Abstract—The value of database in advancing networking — in the paradigm shift from protocols to software-defined networking — was once highlighted by database-inspired management of network states. Moving beyond factual states, this paper considers semantics management a new frontier in the databases-networking knowledge “transfer”, seeking to manage network policies via structural manipulation of the corresponding software (program). As a proof of concept, we make a case of semantics-based network transformation with the datalog structure and the chase, an elegant process for handling data dependencies (semantics). Our main result is an extension of the classic chase to fauré-log, a networking extension of datalog for the richer networking policies.

Index Terms—Software-defined networking, network datalog, semantics-based transformation, the Chase

I. INTRODUCTION

Database has played an active role in advancing networking research, notably, database inspired distributed network state management, a key enabler in the landscape changing movement of software-defined networks (SDN). Before custom built distributed key-value stores were available, production-scale SDN platforms achieved strong consistency among the replicas spanning across many datacenters by adopting the classical ACID notion and existing transactional databases [1]. As a second example, network verification, a topic that garnered wide interest and later borrows heavily from software engineering and formal methods, was first powered by datalog, which was lauded as a general modeling tool that enables declarative specification and fast simulation. With the clean and extensible network state management in datalog, it is not surprising that forerunners like Batfish [2] became a foundation (literally a component system) to many subsequent (often imperative and specialized) verifiers. If databases has helped shaping SDN, what is the next frontier?

One pain point in networking today is semantic management: As networks become more programmable (software-defined), the networks themselves are viewed as programs exhibiting richer semantics (policies), for which semantic management — maintaining the policy properties embedded in a network program — are pursued. Most tools for network semantics management take a primitive behavioral approach in which a network is modeled by a function whose inputs (packets) are exhaustively examined. For example, a network preserves its policy after an update if the tool cannot find a single input packet that exhibits the function — e.g., forwarding path for all packets — different. While great effort went into modeling the network function, the focus is to speed up evaluation on a huge input packet space. More advanced techniques capable of exploiting the network structure itself, however, is rare. The only structural approach we are aware is network transformers [3], [4] that use syntactic heuristics (e.g., based on network symmetry) to compress a network model into a smaller one while preserving certain properties.

In this paper, we consider semantic management a new frontier in network advancement by databases, pursuing the question: can we bring about structural network management in which the intended semantic management — analysis and transformation — intuitively maps to syntactic operation on the corresponding network representation? As a first step towards an affirmative answer, we study the concrete problem of semantics-based network transformation with the chase.

The chase [5], [6] is an elegant syntactic rewrite that takes a datalog query \( Q \) and a data dependency \( \sigma \) as input, transforms \( Q \) into \( Q' \) such that any “element” of \( Q \) that is incompatible with \( \sigma \) is intuitively corrected in \( Q' \) to satisfy \( \sigma \), written as \( \text{chase}(Q, \sigma) = Q' \). To transform a network expressed in a datalog program \( P \), based on its policies given by a set of data constraints (i.e. dependencies) \( \Sigma \), our idea is to repeatedly chase \( P \) with every dependency \( \sigma \in \Sigma \), until we converge to a unique new network \( P' \) that properly reflects all policies.

The key is to extend the classic chase theory to networking. We first identified a limitation to the classic chase: the classic chase uses the standard query evaluation on a datalog program’s instantiated database which is an incomplete database that requires more advanced evaluation. To address this mismatch, we generalize the chase to support richer semantics by developing a new algorithm \( \text{chase}(P, \sigma) \), where both \( P \), \( \sigma \) are datalog rules: by instantiating the \( P \) into an incomplete database instance \( I \), and processing \( \sigma \) as a data query over \( I \) by leveraging fauré-log evaluation, our earlier work on extending datalog to partial network state [8]. Our main finding is that, the new chase with a set of dependencies remains Church-Rosser [7] — the new chasing result remains unique (up to renaming of variable symbols) when terminating. While the classic chase transforms \( P \) with restricted dependencies into a single unique query, the new chase applicable to richer dependencies converts \( P \) into a unique set of programs.

II. A RUNNING EXAMPLE

We motivate semantics-based networking transformation by a running example in Figure 1: reachability between four groups of hosts \( \{A, B, C, D\} \) in the left and right ovals is controlled by the policies distributed at the 5 routers (center); \( R_2 \) is configured to block any packet header with source matching \( B \) (i.e. prefix belonging to group \( B \) and destination...
in D, R₁ (R₅, respectively) is set to rewrite header matching the pattern (B,C) ((A,C)) to (A,C) ((A,D)). That is, if the source of the header is in B (the destination of the header is in C), then modify the source (destination) to a host in A (D). In the presence of such rewrites, does R₂, a node en-route all pair-wise paths, still enforce the semantics — preventing group B from contacting D? The answer is no. A host in B can reach a destination in D by injecting instead a packet with a header that matches (B,C). Detecting such security hole with a existing behavioral analysis (e.g., Batfish [2]) requires insight into what packet to examine: the relevant packets include not, only those with a header in (B,D), but any packet created at group B.

Instead of improving behavioral analysis, we focus on the packet-manipulating network structure itself, that is a forwarding program P collectively driven by a set of policies Σ = {rw₁, rw₂, fw} in Figure 1. While behavioral analysis partitions the packet space of P into the so called equivalent classes (ECs) [9], [10], so as to quickly and thoroughly exercise P’s behavior (semantics) as governed by the policy set Σ, we ask, instead, how does Σ “modify” P structurally? And our goal is to syntactically transform program P, based on Σ, into a set of programs, such that each prescribes the network behavior (packet processing) on a particular EC in a more self-explanatory manner.

III. DATALOG AND THE CLASSIC CHASE

To realize the semantics-based network transformation in §II [1] we present a first attempt with datalog and the classic chase. Datalog has long been accepted as an intuitive specification language for networking: the forwarding behavior along R₁R₂R₃ naturally translates to r in Listing 1, where F(flow, source, destination, location, next – hop) is a predicate expressing that location (a switch interface in the network) forwards packet flow flow with header (source, destination) to the next – hop. To transform the network to incorporate the constraint that the destination of a packet remains unchanged — simply a key dependency k: flow → destination, we only need to chase r with k, “correcting” the body of r — by the substitution y₁/y₂, y₁/y₃, y₁/y₄, y₁/y₅, y₁/y₆, y₁/y — to satisfy k.

```
Listing 1: Example semantics-based network transformation
```

More generally, the classic chase is well-understood for data dependencies in the form of an equality generation dependency (egd) or tuple generation dependency (tgd). An example of egd is the key dependency k given by δ₁ in Listing 2 an example tgd is referential dependency (the presence of certain tuple in a relation implies the presence of another (probably in a different relation)). Both tgd, egd can be written as datalog rules if we allow (inequality. This allows us to apply the chase to a datalog program q by a dependency σ by running σ on the “instantiation” of q (a symbolic database instance D): tgd is just a regular datalog rule h : −b₁, . . . , bₙ, the “evaluation” of which proceeds on D by adding the new atom h to D (program q); for an egd e : −b₁, . . . , bₙ, (e is a substitution y/y’ corresponding to an equality atom y = y’ in δ₁), the evaluation is similar to egd except that instead of adding new goals to the rule, systematically applying the substitution.

```
Listing 2: Limitation of the classic chase
```

Unfortunately, the classic chase is too restricted for networking. The chase is hard to process even for a simple firewall policy for packets along R₁R₂R₃ in Figure 1 δ₃ in Listing 2 specifies a firewall that allows packets to pass R₃ only when its source is not 1.2.3.4, by involving the inequality with 1.2.3.4. δ₃ describes a firewall policy filtering any packets with a source from group B by using an auxiliary predicate B (not a database relation). We observe that the difficulty in chasing with policies given by such (general) datalog rules is that, these general constructs are not “evaluatable” on a symbolic database, because a symbolic database contains tuples with unknown values. For example, consider chasing r with δ₂, we have F(f(x₁, y₁, y₁,3, 2), F(f(x₃, y₃, 3, 4) in the symbolic database D, but we cannot determine whether x₃ ≠ 1.2.3.4 holds or not, because x₃ in D is a “symbolic” constant. Unlike a usual constant whose value we know, it is instantiated from a variable, with an uncertain value! For the same reason, when chasing with δ₃, we cannot decide the auxiliary predicate B(x₃).

IV. EXTENDING THE CHASE TO Fauré-LOG

Our goal is to develop a chase-like process to transform network behavior: given a network expressed in a datalog program p, we seek a network policy expression Σ, and a rewrite (chasing) process →Σ (or abbreviated as → when Σ is clear), such that chasing p with Σ produces p’ (written as p →Σ p’), where p’ is a new program that properly incorporates the intention of Σ; the intention of Σ should be self-evident.
Listing 3: $r \rightarrow \{r_1, r_2, r_3\}$: the “combined effect” of the policies is clearly “pronounced” in the transformed program $\{r_1, r_2, r_3\}$.

For example, the network in Figure 1 (the forwarding behavior) is given by a 4-rule program (along 4 paths, namely $R_1 R_2 B_3, R_1 B_2 R_3, R_2 B_3, R_2 B_3$), each of which computes reachability along a specific path. Specifically, The rule $r$ in Listing 3 specifies the network behavior along $R_2 B_3$. We seek an expression for the three policies, such that chasing the program would embed those policies. Listing 3 illustrates the transformation result of $r$ into three new rules, corresponding to the three equivalent classes determined by the policies. To achieve such network transformation, the rest of the section presents a design of $\Sigma$ and a chase-like $\rightarrow$.

A. Extending the chase to symbolic network

Chasing a datalog program $p$ with a dependency $\delta$ reduces to evaluating $\delta$ on the data instance $D$ obtained from $p$, the challenge is that $D$ is incomplete in the sense that the constant symbols instantiated from variables of $p$ are not real constants, their values are unknown. To address this mismatch, we leverage incomplete databases research [8, 11-13]. Based on our prior work $fau\textordmasculine-log$, a datalog extension for partial network information, we develop a dependency expression called $fau\textordmasculine$-dependency for networking policies, and extend the chase to $fau\textordmasculine$-dependency.

Specifically, we represent the symbolic instance during the chasing, which we call the symbolic network, by conditional tables (c-tables). C-tables allow both constants and variable symbols, while the constant has a “face value”, the variable symbols denote unknown/uncertain values that are constrained by additional conditions (e.g., $\neg (x = 1.234.5)$ denotes an unknown value other than 1.234.5). Evaluation over such symbolic network is thus handled by $fau\textordmasculine-log$ evaluation [8]: in a $fau\textordmasculine-log$ program, the symbols include the usual constants and variables, and a new type of symbols called c-variables that are uncertain constants with additional conditions. The variables symbols now range over the domain of constants as well as the c-variables. The $fau\textordmasculine-log$ evaluation enhances standard datalog evaluation by also properly manipulating the c-variable conditions. $fau\textordmasculine$-dependency is just $fau\textordmasculine-log$ rules with two exceptions (1) all the variable symbols are c-variables to capture the “uncertain constants” in a symbolic network, (2) in the head (left of the rule) we allow the chase actions (substitution and tuple generation), as shown in Listing 3. Intuitively, a $fau\textordmasculine-log$ rule derives a symbolic head $H(u)$ if the symbolic database contains $B_1(u_1), \ldots, B_n(u_n)$ under the conditions $\{C(u_1), \ldots, C(u_n)\}$. That is, a $fau\textordmasculine$-dependency expresses $tgj$ and $egd$ conditionally.

Listing 4: Query symbolic network by $fau\textordmasculine-log$, represent policies as dependency

$Fau\textordmasculine$-dependency can easily express all the network policies in Figure 1 as shown in Listing 4. For example, the header rewrite at $R_3$ is given by $\sigma_3, \sigma_4$: $\sigma_3$ says that “irrelevant” packet headers (not matching the rewrite condition, captured by $\neg(B(x_1), C(y_1))$, where $B, C$ are auxiliary predicates asserting group membership) in line 2) will pass $R_3$ (through ingress interface 1 to egress 2) without change, thus we have the substitution in the head; on the other hand, $\sigma_4$ asserts that for packets matching the condition, the source address will be rewritten to a new address $x_2$ in $A$. The header rewrite at $R_3$ can be formulated similarly. The firewall at $R_2$ is given by $\sigma_3, \sigma_4$: $\sigma_3$ describes the network behavior on packet not to be filtered, similar to $\sigma_2$: $\sigma_4$, for packets to be filtered, is interesting, it uses $\perp$ (falsehood), a special predicate (a 0-ary predicate always evaluating to false), in the head, implying a contradiction. Finally, $\sigma_7$ says that the packet header remains the same as long as it is at an interface not configured with a header rewrite or firewall.

Listing 5: Examples network policies (Figure 1) as $fau\textordmasculine-log$-dependencies

To chase with $fau\textordmasculine$-dependencies, we develop a new algorithm [1]. Given a rule $r$, and a $fau\textordmasculine$-dependency $\sigma$, the intuition is, like the classic chase, to correct $r$ — viewed as an symbolic instance — according to the requirement (substitution in $egd$, or the presence of new tuples in $tgj$) of $\sigma$. The main complexity is in handling the conditions: To decide the proper correction on the symbolic network state which are c-tables, we leverage the $fau\textordmasculine$ evaluation engine to perform $q(D)$ (line 3). When the result $H'_{\sigma}(v_2)$ is empty (line 4), the
dependency is not “applicable” (e.g., the “premise” of the dependency is not satisfiable), so the chase halts; Otherwise, we proceed to compute and evaluate the new conditions under a systematic substitution (line 6): if the new condition is UNSAT (line 7), it signals an “impossible” network state, meaning that $\gamma$ and $\sigma$ are incompatible; on the other hand, if the new condition is satisfiable, we apply the corrections by systematic substitutions (line 6) or new predicate insertions ($H'_\sigma$ in line 8).

Algorithm 1: The chase with fauré-dependency

\begin{algorithm}[h]
\caption{The chase with fauré-dependency}
\begin{algorithmic}
\State \textbf{input} : fauré-log rule $r : B_r \rightarrow \phi$,
\textbf{output} : $\sigma \rightarrow r \rightarrow \sigma'$
\State $\phi_\sigma = \phi \{ x/y, \psi_\sigma : B_\sigma[\phi_\sigma] \}$
\State $\sigma \rightarrow \sigma'$
\Function{instance}{$\phi_\sigma$ into c-tables $D$ ;}
\State $H_\sigma' \{ \psi_\sigma \} = Q(D)$ by fauré-log evaluation ;
\If{$H_\sigma' \{ \psi_\sigma \} \text{ is empty}$}
\State $\sigma \rightarrow \sigma'$
\Else
\State $\phi'_\sigma = \phi \{ x/y \}$, $\phi'_\sigma = \phi \{ x/y \}$
\If{$\phi'_\sigma \land \phi_\sigma \land \psi_\sigma$ is UNSAT}
\State $\sigma \rightarrow \sigma'$
\Else
\State let $\phi'_\sigma = H_\sigma' \{ x/y \}$
\State $\sigma \rightarrow \sigma'$
\EndIf
\State $\sigma \rightarrow \sigma'$
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

B. Discussion: chasing fauré-dependencies is Church-Rosser

Our main conjecture is that chasing with fauré-dependencies, despite being complete for a larger class of network dependencies via a more sophisticated procedure (Algorithm 1), remains “Church-Rosser”. Given a set of policy dependencies $\Sigma$ (multiple network policies), if chasing a fauré-log rule $p$ (we chase a multi-rule program by independently chasing each rule) with $\Sigma$ by repeatedly chasing with $\sigma \in \Sigma$ is terminating, the ordering in which the $\sigma$’s are chosen is insignificant. That is, for any terminating sequence of dependencies from $\Sigma$, $p \rightarrow \ldots \rightarrow \sigma_n \rightarrow p_k \rightarrow \ldots \rightarrow p'$, the end result $p'$ is unique, and we write $p \rightarrow_{\Sigma} p'$. This is particularly appealing for reasoning about the joint effects of a set of distributed policies (Figure 1) since the “interaction” between them is insignificant.

We also point out an interesting twist with the new chase: Let a chase sequence of $r$ by $\Sigma$ be $s_1, \ldots, s_k, \ldots$, such that for each $k$, $s_k$ is the result of applying some $\sigma \in \Sigma$ to $s_{k-1}$ ($s_k$ is the result of chasing $s_{k-1}$ with $\sigma$). The sequence is terminal if it is finite and no dependency in $\Sigma$ can be further applied to it. In such cases, the chase with $\Sigma$ is terminating and the last element is called its result. With these notions, Church-Rosser for the classic chase is shown in Figure 2(a): all (terminating) chasing sequences converge to a single rule $r'$. The unique end result in the case of fauré-dependencies, however, becomes a set of rules $\gamma (\{ r_1, \ldots, r_n \})$: the chase sequences still converge to the unique $\gamma$, but the individual chase sequences can lead to different elements $r_k \in \gamma$. In particular, each $r_k \in \gamma$ represents the policy-based network behavior for a specific equivalent class.