KATch: A Fast Symbolic Verifier for NetKAT

We develop new data structures and algorithms for checking verification queries in NetKAT, a domain-specific language for specifying the behavior of network data planes. Our results extend the techniques obtained in prior work on symbolic automata and provide a framework for building efficient and scalable verification tools. We present KATch, an implementation of these ideas in Scala, featuring an extended set of NetKAT operators that are useful for expressing network-wide specifications, and a verification engine that constructs a bisimulation or generates a counter-example showing that none exists. We evaluate the performance of our implementation on real-world and synthetic benchmarks, verifying properties such as reachability and slice isolation, typically returning a result in well under a second, which is orders of magnitude faster than previous approaches. Our advancements underscore NetKAT's potential as a practical, declarative language for network specification and verification.

1 INTRODUCTION

In the automata-theoretic approach to verification, programs and specifications are each encoded as automata, and verification tasks are reduced to standard questions in formal language theory—e.g., membership, emptiness, containment, etc. [40]. The approach became popular in the mid-1980s, driven by use of temporal logics and model checking in hardware verification, and it remains a powerful tool today. In particular, automata provide natural models of phenomena like transitive closure, which arises often in programs but is impossible to express in pure first-order logic.

NetKAT, a domain-specific language for specifying and verifying the behavior of network data planes, is a modern success story for the automata-theoretic approach. NetKAT programs denote sets of traces (or "histories") where a trace is the list of packets "seen" at each hop in the network. The NetKAT language specifies these traces using a regular expression-like syntax, as follows:

$$p, q ::= \top \mid \bot \mid f = v \mid f \neq v \mid f \leftarrow v \mid \mathrm{dup} \mid p + q \mid p \cdot q \mid p^\star$$

Unlike ordinary regular expressions, which are stateless, NetKAT programs manipulate state in packets. Accordingly, NetKAT's atoms are are not letters, but *actions* that either copy or drop the current packet ($\top$ or $\bot$), modify a header field ($f \leftarrow v$), test a header field against a value ($f = v$, $f \neq v$), or append the current packet to the trace (dup).

A NetKAT program thus gives a declarative specification of the network's global behavior in terms of sets of traces. Specifically, we can model the location of a packet in the network using a special header field (usually named sw for "switch"), that we can update ($\mathsf{sw} \leftarrow \ell$) to logically move the packet from its current location to a new location ℓ. Given a declarative description of the intended behavior, the NetKAT compiler generates a set of local forwarding tables for individual switches that together realize the global behavior [19, 33].

Moreover, NetKAT not only plays a role as an implementation language, but also as a language for expressing verification queries, analogous to verification tools powered by SMT solvers [3, 29, 39]. In particular, as NetKAT includes a union operator ("+"), containment can be reduced to program equivalence—i.e., $p \sqsubseteq q$ if and only if $p + q \equiv q$. Hence, unlike other contemporary tools, which rely on bespoke encodings and algorithms for checking network properties like reachability, slice isolation, etc. [18, 24, 25, 42], NetKAT allows a wide range of practical verification questions to be answered using the same foundational mechanism, namely program equivalence. For example,

- *Network Reachability:* Are there any packets that can go from location 121 to 543 in the network specified by the NetKAT program net? Formally: $(\mathsf{sw} \leftarrow 121) \cdot \mathsf{net}^\star \cdot (\mathsf{sw} = 543) \stackrel{?}{\equiv} \bot$
- *Slice Isolation:* Are slices net_1 and net_2 logically disjoint, even though their implementation on shared infrastructure uses the same devices? Formally: $\mathsf{net}_1^\star + \mathsf{net}_2^\star \stackrel{?}{\equiv} (\mathsf{net}_1 + \mathsf{net}_2)^\star$

NetKAT's dup primitive is a key construct that enables reducing a wide range of network-specific properties to program equivalence. Recall that dup appends the headers of the current packet to the trace. Hence, to verify properties involving the network's internal behavior, we add intermediate packets to the trace using dup, making them relevant for program equivalence. Conversely, if we only care about the input-output behavior of the entire network, we can omit dup from the specification, so intermediate packets are not considered by the check for program equivalence.

The story so far is appealing, but there is a major fly in the ointment. To decide program equivalence, the automata-theoretic approach relies on the translation from NetKAT programs to automata. But standard NetKAT automata have an enormous space of potential transitions—i.e., the "alphabet" has a "character" for every possible packet. So using textbook algorithms for building automata and checking equivalence would clearly be impractical. Instead, what's needed are symbolic techniques for encoding the state space and transition structure of NetKAT automata that avoid combinatorial blowup in the common case.

Prior work on symbolic automata provides a potential solution to the problems that arise when working with large (or even infinite) alphabets. The idea is to describe the transitions in the automata using logical formulas [15, 16] or Binary Decision Diagrams (BDDs) [32] rather than concrete characters. Algorithms such as membership, containment, and equivalence can then be reformulated to work with these symbolic representations. Symbolic automata have been successfully applied to practical problems such as building input sanitizers for Unicode, which has tens of thousands of characters [22]. However, NetKAT's richer semantics precludes the direct adoption of standard notions of symbolic automata. In particular, NetKAT's transitions describe not only predicates but also transformations on the current packet. As Pous writes, it "seems feasible" to generalize his work to NetKAT, but "not straightforward" [32].

In this paper, we close this gap and develop symbolic techniques for checking program equivalence for NetKAT. In doing so, we address three key challenges:

Challenge 1: Expressive but Compact Symbolic Representations. The state space and transitions of NetKAT automata are very large due to the way the "alphabet" is built from the space of all possible packets. Orthogonally, these characters also encode packet transformations, as reflected in NetKAT's packet-processing semantics. It is crucial that the symbolic representations for NetKAT programs be compact and admit efficient equivalence checks.

Challenge 2: Extended Logical Operators. In the tradition of regular expressions, NetKAT only includes sequential composition ($e_1 \cdot e_2$) and union operators ($e_1 + e_2$)—the latter computes the union of the sets of traces described by e_1 and e_2. However, when verifying network-wide properties it is often useful to have operators for intersection ($e_1 \cap e_2$), difference ($e_1 \setminus e_2$), and symmetric difference ($e_1 \oplus e_2$). Having native support for these operators at the level of syntax and in equivalence checking algorithms, allows the reduction of all verification queries to an emptiness check—e.g., $A \equiv B$ reduces to $A \oplus B \equiv \bot$, and $A \sqsubseteq B$ reduces to $A \setminus B \equiv \bot$.

Challenge 3: Support for Counter-Examples. When an equivalence check fails, it can be hard to understand the cause of the failure, and which changes might be needed to resolve the problem. It is therefore important to be able to construct *symbolic counter-examples* that precisely capture the input packets and traces that cause equivalence to fail.

In addressing the above challenges, we make the following technical contributions:

Contribution 1: Efficient symbolic representations *(Section 3).* We design a symbolic data structure called Symbolic Packet Programs (SPPs) for representing both sets of packets and transformations on packets in a symbolic manner. SPPs generalize classic BDDs and are asymptotically

more efficient than the representations used in prior work on NetKAT. In particular, SPPs support efficient sequential composition ($e_1 \cdot e_2$) and can be made *canonical* which, with hash consing, allows us to check equivalence of SPPs in constant time.

Contribution 2: Brzozowski derivatives *(Section 4).* Previous approaches to translating from NetKAT programs to automata were based on the non-deterministic Antimirov derivative, which naturally supports "positive" operators like union, but not "negative" operators like intersection and difference. We develop a deterministic version of the derivatives for NetKAT, analogous to the Brzozowski derivatives for regular expressions, which allows us to support negative operators, which are often useful when verifying network-wide properties.

Contribution 3: Symbolic bisimilarity checking *(Section 5).* Building on the foundation provided by SPPs and deterministic NetKAT automata, we develop symbolic algorithms for checking bisimilarity. We present a symbolic version of the standard algorithm that searches in the forward direction through the state space of the automata under consideration. We also present a novel backward algorithm that computes symbolic counter-examples to equivalence—i.e., the precise set of input packets for which equivalence fails.

Contribution 4: KATch implementation *(Section 6)* ***and evaluation*** *(Section 7).* Finally, we present a new verification tool, called KATch, implemented in Scala. KATch implements symbolic bisimilarity checking, including an extended set of extended logical operators, as well as symbolic counter-examples. We evaluate the performance of KATch on a variety of real-world topologies, and show that it efficiently answers a variety of verification queries and scales to much larger networks than prior work. Due to its use of our efficient data structures, KATch is several orders of magnitude faster than prior implementations of NetKAT on these realistic examples *even when its symbolic nature is not fully exploited.* Moreover, on combinatorial benchmarks, which require symbolic representations to be tractable, KATch is faster than prior work by arbitrary large factors.

2 NETKAT EXPRESSIONS AND AUTOMATA

This section reviews basic definitions for NetKAT, to set the stage for the new contributions presented in the subsequent sections. NetKAT is a language for specifying the packet-forwarding behavior of network data planes [1, 21, 33].[1] The syntax and semantics of NetKAT is presented in Figure 1, and is based on Kozen's Kleene Algebra with Tests (KAT) [27].

To a first approximation, NetKAT can be thought of a simple, imperative language that operates over packets, where a *packet* is a finite record assigning values to fields. The basic primitives in NetKAT are packet tests ($f = v$, $f \neq v$) and packet modifications ($f \leftarrow v$). Program expressions are then compositionally built from tests and packet modifications, using union ($+$), sequencing ($\cdot$), and iteration ($\star$). Conditionals and loops can be encoded in the standard way:

$$\textbf{if } b \textbf{ then } p \textbf{ else } q \triangleq b \cdot p + \neg b \cdot q \qquad \textbf{while } b \textbf{ do } p \triangleq (b \cdot p)^\star \cdot \neg b.$$

In a network, conditionals can be used to model the behavior of the forwarding tables on individual switches while iteration can be used to model the iterated processing performed by the network as a whole; the original paper on NetKAT provides further details [1]. Note that assignments and tests in NetKAT are always against constant values. This is a key restriction that makes equivalence decidable. It also aligns with the capabilities of data plane hardware. The dup primitive makes a copy of the current packet and appends it to the trace, which only every grows as the packet goes

[1]For readers unfamiliar with networking terminology, note that networks have a control plane, which computes paths through the topology using distributed routing protocols (or a software-defined networking controller), and a data plane, which implements paths using high-speed hardware and software pipelines. NetKAT models the behavior of the latter.

Syntax		Description	Semantics
$p, q ::=$		Programs	$[\![p]\!] : (\alpha : \mathsf{Pk}) \to \mathcal{P}(\mathsf{Pk}^\star)$
	$\top$	*True*	$\{\alpha\}$
	$\bot$	*False*	$\emptyset$
	$f = v$	*Test equals*	$\{\alpha\}$ if $\alpha_f = v$, otherwise $\emptyset$
	$f \neq v$	*Test not equals*	$\{\alpha\}$ if $\alpha_f \neq v$, otherwise $\emptyset$
	$f \leftarrow v$	*Modification*	$\{\alpha[f \leftarrow v]\}$
	dup	*Duplication*	$\{\alpha\alpha\}$
	$p + q$	*Union*	$[\![p]\!](\alpha) \cup [\![q]\!](\alpha)$
	$p \cdot q$	*Sequencing*	$\{\mathbf{ab} \mid \mathbf{a} \in [\![p]\!](\alpha),\ \mathbf{b} \in [\![q]\!](\mathbf{a}_{\lvert a \rvert})\}$
	$p^\star$	*Iteration*	$\bigcup_{n \geq 0} [\![p^n]\!]$

Values $v ::= 0 \mid 1 \mid \ldots \mid n$ Fields $f ::= f_1 \mid \ldots \mid f_k$ Packets $\alpha ::= \{f_1 = v_1, \ldots, f_k = v_k\}$

Fig. 1. NetKAT syntax and semantics.

through the network. This primitive is crucial for expressing network-wide properties, as it allows the semantics to "see" the intermediate packets processed on internal switches.

NetKAT's formal semantics, given in Figure 1, defines the action of a program on an input packet α, producing a set of output traces. The semantics for $\top$ gives a single trace containing the single input packet α, whereas $\bot$ produces no traces. Tests ($f = v$ and $f \neq v$) produce a singleton trace or no traces, depending on whether the test succeeds or fails. Modifications ($f \leftarrow v$) produce a singleton trace with the modified packet. Duplication (dup) produces a single trace with two copies of the input packet. Union ($p + q$) produces the union of the traces produced by p and q. Sequential composition ($p \cdot q$) produces the concatenation of the traces produced by p and q, where the last packet in output traces of p is used as the input to q. Finally, iteration ($p^\star$) produces the union of the traces produced by p iterated zero or more times.

Consider the following example:

$$(x = 0 \cdot x \leftarrow 1 \cdot \text{dup} \ + \ x = 1 \cdot x \leftarrow 0 \cdot \text{dup})^\star$$

This program repeatedly flips the value of x between 0 and 1, and traces the packet at each step. On input packet $x = 0$, it produces the following traces:

$$\{[x = 0], [x = 0, x = 1], [x = 0, x = 1, x = 0], [x = 0, x = 1, x = 0, x = 1], \ldots\}$$

Consider what happens if we change the program to:

$$(x = 0 \cdot x \leftarrow 1 \cdot \text{dup} \ + \ x \leftarrow 0 \cdot \text{dup})^\star$$

Unlike the previous example, where the tests are disjoint, this program can generate multiple outputs for a given input, because the right branch of the union can always be taken. Therefore, traces with sequences of $x = 0, x = 0, \ldots$ are also possible. On the other hand, if the $x \leftarrow 1$ assignment is performed, then the left branch cannot be taken the next time, because the test $x = 0$ will fail. Therefore, this program produces traces with sequences of $x = 0$, with singular $x = 1$ packets in between.

As these simple examples show, despite the fact that assignments and tests in NetKAT programs are always against constant values (which aligns with the capabilities of data-plane hardware, and

makes equivalence decidable), the behavior of NetKAT programs can be complicated, particularly when conditionals, iteration, and multiple packet fields are involved.

2.1 NetKAT Automata

NetKAT's semantics induces an equivalence $(p \equiv q) \triangleq (\llbracket p \rrbracket = \llbracket q \rrbracket)$ on syntactic programs. This equivalence is decidable, and can be computed by converting programs to automata and checking for automata equivalence. To convert NetKAT programs to automata, the standard approach is to use Antimirov derivatives. We do not give the details here, but the interested reader can find them in [21], and for our symbolic automata, in Section 4. We continue with a brief overview of NetKAT automata and how to check their equivalence. A NetKAT automaton $(S, s_0, \langle \epsilon, \delta \rangle)$ consists of a set of states S, a start state s_0, and a pair of functions that depend on an input packet:

$$\epsilon : S \times \mathsf{Pk} \to 2^{\mathsf{Pk}} \qquad \delta : S \times \mathsf{Pk} \to S^{\mathsf{Pk}}$$

Intuitively, the observation function ϵ is analogous to the notion of a final state, and models the output packets produced from an input packet at a given state. The transition function δ models the state transitions that can occur when processing an input packet at a given state. Because transitions can modify the input packet, the transition structure $\delta(s, \alpha)$ is itself a function of the modified packet, and tells us to transition to state $\delta(s, \alpha)(\alpha')$ when input packet α is modified to α'.

In terms of the semantics, a transition in the automaton corresponds to executing to the next dup in a NetKAT program, and appends the current packet to the trace. It should also be noted that states in the automaton do not necessarily correspond to network devices; because the location of the packet is modeled as just a field in the packet, there can be states in the automaton that handle packets from multiple devices, and the packets of a device can be handled by multiple states.

Carry-on packets. Unlike traditional regular expressions and automata, NetKAT is stateful, and the output packet of a transition is carried on to the next state. This makes NetKAT fundamentally different from traditional regular expressions and automata and results in challenges in the semantics (which is not a straightforward trace or language semantics) and in designing equivalence procedures, as they have to account for the carry-on packet. Technically it would be possible to fold the packet into the state of the automaton and approach the semantics and problem of checking equivalence using more traditional methods. However, because the number of possible packets is exponential in the number of fields and values, this would cause the state space of the automaton to explode. So while it is possible to do so, it is not practical.

2.2 Bisimulation of NetKAT Automata

NetKAT automata can be used to decide $p \equiv q$. Intuitively, while traces may be infinite, transitions only depend on the current packet, which has a finite number of distinct possible values.

Given two automata $(S, s_0, \langle \epsilon, \delta \rangle)$ and $(S', s_0', \langle \epsilon', \delta' \rangle)$, we check their equivalence by considering all possible input packets separately. We run the two automata in parallel, starting from an input packet α at their respective start states. We first check that the immediate outputs $\epsilon(s_0, \alpha)$ and $\epsilon'(s_0', \alpha)$ are the same. If so, we check that the transitions $\delta(s_0, \alpha)$ and $\delta'(s_0', \alpha)$ are equivalent, by recursively checking the equivalence of the states $\delta(s_0, \alpha)(\alpha')$ and $\delta'(s_0', \alpha)(\alpha')$ for all α'.

We can implement this strategy using a work list algorithm. The work list contains triples (s, s', α), where s and s' are states in the two automata, and α is an input packet. Initially, the work list contains (s_0, s_0', α) for all packets α. We then repeatedly take triples (s, s', α) from the work list, and check that $\epsilon(s, \alpha) = \epsilon'(s', \alpha)$ and add $(\delta(s, \alpha)(\alpha'), \delta'(s', \alpha)(\alpha'), \alpha')$ to the work list for all α'. If at any point we find that $\epsilon(s, \alpha) \neq \epsilon'(t, \alpha)$, then the automata are not equivalent. If the work list stabilizes, the automata are equivalent. This algorithm is shown in Figure 2.

Input: A pair of NetKAT automata $(S, s_0, \delta, \epsilon)$, $(S', s_0', \delta', \epsilon')$.
Returns: A boolean indicating whether the automata are equivalent.

$W \leftarrow \{(s_0, s_0', \alpha) \mid \alpha \in \mathsf{Pk}\}$;
while W changes **do**
  for $(s, s', \alpha) \in W$ **do**
    if $\epsilon(s, \alpha) \neq \epsilon'(s', \alpha)$ **then return false**;
    $W \leftarrow W \cup \{(\delta(s, \alpha)(\alpha'), \delta'(s', \alpha)(\alpha'), \alpha') \mid \alpha' \in \mathsf{Pk}\}$
return true;

Fig. 2. Bisimulation algorithm

Of course, the issue with this algorithm is that the space of packets is huge. The rest of this paper, develops techniques for working with symbolic representations of packets and automata, allowing us to represent and manipulate large sets of packets and to efficiently check equivalence of automata without explicitly enumerating the space of packets.

3 SYMBOLIC NETKAT REPRESENTATIONS

The naive bisimulation algorithm given in the preceding section is rather inefficient, as it iterates over all possible packets. This section develops symbolic representations that allow us to transition an entire set of packets in a single iteration, which vastly reduces the number of iterations required to compute a bisimulation in practice.

We will do this in two steps. First, we introduce a representation for symbolic packets, which represent sets of packets, or equivalently, the fragment of NetKAT where all atoms are tests. This representation is essentially a natural n-ary variant of binary decision diagrams (BDDs) [12]. Second, we introduce Symbolic Packet Programs (SPPs) a new representation for symbolic transitions in NetKAT automata, representing the dup-free fragment of NetKAT—i.e., the fragment in which all atoms are tests or assignments. The primary challenge is to design this representation so that it is efficiently closed under the NetKAT operations. Furthermore, we need to be able to efficiently compute the transition of a symbolic packet over a SPP, both forward and backward.

3.1 Symbolic Packets

We begin by choosing a representation for symbolic packets. A symbolic packet $p \subseteq \mathsf{Pk}$ is a set of concrete packets, represented compactly as a decision diagram. Syntactically, symbolic packets are NetKAT expressions with atoms restricted to tests, represented in the following canonical form:

$$p \in \mathsf{SP} ::= \top \mid \bot \mid (f = v_0) \cdot p_0 + \ldots + (f = v_n) \cdot p_n + (f \neq v_0 \cdots f \neq v_n) \cdot p_{n+1}$$

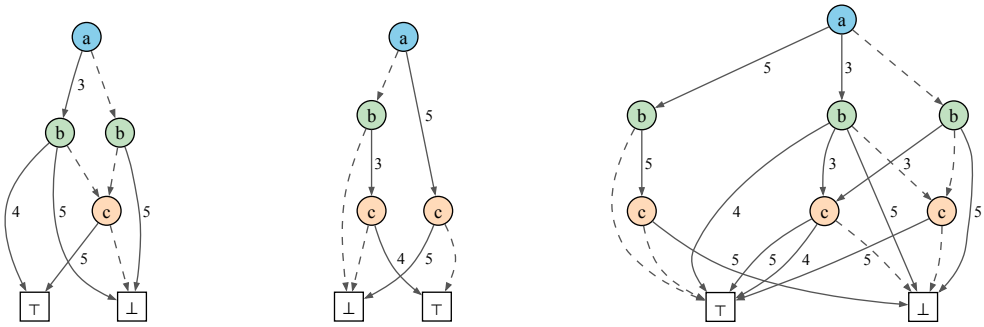
That is, symbolic packets form an n-ary tree, where each child p_i is labeled with a test of the current field f, and the default case p_{n+1} is labeled with the negation of the other tests. This ensures that a given concrete packet has a unique path through the tree.

The following conditions need to be satisfied for a symbolic packet to be in canonical form:

Reduced If a child p_i is equal to the default case p_{n+1}, it is removed. If only the default case remains, the symbolic packet is reduced to the default case itself.

Ordered A path down the tree always follows the same order of fields, and the children p_i are ordered by the value v_i.

This ensures the representation of a symbolic packet is unique, in the sense that two symbolic packets are semantically equal if and only if they are syntactically equal.



(i) $a{=}3 \cdot b{=}4 + b{\neq}5 \cdot c{=}5$ (ii) $a = 3 \cdot c{=}4 + a{=}5 \cdot c{\neq}5$ (iii) Union (i) + (ii)

Fig. 3. Symbolic packets

Representation. As with BDDs, we share nodes in the tree, making the representation of a symbolic packet a directed acyclic graph, as shown in Figure 3. Vertices are labeled with the packet field they test. Solid arrows labeled with a number encode the test value, and dashed arrows represent a default case. The sinks of the graph are labeled with $\top$ or $\bot$, indicating membership in the set. On the left, we have a symbolic packet representing all packets where $a = 3$ and $b = 4$, or $b \neq 5$ and $c = 5$. In the middle, we have a symbolic packet representing all packets where $a = 3$ and $c = 4$, or $a = 5$ and $c \neq 5$. On the right, we have the union of these symbolic packets.

Operations. Symbolic packets are closed under the NetKAT operations. The union of symbolic packets is computed by traversing the two trees in parallel, and taking the union of the children, making sure to reduce the resulting tree to maintain the canonical form. Sequential composition, being equivalent to intersection, is computed similarly, as are the other operators. The iteration operator always produces $\top$. We therefore have the following operations:

$$\hat{+}, \hat{\cdot}, \hat{\cap}, \hat{\oplus}, \hat{-} \ : \ \mathrm{SP} \times \mathrm{SP} \to \mathrm{SP} \qquad \hat{\neg}, \hat{\star} \ : \ \mathrm{SP} \to \mathrm{SP} \qquad f{=}v, f{\neq}v \ : \ \mathrm{SP}$$

3.2 Symbolic Transitions

We now introduce a representation for symbolic transitions in NetKAT automata. Whereas symbolic packets represent the fragment of NetKAT where all atoms are tests, symbolic transitions correspond to the larger dup-free fragment, where all atoms are tests or assignments. This introduces additional challenges for a canonical representation, as well as for the operations. For instance, sequential composition is no longer equivalent to intersection, and the star operator is no longer trivial.

$$\begin{aligned} p \in \mathrm{SPP} ::= {} & \top \mid \bot \mid \\ & (f = v_0) \cdot (f \leftarrow w_0^0 \cdot p_0^0 + \ldots + f \leftarrow w_0^{k_0} \cdot p_0^{k_0}) + \\ & \cdots \\ & (f = v_n) \cdot (f \leftarrow w_n^0 \cdot p_n^0 + \ldots + f \leftarrow w_n^{k_n} \cdot p_n^{k_n}) + \\ & (f \neq v_0 \cdots f \neq v_n) \cdot (f \leftarrow w_{n+1}^0 \cdot p_{n+1}^0 + \ldots + f \leftarrow w_{n+1}^{k_{n+1}} \cdot p_{n+1}^{k_{n+1}} + \\ & \qquad f \neq w_{n+1}^0 \cdots f \neq w_{n+1}^{k_{n+1}} \cdot f \leftarrow w_{n+1}^{k_{n+1}+1} \cdot p_{n+1}^{k_{n+1}+1}) \end{aligned}$$

Like SPs, SPPs have two base cases, $\top$ and $\bot$. Also like SPs, SPPs test a field f of the input packet against a series of values $v_0, \ldots, v_n$, with a default case $f \neq v_0 \cdots f \neq v_n$. However, instead of continuing recursively after the test, SPPs non-deterministically assign a value w_i^j to the field f of the output packet, and continue recursively with the corresponding child p_i^j. This way, SPPs can output more than one packet for a given input packet, and can also output packets with different values for the same field. The default case is further split into two cases, one where the field f is non-deterministically assigned a new value (like in the other cases), and an identity case where the field f keeps the same value as the input packet. However, the packet for the latter case is not produced when the input packet's field f had a value that could also be produced by the explicit assignments in the default case. This is indicated by the test $f \neq w_{n+1}^0 \cdots f \neq w_{n+1}^{k_{n+1}}$, and ensures that for a given input packet, a given output packet is produced by a unique path through the SPP.

The following conditions need to be satisfied for a SPP to be in canonical form:

Reduced If a child p_i^j is equal to $\bot$, it is removed (with the exception of the default-identity case, which is always kept). If one of the default-assignment cases for a value w is $\bot$, it is removed, but to keep the behavior equivalent, an additional test for the same value w is added, with an empty sequence of assignments (if a test for the value w was already present, nothing is added). Further, each of the non-default cases is analyzed, and if we determine that it behaves semantically like the default case for that input value, then it is removed. If only the default case remains, the SPP is reduced to the default case itself.

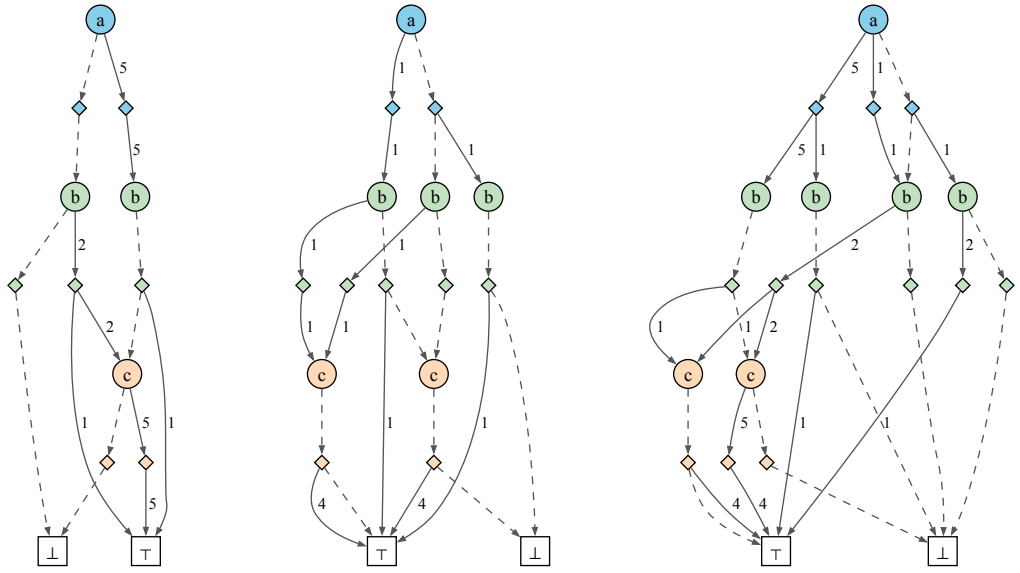
Ordered A path down the tree always follows the same order of fields, and both the tests and the assignments are ordered by the value v_i or w_i^j.

These conditions ensure the representation of a SPP is unique, in the sense that two SPPs are semantically equal if and only if they are syntactically equal.

Representation. Like symbolic packets, we represent SPPs as a directed acyclic graph, with duplicate nodes shared. Examples of SPPsare shown in Figure 4. Vertices are labeled with the packet field that they test. Solid arrows labeled with a number represent the tests, and dashed arrows represent the default case. Each test is followed by a non-deterministic assignment of a new value to the field, indicated by the small diamonds. In the default case, the non-deterministic assignment also has an identity case (keeping the value of the field unchanged), indicated by a dashed arrow emanating from the diamond.

Let us consider the action of the first SPP in Figure 4 on the concrete packet $a=5 \cdot b=3 \cdot c=5$. The first test $a=5$ succeeds, and the field a is assigned a same value 5. The b field is then tested, but the b node only has a default case. The default case does have a non-deterministic assignment, which can set the b field to the new value 1, or it can keep the old value 3. In case the new value of b is 1, the packet is immediately accepted by the $\top$ node, so the packet $a=5 \cdot b=1 \cdot c=5$ is produced. In case the old value of b is kept, the c field is tested, and in our case the value of the c field is 5. In this case, the value of the field is unchanged, and the packet $a=5 \cdot b=3 \cdot c=5$ is produced. In summary, for input packet $a=5 \cdot b=3 \cdot c=5$, the SPP produces the packets $a=5 \cdot b=1 \cdot c=5$ and $a=5 \cdot b=3 \cdot c=5$.

When we sequentially compose the two SPPs on the left, we get the SPP on the right. In other words, if we take a concrete packet, and first apply the first SPP, and then apply the second SPP to all of the resulting packets, then we get the same result as if we apply the SPP on the right directly to the concrete packet. Sequential composition is a relatively complex operation, but algorithmically it is a key strength of our representation. To understand why, consider taking a concrete packet, and applying the first SPP to it. Once we have applied the a-layer of the SPP to the packet, we already know what the value of the a field in the output packet will be. We can therefore immediately



(i) $(a{=}5{+}b{=}2)\cdot(b{\leftarrow}1{+}c{=}5)$ (ii) $b{=}1 + c{\leftarrow}4 + a{\leftarrow}1\cdot b{\leftarrow}1$ (iii) Sequential composition (i) · (ii)

Fig. 4. Symbolic transitions represented as SPPs

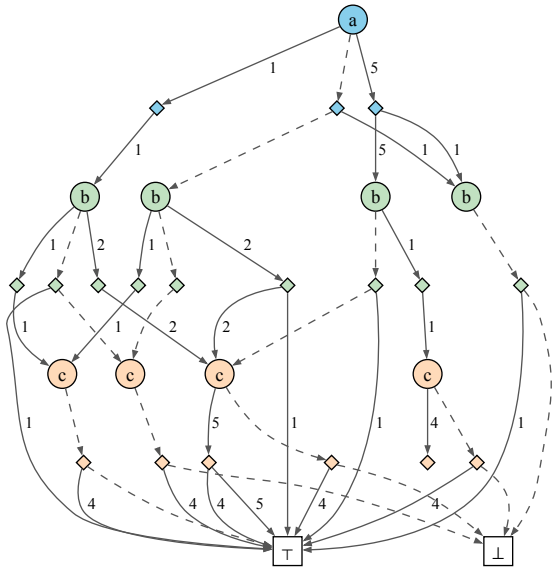
continue with the a layer of the second SPP, without having to consider the other layers of the first SPP. This is in contrast to a naive algorithm, which would have to consider the entire first SPP before being able to apply the second SPP. This property makes it possible to compute the sequential composition of two SPPs efficiently in practice, and is a key contributor to the performance and scalability of our system.

Operations. SPPs are closed not just under sequential composition, but under all of the NetKAT operations. These operations are largely mechanical, but more complex than for SPs, as we need to take assignments into account while respecting the conditions that ensure uniqueness. In particular, sequential composition is no longer equivalent to intersection, as the assignments in the first SPP clearly affect the tests in the second SPP. Furthermore, the star operator is no longer trivial, as the assignments in the SPP can affect the tests in the SPP itself, and a fixed point needs to be taken. We have the following operations on SPPs:

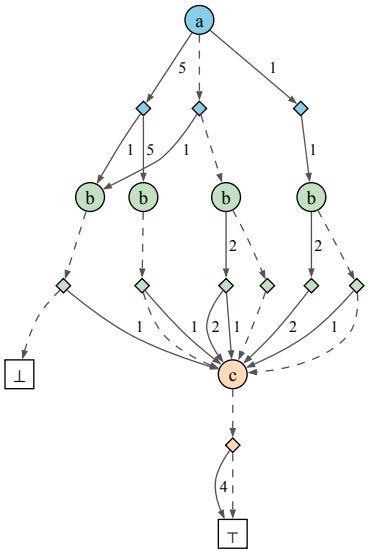
$$\hat{+}, \hat{;}, \hat{\cap}, \hat{\oplus}, \hat{-} \;:\; \mathrm{SPP} \times \mathrm{SPP} \to \mathrm{SPP} \qquad \hat{\star} \;:\; \mathrm{SPP} \to \mathrm{SPP} \qquad f{=}v, f{\neq}v, f{\leftarrow}v \;:\; \mathrm{SPP}$$

The left side of Figure 5 shows the result of applying the symmetric difference to first two SPPs in Figure 4. That is, if we take a concrete packet, and first two SPPs to it, and then take the symmetric difference of the resulting sets of packets, we get the same result as applying the symmetric difference SPP to the concrete packet directly.

The right side of Figure 5 shows the result of applying the star operator to the result of the symmetric difference. That is, if we take a concrete packet, and first apply the symmetric difference SPP to it, and then apply it again to all of the resulting packets, iteratively until we reach a fixed point, we get the same result as if we apply the star SPP to the concrete packet once.



(1) Symmeric difference $(i) \oplus (ii)$



(2) Star $((\text{i}) \oplus (\text{ii}))^\star$

Fig. 5. Symbolic operations (where (i) and (ii) are from Figure 4)

Push and pull. In addition to these operations for combining SPPs, we also have the following operations, which "push" and "pull" a symbolic packet through a SPP:

$$\text{push} : \text{SP} \times \text{SPP} \longrightarrow \text{SP} \qquad \text{pull} : \text{SPP} \times \text{SP} \longrightarrow \text{SP}$$

The push operation computes the effect of a SPP on a symbolic packet, giving a symbolic packet as a result. The new symbolic packet contains all of the packets that are produced by the SPP when applied to the packets in the input symbolic packet.

The pull operation simulates the effect of a SPP in reverse. This operation is used when computing the backward transition of a symbolic packet over a symbolic transition, for counter-example generation. The pull operation answers this question: given a set of output packets (represented symbolically), what are the possible input packets (also represented symbolically) that could have produced them? In other words, a concrete packet is an element of $\text{pull}(p, q)$ if and only if running the SPP p on the concrete packet produces a set of packets that has non-empty intersection with q.

4 SYMBOLIC NETKAT AUTOMATA VIA BRZOZOWSKI DERIVATIVES

With the symbolic packet and symbolic transition representations in place, we can now define symbolic NetKAT automata. The construction of NetKAT automata shares some similarities with the construction of automata for regular expressions. In prior work [21, 33], NetKAT automata were constructed via Antimirov derivatives [2], which can be extended from regular expressions to NetKAT. The Antimirov derivative constructs a non-deterministic automaton, and as such, is well suited for handling NetKAT's union operator by inserting transitions for the two sub-terms. However, the Antimirov derivative is not well suited for our extended set of logical operators, as

we cannot simply insert (non-deterministic) transitions for the operands of intersection, difference, and symmetric difference. Instead, we extend the Brzozowski derivative [13] to NetKAT, which constructs a deterministic automaton directly, and is better suited for the logical operators.

In the rest of this section, we will first describe what symbolic NetKAT automata are, and then describe how to construct them via the Brzozowski derivative.

4.1 Symbolic NetKAT Automata

An automaton for a regular expression consists of a set of states, and transitions labeled with symbols from the alphabet. Additionally, one of the states is designated as the initial state, and a subset of the states are designated as accepting states. This way, an automaton for a regular expression models the set of strings that are accepted by the regular expression.

An automaton for a NetKAT program is similar, but instead of modeling a set of strings, it models the traces that are produced for a given input packet. When a packet travels through the automaton, every state that it traverses acts as a dup operation, appending a copy of the packet to the packet's trace. Therefore, the transition between two states is labeled with a dup-free NetKAT program, represented symbolically as a SPP. Secondly, instead of having a boolean at each state that determines whether it is accepting or not, NetKAT automata have a additional SPP at each state that determines the set of packets that are produced as output of the automaton when a packet reaches that state. Therefore, a symbolic NetKAT automaton consists of the following data:

States A set of states Q.
Initial state A state $q_0 \in Q$.
Transitions A function $\delta : Q \times Q \to \mathsf{SPP}$.
Output A function $\epsilon : Q \to \mathsf{SPP}$.

Deterministic NetKAT automata. The notion of deterministic NetKAT automaton is more subtle than for regular expressions. For regular expressions, a deterministic automaton is one where for every state q and symbol a, there is at most one transition from q labeled with a. For a NetKAT automaton, we need to take into account the fact that the transitions are labeled with SPPs, which may produce multiple packets for a given input packet. Therefore, we define a deterministic NetKAT automaton as one where for every state q and packet α, the set of packets produced by $\mathsf{push}(\alpha, \delta(q, q'))$ are disjoint for all $q' \in Q$. This is equivalent to saying that $\delta(q, q'_1) \mathbin{\hat{\cap}} \delta(q, q'_2) = \bot$ for all $q'_1, q'_2 \in Q$ $(q'_1 \neq q'_2)$. Note that an input packet α may produce multiple different packets at each successor state, but these packet sets at different successor states are disjoint.

4.2 Constructing Automata via Brzozowski Derivatives

We now describe how to construct a symbolic NetKAT automaton for a NetKAT program via the Brzozowski derivative. We take the set of states to be the set of NetKAT expressions, and the initial state to be the NetKAT program itself. Because we want to construct a deterministic automaton, we need a way to represent a non-intersecting outgoing symbolic transition structure (STS). We represent such a transition structure as a NetKAT expression in the following form:

$$r \in \mathsf{STS} ::= p_1 \cdot \mathsf{dup} \cdot q_1 + \ldots + p_n \cdot \mathsf{dup} \cdot q_n$$

where $p_i \in \mathsf{SPP}$ are SPPs, and $q_i \in \mathsf{Exp}$ are NetKAT expressions. Furthermore, we require that the p_i are pairwise disjoint, that is, $p_i \mathbin{\hat{\cap}} p_j = \bot$ for all $i \neq j$.

Operations. The dup operation itself is an STS, by taking $r = \top \cdot \mathsf{dup} \cdot \top$.
We extend the logical operators of extended NetKAT to STSs:

$$\tilde{+}, \tilde{\cap}, \tilde{\oplus}, \tilde{-} \;:\; \mathsf{STS} \times \mathsf{STS} \to \mathsf{STS}$$

These operations need to be defined such that the resulting STS is deterministic. For instance, for $r_1 \cap r_2$, consider the following construction:

$$(p_1 \cdot \mathrm{dup} \cdot q_1 + \ldots + p_n \cdot \mathrm{dup} \cdot q_n) \mathbin{\tilde{\cap}} (p'_1 \cdot \mathrm{dup} \cdot q'_1 + \ldots + p'_n \cdot \mathrm{dup} \cdot q'_n)$$

To bring this in STS form, we distribute the intersection over the union, and combine the terms:

$$(p_1 \mathbin{\hat{\cap}} p'_1) \cdot \mathrm{dup} \cdot (q_1 \cap q'_1) + (p_1 \mathbin{\hat{\cap}} p'_2) \cdot \mathrm{dup} \cdot (q_1 \cap q'_2) + \ldots + (p_n \mathbin{\hat{\cap}} p'_n) \cdot \mathrm{dup} \cdot (q_n \cap q'_n)$$

This is not yet in STS form, as the $q_i \cap q'_i$ terms may not all be different, so we need to collect the terms with the same $q_i \cap q'_i$, and union their SPPs. The other operators need a similar (albeit slightly more complicated) treatment in order to maintain determinism.

Second, we extend sequential composition to STSs, in two forms (denoted with the same symbol). We can compose an STS with a SPP on the left, or with a NetKAT expression on the right:

$$\tilde{\cdot} : \mathsf{SPP} \times \mathsf{STS} \to \mathsf{STS} \qquad \tilde{\cdot} : \mathsf{STS} \times \mathsf{Exp} \to \mathsf{STS}$$

Like the other operators, these need to be defined such that the resulting STS is deterministic.

All of the STS operations are defined in terms of SPP operations. Therefore, an efficient SPP implementation is a key component of our system, not only for the operations on symbolic packets, but also for the operations on STSs that are used to construct the NetKAT automaton.

Brzozowski derivative. With these operations in place, it is straightforward to define the Brzozowski derivative for NetKAT expressions:

$$\begin{aligned}
\epsilon(p + q) &\triangleq \epsilon(p) \mathbin{\hat{+}} \epsilon(q) & \delta(p + q) &\triangleq \delta(p) \mathbin{\tilde{+}} \delta(q) \\
\epsilon(p \cap q) &\triangleq \epsilon(p) \mathbin{\hat{\cap}} \epsilon(q) & \delta(p \cap q) &\triangleq \delta(p) \mathbin{\tilde{\cap}} \delta(q) \\
\epsilon(p \oplus q) &\triangleq \epsilon(p) \mathbin{\hat{\oplus}} \epsilon(q) & \delta(p \oplus q) &\triangleq \delta(p) \mathbin{\tilde{\oplus}} \delta(q) \\
\epsilon(p - q) &\triangleq \epsilon(p) \mathbin{\hat{-}} \epsilon(q) & \delta(p - q) &\triangleq \delta(p) \mathbin{\tilde{-}} \delta(q) \\
\epsilon(p \cdot q) &\triangleq \epsilon(p) \mathbin{\hat{\cdot}} \epsilon(q) & \delta(p \cdot q) &\triangleq \delta(p) \mathbin{\tilde{\cdot}} q \mathbin{\tilde{+}} \epsilon(p) \mathbin{\tilde{\cdot}} \delta(q) \\
\epsilon(p^\star) &\triangleq \epsilon(p)^\star & \delta(p^\star) &\triangleq \delta(p) \mathbin{\tilde{\cdot}} p^\star \\
\epsilon(\mathrm{dup}) &\triangleq \bot & \delta(\mathrm{dup}) &\triangleq \tilde{\mathrm{dup}} \\
\epsilon(f{=}v) &\triangleq f{=}v & \delta(f{=}v) &\triangleq \bot \\
\epsilon(f{\neq}v) &\triangleq f{\neq}v & \delta(f{\neq}v) &\triangleq \bot \\
\epsilon(f{\leftarrow}v) &\triangleq f{\leftarrow}v & \delta(f{\leftarrow}v) &\triangleq \bot \\
\epsilon(\top) &\triangleq \top & \delta(\top) &\triangleq \bot \\
\epsilon(\bot) &\triangleq \bot & \delta(\bot) &\triangleq \bot
\end{aligned}$$

We construct a deterministic symbolic NetKAT automaton for a NetKAT program p as follows:

States $Q \triangleq \mathsf{Exp}$.
Initial state $q_0 \triangleq p$.
Transitions take $\delta(q, q')$ to be the SPP of q' in the Brzozowski derivative of $\delta(q)$.
Output $\epsilon : Q \to \mathsf{SPP}$, defined above.

The Brzozowski derivative is guaranteed to transitively reach only finitely many essentially different NetKAT expressions from a given start state. Therefore, in the actual implementation, we do not use the infinite set Exp for the states, but instead use only the finitely many essentially different NetKAT terms reached from the start state.

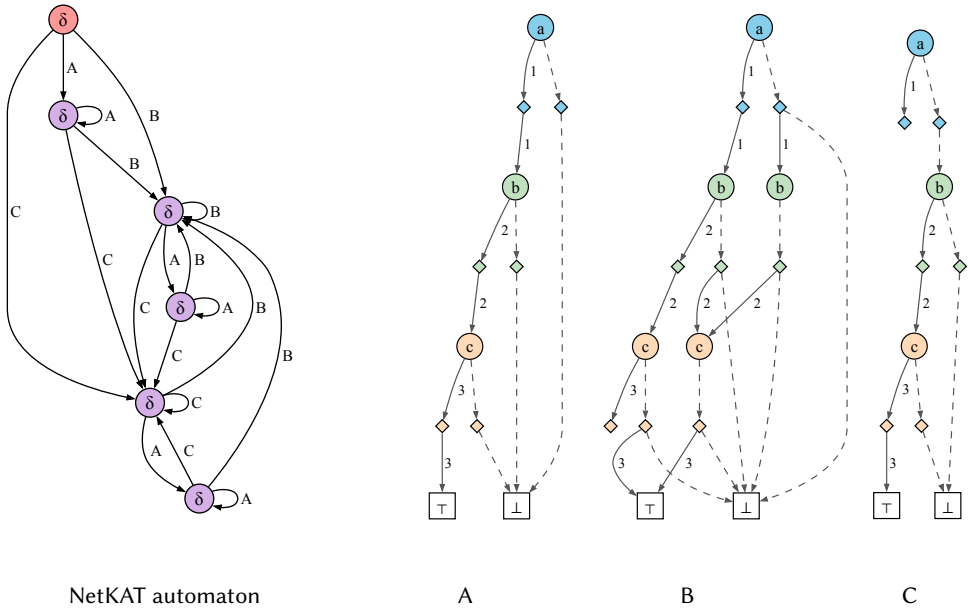


Fig. 6. Symbolic NetKAT automaton for $p \triangleq \left((a \leftarrow 1 \cdot b \leftarrow 2 \cdot c \leftarrow 3 \cdot \mathrm{dup})^\star + (b = 2 \cdot c = 3 \cdot \mathrm{dup})^\star\right)^\star$

Example. Consider the following NetKAT program:

$$p \triangleq \left((a \leftarrow 1 \cdot b \leftarrow 2 \cdot c \leftarrow 3 \cdot \mathrm{dup})^\star + (b = 2 \cdot c = 3 \cdot \mathrm{dup})^\star\right)^\star$$

The NetKAT automaton for this program is shown in Figure 6. The edges are labeled with the SPPs that represent the transitions. The output SPP of every state is $\top$, i.e., the automaton accepts all packets that reach a state.

5 BISIMILARITY AND COUNTER-EXAMPLE GENERATION

Historically, methods for computing bisimulations of automata [8, 17] have been based on either on Moore's algorithm for minimization [31] or Hopcroft and Karp's algorithm [23]. Indeed, the naive approach shown in Figure 2 follows the basic structure of Hopcroft-Karp, in the sense that it relates the start states and proceeds to follow transitions forward in the two automata.

Our situation is different, because we have already added the symmetric difference operator to NetKAT. Therefore, we can reduce equivalence checks $A \equiv B$ to emptiness checks $A \oplus B \equiv \bot$.

Subtlety of symmetric difference. The reader should note that the situation is a bit more subtle than it may seem at first sight. In particular, consider the query $A \equiv B$, which asks whether two NetKAT programs A and B are equivalent. If they are inequivalent, then there must be some input packet that causes A to produce a different output than B. Output in this sense does not just mean the final output packet, but also the trace of the packet. It can be the case that A and B produce the same final output packets, but differ in the trace of the packets. Therefore, the symmetric difference $A \oplus B$ takes the symmetric difference of the traces, not just of the output packets, i.e., the symmetric difference $A \oplus B$ contains precisely the traces of the packets that are produced by A or B, but not

$done(q) \leftarrow \bot$ for $q \in Q$;
$todo(q) \leftarrow \bot$ for $q \in Q \setminus \{q_0\}$;
$todo(q_0) \leftarrow \top$;
while $\exists q.\ todo(q) \neq \bot$ **do**
  set $p := todo(q) \hat{\cdot} done(q)$;
  $todo(q) \leftarrow \bot$;
  $done(q) \hat{+}= p$;
  for $q' \in Q$ **do**
    $todo(q') \hat{+}= \mathsf{push}(p, \delta(q, q'))$;
return $\sum_{q \in Q} \mathsf{push}(done(q), \epsilon(q))$;

(1) Forward algorithm

$done(q) \leftarrow \bot$ for $q \in Q$;
$todo(q) \leftarrow \mathsf{pull}(\epsilon(q), \top)$ for $q \in Q$;

while $\exists q.\ todo(q) \neq \bot$ **do**
  set $p := todo(q) \hat{\cdot} done(q)$;
  $todo(q) \leftarrow \bot$;
  $done(q) \hat{+}= p$;
  for $q' \in Q$ **do**
    $todo(q') \hat{+}= \mathsf{pull}(\delta(q', q), p)$;
return $done(q_0)$;

(2) Backward algorithm

Fig. 7. Forward and backward algorithms for NetKAT automata

by both. In other words, the symmetric difference $A \oplus B$ contains precisely the traces that are counter-examples to the equivalence of A and B.

Subtlety of the emptiness of an automaton. A second subtlety is the emptiness of an automaton. Whereas for regular expressions, it is easy to check whether a DFA is empty (just check whether there are *any* accepting states), this is not the case for NetKAT automata. Because NetKAT automata manipulate and test the fields of packets, it is possible that a packet travels through the automaton, and even causes multiple packets to be produced (e.g., due to the presence of union in the original NatKAT expression), but nevertheless, it is possible that all of these packets eventually dropped by the automaton, before producing any output packets.

The goal of this section is to develop a symbolic algorithm for this check, which is more efficient than the naive algorithm that checks all concrete input packets separately.

5.1 Forward Algorithm

To check whether a NetKAT automaton drops all input packets, we develop a *forward algorithm*. The forward algorithm computes *all* output packets that the automaton can produce, across all possible input packets. The forward algorithm therefore starts with the complete symbolic packet $\top$ at the input state, and repeatedly applies all outgoing transitions δ to it. In this way, the algorithm iteratively accumulates a symbolic packet at every state, which represents the set of packets that can reach that state from the start state. Once we know the set of packets that can reach a state, we can determine the set of output packets by applying the output function ϵ to the symbolic packet of each state, and taking the union of the results.

The forward algorithm is shown in Figure 7. To determine the set of packets that a given NetKAT expression p can produce, we convert it to a NetKAT automaton, and then use the forward algorithm to determine the symbolic output packet. In our implementation, we also have a way to stop the algorithm early, if the user is only interested in a "yes" or "no" answer for the query $p \equiv \bot$. In this case, we can stop the algorithm as soon as any output packet is produced.

5.2 Backward Algorithm

The forward algorithm gives us the set of output packets that a NetKAT automaton can produce, but we are also interested in which input packets cause this output to be produced. For a given check $A \equiv B$, we want to know *which* input packets cause A and B to produce different sets of

Statements c ::=	check e_1 ($\equiv$|$\not\equiv$) e_2	Check equivalence or inequivalence
	print e	Pretty print
	$x = e$	Let
	import "*filename*"	Import NKPL file
	for $i \in n_1..n_2$ do c	For loop
Expressions e ::=	forward e	Compute symbolic packet forward
	backward e	Compute symbolic packet backward
	$e_1 \cap e_2$	Intersection
	$e_1 \oplus e_2$	Symmetric difference (xor)
	$e_1 - e_2$	Set difference
	$f \in n..m$	Range test; sugar for $f = n + \cdots + f = m$
	exists f e	Existential quantifier for symbolic packet e
	forall f e	Universal quantifier for symbolic packet e
	p	NetKAT Program Expression (in Figure 1)

Fig. 8. Syntax for NetKAT Programming Language (NKPL).

output traces. More generally, for a NetKAT automaton, we want to know which input packets cause the automaton to produce a non-empty set of output packets.

To answer this question, we developed a *backward algorithm*, shown in Figure 7. The backward algorithm computes *all* input packets that can cause the automaton to produce a non-empty set of output packets. The backward algorithm therefore starts with the complete symbolic packet $\top$ at every state, and pulls it backwards through all output transitions ϵ. The algorithm then iteratively accumulates a symbolic packet at every state, which represents the set of packets that will cause the automaton to produce a non-empty set of output packets, when starting from that state. The accumulation is done by pulling the symbolic packet backwards through all transitions δ, until fixpoint is reached. Once fixpoint is reached, we simply return the symbolic packet at the start state, which represents the set of input packets that will cause the automaton to produce a non-empty set of output packets when starting from the start state.

6 IMPLEMENTATION

We have implemented the algorithms in a new system, KATch, comprising 2500 lines of Scala. In this section we explain the system's interface, and in the next section we evaluate its performance.

The implementation provides a surface syntax for expressing queries, which extends the core NetKAT syntax from Figure 1. The extended syntax is shown in Figure 8. The language has statements and expressions, which we describe below.

Statements. The check $e_1 \equiv e_2$ command runs the bisimulation algorithm (in the forward direction). The system reports success if the expressions are equivalent (when $\equiv$ is used) or inequivalent (when $\not\equiv$) is used, or reports failure otherwise. The other statements behave as expected: The print statement invokes the pretty printer, let binds names to expressions, import runs the named file making its bound names available, and for run a statement in a loop.

Expressions. The forward e expression computes, in forward-flowing mode, the set of output packets resulting from running the symbolic packet $\top$ through the given expression e (see Figure 7). Conversely backward e computes the set of input packets which generate some output packet when run on the given expression (Figure 7). These are generally used in conjunction with the $\oplus$, $-$, $\cap$ operators to express the desired query. The user may combine these expressions with the

exists and forall operators to reason about symbolic packets, and the print operator to pretty print the symbolic packets, or the check operator to assert (in)equivalence of two expressions.

We note that the generality of the language allows us to express some queries in different, equivalent ways. For example, the two checks:

$$e_1 \equiv e_2 \qquad\qquad e_1 \oplus e_2 \equiv \bot$$

are equivalent. However, the expression on the right lends it self to inspection of counterexample input packets:

$$\mathsf{print}\ (\mathsf{backward}\ e_1 \oplus e_2).$$

This statement pretty prints a symbolic representation of all packets that have different behavior on e_1 and e_2 (i.e., result in some valid history in one expression and not the other). In other words, the command prints the set of all counter-example input packets for the query $e_1 \equiv e_2$.

Topologies and routing tables. While NetKAT can be used to specify routing policies declaratively, it is also possible to import topologies and routing tables into NetKAT. For example, a simple way to define routing tables and topologies in NetKAT is as follows:

$$\begin{array}{c} \mathsf{R} \triangleq \mathsf{sw}{=}5 \cdot (\mathsf{dst}{=}6 \cdot \mathsf{pt}{\leftarrow}3 + \mathsf{dst}{=}8 \cdot \mathsf{pt}{\leftarrow}4) \\ + \cdots + \\ \mathsf{sw}{=}7 \cdot (\mathsf{dst}{=}8 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{dst}{=}9 \cdot \mathsf{pt}{\leftarrow}1) \end{array} \quad \left| \quad \begin{array}{c} \mathsf{T} \triangleq \mathsf{sw}{=}5 \cdot (\mathsf{pt}{=}3 \cdot \mathsf{sw}{\leftarrow}6 + \mathsf{pt}{=}4 \cdot \mathsf{sw}{\leftarrow}8) \\ + \cdots + \\ \mathsf{sw}{=}7 \cdot (\mathsf{pt}{=}2 \cdot \mathsf{sw}{\leftarrow}8 + \mathsf{pt}{=}1 \cdot \mathsf{sw}{\leftarrow}9) \end{array} \right.$$

The routing R tests the switch field (where the packet currently is), and the destination field (where the packet is supposed to end up), and then sets the port over which the packet should be sent out. The topology T tests the switch and port fields, and then transports the packet to the next switch. We define the action of the network by composing the route and topology:

$$\mathsf{net} \triangleq \mathsf{R} \cdot \mathsf{T} \cdot \mathsf{dup}$$

We include a dup to extend the trace of the packet at every hop.

Example 6.1 (All-Pairs Reachability Queries). A naive way to check reachability of all pairs of hosts in a network is to run the following command for each pair of end hosts n_i, n_j:

$$\mathsf{check}\ (\mathsf{sw}{=}n_i) \cdot \mathsf{net}^\star \cdot (\mathsf{sw}{=}n_j) \not\equiv \bot$$

Of course, this requires a number of queries which is quadratic in the number of hosts—quickly becoming prohibitive. One might think that we could reduce the number of queries to n by running:

$$\mathsf{for}\ i \in 1..n\ \mathsf{do}\ \mathsf{check}\ (\mathsf{forward}\ (\mathsf{sw}{=}i \cdot \mathsf{net}^\star)) \equiv (\mathsf{sw} \in 1..n)$$

This query does work if sw is the only field of our packets, because the left hand side contains the packets that can be reached from i via the network, and the right hand side contains packets where the sw field is any value in $1..n$. Unfortunately, this does not quite work if the network also operates on other packet fields, as the packets on the left hand side will have those fields, whereas the packets on the right hand side will only have a sw field. The exists and forall operators allow us to manipulate symbolic packets and give us a way to do all pairs reachability with only n queries:

$$\mathsf{for}\ i \in 1..n\ \mathsf{do}\ \mathsf{check}\ (\mathsf{exists}\ f_1\ (\mathsf{exists}\ f_2\ (\mathsf{forward}\ (\mathsf{sw} = i \cdot \mathsf{net}^\star)))) \equiv (\mathsf{sw} \in 1..n)$$

The operators exists and forall give the programmer the ability to reason about symbolic packets that may be computed, for instance, by forward or backward. The specifications are:

$$\mathsf{exists}\ f\ e = \bigcup_{v \in V} f = v \cdot e \qquad \mathsf{forall}\ f\ e = \bigcap_{v \in V} f = v \cdot e$$

The implementation does not iterate over V (indeed, we need not even know V). Rather, exists and forall are implemented directly as operations on symbolic packets. In fact, the implementation

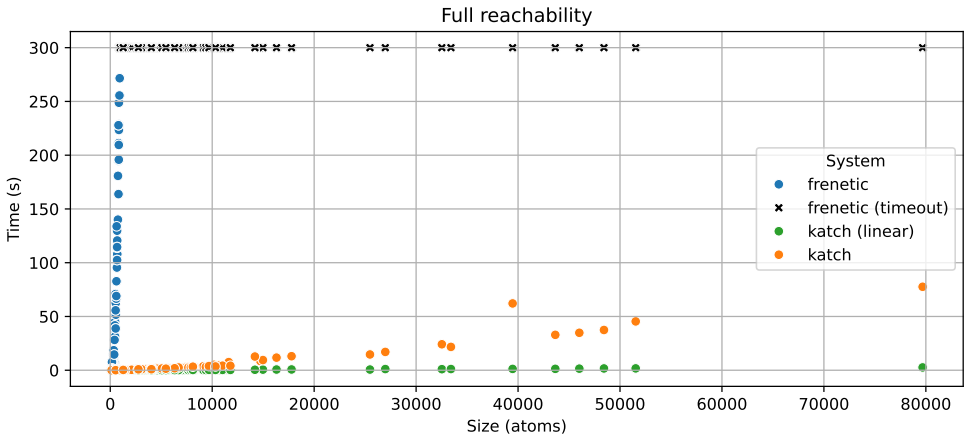


Fig. 9. Full reachability queries on Topology Zoo

does not need to fix the set of fields or the set of values up-front at all, and instead operate on a conceptually infinite set of fields and values. This works because SPPs' default cases handle all remaining fields and all values.

7 EVALUATION

To evaluate KATch, we conducted experiments in which we used it to solve a variety of verification tasks for a range of topologies and routes, as well as challenging combinatorial NetKAT terms. The goal of our evalution is to answer the following three questions:

(1) How does KATch perform compared to the state of the art NetKAT verifier, Frenetic?
(2) How well does KATch scale with the size of topology?
(3) When does KATch perform asymptotically better than prior work?

7.1 Topology Zoo

To begin to answer the first questions, we conducted our experiments using *The Internet Topology Zoo* [26] dataset, a publicly available set of 261 network topologies, ranging in size from just 4 nodes (the original ARPANet with routers at UCLA, UCSB, SRI, and Utah) to 754 nodes (the Kentucky Data Link ISP topology). For each topology, we generated a destination-based routing policy using a simple all-pairs shortest path scheme that connects every pair of routers to each other.

To demonstrate KATch's scalability, we first ran full (i.e., $O(n^2)$) reachability queries for every topology in the zoo using KATch and Frenetic.[2] The results are shown in Figure 10. Note that KATch verifies reachability each topology in less than 2 minutes. Because of the size of the dataset, we set a timeout of 5 minutes per topology. Under these conditions, Frenetic was unable to complete for all but the smallest topologies. KATch handles most of the topologies in well under a second, and all but the largest in under 2 minutes. KATch exceeds the timeout on Kentucky Data Link—using a quadratic number of queries to check full reachability produces over 500k individual queries in a network with 754 nodes!—but is able to finish it in under 2 hours.

To avoid combinatorial blowup in the verification query itself, we also used KATch's high-level verification interface, to check full reachability using a *linear* number of queries, as discussed in

[2]https://github.com/frenetic-lang/frenetic

Name	Size (atoms)	Reachability KATch	Reachability Frenetic	Unreachability KATch	Unreachability Frenetic	Slicing KATch	Slicing Frenetic	Min Speedup
Layer42	135	0.00	0.04	0.00	0.04	0.01	0.07	7×
Compuserv	539	0.01	0.36	0.01	0.38	0.01	0.85	36×
Airtel	785	0.01	0.83	0.01	0.84	0.02	2.08	83×
Belnet	1388	0.01	3.17	0.01	3.16	0.04	7.99	200×
Shentel	1865	0.02	4.01	0.02	4.00	0.04	9.80	200×
Arpa	1964	0.01	4.32	0.02	4.32	0.05	10.99	216×
Sanet	4100	0.04	23.46	0.03	25.23	0.12	62.70	522×
Uunet	5456	0.04	81.54	0.04	81.92	0.15	204.85	1366×
Missouri	9680	0.11	161.28	0.10	165.85	0.27	519.46	1658×
Telcove	10720	0.09	464.15	0.08	465.27	0.28	1274.24	4551×
Deltacom	27092	0.31	2392.56	0.30	2523.03	0.75	7069.54	7718×
Cogentco	79682	0.97	22581.39	0.88	23300.87	1.78	-	23280×

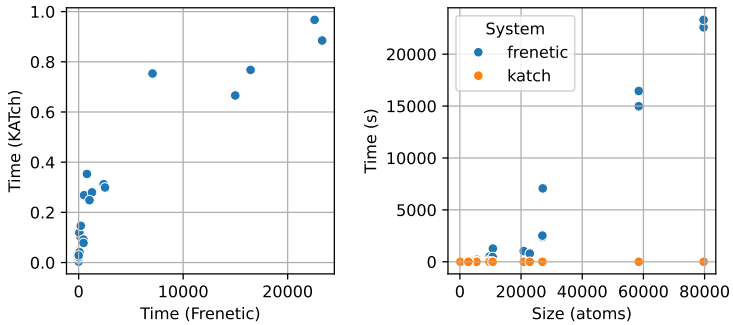


Fig. 10. Results of running KATch and Frenetic on Topology Zoo topologies.

Example 6.1. For Kentucky Data Link, which has 1.1M atoms to encode the network model for 754 nodes, KATch takes 14 seconds for a single query, and under 2 minutes for all queries. Frenetic was still not able to complete even a single reachability query. Indeed, the original NetKAT decision procedure paper excluded this topology from its evaluation as it was too large to handle [21].

Next, for a more fine-grained analysis with a larger (10 hour) timeout, we also randomly sampled a subset of toplogies from the Topology Zoo of varying size and generated reachability, unreachability, and slicing queries to be checked against the routing configurations. For each query, we ran KATch and also generated an equivalent query in the syntax of Frenetic, and ran Frenetic's bisimulation verifier on those queries. We present a charts and a full table of results of these experiments in Figure 10. Note that data for slicing on Cogentco is not shown for Frenetic as it did not terminate in 10 hours. More generally, the table shows that KATch's relative speedup over Frenetic is considerable, and increases as the size of the problem grows. This is encouraging, because it shows that KATch is more scalable.

7.2 Combinatorial Examples

Finally, we ran experiments to test the hypothesis that SPPs have an asymptotic advantage for certain types of queries. The key advantage of SPPs over FDDs is that keeping the updates associated with the fields themselves in the internal node allows avoids a combinatorial blowup associated with sequencing. To test this, we generated the following NetKAT programs:

Inc: Treating the input packet's n boolean fields as a binary number, increment it by one.
Flip: Sequentially flip the value of each of the n boolean fields.

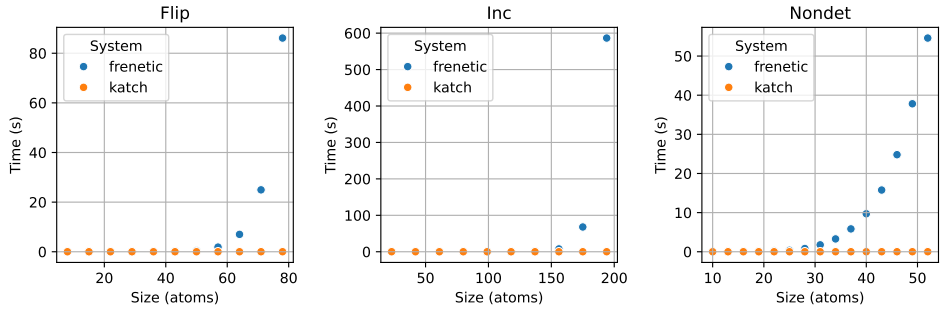


Fig. 11. Results of running KATch and Frenetic on combinatorial benchmarks

Nondet: Set each field of the packet to a range of values from 0 to n.

For Inc, we tested that repeatedly incrementing (using the $\star$ operator) can turn packet $00\cdots0$ into $11\cdots1$. For Flip, we tested that flipping all bits twice returns the original packet. For Nondet, we tested that setting the fields non-deteministically twice is the same as doing it once. The results of this experiment are shown in Figure 11. Because the available fields are hardcoded in Frenetic, we only ran the Inc and Flip experiments up to $n = 10$. We ran the non-determinism test up to $n = 15$. KATch can finish all three queries up to $n = 100$ in less than one minute, demonstrating its aysmptotic advantage for these types of queries.

8 RELATED WORK

This section discusses the most closely related prior work to this paper, focusing on three areas: network verification, NetKAT, and automata-theoretic approaches to verification.

Network Verification. Early work by Xie et al. [41] proposed a unifying mathematical model for Internet routers and developed algorithms for analyzing network-wide reachability. Although the paper did not discuss an implementation, its elegant formal model has been extremely influential in the community and has served as the foundation for many follow-on efforts, including this work. The emergence of software-defined networking (SDN) led to a surge of interest in static data plane verification, including systems such as Header Space Analysis (HSA) [24], Anteater [30], VeriFlow [25], Atomic Predicates (AP) [42]. These systems all follow a common approach: they build a model of the network-wide forwarding behavior and then check whether given properties hold. However, they vary in the data structures and algorithms used to represent and analyze the network model. For instance, Anteater relies on first-order logic and SAT solvers, while VeriFlow uses prefix trees and custom graph-based algorithms. HSA and AP are arguably the most related to our work as they use symbolic representations and BDDs respectively. However, they still rely on ad hoc algorithms for network-wide analysis, which differs from NetKAT's more principled approach based on automata-theoretic foundations.

Another line of work has explored how to lift verification from the data plane to the control plane. Batfish [10, 18] proposed using symbolic simulation to analyze distributed control planes—i.e., generating all possible data planes that might be produced starting from a given control-plane configuration. Like KATch, Batfish uses a BDD-based representation for data plane analysis. MineSweeper [4] improves on Batfish using an SMT encoding of the converged states of the control plane that avoids having to explicitly simulate the underlying routing protocols. Recent work has focused on using techniques like modular reasoning [37, 38] abstract interpretation [6], and a form of symmetry reduction [5] to further improve the scalability of control-plane verification.

NetKAT. NetKAT was originally proposed as a semantic foundation for SDN data planes [1]. Indeed, being based on KAT [27], the language provides a sound and complete algebraic reasoning system. Later work on NetKAT developed an automata-theoretic (or coalgebraic) account of the language [21], including a decision procedure based on bisimulation. However, the performance of this approach turns out to be poor, as shown in our experiments, due to the use of ad hoc data structures ("bases") to encode packets and automata. NetKAT's compiler uses a variant of BDDs, called Forwarding Decision Diagrams (FDDs), as well as an algorithm for converting programs to automata using Antimirov derivatives [33]. The SPPs proposed in this paper improve on FDDs by ensuring uniqueness and supporting efficient sequential composition in the common case. In addition, the deterministic automata used in KATch support additional "negative" operators that are useful for verification. Other papers based on NetKAT have explored use of the language in other settings such as distributed control planes [7] and probabilistic networks [20, 35, 36]. In the future, it would be interesting to consider extending the techniques developed in this paper to these richer settings. Another interesting direction for future work is to build a symbolic verifier for the guarded fragment of NetKAT [34].

Automata-Theoretic Approach and Symbolic Automata. Our work on KATch builds on the large body of work on the automata-theoretic approach to verification and symbolic automata. The automata-theoretic approach was pioneered in the 1980s, with applications of temporal logics and model checking to hardware verification. BDDs, originally proposed by Lee [28], were further developed by Bryant [11], and used by McMillan for symbolic model checking [14].

An influential line of work by D'Antoni and Veanus developed techniques for representing and transforming finite automata where the transitions are not labeled with individual characters but with elements of a so-called effective Boolean algebra [15, 16]. Shifting from a concrete to a symbolic representation requires generalizing classical algorithms, such as minimization and equivalence, but facilitates building automata and transducers that work over enormous alphabets, such as Unicode.

Pous [32] developed symbolic techniques for checking the equivalence of automata where transitions are specified using BDDs. The methods developed in this paper take Pous's work as a starting point but, as highlighted in his paper, develop a non-trivial extension to NetKAT. In particular, SPPs provide a compact representation for "carry-on" packets, which is a critical and unique aspect of of NetKAT's semantics. Bonchi and Pous [8] explored the use of up-to techniques for checking equivalence of automata. In principle, one can view the simplifications enforced by KATch's term representations as a kind of up-to technique, but a full investigation of this idea requires additional work.

Our backward algorithm for computing bisimulations can be seen of a variant of Moore's classic algorithm for computing the greatest bisimulation to NetKAT automata. Doenges et al. [17] proposed an analogous approach for checking equivalence of automata that model the behavior of P4 packet parsers [9]. However, Leapfrog's model is simpler than NetKAT's, being based on classic finite automata, and it achieves scalability primarily due to a novel up-to technique that "leaps" over internal buffering transitions rather than symbolic representations.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* https://doi.org/10.1145/2535838.2535862

[2] Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science* (1996). https://doi.org/10.1016/0304-3975(95)00182-4

[3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. https://doi.org/10.1007/11804192_17

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. https://doi.org/10.1145/3098822.3098834

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. https://doi.org/10.1145/3230543.3230583

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract interpretation of distributed network control planes. POPL (2020). https://doi.org/10.1145/3371110

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*. https://doi.org/10.1145/2934872.2934909

[8] Filippo Bonchi and Damien Pous. 2013. Checking NFA Equivalence with Bisimulations up to Congruence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2429069.2429124

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* (2014). https://doi.org/10.1145/2656877.2656890

[10] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd D. Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*. https://doi.org/10.1145/3603269.3604866

[11] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* (1986). https://doi.org/10.1109/TC.1986.1676819

[12] Randal E. Bryant. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.* (1992). https://doi.org/10.1145/136035.136043

[13] Janusz A Brzozowski. 1962. Canonical regular expressions and minimal state graphs for definite events. In *Proc. Symposium of Mathematical Theory of Automata*.

[14] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1990. Symbolic Model Checking: 10^20 States and Beyond. In *LICS*. https://doi.org/10.1109/LICS.1990.113767

[15] Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2535838.2535849

[16] Loris D'Antoni and Margus Veanes. 2017. Forward Bisimulations for Nondeterministic Symbolic Finite Automata. In *Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205*. https://doi.org/10.1007/978-3-662-54577-5_30

[17] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: Certified Equivalence for Protocol Parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3519939.3523715

[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[19] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *ICFP*. https://doi.org/10.1145/2034773.2034812

[20] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *ESOP*.

[21] Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2676726.2677011

[22] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Conference on Security*.

[23] John E. Hopcroft and Richard M. Karp. 1971. A Linear Algorithm for Testing Equivalence of Finite Automata.

[24] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*.

[25] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. Veriflow: Verifying Network-Wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.* (2012). https://doi.org/10.1145/2377677.2377766

[26] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on* (2011). https://doi.org/10.1109/JSAC.2011.111002

[27] Dexter Kozen. 1996. Kleene algebra with tests and commutativity conditions. In *Tools and Algorithms for the Construction and Analysis of Systems*.

[28] C. Y. Lee. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal* (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x

[29] K. Rustan M. Leino and Valentin Wüstholz. 2014. The Dafny Integrated Development Environment. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.* https://doi.org/10.4204/EPTCS.149.2

[30] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.* (2011). https://doi.org/10.1145/2043164.2018470

[31] Edward F. Moore. 1956. *Gedanken-Experiments on Sequential Machines*.

[32] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. *SIGPLAN Not.* (2015). https://doi.org/10.1145/2775051.2677007

[33] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. https://doi.org/10.1145/2784731.2784761

[34] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. 2019. Guarded Kleene Algebra with Tests: Verification of Uninterpreted Programs in Nearly Linear Time. *Proc. ACM Program. Lang.* 4, POPL, Article 61 (dec 2019), 28 pages. https://doi.org/10.1145/3371129

[35] Steffen Smolka, Praveen Kumar, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *PLDI*.

[36] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. 2017. Cantor Meets Scott: Semantic Foundations for Probabilistic Networks. In *POPL*.

[37] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd D. Millstein, and George Varghese. 2023. Lightyear: Using Modularity to Scale BGP Control Plane Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*. https://doi.org/10.1145/3603269.3604842

[38] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular Control Plane Verification via Temporal Invariants. PLDI (2023). https://doi.org/10.1145/3591222

[39] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. https://doi.org/10.1145/2509578.2509586

[40] Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS*.

[41] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. 2005. On static reachability analysis of IP networks. In *INFOCOMM*.

[42] Hongkun Yang and Simon S. Lam. 2016. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* (2016). https://doi.org/10.1109/TNET.2015.2398197