Update Algebra: Toward Continuous, Non-Blocking Composition of Network Updates in SDN

Geng Li∗†, Y. Richard Yang∗, Franck Le‡, Yeon-sup Lim‡, Junqi Wang§

*Yale University, USA, †Tongji University, China, ‡IBM T.J. Watson Research Center, USA, §Rutgers University, USA

Abstract—The ability to support continuous network configuration updates is an important ability for enabling Software Defined Networks (SDN) to handle frequent or bursty changes. Current solutions for updating SDN configurations focus on one single update at a time, leading to slow, sequential (i.e., blocking) update execution. In this paper, we develop update algebra, a novel, systematic, theoretical framework based on abstract algebra, to enable continuous, non-blocking, fast composition of multiple updates. Specifically, by modeling each data-plane operation in the set of data-plane operations to be executed by an update as a set-theoretical projection, update algebra defines novel operation composition so that the number of projections for the same match remains constant regardless of the number of updates to be composed, leading to substantial performance benefits. Specifying the dependencies of the data-plane operations in updates as a subset of a free monoid provides the advantages of order-agnostic update composition, and the operation ordering is completely decoupled from the match. We conduct asymptotic analysis, extensive benchmarking using a real controller, and integration with a real application to demonstrate the benefits of update algebra. In particular, our asymptotic analysis demonstrates that in independent-update dominant settings, update completion time of update algebra remains asymptotically constant despite growth of the number of updates to be executed. Our benchmarking shows that update algebra can achieve 16x reduction in update latency even in settings with an update arrival rate of only 1.6s. Our integration with Hedera, a real SDN traffic engineering application, shows that update algebra can reduce average link bandwidth utilization by 30% compared with sequential updates.

I. INTRODUCTION

The ability to provide continuous, rapid, non-blocking network configuration updates is an essential capability for Software Defined Networking (SDN). First, it provides a foundation for the development of advanced applications with frequent network updates, which are typically prohibited or discouraged in current SDN systems. A recent trend is the application of machine learning to continuously and rapidly adapt routing strategies to minimize maximum link utilization, achieve proportional fairness, or maximize other objectives [1]–[3]. In addition, with studies having revealed that traffic in several settings (e.g., data centers) is highly dynamic, many solutions are advocating switching traffic at a finer granularity than flows, including sub-flows, and flowlets, to reduce congestion, and optimize path choices more frequently [4]–[6]. These trends and applications emphasize the need for controllers to support continuous, rapid, non-blocking network configuration updates. Second, a network may experience a set of rapid bursts of changes, causing an SDN controller to receive and handle a batch of network configuration updates [7]. For example, such updates may stem from the occurrence of unpredictable events including outages, Denial of Service attacks, BGP re-routes, or flash crowds.

Although a large amount of research effort has recently been devoted to developing efficient algorithms to update forwarding rules in SDN, existing solutions are unsatisfactory, and do not provide the required capability to handle frequent or short sudden groups of network changes. This is because despite having developed algorithms reduce the numbers of changes, or minimize the network update completion times [8], or preserve a range of properties [9]–[12] including loop and blackhole freedom, existing solutions focus on one single network configuration update at a time. In other words, they execute consecutive received network updates individually, and sequentially in a blocking manner [13]. Similar to the development of pipelining executions of instructions that has led to fundamental changes and performance improvements in computer architecture, the ability to execute continuous and non-blocking updates can lead to significant potential improvements in SDN control architecture [14].

Achieving continuous, non-blocking network updates is not straightforward. The first challenge is to guarantee correctness; a naïve execution may lead to unnecessary blocking due to dependencies on updates. To illustrate this, consider two consecutive updates: a first update $U_1$ sets the route for a flow, and a following update $U_2$ changes part of the route. Therefore, in a straightforward execution, the execution of $U_2$ cannot start until $U_1$ is finished, leading to a sequential (i.e., blocking execution) update model. The second challenge is that a network configuration update often consists of multiple operations that must be executed at different switches in a specific order to guarantee properties, such as blackhole freedom and waypoint routing (e.g., to traverse a sequence of VNFs) [12]. When executing consecutive updates in a non-blocking manner, these properties must also be preserved. The third challenge is that because network configuration updates often consist of multiple operations, updates do not operate atomically, and when a new update arrives, previous updates may be mid-execution. The execution status may even be unknown to the controller due to the fundamentally distributed nature of the SDN system.

In this paper, we develop a novel, systematic, and foundational theoretical framework based on abstract algebra to reason about and support continuous, non-blocking updates. The framework is motivated by the following two insights. First, when handling multiple updates (i.e., multiple batches of operations), operations on the same flow rules from consecutive updates may be replaced by fewer equivalent ones. For example, the creation of a flow rule followed by its modification can be replaced by the creation of an updated rule. To realize this insight, we model each operation as a set-theoretical projection, which provides flexible composition...
between operations. As such, the first challenge can be adequately addressed. Second, an update can be represented by a set of feasible sequences of operations whose order ensures the desired properties, and composition of multiple updates can be modeled as the application of different mathematical operations on these sequences. In abstract algebra, such a model can be well defined by a free monoid (of which a typical example is words with letters). By modeling each update as a subset of a free monoid on a set of projections, we can take advantage of the algebraic properties (i.e., associativity, idempotence, selectivity, commutativity) of the structure to identify equivalent operation sequences that preserve correctness and consistency properties, whether the former updates have been completely or partially executed. This insight addresses the second and third challenges.

We conduct asymptotic analysis, extensive benchmarking using a real controller, and integration with a real application to demonstrate the benefits of update algebra. The asymptotic analysis demonstrates that in independent-update dominant settings, the completion time with the existing sequential execution grows linearly, while that of the update algebra remains asymptotically constant. The benchmarking results show that update algebra can achieve 16x reduction in update latency even in settings with an update arrival rate of only 1.6/s. Finally, the integration with a real application shows that by applying update algebra, SDN Traffic Engineering applications (e.g., Hedera [2]) can reduce the average link bandwidth utilization by 30% compared to sequential execution.

II. BASIC MODELS

This section introduces the basic models to represent the network data plane configuration and individual network updates. The update algebra framework for the continuous, non-blocking composition of consecutive network updates will be specified in Section III. TABLE I summarizes the main variables and notations used in the following content.

### TABLE I: Terminologies

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SW = {sw}$</td>
<td>the set of switches (forwarding tables)</td>
</tr>
<tr>
<td>$M = {m}$</td>
<td>the set of all possible match values</td>
</tr>
<tr>
<td>$PR = {pr}$</td>
<td>the set of priorities</td>
</tr>
<tr>
<td>$key = (key_m, key_pr)$</td>
<td>flow rule key, defined by a match value endowed with a priority</td>
</tr>
<tr>
<td>$KEY = {key} = M \times PR$</td>
<td>the set of 2-tuple flow rule keys</td>
</tr>
<tr>
<td>$AC = {ac}$</td>
<td>the set of all possible actions</td>
</tr>
<tr>
<td>$AC^+ = AC \cup {Null}$</td>
<td>the action set with a null value</td>
</tr>
<tr>
<td>$C : SW \times KEY \rightarrow AC^+$</td>
<td>data plane configuration</td>
</tr>
<tr>
<td>$o(sw, key, ac) : C \rightarrow C'$</td>
<td>data plane operation</td>
</tr>
<tr>
<td>$u = o_1 \circ o_2 \circ \ldots \circ o_k$</td>
<td>update representative</td>
</tr>
<tr>
<td>$U = {u_1, u_2, \ldots} : C \rightarrow C'$</td>
<td>data plane update</td>
</tr>
<tr>
<td>$O(U)$</td>
<td>constituent operations of $U$</td>
</tr>
<tr>
<td>$\Omega(U)$</td>
<td>order constraint of update $U$</td>
</tr>
</tbody>
</table>

A. Data Plane Configurations

An SDN is comprised of a set of switches $SW$. A data plane configuration $C$ consists of a collection of flow rules that determine the packets’ forwarding states in the network. A flow rule defines an action $ac \in AC$ for flows matching a $key \in KEY$ at switch $sw \in SW$. A $key$ therefore has two attributes: (1) the matching criteria ($key_m$), and (2) a priority ($key_pr$). Equation (1) illustrates four flow rule keys.

$$key_1 = (pr = 1, m = \{src \_ ip = 10.0.0.1\})$$
$$key_2 = (pr = 1, m = \{src \_ ip = 10.0.0.2\})$$
$$key_3 = (pr = 2, m = \{dst \_ ip = 10.0.0.1\})$$
$$key_4 = (pr = 2, m = \{src \_ ip = 10.0.1.1, dst \_ port = 22\})$$

The matching criteria $key_m$ can have wildcards (*) to match ranges of values, and $key_pr$ is set to a finite integer number where a higher number means a higher priority; if a flow matches the matching criteria from multiple keys, the one with the highest priority is preferred and selected.

An action $ac$ of a flow rule represents the instruction that is applied to the flows matching it. An action can be forwarding to a specified next-hop, modifying a packet, or pipeline processing. The proposed models support the concept of multi-table pipelines in a switch: each flow table can simply be represented as an individual virtual switch. Formally, we define a data plane configuration as follows.

**Definition 1** (Data Plane Configuration). A data plane configuration $C$ of a network is defined as a map from the set of switches and flow rule keys to the set of actions $C : SW \times KEY \rightarrow AC^+$, where $AC^+ = AC \cup \{Null\}$.

Therefore, a configuration $C$ can be expressed with a 2-dimensional matrix over $SW \times KEY$ where each element $C_{sw, key}$ is an action $ac \in AC^+$ for ($sw$, $key$). $C_{sw, key} = Null$ represents the absence of a flow rule with key $key$ at $sw$. Fig. 1 illustrates an example of $C$ with four switches and two keys.

B. Data Plane Updates

A data plane update $U$ consisting of a set of data plane operations $O(U)$ on multiple switches and flow rules can change one configuration to another. Data plane operations act on keys at a particular switch and fall into one of three categories: addition, modification, or deletion. We denote these operations as $\{add(sw, key, ac), mod(sw, key, ac), del(sw, key)\}$; e.g., $add(sw, key, ac)$ means to add an action $ac$ for key $key$ at switch $sw$. For example, consider the network topology depicted in Fig. 2. In the first update $U_1$, flows from source IP address 10.0.0.1 ($key_1$) are forwarded along the route $A \rightarrow B \rightarrow C \rightarrow D$; in the second update $U_2$, flows from source IP address 10.0.0.2 ($key_2$) are forwarded along the route $B \rightarrow C \rightarrow D$, and the forwarding path for $key_1$ is changed to $A \rightarrow E \rightarrow C \rightarrow D$ (e.g., to balance the network load). Fig. 2 illustrates the corresponding data plane operations in both $U_1$ and $U_2$.

For further derivation, we introduce a more general expression $o$, parameterized by $SW \times KEY \times AC^+$: $add$, $mod$, and $del$ are all special cases of $o$; e.g., $add(sw, key, ac)$ or $mod(sw, key, ac)$ can be expressed as $o(sw, key, ac)$, and $del(sw, key)$ as $o(sw, key, Null)$. An operation $o$ transforms an arbitrary configuration $C$ to another $C'$ as follows:

**Definition 2** (Data Plane Operation). A data plane operation $o(sw, key, ac)$ is defined as a morphism between two configurations, i.e., $o : C \rightarrow C'$ or $C \xrightarrow{o} C'$, using:

```
Fig. 2: An example of two consecutive updates involving common flows (match key1). The order constraint ensuring blackhole freedom for each update is illustrated as a DAG of data plane operations.

That is, \( o(sw, key, ac) \) maps \( C \) to \( C' \) by changing \( C'_{sw,key} = ac \) but preserving other values of \( C \). Fig. 1 shows an example of applying the operation \( o_{a1}^{1'} = o(A, key1, fwd_E) \) on configuration \( C \) to obtain \( C' \), where the changed value is labeled in red.

Due to the distributed nature of the data plane, operations in an update can be applied in any order, resulting in different intermediate configuration states. However, some intermediate configurations during an update may violate consistency properties such as blackhole/loop-freedom; an order constraint is required to specify the feasible operation orders in an update.

**Definition 3** (Order Constraint). The order constraint of update \( U \) is defined as \( \Omega(U) \), which specifies the feasible operation orders in \( U \) to ensure consistency properties.

A concrete instantiation of \( \Omega(U) \) will be given in Section III-C. Fig. 2 presents an example of the order constraint to avoid blackholes where no matched rule exists during the updates; the order constraint is represented in the form of a directed acyclic graph (DAG); e.g., the DAG for \( U_1 \) shows that \( o_{a1} \) must be applied after \( o_{b1} \) and \( o_{c1} \). Note that generating the order constraints for various consistency properties is well studied in literature [9]–[12], and therefore not the focus of our work.

**Issues with Multiple Updates.** The model for individual updates is simple and well documented, but new issues arise when attempting to compose and execute multiple consecutive updates in a non-blocking manner. First, a naïve parallel execution of consecutive updates could lead to non-deterministic or incorrect outcomes. For example, in the scenario of Fig. 2, the execution of \( U_1 \) and \( U_2 \) could lead to non-deterministic configurations. This is because the operations \( o_{a1} \) and \( o_{a1}^{1'} \) apply to the same \( key1 \) at switch \( A \), but differ in actions: \( o_{a1}(sw, key) = o_{a1}^{1'}(sw, key) \), but \( o_{a1}.ac \neq o_{a1}^{1'}.ac \). Consequently, depending on the order in which they were applied, the two operations would lead to different configurations. Further, the order constraints in different updates may have dependencies that affect the non-blocking composition of consecutive updates and may be difficult to identify. For example, \( o_{a1}^{1'} \) in \( U_2 \) must be applied after \( o_{a1} \) in \( U_1 \) to guarantee the absence of blackholes. Lastly, when a new update arrives, previous updates may be mid-execution. The execution status may even be unknown to the controller due to the fundamentally distributed nature of the SDN system. For example, if \( U_1 \) is partially executed when \( U_2 \) arrives, the exact execution status may be unknown to the controller.

**III. UPDATE ALGEBRA FRAMEWORK**

In this section, we present our update algebra framework based on advanced abstract algebra. Specifically, data plane operations and updates are modeled by the notions of set-theoretical projection (Section III-A) and free monoid (Section III-B1), which provide the foundation and substantial algebraic properties for further composition. Based on these models, Section III-B2 proposes a general solution to compose consecutive updates. The solution is general as it can preserve any consistency property. Then, Section III-C introduces an efficient representation and composition using partial order to guarantee specific but commonly required properties. Lastly, Section III-D addresses the issue of composition with partially-executed updates to guarantee correctness under uncertainty.

The roadmap of update algebra is illustrated in Fig. 3.

**A. Operation as a Set-theoretical Projection**

As the basic unit of an update, operation and its composition are first introduced in update algebra, providing the foundation and freedom to replace and rearrange the data plane operations within one update or across distinct updates.

Recall Definition 2 where a data plane operation is defined as a morphism from one configuration to another with one value changed. The operation morphism can be viewed as a set-theoretical projection as follows:

**Definition 4** (Operation Projection). Considering the set of all possible configurations over a fixed \( SW \times KEY \), such a set is the Cartesian product \( (AC^\times)^{SW \times KEY} \). Then the operation \( o(sw, key, ac) \) can be considered a projection from \( (AC^\times)^{SW \times KEY} \) to the subset \( \{C|C_{sw,key} = ac\} \).

Definition 4 helps us to visualize an operation as a morphism of the configuration space, i.e., the Cartesian product \( (AC^\times)^{SW \times KEY} \) as shown in Fig. 4. For example, assume \( |SW \times KEY| = 2 \), then the space is two-dimensional. Therefore, \( o(sw, key, ac) \) projects any configuration points onto a line with \( (sw, key)\)-component = ac, and the operation on the other \( (sw, key)\) is “orthogonal” to \( o(sw, key, ac) \).

**Definition 5** (Morphism Equality). Two morphisms \( \pi_1 \) and \( \pi_2 \) are equivalent iff for any configuration \( C \), \( \pi_1(C) = \pi_2(C) \).
Since both the domain and codomain of an operation morphism are the configuration space, a set of data plane operations \{o_1, o_2, \ldots\} can be “composed”, i.e., arranged in a sequence to form a new morphism. Formally, we define a binary operation \circ, called composition of morphisms, such that for any \(o_1 : C_0 \to C_1\) and \(o_2 : C_1 \to C_2\), we have \(o_2 \circ o_1 : C_0 \to C_2\). By modeling an operation as a set-theoretical projection based on Definition 4, the operation composition holds the following properties:

**Theorem 1 (Operation Composition Properties).** The universal operation set \(O = \{o_1, o_2, \ldots\}\) and the binary operator \(\circ\) have the following four properties under Definition 5.

1. **Idempotence** (general): \(\forall o_1\)
   \(o_1 \circ o_1 = o_1\). \hspace{1cm} (4)

2. **Associativity** (general): \(\forall o_1, o_2, o_3\)
   \((o_3 \circ o_2) \circ o_1 = o_3 \circ (o_2 \circ o_1)\). \hspace{1cm} (5)

3. **Selectivity** (conditional): if \(o_1.(sw, key) = o_2.(sw, key)\), \(o_2 \circ o_1 = o_2\). \hspace{1cm} (6)

4. **Commutativity** (conditional): if \(o_1.(sw, key) \neq o_2.(sw, key)\), \(o_2 \circ o_1 = o_1 \circ o_2\). \hspace{1cm} (7)

Here \(o_1.(sw, key) = o_2.(sw, key)\) means both the \(sw\) and the \(key\) of \(o_1\) and \(o_2\) are equal. Theorem 1 can be directly proven by Definition 2. Intuitively, if each morphism is considered as a projection based on Definition 4, we can visualize the four operation properties in the form of projections in a configuration space as shown in Fig. 4.

![Fig. 4: Visualization of the four properties satisfied by projections.](image)

The properties of real-world SDN implementation (e.g., Flow table modification messages in OpenFlow) align exactly with these properties except for **Selectivity**. TABLE II illustrates a concrete example of **Selectivity**, where \(o_1.(sw, key) = o_2.(sw, key)\). Note that if \(o_2\) is a **mod**, all results after composition become an **add**. Because in our framework \(add(sw, key, ac_2) = mod(sw, key, ac_2) = o(sw, key, ac_2)\), and an **add** can act as a potent **mod** (an **add** operation can override an action even though an action for the identical **key** already resides in the requested table **sw**), therefore **Selectivity** property can still hold. Note that \(o.key\) is endowed with a determined priority, so operations on overlapping flow rules can be distinguished in composition.

**TABLE II: Composition rules for \(o_2 \circ o_1\).**

<table>
<thead>
<tr>
<th>(o_2 \setminus o_1)</th>
<th>(add(ac_1))</th>
<th>(mod(ac_1))</th>
<th>(del())</th>
</tr>
</thead>
<tbody>
<tr>
<td>(add(ac_2))</td>
<td>(add(ac_2))</td>
<td>(add(ac_2))</td>
<td>(add(ac_2))</td>
</tr>
<tr>
<td>(mod(ac_2))</td>
<td>(add(ac_2))</td>
<td>(add(ac_2))</td>
<td>(add(ac_2))</td>
</tr>
<tr>
<td>(del())</td>
<td>(del())</td>
<td>(del())</td>
<td>(del())</td>
</tr>
</tbody>
</table>

**B. General Update Representation and Composition**

1. **Update as a Subset of a Free Monoid:** Given the concept of operation composition, an update can be considered as the composition of data plane operations in different sequences, e.g., \(o_1 \circ o_2 \circ o_3\) and \(o_3 \circ o_1 \circ o_2\). Therefore, we extend the definition of an update using a **free monoid** [15] to address possible operation sequences and capture the order constraint. The formal definitions of the monoid and the free monoid are given as follows:

**Definition 6 (Monoid).** A **monoid** (sometimes called a semi-group with identity element) is a 3-tuple \((S, e, \ast)\), where \(S\) is a set, \(e \in S\) is an element, and \(\ast\) is a binary operation \(S \times S \to S\) such that for all \(x, y, z \in S\), \(x \ast (y \ast z) = (x \ast y) \ast z\) and \(e \ast x = x \ast e = x\).

**Definition 7 (Free Monoid).** A **free monoid** \(S^*\) on a generating set \(S\) is a monoid whose elements are all finite sequences (or strings) of zero or more elements from \(S\), with string concatenation as the monoid operation \(\ast\).

**Example:** Letter and Words - A typical **free monoid** example is about letters and words. Start with an alphabet \(S\) of letters, \(S = \{a, b, c, \ldots, z\}\). A word on the generating set \(S\) is a finite sequence of letters, e.g., \(infocom\), and \(paris\). Thus, \(S^*\) is the set of all possible words, the identity element \(e\) is an empty word, and the operation \(\ast\) is word-concatenation. In this **free monoid**, any words can be simply composed together to get a new word, e.g., \(na \ast on \ast noon\).

Consider a **free monoid** \(O^*\), in which the generating set is a data plane operation set \(O\) and the identity element \(e\) is an empty update \(\emptyset\) (i.e., applying nothing on a configuration \(C\)). Then an update can be modeled as follows:

**Definition 8 (Update Representation).** An update is represented as a set \(U = \{u_1, u_2, \ldots\}\) where \(u_i\), called a **representative**, is a sequence of elements from \(O(U)\), and satisfies the order constraint \(\Omega(U)\) and the following conditions:

- **Constitution:** \(\forall o \in O(U), o \in u_i\);
- **Distinction:** \(\forall o_1, o_2 \in u_i, o_1.(sw, key) \neq o_2.(sw, key)\).

**Remark.** \(U\) is a subset of the **free monoid** \(O(U)^*\) on the generating set \(O(U)\). Constitution condition guarantees that all representatives have the constituent operations. Distinction condition avoids configuring a flow rule twice in an update. Each representative \(u_i\) representing an order to compose \(O(U)\) can transform a \(C\) to another as follows:

**Definition 9 (Update Representative Morphism).** An update representative \(u_i = o_1 o_2 \ldots o_k\) can be considered as a mor-
Consider the composition of $U_1$ and $U_2$ in Fig. 2, i.e., $U_2 \circ U_1$. Randomly choose their representatives as $u_1$ and $u_2$ respectively, and then the composition $u_2 \circ u_1$ can be simplified as follows:

$$u_2 \circ u_1 = (o_{a_1} \circ o_{b_1} \circ o_{b_2} \circ o_{c_2} \circ o_{c_1}) \circ (o_{a_1} \circ o_{b_1} \circ o_{c_1})$$

$$= (o_{a_1} \circ o_{b_1} \circ o_{c_1} \circ o_{b_2} \circ o_{c_2}) \circ (o_{a_1} \circ o_{b_1} \circ o_{c_1})$$

$$= o_{a_1} \circ o_{a_1} \circ o_{b_1} \circ o_{b_2} \circ o_{c_1}$$

by Associativity and Commutativity in Theorem 1

$$= o_{a_1} \circ o_{b_1} \circ o_{c_1} \circ o_{b_2} \circ o_{c_2} \circ o_{c_1}$$

by Selectivity Theorem 1

$$= o_{a_1} \circ o_{b_1} \circ o_{c_1} \circ o_{b_2} \circ o_{c_2} \circ o_{c_1}$$

$u_{a_1}$ is changed to $o_{a_1}$ according to TABLE II.

**Example.** Consider the composition of $U_1$ and $U_2$ in Fig. 2, i.e., $U_2 \circ U_1$. Randomly choose their representatives as $u_1$ and $u_2$ respectively, and then the composition $u_2 \circ u_1$ can be simplified as follows:

$$u_2 \circ u_1 = (o_{a_1} \circ o_{b_1} \circ o_{b_2} \circ o_{c_2} \circ o_{c_1}) \circ (o_{a_1} \circ o_{b_1} \circ o_{c_1})$$

$$= (o_{a_1} \circ o_{b_1} \circ o_{c_1} \circ o_{b_2} \circ o_{c_2}) \circ (o_{a_1} \circ o_{b_1} \circ o_{c_1})$$

$$= o_{a_1} \circ o_{a_1} \circ o_{b_1} \circ o_{b_2} \circ o_{c_1}$$

by Associativity and Commutativity in Theorem 1

$$= o_{a_1} \circ o_{b_1} \circ o_{c_1} \circ o_{b_2} \circ o_{c_2} \circ o_{c_1}$$

by Selectivity Theorem 1

$$= o_{a_1} \circ o_{b_1} \circ o_{c_1} \circ o_{b_2} \circ o_{c_2} \circ o_{c_1}$$

$u_{a_1}$ is changed to $o_{a_1}$ according to TABLE II.
The hidden (implicit) dependency element $\Omega$(loss. For example, in because of the transitivity property. When breaking the cycles, however, many of the orders are implicit and not shown).

**Step 2:** Replace operations by composed counterparts in $O(U_k \circ \ldots \circ U_1)$.

**Step 3:** Search cycles in $O(U_k \circ \ldots \circ U_1)$ based on the properties of $\prec$.

**Step 4.1:** In each cycle, remove element $o_1 \prec o_2$ if $\exists o'_1 \prec o'_2 \in O(U_i), o''_1 \prec o''_2 \in O(U_j), s.t. o_1.(sw, key) = o'_1.(sw, key) = o''_1.(sw, key), o_2.(sw, key) = o'_2.(sw, key)$ and $i < j$.

**Step 4.2:** At elements $o_m \prec o_n, \forall m, n$, if $o_m \prec o_1$ and $o_2 \prec o_n$.

**Step 5:** Simplify $O(U_k \circ \ldots \circ U_1)$ based on the properties of $\prec$.

**Example.** Fig. 6 shows the efficient composition of $U_1$ and $U_2$ in the example of Fig. 2. We use the format of DAGs for simple illustration. After Steps 1 and 2, there is a cycle between $o_{a_1''}$ and $o_{b_1'}$, where $o_{a_1''} = o_{a_1'} \circ o_{a_1}$ is computed according to TABLE II. Since $o_{a_1'} \prec o_{a_1''}$ is inherited from $O(U_1)$ but conflicts with $O(U_2)$, it will be removed as depicted in Step 4.1 of our solution.

**Theorem 4.** If the sequential execution from $O(U_1), O(U_2), \ldots$, to $O(U_k)$ guarantee the basic consistency, the non-blocking execution of $O(U_k \circ \ldots \circ U_1)$ in update algebra can guarantee the basic consistency.

The detailed proof of Theorem 4 is omitted due to space limitation. The intuition here is that the first two steps allow $O(U_k \circ \ldots \circ U_1)$ to inherit the orders from $\{O(U_1), O(U_2), \ldots, O(U_k)\}$. Even when there exist orders between operations with different properties, such orders can be captured in the first two steps. In Step 3, any cycle indicates the presence of order conflicts between $U_i$ and $U_j$ for the same flow rules. Since based on Selectivity in Theorem 1, the conflicting operations are overwritten by the last one, applying the same rationale into the order constraint, only the order element in $U_j, (i < j)$, is preserved as depicted in Step 4.1.

**Partial Execution.** Consider an update composition of $U_1$ and $U_2$ in which $U_1$ is partially-executed when $U_2$ arrives at the controller. Let $U_1^+$ denote the part of $U_1$ that has been applied (Completed), and $U_1^−$ the remainder, i.e., $U_1 = U_1^+ \circ U_1^−$. Based on Associativity in Theorem 3, we have:

$$U_2 \circ U_1 = U_2 \circ (U_1^− \circ U_1^+) = (U_2 \circ U_1^−) \circ U_1^+ \tag{8}$$

Fig. 8(a) presents the transitions of configuration states according to executed updates. Note that we compose $U_1$ and $U_2$ at the configuration state $C_i^j$ with a partial execution $U_1^+$, which means $U_2 \circ U_1^−$ is our target composition. The problem is that $C_i^j$ and $U_1^−$ may be unknown to the controller due to the uncertainty state of operations at Scheduled state. A solution is to prohibit Scheduled states during composition; i.e., once $U_2$ arrives, the controller stops scheduling new operations in $U_1$.
and collects responses for all Scheduled ones until their states become steady one such as Idle and Completed. However, this solution inefficiently blocks updates due to the interruption for collecting the responses.

2) Solution: The basic idea behind our solution is to treat Scheduled operations as "not applied" at a data plane during composition. Consider an update composition $U_2 \circ U_1$ in Fig. 8(a). Assume that one operation $o_1$ in the update $U_1$ is in Scheduled state. Suppose that $C'_1$ and $C'^2_1$ are the configurations after failed and successful responses for $o_1$, respectively. Based on the two Invariants, the current configuration $C'_1$ must be either $C^F_1$ or $C^S_1$. Then, we aim to solve:

**Problem 1.** Given $C'_1 \in \{C^F_1, C^S_1\}$, find $U : C'_1 \rightarrow C_2$. Let $U^S : C^S_1 \rightarrow C_2$ to denote the composed update to achieve $U_2$. As depicted in Fig. 8(b), we have

$$C^S_2 = o_1(C^S_1), \quad (9)$$

$$C_2 = U^S \circ o_1(C^F_1). \quad (10)$$

Recall that our solution treats Scheduled operations as "not applied", i.e., $C'_1 = C^F_1$. Based on the assumption $C'_1 = C^F_1$, the solution is $U = U^S \circ o_1$ from Equation (10). We show that the solution yields $U(C'_1) = C_2$ even in the case of $C'_1 = C^S_1$ as follows:

$$U^S \circ o_1(C^S_1) = U^S \circ o_1(C^S_1) \quad (11)$$

$$= U^S \circ o_1(o_1(C^F_1)) \quad (12)$$

$$= (U^S \circ o_1) \circ o_1(C^F_1) \quad (13)$$

$$= U^S \circ o_1(o_1(C^F_1)) \quad (14)$$

$$= U^S \circ o_1(C^F_1) \quad (15)$$

$$= C_2. \quad (16)$$

where Equation (12), (14), (15) and (16) are deduced from (9), **Associativity, Idempotent** in Theorem 1 and Equation (10), respectively.

Therefore, regardless of the uncertainty due to Scheduled operations, the update algebra is able to compute an update composition and achieve correctness based on our solution.

**IV. EVALUATION**

This section evaluates the benefits of update algebra through asymptotic analysis (Section IV-A), extensive benchmarking using a real controller (Section IV-B), and integration with a real application (Section IV-C).

A. Asymptotic Analysis

We conduct an asymptotic analysis to compare the correctness, overhead, and completion time of the composition of consecutive updates using sequential, parallel, and update algebra based executions. We denote $p \in [0, 1]$ the probability that two consecutive updates are related, i.e., involve common flows. In other words, $1 - p$ denotes the probability that two consecutive updates are fully independent. Each update is represented as a sequence of operations. For simplicity, if two consecutive updates $U_1$ and $U_2$ are related (with probability $p$), the two sequences are modeled as overlapping, and the common segment is randomly selected using a uniform distribution. The total length of the update after composition is provided by Equation (17), with $N_1$ and $N_2$ representing the lengths of $U_1$ and $U_2$. The details of the proof are omitted due to space limitation.

$$f(N_1, N_2) = \frac{N_1^2 + N_1N_2 + N_2^2}{N_1 + N_2} - 1 \quad (17)$$

**TABLE III** summarizes the results. First, as explained in Section II, parallel execution may violate correctness as it does not respect update dependencies. Second, **TABLE III** shows that the completion time of a sequential execution increases linearly with the number of updates. In contrast, in independent-update dominant settings, the completion time with update algebra stays asymptotically constant, similar to that of a parallel execution, while guaranteeing correctness.

**TABLE III:** Asymptotic analysis of the composition of two consecutive updates $U_1$ and $U_2.$

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Correctness</th>
<th>Operation Number</th>
<th>Update Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>✓</td>
<td>$N_1 + N_2$</td>
<td>$T_1 + T_2$</td>
</tr>
<tr>
<td>Parallel</td>
<td>×</td>
<td>$N_1 + N_2$</td>
<td>$\max(T_1, T_2)$</td>
</tr>
<tr>
<td>Update</td>
<td>✓</td>
<td>$\approx f(N_1, N_2)p + (N_1 + N_2)(1 - p)$</td>
<td>$\approx f(T_1, T_2)p + \max(T_1, T_2)(1 - p)$</td>
</tr>
</tbody>
</table>

**B. Benchmarking**

**Methodology:** We deploy update algebra in a SDN network running a Ryu 4.24 controller, and twenty Open vSwitch [18] 2.5.4, connected in a FatTree topology, which is common in data centers.

We generate updates as follows: each update involves a random number of flows ranging from 17 to 20. Two successive updates include common flows with a probability of $p$. We vary $p$, and for each case, we synthesize 300 network update events with different Poisson arrival rates $\lambda$.

The controller schedules the arriving updates according to the order constraints (i.e., DAG) derived using the algorithm [9] proposed by Forster et al. to ensure a lack of blackholes and loops during the updates. We compare two composition approaches: in the first, the controller keeps track of the state of each operation, and continuously merges new arriving updates with ongoing ones through update algebra; in the second approach, the controller executes the arriving updates in a sequential (blocking) manner, i.e., according to a First-In-First-Out (FIFO) policy.

To compare the performance, we report two metrics: **average operation number**, and **average update completion time**. The former is the average number of operations applied in the data plane for the 300 updates. The latter is the average duration of an update which begins when it is considered by
As such, with algebra, resulting in a reduction in numbers of operations.

Executions yield similar constant times at low arrival rates as update algebra. Both the sequential and update algebra based update completion times of both the sequential execution and M/M/c system corresponds to the lower bound. The average response time in the M/M/1 system reflects the where c is the number of switches in the network. Therefore, parallel execution can be approximated by an M/M/c system, that updates arrive at switches uniformly at random, the updates can be executed concurrently, the execution bottleneck distributed in our experiments. For the parallel execution, as frequent updates are prohibited in current systems, and the default value is set to 0.2/s (an update every five seconds). In contrast, by merging consecutive updates in a non-blocking manner, update algebra allows updates to be quickly enforced, and the network performance keeps increasing with the update frequency. With an update frequency of 1.2/s, update algebra increases the network utilization by 13% compared to the default value 0.2/s, and outperforms the sequential execution by 30%. These results demonstrate the benefits of update algebra with a real application.

V. RELATED WORK

Consistent Updates: A concerted research effort has recently been made to tackle the problem of network updates in SDN for different aims. Xitao et al. [8] minimize the number and latency of rule updates for TCAM-based switches by eliminating redundant and unnecessary entry moves. Reitblatt et al. [19] introduce the notion of consistent network updates, and propose a two-phase update approach. Solutions supporting a broad range of consistent properties are proposed, including loop freedom [9], congestion-freedom [10], waypoint routing [11] and customizable properties [12]. However, existing work is limited to a single network update at a time, and does not handle consecutive network updates. Peter et al. [13] mention the inter-update scheduling problem in their future work, but only provide a strawman solution as an enhancement to the sequential approach. In contrast, our work allows controllers to efficiently merge multiple network updates to handle continuous and non-blocking network changes while preserving desired properties. To the best of the authors’ knowledge, our work is the first to handle multiple updates as a group.

Policy Composition: Researchers have also investigated composing policies. Several recent SDN policy languages and controller hypervisors (e.g., NetKAT [20], Pyretic [21]) support taking multiple high-level policies and generating flow tables that fulfill the semantics of the sequential and parallel composition. However, network update operations are different from flow rules, and present unique challenges as well as distinct requirements. For example, as discussed in Section III-D, network updates may be partially executed, and controllers may not have a complete and precise up-to-date
view of the update progress. In addition, network updates have distinct consistency requirements that differ from those of policy composition. Consequently, existing work on policy composition cannot be applied to composing network updates. Instead, we developed a theoretical framework that captured the unique characteristics of network updates, and allowed us to reason about their properties and composition.

ACKNOWLEDGMENT

We would like to thank Sanat Khurana who found a counterexample of our earlier algorithm and helped with the design of a new solution. This research was supported in part by NSF grant #1440745, CC*IIE Integration: Dynamically Optimizing Update Data Workflow with a Software Defined Science grant #61701347, NSFC #61702373 and NSFC #61672385; NSF policies,” in Proc. of SIGCOMM, 2016.

REFERENCES


