Action Computation for Compositional Software-Defined Networking

Heng Pan*†, Gaogang Xie*, Peng He*, Zhenyu Li*, Laurent Mathy‡
*ICT, CAS, China, †University of CAS, China, ‡University of Liége, Belgium
{panheng, xie, hepeng, zyli}@ict.ac.cn, laurent.mathy@ulg.ac.be

***Abstract*—Software-defined networking (SDN) envisions the support of multiple applications collaboratively operating on the same traffic. Policies of applications are therefore required to being composed into a rule list that represents the union of application intents. In this context, ensuring the correctness and efficiency of composition for match fields as well as the associated actions is the fundamental requirement. Prior work however focuses only on the composition of match fields and assumes simple concatenation for action composition. We show in this paper that simple concatenation can result in incorrect behavior (for parallel composition) and inefficiency (for sequential composition) for actions composition. To address this issue, we formalized the action composition problem and prove a feasibility condition on the composition of rule actions. We then propose a graph-based approach that facilitates fast composition of action lists without action redundancy. Our proposed approach has been integrated into the CoVisor code base and the evaluation results show its fitness for purpose.**

***Index Terms*—Software-defined Networking, composition, action**

I. Introduction

Software-Defined Networking (SDN) decouples control logic from the forwarding devices to simplify network management and enable complex network applications [1]. Such a separation allows the control plane software and data plane hardware to evolve quickly and independently. Recent interest in SDN has moved to the implementation of various SDN applications upon controllers written in different programming languages. The vision of SDN is to construct an SDN "App Store" [2], [3], [4] for network management services. Similar to the Android Market or the Apple Store, network administrators could download applications suited to their needs from the SDN "App Store" and deploy them into the network. For example, a single network could run simultaneously a firewall written in Java on OpenDaylight [5], a routing application written in Python on Ryu [6] or a monitoring application in C on NOX [7].

To realize this vision, a mechanism that compiles different processing logics of applications to cooperate correctly in the data plane is essential. In general, there are two types of approaches towards such a mechanism: *top-down* and *bottom-up*. The top-down approaches use either domain specific programming languages [8], [9], [10] or a specific programming framework [11], [12], to express each application as a program (module) or an expressive equivalent (*e.g.* graph in [11]). These programs are then translated into a set of low-level OpenFlow rules representing the union of the `intents` of the applications. The bottom-up approaches on the other hand utilize SDN hypervisors [13], [14], lying between the controllers and the underlying forwarding devices, to compose policies[1] into a prioritized list of (OpenFlow [15]) rules. Nonetheless, both types of approaches essentially face the same challenge: composing multiple policies, each representing the intent of an application (program, module), into a single rule list that represents the union of these intents.

In the context of composing multiple policies, two types of composition operators have been proposed in existing SDN programming frameworks: parallel (+) and serial (>>) [9], [16], [10], [17], [18]. Parallel composition gives the illusion that each member policy acts on its own separate copy of the traffic while sequential composition enables multiple policies to operate on traffic in sequence. For example, if the hypervisor applies a composition configuration as follows: `Firewall >> (Monitoring + Routing)`, packets will be processed first by Firewall, and then operated on by Monitoring and Routing concurrently.

A policy consists of match fields and the associated atomic actions, which enable programmers to design abundant expressive behaviour represented as a sophisticated action list. A practical composition mechanism should ensure that the composed rule of multiple policies is correct (in terms of application intents) and efficient (in terms of packet processing) for both match fields and action lists. Prior work on policy composition [13], [19], [14], [11] however mostly discusses how to merge the match fields of rules from different member polices and how to calculate the priority of the composed rules, leaving action composition much overlooked. Indeed, the action composition essentially boils down to the "union" of actions (often implemented as the concatenation of actions) in the previous work. This observation was corroborated by inspection of the released code of the CoVisor system [20] and many language implementations such as Frenetic [21].

We show that simple concatenation for composing action lists not only cannot preserve the semantics (or interests) expected by the original member policies but also can result in wasted compute cycles in the resource constrained forwarding path environment of switches. For example, consider one member policy rule's action list is {`push_vlan(1),`

ISBN 978-3-901882-83-8

[1]To simplify our discussion, we use the terms "policy" and "application" interchangeably.

`tcpdst ← 80, fwd(1)`} while the other is {`dstip ← 10.0.0.1, tcpdst ← 80, fwd(2)`}. If the corresponding two rules are composed to operate on packets in parallel, and the actions lists are simply concatenated, the result becomes {`push_vlan(1), tcpdst ← 80, fwd(1), dstip ← 10.0.0.1, tcpdst← 80, fwd(2)`}, which obviously violates the semantics of the second original rule, since the packet appearing on port 2 are different from the one that would have been generated by this second rule, had it been operating alone. This is because the second rule forwards the input packets with modified IP destination address `10.0.0.1` to port 2, while the composed action list forwards the input packets with both the appropriately modified IP destination address *and* an added vlan header to port 2. Obviously, the second action of `tcpdst ← 80` is redundant, which wastes the compute cycles of underlying switches. Overall, to the best of our knowledge, there is no mechanism to effectively compute action sequences for composing SDN policies.

Motivated by our observations, we in this paper address the challenge of correct and efficient action composition in the context of policy composition. our contributions are four-fold:

1) We show and prove, feasibility conditions on the composition of rule actions in SDN networks. By extension, this result also applies to the feasibility analysis of the composition of the policies themselves;
2) We derive a feasibility test, which can be applied to the "on-the-fly" composition of rules.
3) We propose a graph-based approach for fast computation of the actions of a composed rule. The approach has negligible effect on the performance of the composition operation itself, while resulting in the minimum number of actions to be performed in the data plane of switches;
4) We integrate our action composition algorithms in the CoVisor code base[2].

The rest of the paper is organized as follows: Section II describes the background and motivation for our approach. We present theoretical fundamentals and a model for action list composition in Section III. In Section IV, we detail efficient algorithms based on the model for the composition operators. Section V presents experimental results of these algorithms. We conclude with perspectives on our contributions in Section VI.

II. Background and Motivation

To set the scene, we first briefly present some features of SDN policy, and then review the parallel and sequential composition operations introduced in [14], [13], [19], [9]. Finally, we give examples to motivate our work.

A. SDN Policies

To fix ideas, one can think of OpenFlow [22] policies, although our work is general and not limited to OpenFlow. A policy is expressed as a set of prioritized *rules*. Each rule R is a 3-tuple $R = (p; m; a)$, where $R.p$ is the rule's priority, $R.m$ represents the match field patterns and $R.a$ is a sequential "program" (i.e. list) of the actions to be applied to packets matching the rule (see Figure 1).

Rule	Priority	Match	Actions
R_1	1	0000	*fwd(1)*
R_2	5	01**	*fwd(2)*
R_3	9	00**	*fwd(3)*
R_4	99	****	*fwd(4)*

Fig. 1. Example of policy as a rule table. Smaller priority values imply higher priorities.

The match fields in $R.m$ can, in all generality, consist of any number of adjacent packet bits (although they are usually limited to packet header fields) and ingress port. The set of match fields is the same for each rule in the policy, and their values can be any pattern including exact values, ranges (including prefixes), wildcards (matching any value), etc. If a packet potentially matches several rules, the rule with the highest priority is selected as the actual match, and the associated action list is applied to the packet. How a policy is implemented inside a switch (e.g. hardware table, pipeline of hardware tables, software hash, etc) is not relevant to the considerations of this paper.

We consider that actions are of three types: *modify* actions, whose effect is to modify packets or packet headers; *forwarding* actions, whose effect is to instantiate a packet on a port (i.e. forward the packet through the port); and *miscellaneous (misc)* actions, whose effect does not directly affect a packet (e.g. count actions, action list modification actions, etc.) Note that some of these misc actions have externally observable side-effects (such as actions modifying counters), while others do not (such as actions clearing the action list). To simplify, in this paper, we only consider counters associated with rules (one counter per rule) which count the number of packets for which the corresponding rule was a "hit" (and thus a `count` action simply increments such counter).

In this context, each rule of a policy is a function:

$$F(p) \rightarrow (p', port)^+ | d$$

where $(p', port)$ is a *forwarding pair* representing the packet p' appearing on port $port$, and d represents some statistics data side-effects. The $(\cdot)^+$-notation indicates that a rule can generate 0, 1 or more forwarding pairs for the given input packet p, depending on the packet's input port (which is part of the rule's matching pattern), and the packet itself. d is a positive integer (possibly 0) that represents the increment to be applied to the counter associated with the rule. Switches "implement" these functions by "executing" the actions associated with the rules[3].

From this, we can simply define the notion of *action list equivalence*: two action lists (i.e. two rule programs) are

[2]Our algorithms can be applied to other high-level programming frameworks very easily.

[3]More precisely, switches select the highest priority rule matching the packet and only execute the corresponding actions.

equivalent if and only if, for any packet p, $F_1(p) \equiv F_2(p)$. In other words, two action lists are equivalent, if they 1) produce the same forwarding pairs, and 2) count the same packets.

B. Composition Operators

The composition operators fall into two major categories: parallel composition and sequential composition. Here, we give a simplified overview for these composition operators and their compile algorithms presented in the prior art [13], [19].

Parallel Operator (+): The parallel operator compiles two policies into a single one which behaves as though packets were matched and processed by the two policies operating concurrently on their own copy of the traffic. For example, take a monitoring policy $Monitor$ that counts packets with source IP prefix 3.0.0.0/8 while dropping others. If a routing policy $Route$ forwards packets with destination IP 2.0.0.1 to port 1 and drops others (see Figure 2), then, with the parallel operator, we can generate a single policy $Monitor + Route$ shown in Figure 2.

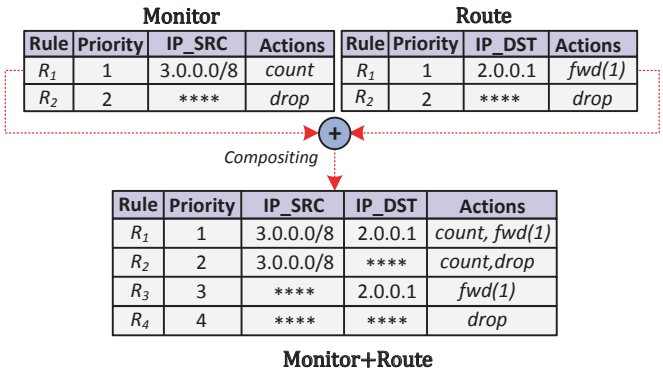


Fig. 2. Example of parallel composition, adapted from [16]

Next, we recall the existing compiler algorithms of the parallel operator using the example in Figure 2. To compile $Monitor + Route$, the compiler algorithms will calculate the cross product of rules from $Monitor$ and $Route$ as follows: any rule $m_i \in Monitor$ and $r_j \in Route$, m_i and r_j can generate a composed rule as long as $m_i.m \cap r_j.m \neq \emptyset$, using the intersection as its match fields and the concatenation of $m_i.a$ and $r_j.a$ as its action list. For example, consider m_1 and r_1 (the first rule in $Monitor$ and $Route$ respectively). As $m_1.m \cap r_1.m$ is {srcip=3.0.0.0/8, dstip=2.0.0.1}, they can generate a composed rule - the first rule in $Monitor + Route$.

Sequential Operator (>>): The sequential operator enables multiple policies to operate packets in series by combining those policies together. For example, suppose we have a load balancer policy LB that distributes traffic to two backend servers by rewriting their IP destination address while a routing policy $Route$ forwards packets based on their IP destination address (see Figure 3). Via the sequential operator, the composed policy will first rewrite the IP destination address and then forward these packets.

For the sequential composition of policies, the compiler algorithms compute the cross product of rules from the two

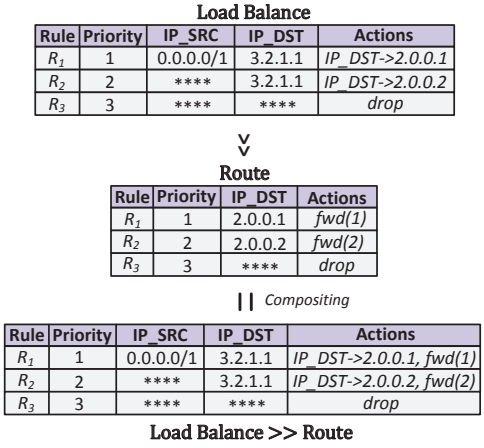


Fig. 3. Example of sequential composition.

policies ($LB >> Route$) as follows: apply the associated action list on the match fields of the rules from LB, and then check, for any rule $l_i \in LB$ and $r_j \in Route$, whether the intersection of $l_i.m$ and $r_j.m$ is empty or not. A composed rule is generated as long as $l_i.m \cap r_j.m \neq \emptyset$, through merging the match fields of $l_i.m$ and $r_j.m$ as the match fields, and concatenating $l_i.a$ and $r_j.a$ as the action list.

C. Motivating Examples

Let us first consider two policies, say P_1 and P_2, to be composed by parallel composition. From the very definition of parallel composition, the parallel composition of these policies should behave as though these policies operated in "parallel" on their own copy of the traffic. In other words, the packets generated by the parallel composition must be the union of the packets that would be generated by each policy operating on the traffic independently.

More formally, if $L_1(p)$, $L_2(p)$ and $L_{//}(p)$ denote the sets of forwarding pairs respectively generated by P_1, P_2 and the parallel composition of these policies, then

$$P_{//}(p) \equiv P_1(p) + P_2(p) \Rightarrow L_{//}(p) = L_1(p) \cup L_2(p)$$

It is trivial to prove that the parallel composition operator is commutative, that is that $P_1(p) + P_2(p) = P_2(p) + P_1(p)$, since $L_1(p) \cup L_2(p) = L_2(p) \cup L_1(p)$, confirming the intuition that the order in which the policies are composed should not affect the result of the parallel composition.

However, existing compositional systems all propose to construct the action list of a rule resulting from parallel composition as a simple concatenation of the action lists of each composed rule ($P_{//}(p) \equiv P_1(p) + P_2(p) \rightarrow a_{//}(p) = a_1(p) \circ a_2(p)$). Concatenation is obviously not commutative: if, for instance, $a_1(p)$ = {`dstip` $\leftarrow$ `8.0.0.2, fwd(2)`} and $a_2(p)$ = {`fwd(1)`}, then $a_1(p) \circ a_2(p)$ forwards the same packet (whose destination address has been changed to 8.0.0.2) on both port 1 and 2, while $a_2(p) \circ a_1(p)$ forwards the original input packet to port 1 and the packet with a modified

destination address to port 2. As parallel composition is a commutative operation, it therefore cannot be realized through simple action concatenation.

For sequential composition, which is not a commutative operation by definition, simple concatenation of action lists is also used. It is however, also easy to show that, while correct, concatenation of actions can lead to redundant actions. Indeed, consider, for instance, a_1 = {`vlan` ← `1`} and a_2 ={`vlan` ← `2`, `fwd(1)`}. $P_1 >> P_2$ yields $a_{>>}$ = {`vlan` ← `1`, `vlan` ← `2`, `fwd(1)`}. Conceptually, the first modification in the composed action list is redundant, leading to wastage in the resource constrained switch fast path[4].

We therefore see that simple concatenation for the composition of action lists cannot always preserve semantic equivalence and correctness, or achieve optimal operations in the data path. As a result, we conclude that action list composition, in the context of policy composition operators, needs to be revisited. We provide deeper analysis and solutions in the next few sections.

III. Action Composition Model

Essentially, actions are used in rules to transform input packets into output packets with specific properties, forward these output packets to output ports, as well as keep statistics on packets or rules. While other use of actions exists, such as circumventing a switch's lack of capabilities, it is the above mentioned observable results of actions that matter for compliance of the implemented policies.

The same is true for the composition operators: as long as the observable forwarding pairs and statistics comply with the intended compositional semantics, the result of the composition is correct.

A. Constructible Sequence and Graph-based Model

With the existence of `set/write` actions capable of setting any sequence of bits and/or fields to any specified value in the packet header, generating a packet with any specific header may seem trivial. However, this is not the case.

Indeed, the composition of policies is computed by the SDN hypervisor (a control plane component), using the policy rules, while the specific packet headers are only known by the switches (the data plane). In other words, the hypervisor can only rely on the rule matching patterns to represent packets, and the crux of the problem is that match patterns can contain "don't-care" bits (e.g. wild-cards, ranges, prefixes, etc.)

This is an issue, because once a part of a packet, corresponding to a match pattern containing "don't-care" bits, has been set to any specific value by a `set` action, there is generally no way to revert such change, as "don't-care" bits always match multiple values (see Figure 4).

The only way to revert a packet field, corresponding to a rule match field containing "don't-care" bits, is constructing switches that can copy and save the original field value from the input packet. However, current switch chipsets are not willing to support such actions for three reasons. First, recording packet values needs extra memory which is expensive in resource limited switch chips; second, enabling copy action causes race conditions because commodity switches usually process packets in parallel; third, each revert needs two memory copy operations (packet to memory and memory to packet), leading to a lower performance. Thus, packet fields fall into two categories:

[4]Any (unnecessary) operation in the data plane potentially leads to a decrease in forwarding rate.

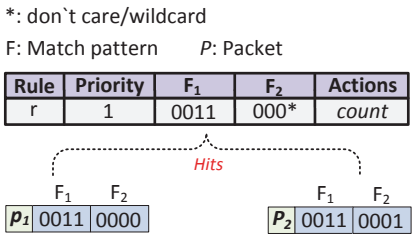


Fig. 4. Example of "don't-care" bits. F_2 in the rule contains one "don't-care" bit and thus matches two different values.

1) *Irreversible fields*: packet fields that 1) cannot be copied from the original (input) packet, *and* 2) correspond to match fields that contain "don't-care" bits in the policy rule.
2) *Reversible fields*: packet fields that either can be copied from the original (input) packet *or* that correspond to match fields specifying an exact (unique) value (no "don't-care" in the bit pattern of the field, the exact original value being thus available to the composing hypervisor).

Consequently, in the presence of changes to irreversible fields (see Figure 5), not every sequence of packets can be generated by a switch, from a given input packet. In fact, a set of output packets is said to be *constructible* from a given input packet if there exists a sequence (i.e. permutation) of those packets, starting with the input packet, such that no change to an irreversible field must be reverted to progress in the sequence. We now prove a fundamental theorem on constructible sequences of packets.

We call IC_i the set of irreversible fields that must change to generate output packet p_i from input packet p_{in}. Note that since changes to reversible fields can always be reversed (i.e. undone), reversible fields can safely be ignored in feasibility considerations.

Theorem 1. (Constructible Sequence Theorem)*: Given an input packet p_{in}, n output packets p_i and their set of irreversible field changes IC_i ($1 \leq i \leq n$), the sequence $< p_{in}, p_1, p_2, \ldots, p_n >$ is constructible iff $IC_1 \subseteq IC_2 \subseteq \ldots \subseteq IC_n$.*

Proof. We prove the forward direction by contradiction. Assume the sequence is constructible. Also assume that there exists an irreversible field if_k that changes to generate p_i, but does not change to generate p_j further in the sequence, that is $\exists if_k : if_k \in IC_i, if_k \notin IC_j$, with $i < j$.

Since $if_k \notin IC_j$, the value of if_k in p_j is the original value

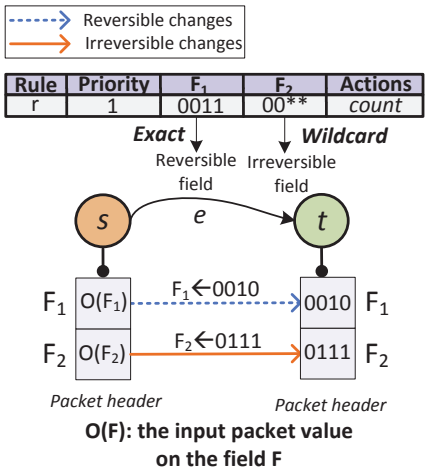


Fig. 5. Example of reversible and irreversible field changes.

of that field in p_{in}. Also, since $i < j$, p_j is constructed after p_i in the sequence, and this can only be possible if the change to irreversible field if_k, that was necessary to generate p_i has been reversed to generate p_j. This is a contradiction, since if_k is an irreversible field. We therefore have that in a constructible sequence, $(i < j, \forall if_k : if_k \in IC_i) \Rightarrow if_k \in IC_j$, which implies that $IC_i \subseteq IC_j, i < j$.

We prove the reverse direction by induction. **Base case**: by definition of IC_k, any packet p_k can be constructed from p_{in} by changing the irreversible fields in IC_k (along with possibly changes to some reversible fields). In particular, p_1 can always be generated from p_{in} by changing the (irreversible) fields in IC_1 (such operation is denote $p_{in} \rightarrow_{IC_i} p_1$). **Inductive case**: assume the prefix subsequence up to packet p_k ($< p_{in}, p_1, p_2, \ldots, p_k >$) is constructible. We show that p_{k+1} is constructible (can be generated) from p_k, given that $IC_k \subseteq IC_{k+1}$. Indeed, $IC_{k+1} = (IC_k \cap IC_{k+1}) \cup (IC_{k+1} \setminus (IC_k \cap IC_{k+1}))$. But since $IC_k \subseteq IC_{k+1}$, we have that $IC_k \cap IC_{k+1} = IC_k$, so that $p_{in} \rightarrow_{IC_k \cap IC_{k+1}} p_k$. This means that p_k can be generated as a step in the construction of p_{k+1}. From this step, the remaining changes in IC_{k+1}, that is all the changes in $IC_{k+1} \setminus (IC_k \cap IC_{k+1})$ can be applied to yield p_{k+1} ($p_k \rightarrow_{IC_{k+1} \setminus (IC_k \cap IC_{k+1})} p_{k+1}$). We therefore have $p_{in} \rightarrow_{IC_k \cap IC_{k+1}} p_k \rightarrow_{IC_{k+1} \setminus (IC_k \cap IC_{k+1})} p_{k+1} = p_{in} \rightarrow_{IC_{k+1}} p_{k+1}$. □

When an SDN hypervisor is composing policies, it does not generally know the exact values of the fields of the packets that will hit the resulting rules. Still, it can "simulate" the effects of applying the actions associated with the rules (according to the composition operators used), so that it can "compute" the packets, in terms of which input packet fields get modified or not, and on which ports these packets get forwarded. The discussion and results describe above therefore suggest that the problem for the hypervisor is thus to find the right sequence for generating the output packets, given that as soon as an output packet has been constructed, it can simply be forwarded to the correct ports by issuing appropriate `forward` actions.

A convenient way to model the process of constructing packets is thus as a graph, where vertices represent each unique packet in the process (that is the input packet and each output packet to be generated), and where there is an oriented edge between two vertices if a series of actions can transform the source packet into the destination packet. The important thing to remember, is that reversible packet fields can always be changed to any value in any order, while irreversible fields can only be set to specific (known) values, but cannot be reverted to their unknown original (input) value. The resulting graph is thus not a "full mesh" (since some packets cannot be constructed from others). Each edge in the graph can then be labelled with the set of packet modification actions needed to actuate the transformation from the source packet to the destination packet (see Figure 6).

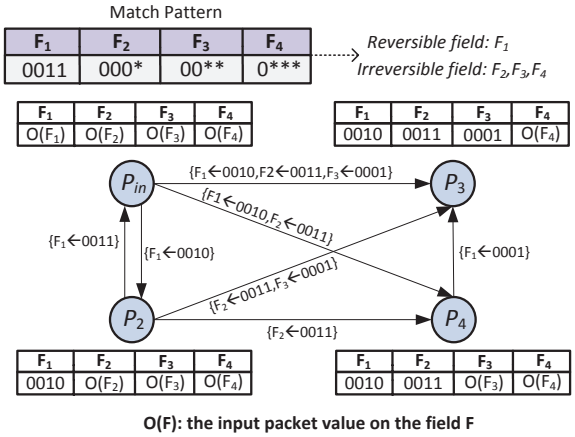


Fig. 6. Example of graph-based action composition. $IC_{p_{in}} = IC_{p_2} = \emptyset$, $IC_{p_3} = \{F_2, F_3\}$, $IC_{p_4} = \{F_2\}$. An oriented edge from p_i to p_j exists *iff* $IC_{p_i} \subseteq IC_{p_j}$. A path starting from the input packet $p_i n$ and visiting each vertex is $p_{in} \rightarrow p_2 \rightarrow p_4 \rightarrow p_3$. The corresponding action list is $F_1 \leftarrow 0010, F_2 \leftarrow 0011, F_1 \leftarrow 0001$.

With such a graph, generating the required packets, and computing the associated action list, reduces to finding a Hamiltonian path [23], starting at the input packet, if such path exists. Indeed, a Hamiltonian path through a graph visits each vertex exactly once, corresponding to every output packets being generated.

However, the Hamiltonian path problem is known to be NP-complete [23], [24]. In section IV, we discuss algorithms to find such a path, while aiming to minimize the number of actions required to actuate the construction of output packets.

B. Misc Action Considerations

A misc action associated with a rule counts the number of packets for which the corresponding rule was a "hit". Let us consider two policies P_1 and P_2 that need to be composed. Suppose $r_1 \in P_1$ has a misc action to count $C(r_1)$ the number of packets that hit r_1. After composition of P_1 and P_2, we still need to get $C(r_1)$ from the composed policy.

Let $S(r_1, P_2)$ denote the set of composed rules that contain the semantic of r_1, i.e., $S(r_1, P_2) = \{r_1.m \cap t_i.m | r_1.m \cap t_i.m \neq \emptyset, t_i \in P_2\}$. For each composed rule $s_i \in S(r_1, P_2)$,

we associate one misc action to count the number of packets that hit s_i. We then have the following Theorem for the restoration of $C(r_1)$ from the composed policy.

Theorem 2. (QUERY STATISTICS)*: Given two policy P_1 and P_2, policy M is composed of P_1 and P_2. The counter $C(r_1)$ associated with the rule $r_1 \in P_1$ can be computed as:*

$$C(r_1) = \sum C(s_i)$$

where $s_i \in S(r_1, P_2) \subseteq M$ and $S(r_1, P_2) = \{r_1.m \cap t_i.m | r_1.m \cap t_i.m \neq \emptyset, t_i \in P_2\}$.

Proof. On the one hand, for any packet that hits r_1, it can hit at least one rule of $S(r_1, P_2)$: the rule composed by r_1 and the `default` rule of P_2. On the other hand, due to the priorities of composed rules, any packet can hit no more than one rule of $S(r_1, P_2)$. As such, any packet that hits r_1 can hit exactly one rule of $S(r_1, P_2)$. In other terms, the number of packets that hit r_1 is equal to the number of packets that hit the rules of $S(r_1, P_2) \subseteq M$. □

IV. ACTION COMPOSITION ALGORITHMS

We showed in section III that the problem of finding a constructible sequence of packets to implement the composition of policies reduces to finding a Hamiltonian path in a graph. While this problem is generally NP-complete, Theorem 1 states a fundamental property of such sequences that can be exploited to efficiently find such sequence.

Indeed, Theorem 1 shows that, in a constructible sequence, changes to irreversible fields must be applied "incrementally", due to the "nesting" of the set of irreversible fields that have changed (compared with the input packet), from one packet in the sequence to the next; in other words, packets further in the sequence, can only be constructed by either "adding" more changed irreversible fields or changing again (to specific know values) some of these fields, compared with earlier packets in the sequence.

This observation leads to a very simple, straightforward and efficient algorithm (see Algorithm 1) to not only test for the existence of a constructible sequence, but also obtain one such sequence of packets (if it exists).

All we need to do is to represent all the irreversible fields in a rule as a bitmap. Remember that what makes a header field irreversible is the presence of "don't care" bits in the pattern of the rule representing that field and the lack of switch capability to save the original value of this header field in the input packet, both properties being known to the compositional hypervisor. Then for each desired output packets (again, these are know to the hypervisor), set to 1 the bits corresponding to changed irreversible fields (lines 1 to 4). Then sort the "output packets" by the *number* of bits set in the bitmap (line 5), because irreversible field changes must be applied incrementally. Then sweep across the ordered packets, checking if `bitmap(k) & bitmap(k+1) == bitmap(k)`, which is equivalent to checking that the set of irreversible field changed in one packet is completely contained in the set of irreversible fields changed in the next packet (as required by Theorem 1). If this test succeeds for each consecutive pair of packets, then not only a constructible sequence of packets exists, but the ordered packets is one such sequence (lines 6 to 11).

Algorithm 1: SIMPLESEARCH(p_{in}, $\{p_{out}\}$, $\{IF\}$)

Input: p_{in}: input packet
Input: $\{p_{out}\}$: set of (unique) output packets
Input: $\{IF\}$: set of irreversible fields in the rule
Output: path: Hamilton path "vector" ("empty" if no such path exists)

```
1  path ← newEmptyVector();
2  for p ∈ {p_out} do
3      bm ←BitMap(p_in, p, {IF});
4      path.append((p, bm));
5  sort(path, ByNumberOfBitSet);
6  thisP ← path.first();
7  while (nextP ← path.next()) ≠ NULL do
8      if thisP.bm & nextP.bm ≠ thisP.bm then
9          return newEmptyVector();
10     thisP = nextP;
11 return path;
```

The complexity of this algorithm, given n output packets to generate, is $O(n)$ for generating the bitmaps; $O(n \lg n)$ for sorting; and $O(n)$ for testing the inclusion relation. Therefore the overall complexity is $O(n \lg n)$.

From the returned sequence of packets (if it exists), the compositional hypervisor can compute the action list for the corresponding (composed) rule by simply concatenating the modify actions required to generate each packet in the path, from the preceding packet, and issuing the required forwarding actions whenever the desired packets have been generated.

While Algorithm 1 finds a constructible sequence of packets if such sequence exists, this sequence may not be optimal in terms of the number of actions required to generate the sequence. This is because packets that have the *same* set of modified irreversible fields (and thus only differ from each other by different sets of modified *reversible fields*) can appear in *any relative order* in the sequence.

See, for instance, packets P_3 and P_4 in Figure 7. The total cost of the path is 6 (① + ② + ③ + ④). But there is another constructible sequence of packets, obtained by exchanging packet P_3 and P_4 in the packet sequence, with reduced cost 5. This is because the cost from P_5 to P_4 is 1 ($F_3 \leftarrow 0001$) while P_4 to P_3 requires 2 modification operations ($F_1 \leftarrow 0011, F_3 \leftarrow 0011$). The reason why we can change the order of P_3 and P_4 to get a lower cost path is that they contain identical sets of modified irreversible fields, i.e. $IC_3 = IC_4$.

While Algorithm 1 actually worked on an *implicit* representation of the graph model for packets described in Section III, finding optimal sequences will require an explicit representa-

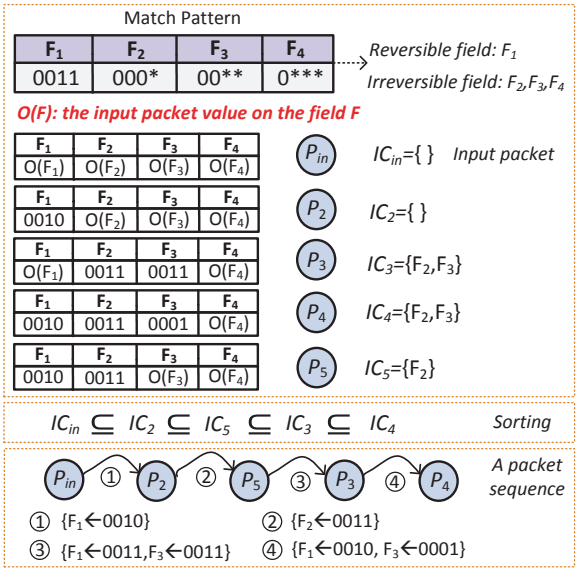


Fig. 7. Example of a Hamilton path. The packets sequence sorted by the number of irreversible changes provides one Hamilton path.

tion of this graph.

The graph for packet generation as described in Section III would have a directed edge between two packets if no modified irreversible field has to be reverted to its original value (in the input packet) to go from the "source" vertex to the "destination" vertex. However, this is far too many edges: indeed, because of the transitivity of the "subset" relationship (i.e. "contain" operations) required of the modified irreversible field subsets of the packets in a constructible sequence (Theorem 1), a sub-sequence $P_1 \rightsquigarrow P_2 \rightsquigarrow P_3$ [5], would also imply one directed edge $P_1 \rightsquigarrow P_3$. However, the $P_1 \rightsquigarrow P_3$ edge is completely useless, because it will never be part of a Hamiltonian path in the graph: a sequence can never go back to P_2 from P_3, as this would mean reverting (at least) one irreversible change.

The output of the simple Algorithm 1 can here help avoid generating these useless edges in the graph, and thus reduce the space to be searched for optimality. Indeed, this simple algorithm outputs packets ordered by their number of modified irreversible fields. Any sub-sequence of adjacent packets with the same number of such modifications thus forms a group of packets whose order can be changed while still conserving a constructible sequence. This is because packets in each group form a "local full-mesh", and they only differ from each other by modifications to reversible fields. The simple algorithm therefore also gives the sequence of groups, and there thus only needs to be an edge from each packet in a group, to each packet in the following group in the sequence (see Figure 8).

While finding an optimal path (in terms of the number of actions needed) in such graph is still an NP-complete Hamiltonian path search, we argue that in practical scenarios, the number of distinct output packets to be generated will be

[5]We suppose $IC_1 \subseteq IC_2 \subseteq IC_3$. $\forall i \in [1,3]$, IC_i corresponds to P_i.

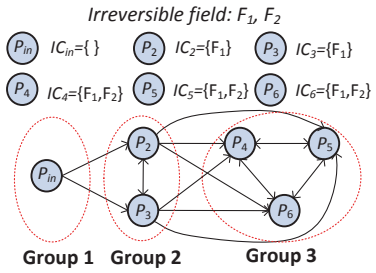


Fig. 8. Example of packet grouping.

kept relatively low (so the number of vertices in the graph will be small). Furthermore, this graph only contains edges that potentially belong to a constructible sequence (so the number of edges has been reduced to a minimum). Because of this "reduced" search space, we believe that a brute-force algorithm, enumerating all the (Hamiltonian) paths in the graph is a plausible solution to the optimal Hamiltonian path finding problem at hand.

Nevertheless, should the search space become too big, the compositional hypervisor can always decide to use a heuristic algorithm (such as one based on a greedy approach) instead, to trade running time for potential deviation from optimality[6].

V. Implementation and Evaluation

We have implemented our model and the related algorithms in CoVisor [13]. Using this implementation we evaluate its performance.

More specifically, we replaced the core logic of action lists composition for both the parallel and sequential operators. Note that we implemented three Hamilton path searching algorithms: the simple algorithm (Algorithm 1), the brute-force (a.k.a. enumeration) algorithm and a greedy algorithm, picking the less weighted edge whenever a choice is available when searching for the Hamilton path: suppose the last added vertex in the path is v, then the next vertex in the path is $u=\text{argmin}_{u \in U(v)} W(v,u)$, where $W(v,u)$ is the weight (i.e. the number of modification actions) of the edge from v to u and $U(v)$ is the set of destination vertices of edges from v. This greedy algorithm works, because we ensure that the graph only contains edges that are potentially part of a Hamilton path (see Section IV).

A. Experimental Setup

We deployed our implementation on an octo-core Intel®Xeon®E5506 CPU, clocked at 2.13GHz. The machine is equipped with 16GB RAM and runs 64-bit Ubuntu Linux 10.04.3. We used two rulesets for our experiments:

1) D1 (real-life policies): L3 Router [26] and L3 Firewall [27].
2) D2 (synthetic policies): rules are generated associated with multiple types of actions (e.g. modification, for-

[6]As an extreme case, the hypervisor could even choose to use the output of the simple algorithm.

warding and misc actions) to reflect more dynamic, complex scenarios.

Each rule of D1 contains one forwarding action. Each rule of D2 on the other hand contains multiple modification/forwarding actions. To generate modification actions in D2, we randomly select one packet header field as the field that is modified by the action, whose value after modification is also randomly assigned. The number of distinct output packets for each rule is controlled through forwarding actions. In the experiment, an action list can generate no more than 10 distinct (different) output packets for one input packet – we believe this value to represent an unrealistic value, chosen to illustrate absolute worst case scenarios. The match pattern for IP address is prefix-based, while for other match patterns (like port, MAC address, vlan), we use exact match.

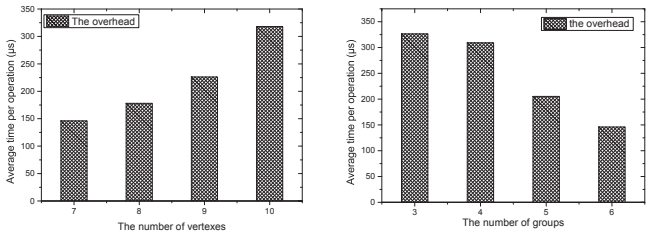
We are interested in four aspects of performance: 1) the computation time; 2) factors that affect the computation time; 3) contribution of the various components to the computation time; 4) comparison between the three path search algorithms in terms of computation time and optimality.

B. Experimental Results

TABLE I
COMPUTATION TIME OVER TWO POLICIES (IN μS).

	average	minimum	maximum
D1	85	72	95
D2	249	125	380

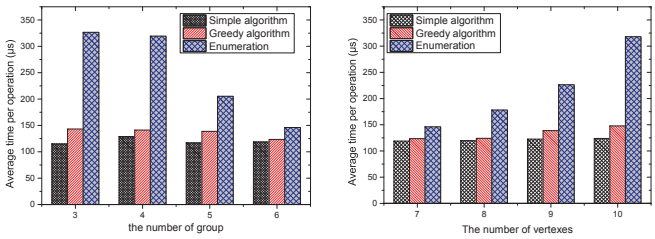
The average, minimum and maximum computation time of the enumeration algorithm are reported in Table I. The action lists of any rule in D1 can be computed within 95 μs. Computation of action lists for rules in D2 takes a longer time and the average time is around 250 μs. This is because rules in D2 have more complex actions and can generate more distinct output packets. Nevertheless, the computation time is relatively small, showing that our approach is practical.



(a) (b)

Fig. 9. Variation of computation time with two factors: (a) the number of vertices in a group, (b) the number of groups.

The computation time depends on both the number of vertices in groups and also the number of groups in our graph-based model. We first select the actions from $D2$ that have 6 groups. Figure 9(a) plots the computation time when varying the number of vertices. As expected, a larger number of vertices leads to a higher computation time. But, even with 10 vertices, the computation time is still within 320 μs. We then select the actions from $D2$ that have 7 vertices. The computation time with different number of groups is reported in Figure 9(b). A larger number of groups leads to a smaller number of vertices per group. Given that the permutation within each group is one of the major contributors on computation time, a smaller number of vertices in each group in turn results in lower computation time.



(a) (b)

Fig. 10. Comparison of three algorithms in terms of computation time: (a) varying the number of groups. (b) varying the number of vertices.

We then compare the three algorithms for the Hamilton path search in terms of computation time. Figure 10 shows the computation time of the three algorithms, where we vary the number of groups while fixing the number of vertexes (Figure 10(a)), and vary the number of vertexes while fixing the number of groups (Figure 10(b)). Since the enumeration and greedy algorithms use extra optimization (necessitating the output of the simple algorithm to generate a "reduced" graph, see Section IV), they requires to use more computation time. Compared with enumeration algorithm, the greedy one can save up to 50% of the computation time, and is less relevant to the number of vertices and the number of groups.

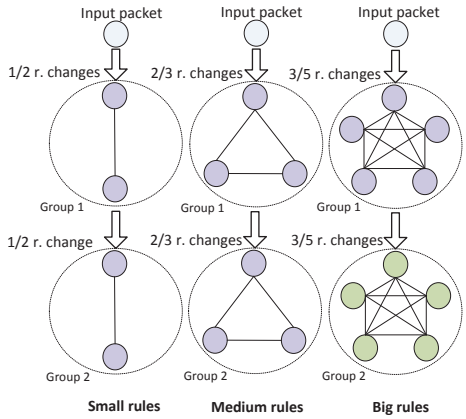


Fig. 11. The three scenarios. r. represents reversible fields.

Finally, we evaluate the amount of actions in the Hamilton path generated by the three algorithms. To this end, we construct three scenarios based on D2 (see Figure 11). In all three scenarios, one input packet would generate two groups of outputs. In the *small rules* scenario, each group contains two packets which change 1 (out of 2) reversible field; in the *medium rules* scenario, each group has three packets which change 2 (out of 3) reversible fields; in the *big rules* scenario,

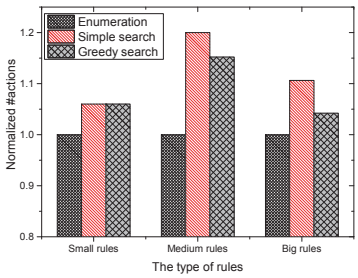


Fig. 12. Number of actions generated by the three path search algorithms, normalized by the number of actions in the path generated by the enumeration algorithm.

each group has five packets which have 3 (out of 5) reversible fields. We apply the three algorithms on each scenario and plot the number of actions in the path generated by each algorithm in Figure 12. We can see that, compared with the enumeration algorithm (which is optimal), the simple search algorithm generates up to 20% more actions, and the greedy algorithm incurs up to 15% more actions.

In summary, given that the composition of policies is performed in servers (like controllers) other than switches themselves, we believe the enumeration algorithms is more applicable in practice in order to obtain optimal results.

VI. CONCLUSION

Policy composition has been emerging as a powerful and important tool for facilitating the creation and deployment of complex network applications. As the developer or network administrator requesting such composition may not master, or even want to know, the details of each policy component being composed, it is of paramount importance that compositional operators be supported in as much a transparent and efficient manner as possible. Previous work introduced important headways in this direction by proposing efficient techniques to compute the matching patterns for composed rules. Our work complements this by tackling the problem of correct and efficient action list computation, another important component of policy rules.

In particular, we formalize an action composition model, and prove a feasibility condition on the composition of rule actions. We abstract the action composition as a Hamilton path search problem in a directed weighted graph, while exploiting fundamental properties specific to the resulting graph to compute solutions, to this otherwise NP-complete problem, efficiently. We show that our approach is not only correct, but also efficient.

ACKNOWLEDGMENTS

We thank the IFIP Networking reviewers for their insightful feedback. This work is supported in part by National High Technology Research and Development Program of China (Grant No. 2015AA016101 and 2015AA010201), National Natural Science Foundation of China (Grant No. 61502458 and 61502462) and Beijing Municipal Natural Science Foundation (Grant No. 4162057).

REFERENCES

[1] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, 2009.
[2] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[3] "The hp sdn app store." http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/devcenter/#sdnAppstore.
[4] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti, *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.
[5] "Opendaylight." http://www.opendaylight.org/.
[6] "Ryu openflow controller." http://osrg.github.io/ryu/.
[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.
[8] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in *ACM CoNEXT*, 2014.
[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, pp. 279–291, ACM, 2011.
[10] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 217–230, 2012.
[11] Y. T. Chaithan Prakash, Jeongkeun Lee and J.-M. Kang., "Pga: Using graphs to express and automatically reconcile network policies," in *Proceedings of the 2015 ACM conference on SIGCOMM*, ACM, 2015.
[12] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," 2015.
[13] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proc. USENIX NSDI*, 2015.
[14] A. Dixit, K. Kogan, and P. Eugster, "Composing heterogeneous sdn controllers with flowbricks," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pp. 287–292, IEEE, 2014.
[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.
[16] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, *et al.*, "Composing software defined networks.," in *NSDI*, pp. 1–13, 2013.
[17] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, *et al.*, "Languages for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 128–134, 2013.
[18] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 113–126, 2014.
[19] X. Jin, J. Rexford, and D. Walker, "Incremental update for a compositional sdn hypervisor," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 187–192, ACM, 2014.
[20] "The opensource code of covisor." https://github.com/CoVisor/CoVisor.
[21] "The code of frenetic language.." http://frenetic-lang.org/pyretic/.
[22] "Openflow switch specification." https://www.opennetworking.org/.
[23] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.
[24] A. A. Bertossi, "The edge hamiltonian path problem is np-complete," *Information Processing Letters*, vol. 13, no. 4, pp. 157–159, 1981.
[25] "Openflow switch specification." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.
[26] "Routereview." http://www.routeviews.org/.
[27] "The rules set of evaluation packet classification." http://www.arl.wustl.edu/~hs1/PClassEval.html.