# Automating SDN Composition: A Database Perspective

Anduo Wang* Jason Croft‡
* Temple University ‡University of Illinois at Urbana-Champaign

## ABSTRACT

To keep up with the complexity of SDN management, it is generally agreed that modular development of the controller software plays a key role. However, forming a *correct* modular composition is still a challenging task. An operator needs to understand the module internals and to manually wire inter-module interactions that often depend on the underlying packets. In contrast, this poster presents a novel database approach towards *automatic*, *packet-agnostic composition* out of *black-box modules*.

## 1. PROBLEM STATEMENT

To keep up with the complexity of SDN management, it is generally agreed that modular development of the controller software plays a key role. Advanced modular SDN platforms [1] enable operators to program their network goals — traffic forwarding, fault tolerance, resource provisioning, stateful middleboxes, service chains, and more — as independently created control modules. However, forming a *correct* modular composition, where multiple control modules collectively realize a coherent network behavior, is still a challenging task. An operator often needs to understand a member module's internals to determine the fine-grained inter-module interactions in all execution runs. Thus, while the SDN paradigm simplifies *how to realize* individual control modules, it relies on the operator to determine *what is the correct composition*.

While there is little hope in building a general-purpose procedure that stitches together arbitrary software modules, we believe it is feasible to automate composition for a large set of SDN modules. We build on the insight that an SDN control module, regardless of the disparate language constructs being adopted, conceptually follows a common pattern of *checking and repair*. A module continuously *checks* the network states for violations of some invariants; if a violation is detected, the module computes a new network state to *repair* the network and restore the invariants. Based on this assumption, the crux to a correct SDN composition is to prevent any module repair from reconfiguring the network into a state that violates other modules.



*SOSR '17, April 03-04, 2017, Santa Clara, CA, USA*

DOI: http://dx.doi.org/10.1145/3050220.3060612

**Correctness criteria.** First, we establish what constitutes a *correct* composition, identifying criteria that guide us through the correct composition. We propose defining composition correctness as a semantic-preserving property: a composite is correct if it respects the semantics (invariants) of every constitute module. Given a set of modules $M$, each $m \in M$ maintains (checks and repairs) some invariant $i_m$ about the network, a composition out of these modules is correct if it continues to enforce every invariants $\bigwedge_{m \in M} i_m$.

To see the strength of this definition, consider two simple modules `p,q` in NetKAT [1] syntax, where · represents sequential composition, and + denotes parallel composition. `p` drops (`0`) any packets whose destination matches `X` (`dst=X`), and `q` sets the destination of some packets (those matching `some_predicate`) to `X` (`dst←X`):

```
p ≜(dst=X)· 0
q ≜ (some_predicate) · (dst←X)
```

There are three different ways to compose `p` and `q`, namely `p+q`, `p·q`, and `q·p`. Both `p+q` and `p·q` will drop packets originally destined to `X` but redirect some other packets (those matching `some_predicate`) to `X`. Only `q·p` will drop all packets destined to `X` (including packets redirected to `X` by `q`). If we view the semantics (invariant) of `p` as removing any packets destined to `X`, only `q·p` respects `p`'s semantics. `p+q` and `p·q` — by leaving some packets destined to `X` unfiltered — both violate `q`'s intention. Thus, the only correct composition is `q·p`.

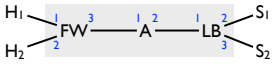

Figure 1: Example network

**Reference composition.** While our criteria enable us to decide whether or not a composition is correct, it does not automatically construct one. To see the subtlety in constructing a correct composition, consider the network in Fig. 1. The network consists of three boxes — firewall `FW`, switch `A`, and load balancer `LB` — that connect two external hosts $H_1, H_2$ to two back-end servers $S_1, S_2$ sharing a public address (`S`). Also suppose the network is controlled by three modules independently developed: routing (`rt`) establishes routes between the clients and the public face of the servers; firewall (`fw`) filters packets originated from certain clients ($H_1$); and load balancer (`lb`) distributes and forwards packets from/to the servers. A possible high-level specification of `fw,lb` is illustrated in the following (we omit `rt`):

```
fw ≜ ¬(src=H1·dst=S) + ¬(src=S·dst=H1)
lb ≜ if_(ec, lb1, lb2) where
        lb1 ≜ src=H1·dst=S·dst←S1·pt←2 + ···
        lb2 ≜ src=S1·dst=H1·src←S·pt←1 + ···
```

Note that lb, with an auxiliary filter ($e_c \triangleq$ sw=FW·pt=1···) denoting packets entering the network from the clients, defines two distinctive repairing actions based on the pattern of the packets. For packets from the clients ($e_c$), $lb_1$ maps the public server address (S) to a particular private address ($S_1$), otherwise it restores the source address from a particular server to S. A correct composition, a *master program* that modularly combines these modules, is the following:

$$\texttt{if\_}(e_c,\ \texttt{fw}\cdot(lb_1+\texttt{rt}),\ lb_2\cdot\texttt{fw}\cdot\texttt{rt})$$

This master program instructs the packets from the clients to be processed by the firewall first, followed by routing and load balancer ($\texttt{fw}\cdot(lb_1+\texttt{rt})$). For packets from the servers, the master program schedules the load balancer before the firewall ($lb_2\cdot\texttt{fw}\cdot\texttt{rt}$) so that the public address (S) can be properly restored for the firewall. In this composition, the member module lb can *not* be treated as a black-box. Rather, the operator needs to understand the *internals* of lb and to explicitly wire its interactions with fw, rt. It is also called the "decompose and re-compose" problem, and additional requirements can quickly add complexity.

In contrast to this manual subtle composition, our objective is to *automatically* generate a *traffic-agnostic composition* out of (possibly finer-grained) *black-box modules*, like the following equivalent composition ($e \triangleq e_s + e_c$ denoting all packets entering the network edges):

$$e\cdot lb_2\cdot\texttt{fw}\cdot(lb_1+\texttt{rt})$$

## 2. A DATABASE SOLUTION

Recall the two simple modules p, q (§1). The correct composition q·p can be derived from the fact that the network repair pushed by q can change the network into a state that activates p — a state that violates the invariants checked by p. We introduce behavioral dependency, denoted by x→y (or y←x), to capture the dependence of module x on y: when x's repair can inadvertently introduce violations to y, x requires further (repairing) service from y. If x→y, we compose by x·y; otherwise, if neither x→y nor y→x, we compose by x+y.

This composition strategy generalizes to a set of modules $M$ as follows: First, we compute a dependency graph $D$ that represents all dependencies. Next, we develop a stratification algorithm to convert $D$ into a correct composition.

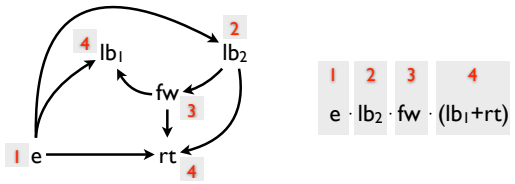


Figure 2: Stratification of dependency graph (left). Constructing the reference composition based on stratification number (right).

**Constructing composition (stratification algorithm).** Fig. 2 (left) depicts the dependency graph for the modules {e, $lb_1$, $lb_2$, fw, rt} collectively driving the network in Fig. 1. In the dependency graph, each module x (node) is associated with a stratification number (sn(x) in red). Intuitively, this number decides the (sequential) ordering in the modular composition. Modules with equal stratification numbers, on the other hand, are independent and are composed in parallel. More precisely, the stratification number is given by:

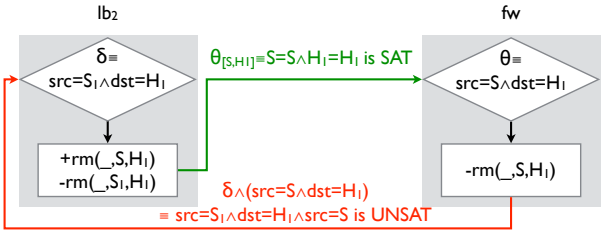


Figure 3: Automatic dependency discovery by SAT/UNSAT reasoning. Invariants $\theta, \delta$ in the diamond, repairing updates in the square.

$$\texttt{sn(x)} = \begin{cases} max(\{\texttt{sn(y)} \mid \text{if y is a parent of x in } D\}) + 1 \\ 1 \text{ if x is the root of } D \end{cases}$$

Fig. 2 (right) shows the composition constructed from the modules' stratification numbers. In general, we have:

$$\forall u, v, x, y \in M, \texttt{sn(u)}+1 = \texttt{sn(v)}+1 = \texttt{sn(x)} = \texttt{sn(y)}$$
$$comp(M) = \cdots(\texttt{u}+\texttt{v}+\cdots)\cdot(\texttt{x}+\texttt{y}+\cdots)\cdots$$

**Generating the dependency graph.** We build $D$ by determining the behavioral dependency relation between every pair of modules. To determine if x→y (or y→x), we leverage automatic reasoning based on database irrelevant updates. The key idea is to represent all network states as database tables [3]: the dataplane as shared, stored tables, while the modules operate on derived tables (database views); and to reduce a module x's operations — the checking and repairing of invariants — to a database query (view $x_v$) and a database update (view update $x_u$). This allows us to apply database irrelevance reasoning [2] to generate the dependency relation: For ∀x, y∈ $M$, x depends on y (x→y) if x's updates can cause a violation to y ($x_u$ is relevant to $y_v$), but y's updates can never affect x ($y_u$ is irrelevant to $x_v$).

Fig. 3 illustrates the reasoning process for deciding $lb_2$ →fw: $lb_2$ update (insert $\texttt{+rm(\_,S,}H_1\texttt{)}$ to reachability table rm) can violate fw because the invariant $\theta$ is satisfiable (SAT); but fw will not inadvertently affect $lb_2$ because the conjunction of $\delta$ and the repairs is not satisfiable (UNSAT).

While our composition method is discussed in a concrete SDN platform (i.e., NetKAT), we believe our strategy will apply to a large family of SDN modules — the particular choice of language is irrelevant; the enabling assumption is an SDN control's check-repair structure.

**Acknowledgment** This material is based upon work supported by the NSF Grant CNS 1657285.

## 3. REFERENCES

[1] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. Netkat: Semantic foundations for networks. In *POPL '14*.

[2] Blakeley, J. A., Coburn, N., and Larson, P.-V. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst. 14*, 3 (Sept. 1989), 369–400.

[3] Wang, A., Mei, X., Croft, J., Caesar, M., and Godfrey, B. Ravel: A database-defined network. In *SOSR '16*.