to the foreign network, is assigned to MN to work as "locator". The original IP address of MN, that belongs to the home network, is still used by MN itself and by any node that is communicating with MN, since it works as "identifier". When a network node (we call such a node *correspondent node* or CN in short) sends a packet to MN, it uses the *identifier* address as destination address. The OFS connecting the CN's network (CNet) to the core network performs an address translation, to substitute the *identifier* with the *locator* address. When the packet reaches the foreign network, the OFS at the edge of such network performs a new translation, substituting the *locator* with the *identifier*, in order to deliver the original packet to the MN.

The current FMC implementation uses NOX [5] as controller platform. To make deployment of FMC in a TCP/IP network as easy as the placement of OFSes at the edge of L2 access networks, we decided to extend an OpenFlow learning switch application with FMC functionalities. A learning switch (LS) application provides Ethernet Switch functionalities, by learning MAC addresses and associating them with switch ports. LS installs proper FTEs to forward to the correct port a packet with a given destination MAC address. Rules 3 and 4 from table II are a typical example of two rules installed by the LS application, to provide connectivity among a node *X* and the gateway of the *X*'s network (*routerA*). OFSes are controlled

TABLE II: OF-Switch installed rules

| # | Matching criteria | | | | Priority | Actions |
|---|---|---|---|---|---|---|
| | DL_DST | DL_TYPE | NW_SRC | NW_DST | | |
| 1 | $MAC_{routerA}$ | IP | * | $identifier$ | 200 | set NW_DST: $locator$; out: $PORT_{routerA}$ |
| 2 | $MAC_x$ | IP | $locator$ | * | 200 | set NW_SRC: $identifier$; out: $PORT_x$ |
| 3 | $MAC_{routerA}$ | * | * | * | 100 | out: $PORT_{routerA}$ |
| 4 | $MAC_x$ | * | * | * | 100 | out: $PORT_x$ |

by the learning switch application, ensuring traditional TCP/IP operations, while only the flows directed to mobile nodes are handled by FMC-related FTEs. As an example, the addition of FMC to a CNet's OFS requires that a destination address translation is performed on any flow destined to an *identifier* address. At the same time, a source address translation must be performed on any flow with *locator* as source network address (See rules 1 and 2 from table II). We have to ensure that FMC-related FTEs are not shadowed by LS-related FTEs. Our FMC implementation uses the interactions detection algorithm to guarantee that newly installed rules are always involved in generalization interactions with LS-related FTEs, i.e., LS-related FTEs are generalizing FMC-related FTEs. The algorithm has been integrated into the FMC OpenFlow Controller and it is used as a runtime tool, to define the required priority value to assign to newly generated FTEs, and as debug and validation tool, to check the absence of shadowing interactions into switches.

## VII. Conclusion and future work

In this paper we presented a formal definition for the interactions of entries installed in a OpenFlow Switch's flow table, an algorithm for the automatic detection of such interactions and we showed how the algorithm has been used to develop a real OpenFlow application. Furthermore, we evaluated our prototype implementation of the proposed algorithm to understand the actual applicability in other real-word scenarios. Our evaluation shows that the dimension of the managed entries set can limit the applicability of the algorithm as a runtime tool for the definition of a complex management policy, since the execution times, when the number of entries is in the order of thousands, can negatively affects the network performance. Anyway, the algorithm is well suited for the development phase of OpenFlow applications, e.g., as debug tool, or in applications where the number of managed FTEs per switch is no more than a few hundreds, which is a dimension currently in line with most hardware OpenFlow switches' flow table size. In future work we plan to improve the algorithm execution times by exploiting, e.g., smart ordering of entries or FTEs set reduction strategies. Moreover, we are extending its applicability to the detection of network-wide FTEs interactions.

## References


[1] Al-Shaer, E., and Al-Haj, S. Flowchecker: configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration* (New York, NY, USA, 2010), SafeConfig '10, ACM, pp. 37–44.

[2] Al-Shaer, E., and Hamed, H. Modeling and management of firewall policies. *Network and Service Management, IEEE Transactions on 1*, 1 (april 2004), 2 –10.

[3] Canini, M., Venzano, D., Peresini, P., Kostic, D., and Rexford, J. A nice way to test openflow applications. *EPFL Technical Report* (Oct. 2011).

[4] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: a network programming language. *SIGPLAN Not. 46*, 9 (Sept. 2011), 279–291.

[5] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev. 38*, 3 (2008), 105–110.

[6] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 9–9.

[7] Monsanto, C., Foster, N., Harrison, R., and Walker, D. A compiler and run-time system for network programming languages. *SIGPLAN Not. 47*, 1 (Jan. 2012), 217–230.

[8] Openflow - http://www.openflow.org/.

[9] Openflow specification 1.1.0 - http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.