# Verifying multiple virtual networks in Software Defined Networks


Igor Burdonov
*Software Engineering department*
*Ivannikov Institute for System Programming of RAS*
Moscow, Russia
igor@ispras.ru

Nina Yevtushenko
*Software engineering deparment*
*Ivannikov Institute for System Programming*
Moscow, Russia
evtushenko@ispras.ru

Alexandr Kossachev
*Software Engineering department*
*Ivannikov Institute for System Programming of RAS*
Moscow, Russia
kos@ispras.ru



***Abstract*— Software Defined Network (SDN) technology is one of the modern network virtualization technologies. When implementing a virtual network on the SDN data plane, undesired effects may occur: the appearance of undesired paths where packages can be sent, "looping" when the packages are infinitely cycled and infinitely cloned, duplicate paths when the host receives the same package several times. We show that these effects can occur for the joint implementation of several virtual networks even if the implementation of each separate virtual network does not cause these effects. A method for verifying the implementation of several virtual networks is proposed. A sufficient condition is established for the graph of physical connections when any set of virtual networks can be implemented without the occurrence of the above undesirable effects.**



***Keywords*— *Software Defined Networks (SDN), Network Virtualization, Graph paths, Edge Simple Paths, Arc Closure, Verification***


## I. Introduction

Software defined networking [1][2][3] with separated data and control planes is one of the main technologies for implementing virtual networks. On the data plane, hosts exchange packets through intermediate switches. A switch after receiving a packet forwards it without any changes to one or more of its neighbors (hosts or switches); the choice of neighbor nodes depends on the neighbor (a host or a switch) from which the packet has been received and on the parameter values of the packet header. If the switch forwards a packet to several neighbors, then this means that the packet is cloned and then its clones are moved over the network independently of each other. The switch rules are set by the SDN controller(-s) [4]. The set of parameter values of the packet header affecting the packet forwarding is called the packet identifier. Packets with different identifiers are moved over the network independently of each other using different paths from a host-sender to a host-recipient.

The virtual network is described by the set of (ordered) pairs of hosts (a sender, a recipient) with corresponding paths for packets with the given identifiers. This, in turn, defines the switch rules that must be installed when the switches are configured.

When implementing a virtual network for a given set of paths, the problem of the implementation of undesired paths can occur [5][6][7]. First, it is a problem of duplicating paths when a host receives the same packet several times. Second, the cycling paths where packets can infinitely move can also occur. This problem was studied in a number of works [5][6][7] in which a number of solutions have been proposed.

Here we note that the problem of undesired paths can occur when implementing several virtual networks even in the case when the implementation of each separate virtual network does not induce undesired paths. The reason is the interference of the implementations of different virtual networks for packets with the same packet identifier, and this paper is devoted to the study of such influence. The main attention is paid to the problem of the cycle appearance due to the composition of path segments that appear in the implementation of different virtual networks for the same packet identifiers.

The structure of the paper is as follows. Section 2 contains the preliminaries and problem setting. Sections 3, 4 and 5 are devoted to the verification of the implementations of several virtual networks, especially for the absence of cycles. In Section 3, the algorithm is presented [6] for checking the existence of undesired paths and, in particular, the absence of cycling paths when implementing one virtual network. In Section 4.A, an algorithm is proposed for checking the absence of cycling paths when implementing several virtual networks. In Section 4.B, the algorithm of Section 4.A is modified to be more efficient but without specifying packet identifiers for cycling paths. A sufficient condition for implementing any set of virtual networks without undesired paths is proposed in Section 5. This condition is an extension of a similar condition for implementing a separate virtual network proposed in [5][7]. In conclusions, the results are summed up and the directions for future research are discussed.

## II. Preliminaries and the problem setting

The data plane of the software defined network is modeled by a finite undirected connected graph $G = (V, E)$ without multiple edges and loops, where $V$ is a set of vertices that are hosts and switches, and $E \subseteq V \times V$ is a set of edges displaying physical communication channels between vertices. We assume that each host is connected with exactly one switch. Without loss of generality, we can assume that all the vertices of degree one are hosts.

Since the graph $G$ is undirected and there are no multiple edges, the edge can be defined as a pair of vertices $a$ and $b$, which are connected by this edge: $ab$ or $ba$. A path is a sequence of neighbor vertices through which it passes. The path is called *complete* if the head and tail vertices are hosts, while all intermediate vertices are switches. The path where all the vertices (arcs) are pair-wise different, is called the *vertex simple* (or *edge simple*).

We further denote the vertices of the graph by lowercase Latin letters $a$, $b$, … $y$, $z$, the paths by bold letters $\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{r}$,…, the sets by capital letters $A$, $B$, … $Y$, $Z$ and the families of sets by capital bold letters $\boldsymbol{A}$, $\boldsymbol{B}$, … $\boldsymbol{Y}$, $\boldsymbol{Z}$.


978-1-7281-9899-6/20/$31.00 ©2020 IEEE

The switch $s$ rule is specified as $\iota{:}asb$ where $\iota$ is the packet identifier, $a$ and $b$ are vertices (switches or hosts) connected with the switch $s$ by the edges. This rule means that the switch $s$ after receiving a packet with the identifier $\iota$ from its neighbor $a$ forwards it to the neighbor $b$ without changes. The cloning of the packet occurs when there are several rules in the switch $s$ differing only by the neighbor $b$.

**Proposition 1.** The complete path $a_1...a_n$ is implemented for packets with an identifier $\iota$ if and only if each switch $a_i$, $i = 2..n-1$, has a rule $\iota{:}a_{i-1}a_ia_{i+1}$.

**Corollary 1.** If on the data plane, there are two complete paths $\boldsymbol{p}ab\boldsymbol{q}$ and $\boldsymbol{p}'ab\boldsymbol{q}'$ for the identifier $\iota$ which have a common arc $ab$, then the data plane has complete paths $\boldsymbol{p}ab\boldsymbol{q}'$ and $\boldsymbol{p}'ab\boldsymbol{q}$ for this identifier.

Thus, when implementing a set $P$ of complete paths on the data plane, the corresponding set $P{\downarrow}$ of the switch rules is installed, which in turn induces the set of implemented pats that is a superset of the set $P$; this superset denoted by $P{\downarrow\uparrow}$ is called the *arc closure* of the set $P$. In [6], the following statements are established.

**Proposition 2.** The set $P{\downarrow\uparrow}$ is induced by the following inference rules:

$\boldsymbol{p} \in P$ implies $\boldsymbol{p} \in P{\downarrow\uparrow}$,

$\boldsymbol{p}ab\boldsymbol{q} \in P{\downarrow\uparrow}$ & $\boldsymbol{p}\grave{}ab\boldsymbol{q}\grave{} \in P{\downarrow\uparrow}$ implies $\boldsymbol{p}ab\boldsymbol{q}\grave{} \in P{\downarrow\uparrow}$.

Therefore, the virtual network is set by a pair of sets $(Z, P)$ where $Z$ is a set of packet identifiers while $P$ is a set of complete paths. The packet with the identifier of the set $Z$ passes through the path(s) of the set $P{\downarrow\uparrow}$.

**Proposition 3.** There are no undesired paths when implementing a set $P$ if and only if $P = P{\downarrow\uparrow}$, i.e. the set $P$ is arc closed.

The generation of undesired paths not necessary is a problem if all the paths are edge simple, i.e. there are no cycles. For example, the path $\boldsymbol{p}ab\boldsymbol{q}ab\boldsymbol{r}$ is not edge simple and induces an infinite number of paths $\boldsymbol{p}ab(\boldsymbol{q}ab)^n\boldsymbol{r}$, $n = 1, \ldots$ .

**Proposition 4.** The set $P{\downarrow\uparrow}$ is finite if and only if all paths of $P{\downarrow\uparrow}$ are edge simple.

In the following section, we present the algorithm [6] for checking the presence of undesired and cycling paths for a given pair $(Z, P)$.

## III. VERIFICATION OF ONE VIRTUAL NETWORK

Let $G = (V, E)$ be the graph of physical connections while $(Z, P)$ be the specification of the virtual network to be implemented. It is necessary to check whether there are undesired paths generated (1) and if so whether there are cyclic paths (2). Due to the above, undesired paths occur if $P \neq P{\downarrow\uparrow}$. Since $P \subseteq P{\downarrow\uparrow}$ and $P$ is finite, the inequalities $P \neq P{\downarrow\uparrow}$ and $|P| \neq |P{\downarrow\uparrow}|$ are equivalent. Apparently, there are cyclic paths among undesired paths if and only if $P{\downarrow\uparrow}$ is infinite.

In [6], an algorithm for verification of the presence of undesired paths based on the directed graph $L(P)$ is proposed. Graph $L(P)$ is a subgraph of the line graph of $G$ generated by the set of paths $P$. The set of vertices of the graph $L(P)$ is the set of all arcs of paths of $P$, along with two additional vertices *source* and *sink*. An arc $(ab, b'c)$ of the graph $L(P)$ corresponds to a path $abc$ of length two in the graph $G$, i.e. $b = b'$, and is carried out if and only if in $P$ there is a path that has the fragment $abc$. The arcs $(source, xa)$ are leading from *source* to all the vertices $xa$, where $x$ the head host of a path of $P$. The arcs $(ax, sink)$ lead from all vertices $ax$ where $x$ is the tail host of a path from $P$, to *sink*.

**Proposition 5.** An undesired path can only be generated by paths of length more than two.

If a complete path in $P{\downarrow\uparrow}$ is an undesired path, i.e., this path is absent in $P$, then this path has the form $\boldsymbol{p}ab\boldsymbol{q}'$ or $\boldsymbol{p}'ab\boldsymbol{q}$ and is generated by two complete paths $\boldsymbol{p}ab\boldsymbol{q}$ and $\boldsymbol{p}'ab\boldsymbol{q}'$, where $\boldsymbol{p} \neq \boldsymbol{p}'$ and $\boldsymbol{q} \neq \boldsymbol{q}'$. Therefore, one of paths $\boldsymbol{p}$ or $\boldsymbol{p}'$, as well as of $\boldsymbol{q}$ or $\boldsymbol{q}'$ has nonzero length. Thus, since the paths $\boldsymbol{p}ab\boldsymbol{q}$ and $\boldsymbol{p}'ab\boldsymbol{q}'$ are complete, $a$ and $b$ are switches, and then all four fragments $\boldsymbol{p}$, $\boldsymbol{p}'$, $\boldsymbol{q}$, $\boldsymbol{q}'$ have nonzero length, i.e., length of $\boldsymbol{p}ab\boldsymbol{q}$ as well as of $\boldsymbol{p}'ab\boldsymbol{q}'$ is bigger than two. Therefore, we can restrict ourselves with the paths of $P$ having length bigger than two. In [6] the following algorithm for deriving graph $L(P)$ is proposed.

**Algorithm 1:** Deriving graph $L(P) = (V_L, E_L)$
**Input:** A set $P$ of complete paths
**Output:** A graph $L(P)$
Derive a subset $Q = \{q_1, ..., q_k\}$ of $P$ that contains all the paths of length greater than two; we denote as $k_j$ the length of a path $q_j$, $j \in \{1, ..., k\}$;
$V_L = \{source, sink\}$;
$E_L = \varnothing$;
$j = 1$;
***while*** $j < k$ ***do***
  $V_L = V_L \cup \{(q_j(1), q_j(2))\}$;
  $E_L = E_L \cup \{(source, (q_j(1), q_j(2))\}$;
  $m = 2$;
  ***while*** $m < k_j+1$ ***do***
    $V_L = V_L \cup \{(q_j(m), q_j(m+1))\}$;
    $E_L = E_L \cup \{((q_j(m-1), q_j(m)), (q_j(m), q_j(m+1)))\}$;
    $m$++;
  $E_L = E_L \cup \{(q_j(k_j), q_j(k_j+1)), sink)\}$;
  $j$++;
***return*** $L(P)$;

The complexity of Algorithm 1 depends on the number of pair-wise comparison of arcs of paths of $P$ that can be estimated as $O(L^2)$ where $L$ is the sum of lengths of paths of $P$. Thus, the following statement holds.

**Proposition 6. Algorithm** 1 has the complexity $O(L^2)$ where $L$ is the sum of lengths of paths of $P$.

There is one-to-one correspondence between complete paths of $P{\downarrow\uparrow}$ which have length bigger than two and paths of the graph $L(P)$ from *source* to *sink*. At the same time, the edge simple paths from $P{\downarrow\uparrow}$ correspond to the vertex simple paths in the graph $L(P)$. Thus, to check the presence or absence of undesired paths in $P{\downarrow\uparrow}$, it is enough to calculate the number of paths in $L(P)$ from *source* to *sink* and compare it with the number of paths in $P$. Therefore, as it is said in [6], the paths of $P{\downarrow\uparrow}$ are edge simple if and only if there are no cycles in the graph $L(P)$.

To verify the above property in graph $L(P)$, it is possible to use the DFS algorithm [8], that will be a bit modified for detecting cycles and calculating the number of paths from *source* to *sink*. The running time of the depth first search algorithm on the graph $L(P)$ is evaluated as $O(m)$, where $m$ is the number of arcs of the graph $L(P)$, $m \leq |V|^3$. On the other hand, an arc of graph $L(P)$ corresponds a pair of different arcs of paths of P and thus, $m < L^2$.

**Proposition 7.** The total complexity of Algorithm 1 and modified DFS algorithm is $O(L^2)$.

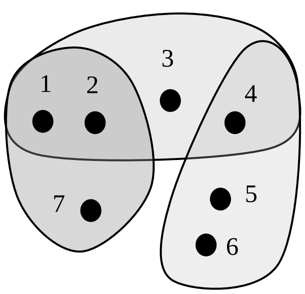
## IV. Verification of multiple virtual networks

When $n$ virtual networks should be implemented, two equinumerous sets $\boldsymbol{Z} = \{ Z_1, ..., Z_n \}$ and $\boldsymbol{P} = \{ P_1, ..., P_n \}$ are given where every virtual network implements a pair $(Z_i, P_j)$, $i = 1...$, $j = 1....$ Thus, we have two indexed families and assume that a virtual network implements a pair $(Z_i, P_i)$, $i = 1....$ The question arises whether it could happen that there are cyclic paths when the given virtual networks are implemented together. The reply is 'yes' and in Sections 4.A and 4.B two algorithms for such verification are proposed. Here we note that the reply is 'yes' even in the case when every virtual network can be implemented separately without cyclic paths.

### A. *Algorithm 2 for verifying multiple virtual networks*

In this section, we assume that we are given the graph of physical connections $G = (V, E)$ and $n$ virtual networks as a pair of equinumerous families: a family of sets of identifiers $\boldsymbol{Z} = \{ Z_1, ..., Z_n \}$ and a family of sets of paths $\boldsymbol{P} = \{ P_1, ..., P_n \}$. A pair $(Z_i, P_i)$, $i = 1..n$, sets the implementation of the $i$-th virtual network. It is required to determine whether there are undesired and / or cycling paths when implementing these virtual networks and, if it is the case, to determine for which packet identifiers this happens.

If the family $\boldsymbol{Z}$ is a partition of the union $\cup\boldsymbol{Z}$, i.e., a partition of the set of all identifiers, it is sufficient to verify independently every implementation of the virtual network $(Z_i, P_i)$, $i = 1..n$, since for $i \neq j$, the sets of $Z_i$ and $Z_j$ are disjoint, i.e., these sets have no common identifiers. But in the general case, the family $\boldsymbol{Z}$ can be a cover of the union $\cup\boldsymbol{Z}$, since the sets of $Z_i$ and $Z_j$ can intersect. Therefore, the problem is reduced to constructing the largest partition $\boldsymbol{A} = \{ A_1, ..., A_m \}$ w.r.t. the refinement such that for each $i = 1..n$ and $j = 1..m$, it holds that if $Z_i \cap A_j \neq \varnothing$ then $A_j \subseteq Z_i$. The independent verification for this partition item can be performed as described in Section 3. Each item $A_j$ of the partition $\boldsymbol{A}$, $j = 1..m$, corresponds to the subset $U_j$ of indexes such that $A_j = (\cap\{ Z_i \mid i \in U_j \}) \setminus (\cup\{ Z_i \mid i \notin U_j \})$ is not empty (Fig. 1). The corresponding set $R_j$ of paths is calculated as $\cup\{ P_i \mid i \in U_j \}$, which requires no more than $n$ union operations, and the family of paths $\boldsymbol{R} = \{ R_1, ..., R_m \}$ is calculated for run time $O(nm)$.




$\boldsymbol{Z} = \langle Z_1 = \{1, 2, 3, 4\}, Z_2 = \{4, 5, 6\}, Z_3 = \{7, 1, 2\}\rangle$

$\boldsymbol{A} = \langle A_1 = \{1, 2\}, A_2 = \{3\}, A_3 = \{4\}, A_4 = \{5, 6\}, A_5 = \{7\}\rangle$

$U_1 = \{1, 3\}, U_2 = \{1\}, U_3 = \{1, 2\}, U_4 = \{2\}, U_5 = \{3\}$

Fig. 1. The cover $\boldsymbol{Z}$ and the partition $\boldsymbol{A}$

Given the cover $\boldsymbol{Z}$. the partition $\boldsymbol{A}$ is constructed iteratively as follows. Suppose for the first $i$ elements of the cover, the partition of their union that contains $x_i$ elements and the union of corresponding subsets of $\boldsymbol{P}$ are already constructed. Consider the $(i+1)^{\text{th}}$ cover element. To do the above, it is necessary to construct the intersection of the $(i+1)^{\text{th}}$ cover item with each $j^{\text{th}}$ partition item, $j = 1..x_i$, and if the intersection is not empty then to construct the difference of the $j^{\text{th}}$ element of the partition and $(i+1)^{\text{th}}$ element of the cover. In addition, it is necessary to construct the difference of the $(i+1)^{\text{th}}$ element of the cover and the union of the first $i$ cover elements, as well as the union of the first $(i+1)$ elements of the cover as the union of the $(i+1)^{\text{th}}$ element of the cover and the union of the first $i$ cover elements.

**Algorithm 2:** Derivation a partition from family of sets
**Input:** A families of sets $\boldsymbol{Z} = \{ Z_1, ..., Z_n \}$ and $\boldsymbol{P} = \{ P_1, ..., P_n \}$
**Output:** The largest partition $\boldsymbol{A} = \{ A_1, ..., A_m \}$ w.r.t. the refinement such that for each $i = 1..n$ and $j = 1..m$, it holds that if $Z_i \cap A_j \neq \varnothing$ then $A_j \subseteq Z_i$ and corresponding family of sets $\boldsymbol{R} = \{ R_1, ..., R_m \}$

/* $\boldsymbol{U} = (U_1, \ldots, U_{|\boldsymbol{U}|})$ where $U_j \subseteq \{1..n\}$, $j = 1..|\boldsymbol{U}|$, is a family of subsets of indexes of $Z$, which corresponds to the current already derived $\boldsymbol{A} = (A_1, \ldots, A_{|\boldsymbol{U}|})$ where
$A_j = (\cap\{ Z_i \mid i \in U_j \}) \setminus (\cup\{ Z_i \mid i \notin U_j \})$, $j = 1..|\boldsymbol{U}|$;
$B = \cup\boldsymbol{A}$;
$\boldsymbol{U}'$ and $\boldsymbol{A}'$ correspond to the next partition which is still constructing; */

$U = ()$; $\boldsymbol{A} = ()$; $B = \varnothing$; $\boldsymbol{U}' = ()$; $\boldsymbol{A}' = ()$;
***for*** $i = 1..n$ ***do***
  ***for*** $j = 1..|\boldsymbol{U}|$ ***do***
    $X = A_j \cap Z_i$;
    ***if*** $X = \varnothing$ ***then***
      $\boldsymbol{A}' = \boldsymbol{A}' \cdot (A_j)$;
    $\boldsymbol{U}' = \boldsymbol{U}' \cdot (U_j)$;
    ***else*** $\boldsymbol{A}' = \boldsymbol{A}' \cdot (X)$;
      $\boldsymbol{U}' = \boldsymbol{U}' \cdot (U_j \cup \{i\})$;
      $X = A_j \setminus Z_i$;
      ***if*** $X \neq \varnothing$ ***then***
        $\boldsymbol{A}' = \boldsymbol{A}' \cdot (X)$;
        $\boldsymbol{U}' = \boldsymbol{U}' \cdot (U_j)$;
  $X = Z_i \setminus B$;
  ***if*** $X \neq \varnothing$ ***then***
    $\boldsymbol{A}' = \boldsymbol{A}' \cdot (X)$;
    $\boldsymbol{U}' = \boldsymbol{U}' \cdot (\{i\})$;
    $B = B \cup X$;
  $\boldsymbol{A} = \boldsymbol{A}'$; $\boldsymbol{A}' = ()$;
  $\boldsymbol{U} = \boldsymbol{U}'$; $\boldsymbol{U}' = ()$;
$\boldsymbol{R} = \{ \cup\{ P_i \mid i \in U_j \} \mid j = 1..|\boldsymbol{U}| \}$;
***return*** $(\boldsymbol{A}, \boldsymbol{R})$;

**Proposition 8.** Algorithm 2 returns the largest partition $\boldsymbol{A}$ w.r.t. the refinement such that for each $i = 1..n$ and $j = 1..m$, it holds that if $Z_i \cap A_j \neq \varnothing$ then $A_j \subseteq Z_i$, and the corresponding family of paths $\boldsymbol{R}$. The complexity of the algorithm is equal to $O(nm) = O(nk)$, where $m$ is the number of partition elements, $k = |\cup\boldsymbol{Z}|$ is the number of different identifiers in the family $\boldsymbol{Z}$.

At the $(i+1)^{\text{st}}$ step of Algorithm 2, we perform not more than $2x_i + 2$ operations such as intersection, difference or union of two sets, and add the number of elements that can be fluctuated from 0 to $x_i + 1$ to the partition. Thus, $x_i \le x_{i+1} \le 2x_i + 1$, $i = 1..n–1$. Denote by $y_i$ the total number of operations by the end of the $i^{th}$ step. We have as follows: $x_1 = 1$, $y_1 = 0$ as in the first step we simply select the first element of the cover; $x_i \le x_{i+1} \le 2x_i + 1$ for $i = 1..n-1$; $m = x_n$, $y_n \le (2x_1 + 2) + \ldots + (2x_{n-1} + 2) = 2(x_1 + \ldots + x_{n-1}) + 2(n-1) \le 2(x_n + \ldots + x_n) + 2(n-1) = 2(n-1)x_n + 2(n-1) = 2(n-1)(x_n + 1) = 2(n-1)(m + 1)$. This estimate for $n > 1$ is achieved when all $x_i$ are equal to 1, i.e., the cover consists of $n$ identical sets, $m = 1$, $y_n = 4n – 2$.

Thus, the complexity of constructing the partition is equal to $y_n = O(nm)$. Note that the number $m$ of partition elements does not exceed the number $k = |\cup\boldsymbol{Z}|$ of different identifiers in the family $\boldsymbol{Z}$. Therefore, $y_n = O(nk)$. Due to the above and the results of the previous section, the following statement is valid.

**Proposition 9.** The total complexity of constructing the partition and verification of its elements is $O(mL^2) = O(kL^2)$.

The complexity of constructing the partition $\boldsymbol{A}$ and corresponding family $\boldsymbol{R}$ of paths is $O(nm)$ while the construction of a graph $L(P)$ and verification this graph for one partition item has the complexity $O(L^2)$, and the number of such items is $m$. Thus, the total complexity of constructing the partition and verifying its elements is $O(nm + mL^2)$. Without loss of generality, we can assume that all the sets of the family $\boldsymbol{P}$ are not empty, and thus, $n \le L$ and $O(nm + mL^2) = O(mL^2) = O(kL^2)$.

### *B. Algorithm 3 for verifying multiple virtual networks*

Similar to the previous section, given the graph of physical connections $G = (V, E)$ and $n$ virtual networks as a pair of equinumerous families: a family of sets of identifiers $\boldsymbol{Z} = \{ Z_1, ..., Z_n \}$ and a family of sets of paths $\boldsymbol{P} = \{ P_1, ..., P_n \}$, a pair $(Z_i, P_i)$, $i = 1..n$, sets the implementation of the $i^{th}$ virtual network. The question is whether there are cycling paths on the data plane when implementing these virtual networks together. Differently from the problem statement in the previous section, it is not necessary to determine the identifiers of packets that can move along undesired edge simple paths and cycling paths.

The idea behind a proposed algorithm is as follows. Suppose that there are three subfamilies $\boldsymbol{V} \subseteq \boldsymbol{W} \subseteq \boldsymbol{Z}$ such that the intersection of their sets is not empty, i.e., $\cap\boldsymbol{V} \ne \varnothing$ and, therefore, $\cap\boldsymbol{W} \ne \varnothing$. These subfamilies can be represented as $\boldsymbol{V} = \{ Z_i \mid i \in U_V \}$ and $\boldsymbol{W} = \{ Z_i \mid i \in U_W \}$, where $U_V, U_W \subseteq \{1..n\}$ and $U_V \subseteq U_W$. Then the inclusion holds for corresponding sets of paths: $R_V \subseteq R_W$ where $R_V = \{ P_i \mid i \in U_V \}$ and $R_W = \{ P_i \mid i \in U_W \}$. If all the paths in $R_W{\downarrow\uparrow}$ are edge simple, then all the paths in the $R_V{\downarrow\uparrow}$ are also edge simple. Therefore, it is sufficient to check cycling paths corresponding to the maximum subfamilies w.r.t. the inclusion of the family $\boldsymbol{Z}$ with the non-empty intersection of sets of each subfamily, or the *M-subfamilies* of the family $\boldsymbol{Z}$, for short. For example in Fig. 1, $M$-subfamilies are $\boldsymbol{V} = \langle Z_1 = \{1, 2, 3, 4\}, Z_2 = \{4, 5, 6\}\rangle$, $\boldsymbol{W} = \langle Z_1 = \{1, 2, 3, 4\}, Z_3 = \{7, 1, 2\}\rangle$. Here $U_V = \{1, 2\}$, $U_F = \{1, 3\}$, $\cap\boldsymbol{V} = \{4\}$, $\cap\boldsymbol{W} = \{1, 2\}$. Subfamilies $\boldsymbol{V}$ и $\boldsymbol{W}$ are included only in one subfamily $\boldsymbol{Z}$, but $\cap\boldsymbol{Z} = \varnothing$.

**Proposition 10.** The intersection of the sets of the $M$-subfamily of the family $\boldsymbol{Z}$, is an element of the partition $\boldsymbol{A}$.

Indeed, let $\boldsymbol{V} \subseteq \boldsymbol{Z}$ be some maximum $M$-subfamily of the family $\boldsymbol{Z}$. Then, due to the maximality of the subfamily $\boldsymbol{V}$, for any $X \in \boldsymbol{Z} \setminus \boldsymbol{V}$, it holds that $(\cap\boldsymbol{V}) \cap X = \varnothing$. Hence $(\cap\boldsymbol{V}) \cap (\cup(\boldsymbol{Z} \setminus \boldsymbol{V})) = \varnothing$. Consequently, $(\cap\boldsymbol{V}) \setminus (\cup(\boldsymbol{Z} \setminus \boldsymbol{V})) = \cap\boldsymbol{V} \ne \varnothing$, and the latter means that $\cap\boldsymbol{V}$ is an element of the partition $\boldsymbol{A}$.

When comparing the complexity of verification using Algorithms 2 and 3, it is possible to compare the number of path sets, for each of which it needs to construct a graph $L(P)$ and verify this graph. For Algorithm 2, this is the number of partition elements that is at most $2^n–1$. For Algorithm 3, this is the number of $M$-subfamilies of the family $\boldsymbol{Z}$. Since such M-subfamilies of the family $\boldsymbol{Z}$ form an antichain w.r.t. inclusion, their number does not exceed the length of the maximum antichain. The latter is called the *width* of the Boolean lattice $B_n$, which, by the Sperner's Theorem [9], does not exceed $C_n^{\lfloor n/2 \rfloor}$. We have $C_n^{n/2} \sim 4^{n/2}/(\pi(n/2))^{1/2} = (1/((\pi/2)^{1/2}))2^n$. This is in $(\pi/2)^{1/2} \approx 1{,}25{*}n^{1/2}$ times less than $2^n–1$ for Algorithm 2.

Algorithm 2 constructs a family $\boldsymbol{U} = (U_1, \ldots, U_m)$, where $U_j \subseteq \{1..n\}$ for $j = 1..m$ is a family of index subsets of $\boldsymbol{Z}$ corresponding to the partition $\boldsymbol{A} = (A_1, \ldots, A_m)$, where $A_j = (\cap\{ Z_i \mid i \in U_j \}) \setminus (\cup\{ Z_i \mid i \notin U_j \}) \ne \varnothing$ for $j = 1..m$, and the corresponding family of paths $\boldsymbol{R} = (R_1, \ldots, R_m)$ where $R_j = \{ \cup\{ P_i \mid i \in U_j \}$. Therefore, it is enough to look for cycling paths for the sets $R_j$, corresponding to the maximum sets $U_j$ w.r.t. the nesting.

**Algorithm 3:** Derivation a family $\boldsymbol{R}$ of sets of paths, corresponding to the $M$-subfamilies of the family $\boldsymbol{Z}$.
**Input:** Families of subsets $\boldsymbol{Z} = \{ Z_1, ..., Z_n \}$ and $\boldsymbol{P} = \{ P_1, ..., P_n \}$
**Output:** A family $\boldsymbol{R} = \{ R_1, ..., R_{m'} \}$

```
/*  U = (U_1, …, U_|U|), where U_j ⊆ {1..n} for j = 1..|U|, is a family
    of subsets of indexes by Z, corresponding to the current
    (already constructed) partition A = (A_1, …, A_|U|), where
    A_j = (∩{ Z_i | i ∈ U_j }) \ (∪{ Z_i | i ∉ U_j }) for j = 1..|U|;
    B = ∪A;
    U′ and A′ — U and A corresponding to the next partition
    which is still constructing; */
U = (); A = (); B = ∅; U′ = (); A′ = ();
for i = 1..n do
|     for j = 1..|A| do
|     |     X = A_j ∩ Z_i;
|     |     if X = ∅ then
|     |     |     A′ = A′·(A_j);
|     |     └     U′ = U′·(U_j);
|     |     else  A′ = A′·(X);
|     |     |     U′ = U′·(U_j ∪ {i});
|     |     |     X = A_j \ Z_i;
|     |     |     if X ≠ ∅ then
|     |     |     |     A′ = A′·(X);
|     └     └     └     U′ = U′·(U_j);
|     X = Z_i \ B;
|     if X ≠ ∅ then
|     |     A′ = A′·(X);
|     |     U′ = U′·({i});
|     └     B = B ∪ X;
|     A = A′; A′ = ();
└     U = U′; U′ = ();
i = 1;
while i < |U| do
|     j = i+1;
|     while j ≤ |U| do
```

```
|       |       if U_i ⊆ U_j then
|       |       |       U = U \ {U_i};
|       |       └       break;
|       |       else    if U_j ⊂ U_i then
|       |       |               └       U = U \ {U_j};
|       └       └               else    j = j+1;
|       if j > |U| then
└       └       i = i+1;
R = { ∪{ P_i | i ∈ U_j } | j ∈ 1..|U| };
return R;
```

**Proposition 11.** Algorithm 3 returns a family of paths $\boldsymbol{R}$ corresponding to the $M$-subfamilies of the family $\boldsymbol{Z}$,. The complexity of the algorithm is equal to $O(nm + m^2) = O(nk + k^2)$, where $m$ is the number of partition elements, $k = |\cup\boldsymbol{Z}|$ is the number of different identifiers in the family $\boldsymbol{Z}$.

The time complexity of constructing a partition is $O(nm)$. The pair-wise comparison of the subsets of the family $\boldsymbol{U}$ has the complexity $O(m^2)$. We then select the subfamily of $m'$ maximum sets of indices w.r.t. the nesting and based on it a family $\boldsymbol{R}$ of paths is constructing with the time complexity $O(nm')$ . As $m' \le m \le k$, the total complexity is $O(nm + m^2 + nm) = O(nm + m^2) = O(nk + k^2)$.

**Proposition 12.** The total complexity of Algorithm 3 and corresponding verification is $O(mL^2) = O(kL^2)$.

The complexity of Algorithm 3 is equal to $O(nm + m^2)$, the constructing of the graph $L(P)$ and the verification of this graph for one set of paths has the complexity $O(L^2)$, and the number of such elements is $m' \le m \le k$. Thus, the total complexity of construction and verification is equal to $O(nm + m^2 + mL^2)$. Without loss of generality, we assume that all the sets of the family $\boldsymbol{P}$ are not empty, and thus, $n \le L$, $m \le L$ and $O(nm + m^2 + mL^2) = O(mL^2) = O(kL^2)$.

## V. A SUFFICIENT CONDITION FOR THE POSSIBILITY OF IMPLEMENTING ANY SET OF VIRTUAL NETWORKS WITHOUT UNDESIRED PATHS

In [5], the following problem is investigated: which properties should have a graph $G$ of physical connections in order it could be possible to implement any virtual network without generating undesired, duplicate and / or cycling paths.

In this section, the specification of a virtual network is a pair $(Z, D)$, where $Z$ is a set of packet identifiers that can be sent over this virtual network, and $D$ is a set of (ordered) pairs of hosts (a sender, a recipient). The set $D$ is called *normal* if it does not contain a pair of identical hosts $(x, x)$. The implementation of the virtual network $(Z, D)$ is a pair $(Z, P)$, where $P$ is a set of complete paths that connect all pairs of hosts of $D$. Given a complete path $p$, $h(p)$ denotes a pair (the head host of the path $p$, the tail host of the path $p$). The set of host pairs connected by paths of $P$ is denoted by $H(P) = \{h(p) \mid p \in P\}$. The pair $(Z, P)$ is the implementation of a virtual network $(Z, D)$ if $H(P) = D$. However, as shown in Section 2, when the switch rules defined by paths of $P$ are implemented, additional paths can be generated, and in general, these rules generate the superset of paths $P{\downarrow\uparrow} \supseteq P$. The path $p \in P{\downarrow\uparrow}$ is undesired if $h(p) \notin D$. The implementation is *strict* if it no undesired paths are generated, i.e. $H(P{\downarrow\uparrow}) = D$.

Graph $G$ for which any normal set of pairs of hosts can be strictly implemented without cycling paths, is called *perfect*.

A finite set $P$ of complete paths connecting all pairs of different hosts, i.e. set $P$ for which $H(P)$ is the greatest normal set of pairs of hosts, is called *perfect* if $P$ does not have two paths $\boldsymbol{p}ab\boldsymbol{r}$ and $\boldsymbol{p}'ab\boldsymbol{r}'$, where $\boldsymbol{p} \ne \boldsymbol{p}'$ and $\boldsymbol{r} \ne \boldsymbol{r}'$. In this case $P$ is arc closed, i.e. in this case, the arc closure does not induce undesired paths.

**Proposition 13.** Graph $G$ for which there is a perfect set $P$ of paths, is perfect. In this case, the set $P$ for each normal set of host pairs contains its strict implementation without cycling paths as a subset, as well as without duplication if $P$ does not contain duplicate paths.

In [7], the above condition is weakened: undesired paths are allowed, but only such that do not violate the security policy. This policy is described in terms of priorities assigned to hosts, and is formulated as a requirement: the priority of the host receiving the packet must be no less than the priority of the host sending the packet. The pair of hosts $(x, y)$ is called *permissible* if the priority of the host $x$ is not more than the priority of the host $y$. A complete path $p$ which does not violate the security requirements, i.e. is the pair $h(p)$ is permissible, is called *permissible*. The implementation $(Z, P)$ of the virtual network $(Z, D)$ is called *permissible* if the arc closure $P{\downarrow\uparrow}$ contains only permissible paths. Graph $G$ where a normal set of permissible pairs of hosts can be implemented without cycles, is called *permissibly perfect* or *p-perfect* for short. Finite set $P$ of complete paths that connects all permissible pairs of different hosts is called *p-perfect* if $P$ has no two paths $\boldsymbol{p}ab\boldsymbol{r}$ and $\boldsymbol{p}'ab\boldsymbol{r}'$ where $\boldsymbol{p} \ne \boldsymbol{p}'$ and $\boldsymbol{r} \ne \boldsymbol{r}'$.

**Proposition 14.** Graph $G$ for which there is a $p$-perfect set $P$ of paths, is $p$-perfect. In this case, the set $P$ for each permissible normal set of host pairs contains its strict implementation without cycles as a subset, as well as without duplication if $P$ does not contain duplicate paths.

At the same time, as shown in [7], the presence in the graph $G$ of the perfect ($p$-perfect) set paths is sufficient, but is not a necessary condition that the graph $G$ is perfect ($p$-perfect).

The concept of "perfection" is now extended to a set of virtual networks specified by a pair of equinumerous families: a family of sets of identifiers $\boldsymbol{Z} = \{ Z_1, ..., Z_n \}$ and a family of normal sets of pairs of hosts $\boldsymbol{D} = \{ D_1, ..., D_n \}$. If hosts have priorities and there is the security policy based on them, the virtual network $(Z_i, D_i)$ is *permissible* if the set $D_i$ of hosts is permissible.

**Proposition 15.** If the graph $G$ has a perfect ($p$-perfect) set $P$ paths, then any set $(\boldsymbol{Z}, \boldsymbol{D})$ of (permissible) virtual networks is (permissibly) strictly implemented without cycles. In this case, the set $P$ for each (permissible) normal set $D_i$ of pairs of hosts contains its strict (permissible) implementation $P_i$ without cycles as a subset $P_i \subseteq P$. If $P$ does not contain duplicate paths, then $P_i$ also does not contain duplicate paths.

Suppose that the graph $G$ has a perfect ($p$-perfect) set $P$ of paths. Propositions 13 and 14 imply as follows. First, any two paths from $P$ do not generate paths other than themselves, and, second, the fact that for any (permissible) normal set $D$ of pairs of hosts the set $P$ contains a subset $P_D \subseteq P$ of paths connecting these pairs of hosts, i.e. $H(P_D) = D$. Thus, for each virtual network $(Z_i, D_i)$, there exists the cycle-free implementation $(Z_i, P_i)$ where $P_i \subseteq P$, which is strict (permissible). If $P$ does not contain duplicate paths, then any subset $P_i$ of $P$ has no duplicate paths.

## VI. CONCLUSIONS

In this article, algorithms are proposed for verifying the implementation of virtual networks on the SDN data plane; the purpose of such checking is the detection of undesired paths for which packets are sent, cyclic paths and duplicate paths. Even if each virtual network is verified separately and has no effects specified above, the joint implementation of several virtual network still can have these effects. Consequently, two modifications of the algorithm for verifying the implementation of one virtual network to verify the joint implementation of several virtual networks are proposed. All proposed algorithms have polynomial complexity w.r.t. the source data. The possibility of implementing any virtual networks on the SDN data plane without undesirable effects is also considered in the paper for which a sufficient condition is established. In the future, we plan to investigate more effects, for example, such as network overload problems and other kinds of specifications for user requests.

## Acknowledgment

This work is partly supported by RFBR project N 20-07-00338 A.

## References

[1] Sezer, S., Scott-Hayward, S., Chouhan, P. K., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., and Rao, N. (2013). Are we ready for sdn? implementation challenges for software-defined networks. IEEE Communications Magazine, 51(7):36–43.

[2] Mohammed, A. H., Khaleefah, R. M., k. Hussein, M., and Amjad Abdulateef, I. (2020). A review software defined networking for internet of things. In 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), pages 1–8.

[3] OpenNetworkingFoundation (2012). Software-defined networking: The new norm for networks. ONF White Paper.

[4] OpenNetworkingFoundation (2015). Openflow switch specification version 1.5. 0. ONF Specification.

[5] Igor Burdonov, Nina Yevtushenko, Alexandr Kossachev. Implementing a Virtual Network on the SDN Data Plane. Proceedings 2020 IEEE East-West Design & Test Symposium (EWDTS). Varna, Bulgaria, September 4 – 7, 2020. pp. 279-283. ISBN: 978-1-7281-9898-9.

[6] Burdonov, I.; Kossachev, A.; Yevtushenko, N.; López, J.; Kushik, N. and Zeghlache, D. (2021). Preventive Model-based Verification and Repairing for SDN Requests. In Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE, ISBN 978-989-758-508-1 ISSN 2184-4895, pages 421-428. DOI: 10.5220/0010494504210428.

[7] Burdonov I.B., Yevtushenko N.V., Kossatchev A.S. Secure Implementing a Virtual Network on the SDN Data Plane. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2021;33(1):123-136. (In Russ.).

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to algorithms. MIT press.

[9] Sperner, Emanuel (1928), "Ein Satz über Untermengen einer endlichen Menge", Mathematische Zeitschrift (in German), 27 (1): 544–548.