Network Change Validation with Relational NetKAT

HAN XU, Princeton University, United States
ZACHARY KINCAID, Princeton University, United States
RATUL MAHAJAN, University of Washington, United States
DAVID WALKER, Princeton University, United States

Relational NetKAT (RN) is a new specification language for network change validation. Engineers use RN to specify intended changes by providing a trace relation R, which maps existing packet traces in the pre-change network to intended packet traces in the post-change network. The *intended* set of traces may then be checked against the *actual* post-change traces to uncover errors in implementation. Trace relations are constructed compositionally from a language of combinators that include trace insertion, trace deletion, and packet transformation, as well as regular operators for concatenation, union, and iteration of relations. We provide algorithms for converting trace relations into a new form of NetKAT transducer and also for constructing an automaton that recognizes the image of a NetKAT automaton under a NetKAT transducer. These algorithms, together with existing decision procedures for NetKAT automaton equivalence, suffice for validating network changes. We provide a denotational semantics for our specification language, prove our compilation algorithms correct, implement a tool for network change validation, and evaluate it on a set of benchmarks drawn from a production network and Amazon's Batfish toolkit.

CCS Concepts: • **Theory of computation** → **Formal languages and automata theory**; • **Networks**;

Additional Key Words and Phrases: Network verification; Change validation; NetKAT; Automata theory

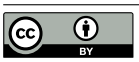
ACM Reference Format:
Han Xu, Zachary Kincaid, Ratul Mahajan, and David Walker. 2026. Network Change Validation with Relational NetKAT. *Proc. ACM Program. Lang.* 10, POPL, Article 14 (January 2026), 54 pages. https://doi.org/10.1145/3776656

1 Introduction

As industrial networks have grown in size and complexity, so too has the difficulty of keeping them running smoothly, and when networks go down, critical services of all kinds are disrupted. Recognizing the threat, researchers looked to formal methods to systematically detect and prevent problems, and over the course of the 2010s, the foundations for sound and scalable network verification were born through systems such as Anteater [Mai et al. 2011], Header Space Analysis [Kazemian et al. 2012], Veriflow [Khurshid et al. 2013], Batfish [Fogel et al. 2015], NetKAT [Anderson et al. 2014a], Minesweeper [Beckett et al. 2017], and Self-Starter [Kakarla et al. 2020], among others. Today, thanks to the transfer of technology, ideas, and people, most if not all large hyperscale cloud providers, including Microsoft, Amazon, and Google, use verification tools to make their networks more reliable [Backes et al. 2019; Jayaraman et al. 2019; Zeng et al. 2014].

Despite the successes and wide adoption of these techniques, making changes to network configurations continues to be a risky activity. A key problem is that *existing specification languages are poorly suited for concisely describing the full effect of common network changes.* They focus on

Authors' Contact Information: Han Xu, hx3501@princeton.edu, Princeton University, United States; Zachary Kincaid, zkincaid@cs.princeton.edu, Princeton University, United States; Ratul Mahajan, ratul@cs.washington.edu, University of Washington, United States; David Walker, dpw@princeton.edu, Princeton University, United States.



ACM 2475-1421/2026/1-ART14
https://doi.org/10.1145/3776656

single-snapshot verification, that is, checking the properties of a *single* network configuration. Using single-snapshot verification, one can check coarse properties of the changed network such as reachability, isolation, or access control. However, a comprehensive analysis of a change will often involve (1) identifying all of the paths in the old network that satisfy some property (the need to be changed), (2) checking that the new network contains all and exactly the corresponding changed paths and that those changed paths have the desired properties, and (3) checking that none of the other paths in the old network (those that should not have been altered) have been corrupted, blocked, or rerouted in the new network. Single-snapshot verifiers provide no means of "getting one's hands on" the old paths and ensuring they are properly changed in the updated network—by definition single-snapshot verifiers do not reference the old paths or old network at all.

To overcome such issues, Xu et al. [2024] recently proposed *relational network verification*. In their system, called Rela, networks were modeled as sets of *paths*, where each path is a sequence of locations (device groups, switches, or interfaces) along which a packet will travel. A language of *rational relations* over paths was used in combination with the old network to specify the intended network change. More precisely, network engineers would write a transformer R to indicate the intent of a change and specifications took (roughly) the form "old $\triangleright R$ = new" where old and new encoded the paths through the old and new networks respectively. The relation R transforms the old network into a set of expectations and those expectations are compared with the new network. Their change specifications were compact, often being just 10s of lines of code, even for networks with millions of flows, because even in large networks, individual changes are often small.

While Rela is a strong progess on an important problem, its "set of paths" model is a weak model for a modern network. It cannot represent internal decision-making made by network forwarding tables, which classify and act on packets with different headers in different ways, nor can it represent packet transformations, such as those used in network address translators (NATs), tunnels, or other purposes in software-defined networks (SDN) (*e.g.*, in-band network telemetry) .

In this paper, we introduce Relational NetKAT (RN for short), a new language capable of fully and accurately specifying changes to stateless modern networks. Whereas Rela uses regular sets of paths to describe networks, we adopt a stronger base model—that of NetKAT [Anderson et al. 2014a]—in which networks are described as sets of *traces*. Here, a *trace* is a sequence of *located packets*—that is the headers of packets (destination IP, source IP, *etc.*) together with their network location. By using NetKAT to describe traces, one can easily model packet forwarding tables, with their match-action semantics compactly, as well as the various ways that stateless modern networks modify packets including in devices such as stateless NATs, SDN devices, and tunnels.

Relational NetKAT augments traditional NetKAT with a new sublanguage of trace relations R. Users can write relations that specify desired network changes by inserting new subpaths into existing traces, or deleting old ones that should no longer be present. Alternatively, users can map packet relations across the elements of a trace to represent changes to packet contents between old and new networks. This new language of trace relations is compositional, conferring the ability to construct complex specifications from simpler parts through the use of the regular operators for concatenation, union, and iteration, and has a well-defined denotational semantics.

The main technical contribution of this paper is the automata-theoretic machinery that enables verification of relational NetKAT specifications. First, we introduce a class of *NetKAT transducers*, which are used to represent trace relations. NetKAT transducers may be thought of as an extension of NetKAT automata with an additional tape. Second, we give an algorithm for constructing a NetKAT automaton that recognizes the image of a NetKAT automaton under a NetKAT transducer. In principle, this can be accomplished using classical algorithms for transducers, but the size of the resulting NetKAT automaton is exponential in the size of the input. We avoid this state-space explosion by making careful use of our domain: While networks are large, most changes are

intended to be small [Xu et al. 2024], affecting a limited number of locations and/or flows in the network. Consequently, "most" of the trace relation between old and new networks is the identity relation, and we take care to exploit this property during image computation.

We have proven the correctness of our compilation algorithms and implemented them in a tool for network verification. We evaluate the effectiveness of our tool in multiple ways. First, we demonstrate the use of Relational NetKAT on Rela's open benchmark, which is drawn from an example change implemented in one of Alibaba's production networks. While our prototype tool is considerably slower than Rela due to the much more expressive trace specifications we process,[1] we demonstrate it is fast enough for large-scale industrial use. Moreover, the fact one may process all packet equivalence classes in parallel means that the slowdown may have little practical significance. Second, we demonstrate the expressiveness of the RN specification language by tackling a series of examples drawn from the Batfish toolkit. Here, we craft relational specifications for a corporate network with NATs and IP tunneling, demonstrating how changes to forwarding may be specified and checked.

2 Background: (Modestly Extended) NetKAT

NetKAT [Anderson et al. 2014b] is a decidable language for specifying properties of stateless network data planes. While there are a number of frameworks for verifying stateless data planes, some in heavy use in industry, NetKAT stands out as it provides a general, compositional specification language in which new properties are easily constructed from smaller, simpler parts. When first presented, NetKAT was equipped with an equational theory proven sound and complete with respect to its denotational semantics. In this paper, we ignore NetKAT's equational theory and focus entirely on its denotational semantics and algorithms for compilation to automata. We need only these latter elements for clearly defined specifications and decidable verification procedures.

In this paper, we present a slight variant and minor refactoring of traditional NetKAT, which we call "Modestly Extended" NetKAT when we need to point out the differences. Modestly Extended NetKAT contains a few additional primitives for crafting relations between one packet and the next. These extensions are not very important for NetKAT itself; they play a larger role in our new Relational NetKAT. Readers familiar with NetKAT may skip ahead to the syntax and semantics presented in Figure 1.

Networks as Sets of Traces. NetKAT models a network as a set of possible *traces*, where each trace captures the sequence of steps a packet takes from some network source to some network sink. More specifically, a *trace* (tr) is a sequence of *located packets* (pk), where a *located packet* is record mapping packet header field names to values and mapping the special `loc` field to the packet's current location in the network. Networks locations are often identified using a switch and a port on that switch[Anderson et al. 2014b]. For simplicity, most of our examples are presented at a slightly higher level of abstraction, calling out only the switch at which a packet has just arrived, not the port at which it sits. For instance, if pk_1 is $\{\texttt{loc} = A; \texttt{dst.ip} = 1.1.1.1; \cdots\}$ and pk_2 and pk_3 are similar but with location fields B and C respectively then the trace $(pk_1\ pk_2\ pk_3)$ represents a path starting at the ingress to A, through B, and ending at the ingress to C for a packet with destination IP address 1.1.1.1. In this case, the packet contents remain unchanged as it travels that path, but in general, packet contents may change as it travels from A to B. We write $pk[\texttt{f} \leftarrow \texttt{v}]$ to update field `f` of pk with value `v`.

Two traces may be *concatenated* only when the last packet of one trace equals the first packet of the next trace, and when they do, the equal packets are dropped from the result. For instance,

[1] and less time optimizing our prototype—Rela uses a heavily optimized, off-the-shelf library for equivalence of ordinary finite automata.

concatenating ($pk_1\ pk_2\ pk_3$) with ($pk_4\ pk_5\ pk_6$) results in ($pk_1\ pk_2\ pk_5\ pk_6$) when $pk_3 = pk_4$ and is undefined otherwise. Hence, concatenating two 2-element traces leaves us with a 2-element trace. The concatenation of two trace sets ($S_1 \circ S_2$) is defined as follows

$$S_1 \circ S_2 = \{pk_{11}\ pk_{12}\ \cdots\ pk_{1(n-1)}\ pk_{22}\ \cdots\ pk_{2m} \mid pk_{11}\ \cdots\ pk_{1(n-1)}\ pk_{1n} \in S_1, pk_{21}\ pk_{22} \cdots\ pk_{2m} \in S_2, pk_{1n} = pk_{21}\}$$

Compact Network Specifications. For a network of even modest size, its set of traces is enormous as each packet header may be 100+ bits and paths can vary in length. Fortunately, NetKAT provides a means to specify these sets of paths compactly. NetKAT contains two essential sublanguages: (1) a language of boolean predicates, and (2) a language of packet relations.

The boolean predicates (*pred*) are used to classify packets so we can describe how sets of related packets are forwarded efficiently. The basic predicate ($\mathtt{f} = \mathtt{v}$) is true of packets with field $\mathtt{f}$ containing value $\mathtt{v}$. Boolean combinations of basic predicates are possible using conjuction ($\cdot$), disjunction(+) and negation ($\neg$). For instance, $(\mathtt{src.ip} = 1.1.1.1) \cdot (\mathtt{loc} = A + \mathtt{loc} = C)$ describes the set of packets with source IP 1.1.1.1 at locations A or C. We write ($\mathtt{f} \neq \mathtt{v}$) for $\neg(\mathtt{f} = \mathtt{v})$.

The packet relations (*PkR*) explain how to forward a packet one hop along its journey through a network. Semantically, such relations denote sets of pairs of located packets. Traditional NetKAT contains a basic relation to update a field of a packet, write $\mathtt{f} \leftarrow \mathtt{v}$. When $\mathtt{f}$ is the location field, $\mathtt{loc} \leftarrow L$ models the movement of a packet from its current location to some (likely new) location L. NetKAT also contains the identity relation ($\overline{1}$), which relates every packet to itself, and the empty relation ($\overline{0}$). More complex relations can be constructed through composition of packet relations ($PkR_1 \circ PkR_2$), intersection of packet relations ($PkR_1 \cap PkR_2$), and cartesian product of predicates ($pred_1 \times pred_2$). The latter relates every packet in the set $pred_1$ with every packet in the set $pred_2$—it may be used to model a nondeterministic forwarding relation, for instance. The cartesian product is a new primitive for NetKAT, though its semantics could already be encoded through a disjunction of many alternatives. We choose to add it as a primitive because doing so leads to a more efficient implementation for a variety of our specifications. Without it, our encodings would be quadratic in the size of the sets represented by $pred_1$ and $pred_2$. Of particular importance is the *havoc* relation, which we define as 1×1, and which relates any two packets. We write $\overline{pred}$ for the subset of the identity relation that includes only packets satisfying *pred*. In this version of NetKAT, $\overline{pred}$ is an abbreviation for $\overline{1} \cap (pred \times pred)$.

Full NetKAT expressions (K) denote sets of traces, with each trace at least two elements long. NetKAT expressions include packet relations (*PkR*), which give rise to sets of 2-element traces, concatenation ($K_1 \circ K_2$), union ($K_1 + K_2$), and iteration of trace sets (K^*). Finally, *dup* is the set of 3-element traces: $\{pk\ pk\ pk \mid pk \in Pk\}$. While the concatenation of any trace tr of length n with a 2-element trace leaves a trace with n elements, concatenation of an n-element trace with *dup* leaves a trace with $n + 1$ elements. One common abbreviation used in our specifications is *alltraces*(*pred*)—the set of all traces that satisfy *pred* on every hop. *alltraces*(*pred*) is defined as $(pred \times 1 \circ dup)^* \circ pred \times pred$. The syntax and semantics of NetKAT is summarized in Figure 1.

Examples. The data plane of a network is composed of a set of *forwarding tables*—for simplicity here, one forwarding table per switch. Each forwarding table consists of a series of *match-action rules*. The "match" component of a given rule is modeled as a NetKAT predicate and the action is modeled by a packet relation. For example, a forwarding table containing rules

(1) forward packets with $\mathtt{dst.ip}$ 1.0.0.0 to switch B
(2) modify packets with $\mathtt{dst.ip}$ 1.0.0.1 so their $\mathtt{src.ip}$ is 2.0.0.0 before forwarding to C
(3) drop all other packets

Syntax:

$$
\begin{array}{lcl}
Pred & ::= & 0 \mid 1 \mid f = v \mid Pred_1 + Pred_2 \mid Pred_1 \cdot Pred_2 \mid \neg Pred \\
PkR & ::= & \overline{0} \mid \overline{1} \mid f \leftarrow v \mid Pred \times Pred \mid PkR_1 \circ PkR_2 \mid PkR_1 \cup PkR_2 \mid PkR_1 \cap PkR_2 \mid \neg PkR \\
K & ::= & PkR \mid K_1 + K_2 \mid K_1 \circ K_2 \mid K^* \mid dup
\end{array}
$$

Common Abbreviations:

$$
\begin{array}{lcl}
havoc & = & 1 \times 1 \\
\overline{pred} & = & \overline{1} \cap (pred \times pred) \\
alltraces(pred) & = & (pred \times 1 \circ dup)^* \circ pred \times pred \\
alltraces & = & alltraces(1)
\end{array}
$$

Semantics:

$$
\begin{aligned}
[\![0]\!]_{Pred} &= \emptyset \\
[\![1]\!]_{Pred} &= \{pk \mid pk \in Pk\} \\
[\![\texttt{loc} = \texttt{v}]\!]_{Pred} &= \{pk \mid pk.\texttt{loc} = \texttt{v}\} \\
[\![\neg Pred]\!]_{Pred} &= Pk \setminus [\![Pred]\!]_{Pred} \\
[\![Pred_1 + Pred_2]\!]_{Pred} &= [\![Pred_1]\!]_{Pred} \cup [\![Pred_2]\!]_{Pred} \\
[\![Pred_1 \cdot Pred_2]\!]_{Pred} &= [\![Pred_1]\!]_{Pred} \cap [\![Pred_2]\!]_{Pred}
\end{aligned}
$$

$$
\begin{aligned}
[\![\overline{0}]\!]_{PkR} &= \emptyset \\
[\![\overline{1}]\!]_{PkR} &= \{(pk, pk) \mid pk \in Pk\} \\
[\![f \leftarrow v]\!]_{PkR} &= \{(pk, pk[f \leftarrow v]) \mid pk \in Pk\} \\
[\![Pred_1 \times Pred_2]\!]_{PkR} &= \{(pk_1, pk_2) \mid pk_1 \in [\![Pred_1]\!]_{Pred}, pk_2 \in [\![Pred_2]\!]_{Pred}\} \\
[\![PkR_1 \circ PkR_2]\!]_{PkR} &= \{(pk_1, pk_3) \mid \exists pk_2.\, (pk_1, pk_2) \in [\![PkR_1]\!]_{PkR} \wedge (pk_2, pk_3) \in [\![PkR_2]\!]_{PkR}\} \\
[\![\neg PkR]\!]_{PkR} &= Pk \times Pk \setminus [\![PkR]\!]_{PkR} \\
[\![PkR_1 \cup PkR_2]\!]_{PkR} &= [\![PkR_1]\!]_{PkR} \cup [\![PkR_2]\!]_{PkR} \\
[\![PkR_1 \cap PkR_2]\!]_{PkR} &= [\![PkR_1]\!]_{PkR} \cap [\![PkR_2]\!]_{PkR}
\end{aligned}
$$

$$
\begin{aligned}
[\![PkR]\!]_K &= \{pk1\ pk2 \mid (pk1, pk2) \in [\![PkR]\!]_{PkR}\} \\
[\![K_1 + K_2]\!]_K &= [\![K_1]\!]_{PkR} \cup [\![K_2]\!]_{PkR} \\
[\![K_1 \circ K_2]\!]_K &= [\![K_1]\!]_{PkR} \cdot [\![K_2]\!]_{PkR} \\
[\![K^*]\!]_K &= \bigcup_{n \geq 0} [\![K^n]\!]_K, \quad \text{where } K^0 = \overline{1},\ K^{n+1} = K^n \circ K \\
[\![dup]\!]_K &= \{pk\ pk\ pk \mid pk \in Pk\}
\end{aligned}
$$

Fig. 1. Syntax and Semantics of (Modestly Extended) NetKAT

may be modeled using the NetKAT expression FTA =

$$
\begin{array}{lll}
(\overline{\texttt{dst.ip} = 1.0.0.0} \circ \texttt{loc} \leftarrow B) + & & \texttt{\% rule 1} \\
(\overline{\texttt{dst.ip} = 1.0.0.1} \circ \texttt{src.ip} \leftarrow 2.0.0.0 \circ \texttt{loc} \leftarrow C) + & & \texttt{\% rule 2} \\
(\overline{\texttt{dst.ip} \neq 1.0.0.0 \cdot \texttt{dst.ip} \neq 1.0.0.1} \circ \overline{0}) & & \texttt{\% rule 3}
\end{array}
$$

FTA's denotation contains a set of 2-element traces, including for instance, the trace $\{\texttt{loc} = L; \texttt{dst.ip} = 1.0.0.0; \ldots\}\ \{\texttt{loc} = B; \texttt{dst.ip} = 1.0.0.0; \ldots\}$ for all locations L since the position of the

forwarding table in the network is not specified. Further, since $[\![K \circ \overline{0}]\!]_K = [\![\overline{0}]\!]_K$, the third rule is unnecessary.

Now, suppose forwarding table FTA is located at switch A and switches B and C have their own forwarding tables FTB and FTC respectively. A network containing those three switches may be modeled with the following expression K_{net}.

$$(((\overline{\texttt{loc} = A} \circ \texttt{FTA}) + (\overline{\texttt{loc} = B} \circ \texttt{FTB}) + (\overline{\texttt{loc} = C} \circ \texttt{FTC})) \circ \mathit{dup})^*$$

Intuitively, the expression states that a packet travelling through the network may encounter any of switches A, B or C, and when it does it is processed using their forwarding table. The Kleene star indicates a packet may encounter 0 or more switches along its journey through the network. The *dup* expression is used to control what is observed as a packet travels through the network—it makes a copy of a packet and adds that copy to the trace. Without it, we would have only 2-element traces that record start and end points of a packet's journey. When making a change to a network, engineers often care about the details of the path being used, for instance, in cases where the change involves decommisioning an internal switch, eliminating internal congestion, or obeying a security that mandates only certain devices (perhaps in certain countries) are used.

Single-snapshot Verification with NetKAT. NetKAT has regularly been used as a *single-snapshot* specification and verification platform. To do so, one uses the techniques of the previous paragraphs to encode the network of interest as a NetKAT expression K_{net}. Next, to specify a property, one adds constraints to K_{net} and constructs an equation or inequation. For example, to check that a particular class of packets, say those satisfying predicate *blocked*, are not forwarded through the network, one might check the equation $[\![\overline{\mathit{blocked}} \circ K_{net}]\!]_K = [\![\overline{0}]\!]_K$. To check that some packets satisfying $\overline{\mathit{reach}}$ will be forwarded from A to C, one might check $[\![\overline{\texttt{loc} = A} \circ \overline{\mathit{reach}} \circ K_{net} \circ \overline{\texttt{loc} = C}]\!]_K \neq [\![\overline{0}]\!]_K$.

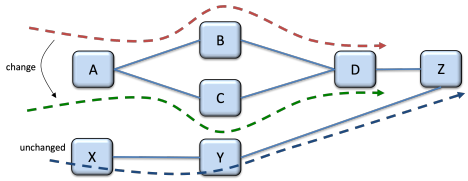
Change Validation with NetKAT. To our knowledge, NetKAT and its variants have only ever been used for single-snapshot verification. However, we observe that due to the nature of its equational specifications, NetKAT can validate the simplest kinds of changes—it can compare subsets of the old, pre-change network to the new post-change network with an equation of the form $K_{pre} \cap \mathit{subset} = K_{post} \cap \mathit{subset}$, using the intersection operator of KATch [Moeller et al. 2024]. More general change validation is out of reach for NetKAT. There is no general way to identify a set of traces in the pre-change network, apply a transformation to them, and then check that all and exactly the transformed traces appear as desired in the post-change network. As we demonstrate via examples below, this capability is key to validating many types of changes.

3 Relational NetKAT

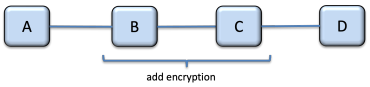
Relational NetKAT (RN) extends NetKAT with a new sublanguage for expressing *trace relations* R. Such trace relations denote subsets of Trace × Trace and may be used to capture fine-grained correspondences between traces that occur in a network before a change and traces that occur in a network after a change. These trace relations may be included in NetKAT expressions using the relational image operation $K \triangleright R$, which applies relation R to K yielding some new set of traces K'. Trace relations enable a new, modular style of network specification—one focused on transforming networks and comparing the transformations to desired specifications. This style is particularly well-suited to verification of changes to networks.

Normally, to use RN to verify a change, a network operator will supply a pair of trace relations—one to transform the pre-change network (R_{pre}) and one to transform the post-change network (R_{post}). Given those two specifications, the RN system:

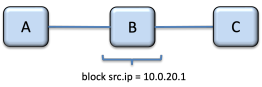
- translates the pre-change network into a NetKAT expression K_{pre},



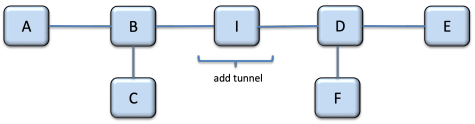
(a) Change path



(b) Add encryption



(c) Change firewall



(d) Add tunnel

Fig. 2. Four network updates.

- translates the post-change network into a NetKAT expression K_{post}, and
- implements a decision procedure for the specification $K_{pre} \triangleright R_{pre} = K_{post} \triangleright R_{post}$, where $K \triangleright R$ is the set of traces computed by finding the image of R applied to K.

When the second relation, R_{post} is the identity, we are left with the equation $K_{pre} \triangleright R_{pre} = K_{post}$. In this case, R_{pre} represents the engineer's (possibly incorrect) beliefs about the change process, and $K_{pre} \triangleright R_{pre}$ specifies the expected impact of the change on the original network. On the other hand, K_{post} is the *actual* effect of the change. The goal is to uncover bugs by checking the engineer's beliefs against reality.

3.1 Trace Relations

To introduce and illustrate the use of trace relations in network change specifications, we provide a series of examples before supplying a formal semantics.

Example: Specifying a Path Change. Consider a simple scenario, presented in Figure 2a, in which a network engineer must take a switch (say, switch B) offline for maintenance to upgrade or replace it. Prior to doing so, any existing traffic traversing B should instead be redirected along another path, perhaps the one using switch C. A strong specification of such a change should include two elements:

(1) Traces in the pre-change network that do not include B should stay the same in the post-change network.
(2) Traces in the pre-change network that include B should instead use C in the post-change network, but should otherwise stay the same.

This specification is easy to state and understand precisely because it refers to the traces *from the old network*. If one attempted a single-snapshot specification of the property (i.e., a specification that does not include direct reference to "the pre-change network"), one would effectively have to enumerate all of the (possibly millions) of paths in the pre-change network by hand, or by some other process, patching those paths by replacing B with C where necessary. Alternatively—and this has been the modus operandi of past NetKAT verification efforts, which have not focused on change validation—one checks weaker properties such as reachability. However, in industrial practice, the stronger properties are needed to have confidence in the change. Hence, despite the availability of in-house verification tools, much change checking has, somewhat surprisingly, been done manually and at great cost in terms of human time and effort[Xu et al. 2024].

In RN, specifying and checking such changes is straightforward. To begin, consider part (1) of the specification above, which demands that a subset of the network *stay the same*. Ironically,

"staying the same" is amongst the most common components of our change specifications. Such specifications are instances of the *identity relation* on traces, written $Id(K)$, which relates every trace $t \in [\![K]\!]_K$ to itself. Hence, in this case, the first part of the specification may be written using the following trace relation.

$$stay\_the\_same = Id(alltraces(\texttt{loc} \neq B))$$

To implement the second part of the specification, we must transform traces from using B to using C instead. To this end, Relational NetKAT supplies $DeleteSeg(K)$ to delete a segment matching K and $InsertSeg(K')$ to add a segment matching K'. More precisely, $DeleteSeg(K)$ relates all traces in $[\![K \circ dup \circ \texttt{havoc}]\!]_K$ to an "empty" trace[2]. Recall that the *havoc* relation leaves the relationship between one packet and the next in a trace completely unconstrained. Hence, the "$dup \circ havoc$" extension may be read as saying "and any other packet may come next." In our experience, when writing specifications, one typically writes them in "chunks," where one chunk does not constrain the next; The $DeleteSeg(K)$ (also $InsertSeg(K')$) combinator conveniently admits this style as the following examples show.

Now that we have the combinators we need, changing a segment that uses B to one that uses C is achieved via the relation $DeleteSeg(\texttt{loc} \leftarrow B) \cdot InsertSeg(\texttt{loc} \leftarrow C)$, which "concatenates" an insertion after a deletion, giving us a replacement of one by the other. To extend these trace relations from one-hop relations to full paths, one can concatenate the desired prefix and/or suffix. Usually, we assume no loop in a desired network, hence part (2) of the change may be expressed in its entirety as follows.

$$change\_path = Id(alltraces) \cdot DeleteSeg(\texttt{loc} \leftarrow B) \cdot InsertSeg(\texttt{loc} \leftarrow C) \cdot Id(alltraces)$$

With both parts (1) and (2) in hand, the full relation is constructed using a union.

$$fullchange = stay\_the\_same + change\_path$$

To check this change, we simply apply *fullchange* to the pre-change network and compare the result with the post-change network—that is, we verify the equation $K_{pre} \triangleright fullchange = K_{post}$.

Example: Encrypting payloads. Protocols such as IPsec will encrypt packet payloads to ensure client privacy. One way to model encryption is to extend packets with an additional field `encrypted`, set to 1 when the packet payload is encrypted and set to 0 when the packet payload is unencrypted. Now, consider the scenario presented in Figure 2b, where a network operator adds encryption and decryption protocols at the network endpoints. One might want to verify that packets traversing internal network node B and C are encrypted after the change, but otherwise act similarly, following similar paths, and with headers undisturbed.

Reasoning about such changes may be accomplished via the *Map* combinator, which is a generalization of *Id*, used earlier. $Map(PkR, K)$ applies packet relation PkR at each hop in every trace in $[\![K]\!]_K$; $Id(K)$ is an abbreviation for $Map(\overline{1}, K)$. In our case, relating any packet to its encrypted counterpart is achieved by the packet relation $(\texttt{encrypted} \leftarrow 1)$. Since we wish packets be encrypted when traversing nodes B and C, we should restrict the relation so it only applies in those locations: $(\overline{\texttt{loc} = B + \texttt{loc} = C}) \circ (\texttt{encrypted} \leftarrow 1)$. Elsewhere in the network, packets should be unchanged: $\overline{\neg(\texttt{loc} = B + \texttt{loc} = C)} \circ \overline{1}$. The desired transformation is the union of the two packet

[2]The "empty" trace is represented here as any length-1 trace.

relations applied universally:

$$\begin{aligned} \mathit{inside} &= \texttt{loc} = B + \texttt{loc} = C \\ \mathit{outside} &= \neg \mathit{inside} \\ \mathit{encrypt\_it} &= \overline{\mathit{inside}} \circ (\texttt{encrypted} \leftarrow 1) \\ \mathit{cleartext} &= \overline{\mathit{outside}} \circ \overline{1} \\ \mathit{spec} &= \mathit{Map}(\mathit{encrypt\_it} + \mathit{cleartext}, \mathit{alltraces}) \end{aligned}$$

Example: Firewall. Firewalls are widely used in all kinds of networks. Common changes involve adding, deleting, or modifying firewall policies. Figure 2c presents a network where a firewall is placed at node B. The network administrator has decided to block all traffic with source IP address 10.0.20.1, due to uncertainty about whether such traffic is malicious. The goal is to verify that the firewall rule is correctly enforced at node B, while ensuring that all other traffic remains unaffected.

Such changes can be specified using the *Filter* combinator. Whereas *Map* applies a packet relation to every "hop" in a trace, *Filter* requires a particular packet relation hold at a specific point in a trace relation. In this case, for each trace tr_1, tr_2 from an element (tr_1, tr_2) of the *firewall_path* relation, defined below. Such traces must all include a packet at node B with a $\texttt{src.ip} \neq 10.0.20.1$. Other (tr_1, tr_2) pairs are excluded from *firewall_path*.

$$\mathit{firewall\_path} = \mathit{Id}(\mathit{alltraces}) \cdot \mathit{Filter}(\overline{\texttt{loc} = B \cdot \texttt{src.ip} \neq 10.0.20.1}) \cdot \mathit{Id}(\mathit{alltraces})$$

To include all other traces that do not pass through B (thereby excluding just those traces that do pass through B but have $\texttt{src.ip} = 10.0.20.1$) we use the familiar identity relation:

$$\mathit{stay\_the\_same} = \mathit{Id}(\mathit{alltraces}(\texttt{loc} \neq B))$$

Combining *stay_the_same* and *firewall_path*, gives us the firewall update relation we desire: $K_{pre} \triangleright (\mathit{stay\_the\_same} + \mathit{firewall\_path}) = K_{post}$, which asserts that the post-change network behaves like the pre-change network except for traffic passing through B, which is subject to the firewall rule.

Example: Tunneling. Figure 2d presents a system involving two corporate networks, one with nodes A, B, C, and the other with nodes D, E, F. The internet I sits between the two. We will assume that A, B, C use IP addresses identified by predicate net_1 and D, E, F use IP addresses identified by predicate net_2. In this example, the user makes a change that adds a tunnel between C and D. The tunnel entry after C will add a new set of headers to every packet passing through it, with $\texttt{dst.ip}$ 128.112.0.0 (C's destination IP). The tunnel exit prior to D will strip the header, revealing the original header underneath. A similar transformation occurs in the opposite direction (D and C) with IP 128.112.0.1 (D's destination IP) used.

One of the limitations of NetKAT is that it demands all packets use the same set of headers throughout the network (header fields cannot be pushed and popped arbitrarily). Still, we can model a network that admits one extra layer of headers where needed by assuming all packets have a second copy of each header field. We name the two copies $\texttt{dst.ip}_1$ and $\texttt{dst.ip}_2$ with $\texttt{dst.ip}_1$ the inner header and $\texttt{dst.ip}_2$ the outer header. When $\texttt{dst.ip}_2$ is not in use, the field is set to 0. With this set-up, relating packets moving through the tunnel is achieved via *tunnel_change* =

$$\begin{aligned} &(\mathit{loc} = I \cdot \mathit{net}_2 \cdot \texttt{dst.ip}_2 \leftarrow 128.112.0.0) + && \texttt{\% in the tunnel, headed to net2} \\ &(\mathit{loc} = I \cdot \mathit{net}_1 \cdot \texttt{dst.ip}_2 \leftarrow 128.112.0.1) && \texttt{\% in the tunnel, headed to net1} \end{aligned}$$

The relations for (1) all traces that travel through the tunnel one from one network to the other (*net2net_traces*), (2) all traces that remain in one network or the other, not using the tunnel

Syntax:

$$R ::= Filter(PkR) \mid Map(PkR, K) \mid Delete(K) \mid Insert(K) \mid R_1 \cdot R_2 \mid R1 + R2 \mid R^*$$
$$K ::= \cdots \mid K \triangleright R$$

Abbreviations:

$$\begin{aligned} Id(K) &= Map(\overline{1}, K) \\ DeleteSeg(K) &= Delete(K \circ dup \circ havoc) \\ InsertSeg(K) &= Insert(K \circ dup \circ havoc) \\ MapSeg(PkR, K) &= Map(PkR, K) \cdot Map(havoc, havoc) \end{aligned}$$

Semantics:

$$\begin{aligned} [\![Filter(PkR)]\!]_R &= \{(pk_1, pk_2) \mid (pk_1, pk_2) \in [\![PkR]\!]_{PkR}\} \\ [\![Map(PkR, K))]\!]_R &= \{(pk_1 \ \cdots \ pk_n, pk'_1 \ \cdots \ pk'_n) \mid pk_1 \ \cdots \ pk_n \in [\![K]\!]_K, \\ &\qquad\qquad \forall i \in [1, n].(pk_i, pk'_i) \in [\![PkR]\!]_{PkR}\} \\ [\![Delete(K)]\!]_R &= \{(pk_1 \ \cdots \ pk_n, pk) \mid pk \in Pk, pk_1 \ \cdots \ pk_n \in [\![K]\!]_K\} \\ [\![Insert(K)]\!]_R &= \{(pk, pk_1 \ \cdots \ pk_n) \mid pk \in Pk, pk_1 \ \cdots \ pk_n \in [\![K]\!]_K\} \\ [\![R_1 + R_2]\!]_R &= [\![R_1]\!]_R \cup [\![R_2]\!]_R \\ [\![R_1 \cdot R_2]\!]_R &= \left\{ \begin{array}{l} (pk_1 \ \cdots \ pk_{n1} \ \cdots \ pk_{n2}, \ pk'_1 \ \cdots \ pk'_{m1} \ \cdots \ pk'_{m2}) \mid \\ \quad (pk_1 \ \cdots \ pk_{n1}, \ pk'_1 \ \cdots \ pk'_{m1}) \in [\![R_1]\!]_R, \\ \quad (pk_{n1} \ \cdots \ pk_{n2}, \ pk'_{m1} \ \cdots \ pk'_{m2}) \in [\![R_2]\!]_R \end{array} \right\} \\ [\![R^*]\!]_R &= \bigcup_{n \geq 0} [\![R^n]\!]_R, \quad \text{where } R^0 = Filter(havoc), \quad R^{n+1} = R^n \cdot R \end{aligned}$$

$$[\![K \triangleright R]\!]_K = \{tr' \mid tr \in [\![K]\!]_K, (tr, tr') \in [\![R]\!]_R, |tr'| \geq 2\}$$

Fig. 3. Syntax and Semantics of Trace Relations.

(*other_traces*), and (3) the overall specifcation (*spec*) are presented below.

$$\begin{aligned} net2net\_traces &= Id(alltraces) \cdot MapSeg(tunnel\_change, \texttt{loc} = I) \cdot Id(alltraces) \\ other\_traces &= Id(alltraces(\texttt{loc} \neq I)) \\ spec &= net2net\_traces + other\_traces \end{aligned}$$

Summary. These examples, while idealized, illustrate a number of the (subjective) benefits of this style of specification:

- *Change specifications are clear*: Specifications say what changes directly as a relation between old and new, and say what does not change equally clearly.
- *Specifications are compact*: Specifications are usually proportional the size of a change rather than proportional to the number of devices, forwarding rules, or paths in a network. Specifications for real changes in networks in our study in Section 5 are only slightly longer than the idealized variants presented here, since most of the time, most of the network says unchanged.

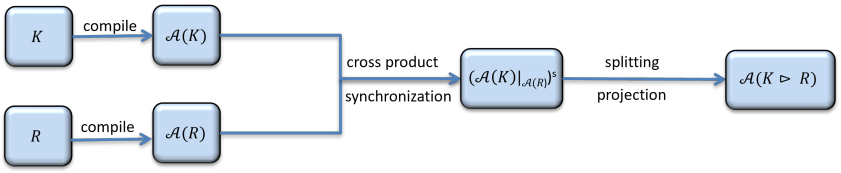


Fig. 4. Compilation process of Relational NetKAT

- *Specifications are expressive*: Using regular languages over trace relations and four simple but expressive atomic primitives, we can encode a wide variety of change-validation properties—many of which cannot be captured in prior work [Xu et al. 2024].
- *Specifications are compositional*: Sub-regions of a network may be specified separately then composed with specifications for other sub-regions.
- *Specifications are modular*: The specification need not be heavily "intermixed" with elements that represent parts of the post-change (or pre-change) networks. The specification can be defined separately from the network to which it is applied. A single specification could be reused and applied to multiple networks, or parts of the same network over time, if similar changes occur.

The formal syntax and semantics of trace relations, as well as some common abbreviations we use, are summarized in Figure 3. While most of the semantics is unsurprising, there are a couple of subtle technicalities to note. In particular, an element of a relation (tr_1, tr_2) may contain a length-1 trace. For example, a filter is implemented as a pair of length-1 traces. When a relation R is applied to a set of traces K, yielding a new set of traces, any remaining length-1 traces are excluded from the result. Concatenation in the sublanguage is also subtly different: It eliminates one intermediate packet rather than two, as in NetKAT.

4 Automata

The Relational NetKAT compiler architecture is presented in Figure 4 and proceeds as follows.

(1) The NetKAT expression K is compiled into a standard NetKAT automaton, denoted $\mathcal{A}(K)$.
(2) The Relational expression R is compiled into a NetKAT transducer, denoted $\mathcal{A}(R)$.
(3) These two automata are combined to form a transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$, which captures the synchronized behavior of K and R.
(4) Finally, the transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ is transformed into a projected automaton $\mathcal{A}(K \triangleright R)$ that recognizes $[\![K \triangleright R]\!]_K$ by applying projection. While doing so, a key *relational splitting* optimization decomposes relational transitions to reduce the state space.

We establish the correctness of this compilation process by showing that the behavior of the final automaton $\mathcal{A}(K \triangleright R)$ aligns with the denotational semantics of $K \triangleright R$.

4.1 Automaton Model for NetKAT Expression K and Relation R

Our automata model for NetKAT expressions K are similar to prior work [Foster et al. 2015; Smolka et al. 2015], following the Antimirov derivatives construction as used in work [Smolka et al. 2015] to generate finite automata from NetKAT programs. Since our language equivalence decision procedure for K requires one step of determinization, we also adopt the explicit state representation from this line of work.

The key difference in our automata is that we unify the two different kinds of transitions: ϵ-transitions that lead to accepting states, and δ-transitions that lead to non-final states into one

unified transition Δ representing all of the cases. Using this Δ transition, we further give a different definition of the transition relation $(s_0, pk_0) \xrightarrow{w} (s_n, pk_n)$, characterizing the language acceptance.

This unified Δ transition-based formulation and Antimirov derivatives construction make the compilation process readily extensible to relational expressions R, enabling us to define a uniform cross-product construction between automata for K and R in the subsequent steps.

NetKAT Automata. We begin by defining a general automaton model that captures the structure needed to represent NetKAT programs.

Definition 1. *A **NetKAT automaton** is a tuple $M = (S, S_0, S_f, \Delta)$ where S is a finite set of states, $S_0 \subseteq S$ is the set of start states, $S_f \subseteq S$ is the set of final states, and $\Delta : S \times S \to 2^{Pk \times Pk}$ is a transition relation.*

Unlike classical automata, where transitions depend solely on the current input symbol, network devices operate on the entire packet and update it in place. Upon receiving a packet, a device inspects and possibly modifies it before forwarding. To accurately model this behavior, each transition in a NetKAT automaton involves a configuration pair (s, pk), consisting of a state and a packet.

To distinguish packets on different tapes in the relational setting, throughout this section we use x and w to denote packets and traces on the NetKAT automaton tape, and y and z for those on the other tape of the NetKAT transducer.

The transition relation of M is defined inductively over packet traces. A labeled transition

$$(s_0, x_0) \xrightarrow{w} (s_n, x_n)$$

indicates that, starting from state s_0 and input packet x_0, the automaton can process the trace w and reach state s_n with final packet x_n.

- **Base case:** $(s_0, x_0) \xrightarrow{\epsilon} (s_0, x_0)$
- **Inductive case:** If $(x_0, x_1) \in \Delta(s_0, s_1)$ and $(s_1, x_1) \xrightarrow{w} (s_n, x_n)$, then $(s_0, x_0) \xrightarrow{x_1 w} (s_n, x_n)$.

The language accepted by M is the set of input/output traces that begin from an initial state and terminate in an accepting state:

$$L(M) = \{x_0 w \mid s_0 \in S_0, s_f \in S_f, x_0, x_n \in Pk. (s_0, x_0) \xrightarrow{w} (s_f, x_n)\}.$$

Compilation of K. To compile a NetKAT expression K into an automaton, we adopt the Antimirov derivative approach [Smolka et al. 2015], adapted to our explicit automaton model. Although this construction yields a non-deterministic (i.e, the out transition of each state may not be disjoint) automaton, it guarantees that the number of states is bounded by the syntactic length of the expression $l(K) + 1$ (with proof in Appendix), a desirable property inherited from the derivative framework [Antimirov 1996; Sakarovitch 2009]. The compilation process is straightforward and is detailed in the appendix. We also formally prove its correctness:

Theorem 4.1. *For all NetKAT expressions K, let $\mathcal{A}(K)$ be the automaton constructed via the Antimirov derivative. Then:*

$$L(\mathcal{A}(K)) = [\![K]\!]_K.$$

NetKAT Transducer. NetKAT transducers model the semantics of relational NetKAT programs R, which operate on *pairs* of traces. Intuitively, a NetKAT transducer behaves like a two-tape automaton, where each tape corresponds to a trace in the pair. Each state in the automaton is implicitly associated with the current packet being processed on each tape.

In our syntax, we support four kinds of atomic relational constructs: $Map(\mathsf{PkR}, K)$ advances both tapes simultaneously, $Delete(K)$ advances only the left tape, $Insert(K)$ advances only the right tape, and $Filter(\mathsf{PkR})$ without advancing either tape.

To model the behavior of these constructs, the NetKAT transducer includes four basic transition types, corresponding directly to the above operations:

Definition 2. *A **NetKAT transducer** is a tuple $T = (S, S_0, S_f, \Delta_S, \Delta_L, \Delta_R, \Delta_E)$ where S is a finite set of states, $S_0 \subseteq S$ is a set of start states, $S_f \subseteq S$ is a set of final states, $\Delta_S : S \times S \to (Pk \times Pk) \times (Pk \times Pk)$ is a synchronous transition relation, $\Delta_L : S \times S \to Pk \times Pk$ is an asynchronous left transition relation, $\Delta_R : S \times S \to Pk \times Pk$ is an asynchronous right transition relation, and $\Delta_E : S \times S \to Pk \times Pk$ is an epsilon transition relation.*

Similar to NetKAT automaton M, the transition relation of T is defined inductively over pairs of packet traces. A labeled transition of the form:

$$(s_0, (x_0, y_0)) \xrightarrow{(w,z)} (s_n, (x_{n_1}, y_{n_2}))$$

indicates that, starting from (x_0, y_0), the transducer produces the output traces w and z along a path to state s_n with final packets (x_{n_1}, y_{n_2}).

- **Base case:** $(s_0, (x_0, y_0)) \xrightarrow{(\epsilon,\epsilon)} (s_0, (x_0, y_0))$.
- **Both tapes move:** If $((x_0, y_0), (x_1, y_1)) \in \Delta_S(s_0, s_1)$ and $(s_1, (x_1, y_1)) \xrightarrow{(w,z)} (s_n, (x_{n_1}, y_{n_2}))$, then $(s_0, (x_0, y_0)) \xrightarrow{(x_1 w, y_1 z)} (s_n, (x_{n_1}, y_{n_2}))$.
- **First tape only:** If $(x_0, x_1) \in \Delta_L(s_0, s_1)$ and $(s_1, (x_1, y_0)) \xrightarrow{(w,z)} (s_n, (x_{n_1}, y_{n_2}))$, then $(s_0, (x_0, y_0)) \xrightarrow{(x_1 w, z)} (s_n, (x_{n_1}, y_{n_2}))$.
- **Second tape only:** If $(y_0, y_1) \in \Delta_R(s_0, s_1)$ and $(s_1, (x_0, y_1)) \xrightarrow{(w,z)} (s_n, (x_{n_1}, y_{n_2}))$, then $(s_0, (x_0, y_0)) \xrightarrow{(w, y_1 z)} (s_n, (x_{n_1}, y_{n_2}))$.
- **No tape moves:** If $(x_0, y_0) \in \Delta_E(s_0, s_1)$ and $(s_1, (x_0, y_0)) \xrightarrow{(w,z)} (s_n, (x_{n_1}, y_{n_2}))$, then $(s_0, (x_0, y_0)) \xrightarrow{(w,z)} (s_n, (x_{n_1}, y_{n_2}))$.

The language accepted by the NetKAT transducer T is then defined as the set of trace pairs processed from a start state to an accepting state:

$$L(T) = \{(xw, yz) \mid s_0 \in S_0, s_f \in S_f, x, x', y, y' \in Pk. (s_0, (x, y)) \xrightarrow{(w,z)} (s_f, (x', y'))\}.$$

Correctness of Compilation. Similar to the case for NetKAT expressions, we can ensure the correctness of the NetKAT transducer with respect to the denotational semantics. The number of states is again bounded by the syntactic length of the expression $l(R) + 1$. The full details of compilation and proof is in the Appendix.

Theorem 4.2. *For all relational NetKAT expressions R, let $\mathcal{A}(R)$ be the transducer constructed via the Antimirov derivative. Then:*

$$L(\mathcal{A}(R)) = [\![R]\!]_R.$$

4.2 Cross Product and Synchronization

Our goal in this section is to compile an automaton that faithfully captures the semantics of the projection expression $K \triangleright R$. As an intermediate step, we construct the cross-product transducer $\mathcal{A}(R)|_{\mathcal{A}(K)}$, which combines the behavior of the NetKAT automaton $\mathcal{A}(K)$ and the NetKAT transducer $\mathcal{A}(R)$. We then eliminate all forms of asynchronous and ϵ transitions from $\mathcal{A}(R)|_{\mathcal{A}(K)}$ to form a synchronized version of the NetKAT transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$. We break the compilation process into two steps:

(1) **Cross Product:** Construct the cross-product transducer $\mathcal{A}(R)|_{\mathcal{A}(K)}$ from $\mathcal{A}(K)$ and $\mathcal{A}(R)$ by aligning transitions from $\mathcal{A}(K)$ with the first tape of $\mathcal{A}(R)$. The resulting transducer $\mathcal{A}(R)|_{\mathcal{A}(K)}$ accepts the relation $\{(w, z) \in \mathcal{A}(R) \mid w \in \mathcal{A}(K)\}$.
(2) **Synchronization:** Eliminate all Δ_E, Δ_L, and Δ_R transitions to obtain the synchronized transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$, in which transitions occur simultaneously on both tapes. Note that $L((\mathcal{A}(R)|_{\mathcal{A}(K)})^s) \neq L(\mathcal{A}(R)|_{\mathcal{A}(K)})$; however, they agree on the second tape of valid NetKAT traces (length greater than 2):

$$\{z \mid (w, z) \in (\mathcal{A}(R)|_{\mathcal{A}(K)})^s, |z| \geq 2\} = \{z \mid (w, z) \in \mathcal{A}(R)|_{\mathcal{A}(K)}, |z| \geq 2\}.$$

Cross Product Construction. Suppose we have NetKAT automaton $\mathcal{A}(K) = (S_k, S_{k0}, S_{kf}, \Delta)$ that accepts single traces, while the NetKAT transducer $\mathcal{A}(R) = (S_r, S_{r0}, S_{rf}, \Delta_S, \Delta_L, \Delta_R, \Delta_E)$ accepts pairs of traces. The purpose of the cross-product $\mathcal{A}(R)|_{\mathcal{A}(K)}$ is to match every trace on the first tape of $\mathcal{A}(R)$ to a valid trace accepted by $\mathcal{A}(K)$.

We define the cross-product transducer $\mathcal{A}(R)|_{\mathcal{A}(K)} = (S_{kr}, S_{kr0}, S_{krf}, \Delta'_S, \Delta'_L, \Delta'_R, \Delta'_E)$ as follows:

- $S_{kr} = S_k \times S_r$ is the set of composite states.
- $S_{kr0} = S_{k0} \times S_{r0}$ is the set of initial state pairs.
- $S_{krf} = S_{kf} \times S_{rf}$ is the set of accepting state pairs.

The transition relation is defined as follows:

- **Synchronized transitions on both tapes:**

$$\Delta'_S((s_k, s_r), (s'_k, s'_r)) = \{((x_1, y_1), (x_2, y_2)) \mid (x_1, x_2) \in \Delta(s_k, s'_k) \wedge ((x_1, x_2), (y_1, y_2)) \in \Delta_S(s_r, s'_r)\}$$

- **Transition on the first tape only:**

$$\Delta'_L((s_k, s_r), (s'_k, s'_r)) = \left\{(x_1, x_2) \;\middle|\; (x_1, x_2) \in \Delta(s_k, s'_k) \wedge (x_1, x_2) \in \Delta_L(s_r, s'_r)\right\}$$

- **Transition on the second tape only:**

$$\Delta'_R((s_k, s_r), (s'_k, s'_r)) = \begin{cases} \Delta_R(s_r, s'_r) & \text{if } s_k = s'_k \\ \emptyset & \text{otherwise} \end{cases}$$

- **Transition on no tape:**

$$\Delta'_E((s_k, s_r), (s'_k, s'_r)) = \begin{cases} \Delta_E(s_r, s'_r) & \text{if } s_k = s'_k \\ \emptyset & \text{otherwise} \end{cases}$$

That is, when the relational transition consumes a packet on the X tape, we synchronize it with a corresponding transition in $\mathcal{A}(K)$. When the relational transition doesn't consume on the X tape, we keep the NetKAT state unchanged (i.e., $s_k = s'_k$) and perform only the transition on NetKAT transducer states.

Synchronization. After constructing the intermediate transducer $\mathcal{A}(R)|_{\mathcal{A}(K)}$, we eliminate all transitions involving Δ_E, Δ_L, and Δ_R, yielding a standard two-tape NetKAT transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ in which each transition consumes exactly one packet from both tapes. Formally, we refine $\mathcal{A}(R)|_{\mathcal{A}(K)}$ from:

$$(S_{kr}, S_{kr0}, S_{krf}, \Delta'_S, \Delta'_L, \Delta'_R, \Delta'_E) \quad \text{to} \quad (S_{kr}, S_{kr0}, S_{krf}, \Delta^*_S, \emptyset, \emptyset, \emptyset).$$

where Δ^*_S represent the synchronized transition for the new transducer.

A key observation is that synchronization does not preserve the full trace language:

$$L((\mathcal{A}(R)|_{\mathcal{A}(K)})^s) \neq L(\mathcal{A}(R)|_{\mathcal{A}(K)}).$$

However, since the projection $K \triangleright R$ only concerns the trace sequence on the *second tape*, we need only preserve the second-tape behavior. That is, we require:

$$\{z \mid (w,z) \in L((\mathcal{A}(R)|_{\mathcal{A}(K)})^s), |z| \geq 2\} = \{z \mid (w,z) \in L(\mathcal{A}(R)|_{\mathcal{A}(K)}), |z| \geq 2\}.$$

To achieve this, we apply a two-step strategy:

- **Simulate right-only transitions:** For every transition in Δ'_R that advances the second tape while leaving the first unchanged, we replace it with a synchronized transition:

  $$\Delta_{SR}(s_1,s_2) = \left\{((x_1,y_1),(x_1,y_2)) \mid (y_1,y_2) \in \Delta'_R(s_1,s_2)\right\}.$$

  This construction preserves the effect of the original right-only transition while achieving synchronization by introducing a no-op (i.e., static) move on the first tape.
- **ϵ-Elimination:** We compute the transitive closure of Δ'_E (transitions that move neither tape) and Δ'_L (transitions that move only the first tape). Using this closure, we propagate the effects of these transitions into Δ'_S and Δ_{SR}, effectively incorporating all intermediate ϵ-moves into synchronized transitions. This process mirrors standard ϵ-elimination in automata theory and yields a fully synchronized transducer.

The resulting NetKAT transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ is compatible with the projection operator and preserves the desired second-tape behavior. Full algorithmic details of the synchronization and ϵ-elimination procedures are deferred to the appendix. We summarize the correctness of the construction with the following theorem:

Theorem 4.3. *Let $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ be the synchronized NetKAT transducer constructed from the cross-product and synchronization procedures. Then:*

$$\left\{z \;\middle|\; (w,z) \in L((\mathcal{A}(R)|_{\mathcal{A}(K)})^s)\right\} = [\![K \triangleright R]\!]_K.$$

Once $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ is constructed, we apply standard reachability analysis to prune unreachable states and transitions. This optimization is important for performance in practice, as shown in our evaluation in Section 5, though we defer the technical details to the appendix.

4.3 Relational Splitting and Projection

The ultimate goal of this section is to project the NetKAT transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ into the final form $\mathcal{A}(K \triangleright R)$, while avoiding state explosion.

High-Level Idea. Following the synchronization procedure, we obtain a NetKAT transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ where every transition consumes a pair of packets—one from each tape. Our objective now is to project out the first tape X and extract the trace set from the second tape Y.

A naive approach to projection is to explicitly track the first-tape packet as part of the state. That is, we construct a NetKAT automaton where the states are of the form (s, pk) and where $\Delta((s,x),(s',x')) \triangleq \{(y,y') : ((x,y),(x',y')) \in \Delta_S(s,s')\}$. Thus the configuration space of the NetKAT transducer $S \times (Pk \times Pk)$ is isomorphic to the configuration space of the NetKAT automaton $(S \times Pk) \times Pk$, but in the NetKAT automaton the first packet in the configuration is part of the explicit state rather than the symbolic state.

However, this strategy leads to an exponential blowup in the state space. For each automaton state s, we would need to generate up to $|Pk|$ copies to account for all possible packet values pk, resulting in an infeasible number of states. For instance, even a packet with just 32-bit source and destination IP fields would yield at least 2^{64} distinct combinations—making equivalence-checking impossible in practice.

A key observation is that, in practice, relational NetKAT programs typically induce only "small" changes while preserving most of the structure at each transition. For example, suppose a NetKAT

relation R is of the form $Id(K)$ for some NetKAT expression K. In this case, we need not track the entire packet state on the X tape, because it is always equal to the packet on the Y tape!

Thus, for this example, we can construct a NetKAT automaton that recognizes the projection of the relation $[\![Id(K)]\!]_R$ using a NetKAT transducer T with the same state space as $\mathcal{A}(Id(K))$, and with the transition relation

$$\Delta(s, s') \triangleq \{(y, y') \mid ((x, y), (x', y')) \in \Delta_S(s, s'),\ x = y,\ x' = y'\}.$$

What makes this optimization possible is that, in all transitions $(s_1, (x_1, y_1)) \xrightarrow{(x_2, y_2)} (s_2, (x_2, y_2))$ of $\mathcal{A}(R)$, the value of x is completely determined by s and y. While this is not true for all NetKAT transducers, we can always partition the transitions of $\mathcal{A}(R)$ into finitely many cases in which the Y-tape value determines the X-tape value. Although the worst-case size of such an injective partition (i.e., partitioning until y determines x) is $O(|Pk|)$, our "small change" observation suggests that the size of this partition is often small—in the case of $Id(K)$, it has only one cell. The definition below formalizes our intuition about *efficiently projectable transducers*.

Definition 3 (Efficiently Projectable Transducer). *Let $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ be a synchronous NetKAT transducer. Its projection $|T| \triangleq (S, S_0, S_f, \Delta)$ onto the second tape is a NetKAT automaton with the same state space, initial states, and final states as T, and where the transition relation is defined as:*

$$\Delta(s, s') \triangleq \{(y, y') \mid \exists x, x' \in Pk.((x, y), (x', y')) \in \Delta_S(s, s')\}.$$

We say that T is efficiently projectable *if:*

$$L(|T|) = \{z \mid (w, z) \in L(T)\}.$$

Notice that $L(|T|) \supseteq \{z \mid (w, z) \in L(T)\}$ for any T, and that equality of the two sets holds only in special cases. One case in which the a NetKAT transducer is efficiently projectable is when the y-packet uniquely determines its corresponding x-packet. Under this assumption, each configuration $(s, (x, y))$ of the original transducer is in one-to-one correspondence with (s, y) in the projected automaton. From a bisimulation perspective, this yields a strong invariant:

$$\exists x_1, x_2 \in Pk.(s_1, (x_1, y_1)) \xrightarrow{(x_2, y_2)}_T (s_2, (x_2, y_2)) \iff (s_1, y_1) \xrightarrow{y_2}_{|T|} (s_2, y_2).$$

However, in real-world settings, this uniqueness assumption is often too restrictive. Some fields—such as VLAN tags, source ports, or other metadata—may be abstracted or ignored, meaning y cannot fully determine x. Enforcing strict uniqueness in such cases would require explicitly enumerating all values of these abstracted fields, leading to an exponential blowup of size $2^{|f|}$, where $|f|$ is the length of hidden fields.

To avoid this explosion, we relax the uniqueness condition and introduce the notion of *x-bisimulation*. The observation here is that for hidden fields, all of the different values associated with them have identical behavior on the y-tape across all subsequent transitions. They are safe to collapse even if multiple x values may map to the same y. We relax the uniqueness condition to y uniquely determines a set of *behaviourly equivalent* values of x (with respect to the y-tape). Since the state s and the y-tape remain observable after projection, we only need to consider bisimulation of the x-tape for a fixed state s and y. This motivates the following definition.

Definition 4 (x-bisimulation). *For a synchronous NetKAT transducer $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$, a relation $x_1 \equiv_{s,y} x_2$ is an x-bisimulation if:*

- *For all $x_1 \equiv_{s,y} x_2$ and $((x_1, y), (x_3, y')) \in \Delta_S(s, s')$, there exists x_4 such that $((x_2, y), (x_4, y')) \in \Delta_S(s, s')$ and $x_3 \equiv_{s',y'} x_4$.*

- *For all* $x_1 \equiv_{s,y} x_2$ *and* $((x_2, y), (x_4, y')) \in \Delta_S(s, s')$*, there exists* x_3 *such that* $((x_1, y), (x_3, y')) \in \Delta_S(s, s')$ *and* $x_3 \equiv_{s',y'} x_4$*.*

We define the maximum x-bisimulation relation $R_{\max}$ *to be the union of all x-bisimulations.*

If the y-packet determines a set of x-packets that are behaviorally indistinguishable on the y-tape, then transitions in the original transducer involving (x, y) pairs can be faithfully simulated using only y. That is, for all reachable configurations (x_1, y_1) at state s_1, the original transducer makes a transition

$$\exists x_1, x_2 \in Pk.(s_1, (x_1, y_1)) \xrightarrow{(x_2,y_2)} (s_2, (x_2, y_2))$$

if and only if the projected automaton makes a transition

$$(s_1, y_1) \xrightarrow{y_2} (s_2, y_2).$$

This effectively means that the set of reachable configurations $(s, (x, y))$ is bisimilar to (s, y)—and therefore, projection preserves the trace semantics.

We introduce the notion of *bisimulation witnesses* as a sufficient condition for bisimilarity of a synchronous NetKAT transducer T and its projection $|T|$. A bisimulation witness for the transducer T is a function $B : S \to 2^{Pk \times Pk}$ imply that there is a bisimulation relation $\sim$ between T and $|T|$ defined by

$$(s, (x, y)) \sim (s', y') \iff s = s', y = y', \text{ and } (x, y) \in B(s)$$

Definition 5 (Bisimulation Witness). *Let* $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ *be a synchronous NetKAT transducer. A function* $B : S \to 2^{Pk \times Pk}$ *is a* bisimulation witness *if the following conditions holds:*

(1) *(Bisimulation Witness) The relation* $(x_1 \equiv_{s,y} x_2) \triangleq ((x_1, y) \in B(s)) \wedge ((x_2, y) \in B(s))$ *is a x-bisimulation for tranducer* T*.*
(2) *(Domain coverage) For all* $s \in S$*,* $B(s) \supseteq \bigcup_{s' \in S} dom(\Delta(s, s'))$
(3) *(Range coverage) For all* $s \in S$*,* $B(s') \supseteq \bigcup_{s \in S} range(\Delta(s, s'))$

This bisimulation witness B captures the observable behavior of the transducer at each state. Condition (1) weakens the strict uniqueness requirement on x and replaces it with behavioral equivalence: all x values compatible with the same y must exhibit indistinguishable behavior on y-tape across transitions. Condition (2) ensures that B covers all source pairs involved in any outgoing transition, while condition (3) propagates this property to target states, ensuring that the image of any transition remains within the bisimulation witness.

Then we can conclude that this transducer is efficiently projectable if such a witness is found.

Theorem 4.4 (Correctness of Projection). *Let* $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ *be a synchronous NetKAT transducer. If there exists a bisimulation witness B, then* T *is* efficiently projectable.

Relational Splitting. After defining the class of transducers that can be efficiently projected, the next step is to transform a general $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ transducer into one that belongs to this class. The key challenge lies in discovering a bisimulation witness. However, the bisimulation witness defined in the previous section may not exist for all transducers T. Therefore, we aim to split the transducer by states or transitions until a bisimulation witness can be identified for every state s.

A natural approach is to compute the maximum bisimulation at each state and then split states according to the equivalence classes of this relation. These equivalence classes form a bisimulation witness for the resulting transducer, thereby ensuring that it is *efficiently projectable*. Moreover, this approach yields a transducer with the minimal number of states after splitting. We describe this method in the Appendix.

However, this approach is inefficient in practice, as computing the global bisimulation relation is computationally expensive. Prior work [Fisler and Vardi 2002] has shown that computing

the maximum bisimulation relation for a symbolic model can be more costly than verifying an invariant for that model. This overhead becomes prohibitive when handling industrial networks with thousands of devices.

To avoid this cost, we propose a "local" approach that computes equivalence classes on a per-transition basis. Specifically, we check whether (x_1, y) and (x_2, y) reach the same (x', y') in the next state. This yields the following equivalence relation:

$$x_1 \equiv_{s,y} x_2 \quad \triangleq \quad \forall s',\ ((x_1, y), (x', y')) \in \Delta(s, s') \iff ((x_2, y), (x', y')) \in \Delta(s, s')$$

It is straightforward to verify that this relation defines an x-bisimulation for transducer T. This localized x-bisimulation enables us to generate bisimulation witnesses per transition. Our experiments in Section 5 demonstrate that this approach yields a 2.5× speedup over the global bisimulation method. We adopt this local splitting approach in the paper, and the resulting transducer construction based on this equivalence relation is defined as follows:

Step 1: Transition-Based Partitioning. The first step is to partition each transition relation $\Delta_S(s, s')$ into a set of subrelations r' such that each subrelation's input pairs is indistinguishable towards y-tape. i.e., for all $(x_1, y), (x_2, y) \in dom(r'), x', y'$ we have

$$((x_1, y), (x', y')) \in r' \iff ((x_2, y), (x', y')) \in r'$$

We do not prescribe a concrete algorithm for this partitioning, as efficient strategies may depend heavily on the structure of the transition r. In principle, a naive partitioning can always be achieved by enumerating all x values corresponding to each fixed y, although this may lead to exponential blowup. In practice, more optimized approaches are often possible. For example, one such strategy is discussed in Section 5, where we exploit the low-level BDD representation used in our implementation to enable efficient symbolic partitioning.

Formally, we suppose that we have partition function $P : S \times S \to 2^{(Pk \times Pk) \times (Pk \times Pk)}$ such that:

- For each pair of states (s, s'), each $r \in P(s, s')$ satisfies a localized bisimulation condition required for efficient projection. More precisely, for every $r \in P(s, s')$, the following holds for all $(x_1, y), (x_2, y) \in dom(r), x', y'$ we have

  $$((x_1, y), (x', y')) \in r \iff ((x_2, y), (x', y')) \in r$$

  This condition ensures that all pairs (x, y) appearing as inputs to the transition relation r' are indistinguishable towards the output at next states.
- The partition covers the transition relation:

  $$\left(\bigcup_{r \in P(s,s')} r \right) = \Delta(s, s').$$

Step 2: Bisimulation Witness Generation. The ultimate goal of splitting is to transform the transducer into one that is efficiently projectable, with bisimulation witnesses serving as the guarantee. In this step, we aim to assign each state $s \in S$ a set of bisimulation witnesses such that, in the final split transducer, the states take the form (s, b), where the bisimulation witness of (s, b) is b. Formally, we define the target function $\mathcal{B} : S \to 2^{2^{Pk \times Pk}}$ as follows:

$$\mathcal{B}(s) = \begin{cases} \{\mathrm{dom}(r) \mid \exists s' \in S, r \in P(s, s')\}, & \text{if } s \notin S_f \\ \{Pk \times Pk\}, & \text{if } s \in S_f \end{cases}$$

The idea behind this construction is that for every non-final state s, and every transition $r \in P(s, s')$, there exists a domain-covering (one that satisfy property (2)) witness $\mathrm{dom}(r)$ at state

$(s, \mathrm{dom}(r))$. For final states, we assume without loss of generality[3] that there are no outgoing transitions. Thus, domain coverage is not a concern, and we may safely assign $\mathcal{B}(s)$ to be the singleton set $Pk \times Pk$ to ensure coverage for any incoming transition (i.e., range coverage).

Combined with the previous step of transition partitioning, this witness generation ensures that in the final split transducer, each witness b at state (s, b) satisfies both condition (1) (bisimulation witness) and condition (2) (domain coverage). In the next step, we will ensure condition (3) (range coverage) through output restriction.

Step 3: State Splitting and Output Restriction. Using $\mathcal{B}(s)$, we now construct a new transducer where states are paired with their bisimulation witness. Define the split transducer:

$$T^p \triangleq (S^p, S_0^p, S_f^p, \Delta_S^p, \emptyset, \emptyset, \emptyset)$$

as follows:

- $S^p \triangleq \{(s, b) \mid s \in S, b \in \mathcal{B}(s)\}$.
- $S_0^p \triangleq \{(s_0, b) \mid s_0 \in S_0, b \in \mathcal{B}(s_0)\}$.
- $S_f^p \triangleq \{(s_f, b) \mid s_f \in S_f, b \in \mathcal{B}(s_f)\}$.
- Transition relation:

$$\Delta_S^p((s_1, b_1), (s_2, b_2)) \triangleq \bigcup_{b_1 = \mathrm{dom}(r) \wedge r \in P(s_1, s_2)} \{((x, y), (x', y')) \in r \mid (x', y') \in b_2\}$$

The goal of this construction is that, attached to each state (s, b) after splitting, the packet relation b serves as the bisimulation witness for that state, therefore an effecient projection is then doable. As discussed above, without output restriction, the split transition would be:

$$\Delta_S^p((s_1, b_1), (s_2, b_2)) \triangleq \bigcup_{b_1 = \mathrm{dom}(r) \wedge r \in P(s_1, s_2)} r$$

In this form, the b in each state (s, b) satisfies condition (1) and condition (2). Now, we apply the final output restriction step to ensure condition (3) (range coverage).

Although the output restriction prunes the output of each transition, it does not violate bisimulation conditions (1) and (2). For condition (2), the filtered transition is a subset of the unfiltered one, so domain coverage still holds. For condition (1), the key insight is this: because we adopt a x-bisimulation relation that they are behavioural indistinguishable in terms of output transition behavior, applying uniform truncation on their output transitions will preserve another uniform output transition behavior, thus x-bisimulation is retained. We thus conclude with the following theorem:

Theorem 4.5 (Correctness of Splitting). *Let $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ be a synchronous NetKAT transducer. Without loss of generality, we suppose that final states have no outgoing transitions. Then the split transducer T^p constructed as above satisfies:*

- $L(T^p) = L(T)$
- *T^p is in the class of efficiently projectable transducers.*

Then we summarize all of the compilation pipeline to the projected automaton $\mathcal{A}(K \triangleright R)$ as:

$$\mathcal{A}(K \triangleright R) \triangleq |((\mathcal{A}(R)|_{\mathcal{A}(K)})^s)^p|$$

Having constructed the projected automaton $\mathcal{A}(K \triangleright R)$ from the pipeline, we obtain an automaton of the form as a standard NetKAT automaton. This allows us to complete the verification of relational specifications $K_1 \triangleright R_1 = K_2 \triangleright R_2$. To this end, we apply two standard procedures:

[3]This holds trivially for $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$, as shown in the appendix.

determinization, followed by *bisimulation checking*. These procedures follow the same methodology as prior works [Moeller et al. 2024; Smolka et al. 2015], and are deferred to the Appendix.

5 Evaluation

In this section, we describe the implementation of our Relational NetKAT compiler and compare its performance against existing tools, such as Batfish and Rela, for verification tasks that these other tools support. Our goal is to quantify the overhead of the more general approach of RN. We also benchmark RN's performance on tasks that are uniquely supported by RN and quantify the value of our optimizations techniques. Our artifact is available at [Xu et al. 2025].

5.1 Implementation

Our implementation has approximately 3000 lines of OCaml code, includes the following key components:

(1) A compiler that translates a NetKAT expression K and a Relational NetKAT expression R into a NetKAT automaton $\mathcal{A}(K \triangleright R)$.
(2) An equivalence checker that checks the equivalence of two NetKAT automata.
(3) An interface for translating input/output formats of Batfish and Rela into our representation.

$\mathcal{A}(K \triangleright R)$ Compiler. We follow the compilation pipeline outlined in Section 4. At the lowest level, we encode packets using binary representations and use Binary Decision Diagrams (BDDs) for symbolic manipulation. Specifically, we use the MLBDD library, where each boolean variable corresponds to the value of a particular field bit.

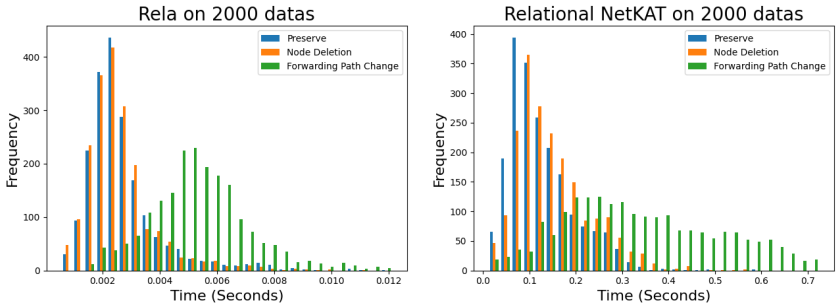
Since automaton transitions involve pairs such as (x, x') and $((x, y), (x', y'))$, we interleave the bit positions of the four packets within the BDD variable ordering. Concretely, for each bit index k, we assign positions $6k$, $6k+1$, $6k+2$, and $6k+3$ to the bits of x, x', y, and y' respectively, while reserving $6k+4$ and $6k+5$ for BDD composition. This encoding is especially efficient for common cases like identity relations or slight field modifications.

After BDD encoding, we implement the functions `delta_k`, `delta_r`, and `delta_kr` to construct the automata $\mathcal{A}(K)$, $\mathcal{A}(R)$, and $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ respectively. We apply a "reachability pruning" optimization to prune unreachable states before performing relational splitting, reducing the number of (y, x) pairs that must be considered. With this reachability optimization, we are able to do an emptiness check using the function `emptiness_check` at this phase.

Next, we implement a naive but effective BDD-based splitting strategy to ensure that each y value is associated only with indistinguishable x values. To achieve this, we reorder the BDD variable layout from (x, x', y, y') — i.e., $6k, 6k+1, 6k+2, 6k+3$ — to (y, x, y', x') — i.e., $6k+1, 6k+3, 6k, 6k+2$. This reordering enables us to analyze, for each y, whether multiple x values appear before the next y variable is encountered in the BDD. If so, we split the BDD branches; otherwise, we retain the structure. Recursively applying this process yields a partition satisfying the criteria for splitting.

We then combine this splitting logic with the algorithms described in Section 4, implementing the procedures `generate_all_transition` and `simplify_all_transition` for relational splitting and projection. Finally, we support automaton determinization through the function `determinization`, completing the compilation pipeline.

Equivalence Checker. The equivalence checker and counter-example generation follows the bisimulation-based approach introduced by Moeller et al. [2024]. Specifically, we implement the bisimulation procedure via the function `bisim`, which symbolically compares two deterministic automata for language equivalence. To generate counterexamples when equivalence fails, we compile the symmetric difference of two NetKAT expressions $e_1 \oplus e_2$ (in our case $\mathcal{A}(K_1 \triangleright R_1) \oplus$



Rela	Average Time (s)
Preserve	0.002841 (1.00x)
Delete	0.002735 (1.00x)
Change	0.005587 (1.00x)

Ours	Average Time (s)
Preserve	0.134055 (47.19x)
Delete	0.157056 (57.42x)
Change	0.355372 (63.61x)

Fig. 5. Arithmetic runtime comparison between Rela and our system across three verification goals

$\mathcal{A}(K_2 \triangleright R_2)$), implemented by the function `symmetric_difference`. This process enables us to identify traces accepted by one expression but not the other.

Interfaces for Batfish and Rela. To interface with Batfish and Rela, the first step is to encode packet structures into BDDs. We adopt two different encoding strategies for packet fields. For source/destination IP addresses and ports, we use a bit-aligned binary encoding that matches their natural layout in packets. This representation supports efficient prefix matching (e.g., for IP prefixes such as `10.20.1.0/24`) and is well-suited to forwarding table lookups. For other fields such as `loc = A` or `typ = ssh`, we assign unique identifiers to each field value and encode them into binary form. This hybrid approach avoids wasting bits on unused values while preserving efficient predicate evaluation. Combined, these strategies enable us to implement backends compatible with both Batfish and Rela.

For Batfish, RN consumes its output as input. Batfish ingests network configuration and exposes the network's routing and topology information using higher-level APIs. We use the `routes()` API to extract routing tables for each node, and `layer3Edges()` API to extract the network's topology. Interface-level properties and access control lists are retrieved using `interfaceProperties()` and `namedStructures()` APIs. IP tunnel information is extracted using `ipsecEdges()` API. Since Batfish currently lacks APIs that yield firewall and NAT, we directly parse configuration files to extract these features as needed.

For Rela, Relational NetKAT acts as a natural extension of its IR. In particular, Rela-style queries of the form $P_1 \triangleright R_1 = P_2 \triangleright R_2$ are translated into Relational NetKAT expressions. We compile path sets P_1 and P_2—representing networks annotated with IP constraints—into pre- and post-network NetKAT expressions K. Similarly, we promote path relations R_1 and R_2 to relational programs.

5.2 Comparison to Rela

Rela [Xu et al. 2024] is a relational network verification tool for verifying path changes in networks that do not transform packets. It models the network as a set of DAGs annotated with source and destination IP constraints. As shown earlier, our language subsumes the syntax of Rela's intermediate representation, enabling a direct comparison.

Xu et al. [2024] released a dataset with pre- and post-network paths from Alibaba's global backbone. Because the data is not accompanied by specific verification goals, we consider three common verification goals based on the list of changes in the paper's appendix:

Table 1. Runtime of Batfish and RN on Example Batfish Benchmarks

Forwarding Change Validation		
Function	**Batfish (s)**	**Ours (s)**
traceroute 1	1.013	0.641
reachability 1	0.440	1.391
differential 1	0.396	1.218
reachability 2	0.254	0.875
differential 2	0.246	0.391
traceroute 2	0.429	0.140
traceroute 3	0.372	0.344
reachability 3	0.377	1.046
differential 3	0.378	1.125
reachability 4	0.234	0.782
differential 4	0.398	0.625
Average	0.414	0.782(1.8x)

Hybrid Cloud Network		
Function	**Batfish (s)**	**Ours (s)**
traceroute 1	0.399	40.657
traceroute 2	0.230	3.406
traceroute 3	0.222	2.797
traceroute 4	0.380	2.281
traceroute 5	0.212	2.828
reachability 1	0.224	7.656
Average	0.278	9.271(33.3x)

- **Preserve**: Verify equivalence between the original and updated DAGs.
- **Node Deletion**: Randomly delete one node and its associated traffic, then verify equivalence.
- **Forwarding Path Change**: Redirect traffic through a randomly selected node to another random node (e.g., changing the path $A \to B \to C$ to $A \to D \to C$), and verify equivalence.

The dataset includes 2460 devices and 21,112 traffic DAGs. We randomly selected 2000 DAGs and applied each verification tasks above. As shown in Figure 5, our system is approximately 50–60 times slower than Rela. This slowdown stems from two factors:

- We use BDDs to encode location and IP headers, resulting in heavier symbolic manipulation. In contrast, Rela uses a single-character symbol per location.
- Our verification relies on symbolic automata due to the large packet space. Symbolic transitions are not necessarily disjoint, requiring $O(n)$ checks for disjointness, whereas regular automata alphabets are disjoint and allow $O(\log n)$ disjointness checks.

Despite the slowdown, our system exhibits acceptable performance (sub-second validation time). We deem that RN's slower relative performance is worthwhile given its support for richer trace relations and stronger guarantees for change types that both tools support. Unlike Rela, which considers concrete packets, RN verifies changes over extremely large packet spaces ($> 2^{32}$ packets).

5.3 Comparison to Batfish

Batfish is a widely used network verification tool that supports a broad range of single-snapshot verification tasks, along with limited forms of relational verification. To compare RN with Batfish, we replicate forwarding analyses from two tutorial examples: *Forwarding Change Validation* and *Hybrid Cloud Network*. These examples demonstrate Batfish's capabilities in relational verification and its scalability on realistic network topologies. The first example features a network with 9 devices and approximately 8.5k lines of JSON-based routing configuration. The second example involves a more complex network with 17 devices and over 200k lines of routing policies.

Our system successfully reproduces a wide range of Batfish-supported queries. In particular, we focus on data plane analysis queries which are:

(1) `reachability` – Determines whether packets satisfying header constraints H (e.g., `dst.ip`= 10.0.0.1) can traverse a path matching path constraints P (e.g., from A to C via B).

Table 2. Relational Verification Queries Only Supported by Relational NetKAT

Hybrid Cloud Network		
Verification	**Ours (s)**	**Specification**
NAT unchanged 1	26.152	$K_{pre} \triangleright Id(alltraces(\texttt{src.ip} \neq NAT\_ip \cdot \texttt{dst.ip} \neq NAT\_ip))$ $= K_{post} \triangleright Id(alltraces(\texttt{src.ip} \neq NAT\_ip \cdot \texttt{dst.ip} \neq NAT\_ip))$
NAT unchanged 2	21.141	$K_{pre} \triangleright Id((\neg(NAT\_entry) \circ dup)^*)$ $= K_{post} \triangleright Id((\neg(NAT\_entry) \circ dup)^*)$
NAT changed 1	7.469	$K_{post} \triangleright Id(\overline{\texttt{loc} = outer \cdot \texttt{dst.ip} = public\_ip} \circ alltraces \circ \overline{\texttt{loc} = inner}) = \emptyset$
Tunneling 1	28.687	$K_{pre} = K_{post}$
Tunneling 2	20.515	$K_{pre} \triangleright Map(encrypt\_it + cleartext, alltraces) = K_{post}$
Average	**20.792**	

(2) `traceroute` – Similar to `reachability`, but without user-defined path constraints; only the start location and header constraint are specified.
(3) `differentialReachability` – Verifies whether packets satisfying H reach a path P in one snapshot but are filtered in another.

We specify general reachability in relational NetKAT as:

$$\texttt{reachability} = Filter(\overline{1})Delete(\texttt{alltrace})Insert(havoc)Filter(\overline{1})$$

This relation maps all traces in the network into its input/output pair, where the verification task is: is formulated as:

$$K_{pre} \triangleright \texttt{reachability} = K_{post} \triangleright \texttt{reachability}$$

The `traceroute` query focuses the `reachability` query with a header constraint H and a path constraint P. We can express these constraints using a packet predicate P_h and NetKAT expression K_p, respectively. For instance, in the Forwarding Change Validation example, one goal is to ensure no flow traverses `"core1"` after a configuration change. This corresponds to $K_p = alltrace \circ loc \leftarrow \texttt{core1} \circ dup \circ alltrace$. A constraint like `dst.ip= "54.191.42.182"` from the Hybrid Cloud Network is similarly encoded as a packet predicate. We define a strengthened reachability as:

$$\texttt{reachability'} = Filter(P_h)Delete(K_p)Insert(havoc)Filter(\overline{1})$$

This can be used for input-output reachability queries, but to extract full trace sets (as in Batfish's `traceroute`), we define:

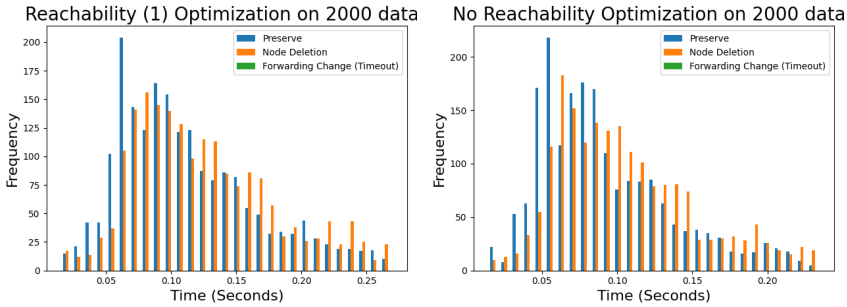
$$\texttt{traceroute} = Filter(P_h)Map(\overline{1}, K_p)$$

Finally, differential reachability across network snapshots is formulated as:

$$K_{pre} \triangleright \texttt{reachability'} = K_{post} \triangleright \texttt{reachability'}$$

This strengthens Batfish's version by enforcing output and end-location equivalence.

Table 1 shows the performance of Batfish and RN on the forwarding analyses in the two examples. While RN shows comparable performance to Batfish for most analyses, it has much higher runtime for some, where BDD and initialization overhead dominate. For instance, in the hybrid cloud benchmark, Traceroute 1 takes over 40s in our system due to symbolic automata and BDD initialization for over 200k lines of routing policies, whereas Batfish completes parsing in under 1s. We cache automata computations using hash tables to reduce recomputation costs.



Task	$R(\infty)$	$R(1)$	$R(0)$
Preserve	0.134 (1.00x)	0.119 (0.89x)	0.097 (0.72x)
Delete	0.157 (1.00x)	0.135 (0.86x)	0.114 (0.73x)
Change	0.355 (1.00x)	**TIMEOUT**	**TIMEOUT**

Fig. 6. Comparison of verification times under different levels of reachability pruning. TIMEOUT > 1000s.

5.4 Relational Verification for Networks with Transformations

Our system supports relational verification tasks that no current tool (including Batfish and Rela) can express. Table 2 lists several such relational queries and our system's performance for them on the same network as the Hybrid Cloud example above. The queries use NAT (Network Address Translation) and tunneling.

NAT queries. NAT is often deployed to bridge internal (private) and external (public) networks. It rewrites packet headers such that internal hosts appear under public IPs. In our experiment, the network operator disables the NAT entry to isolate the internal network from external traffic.

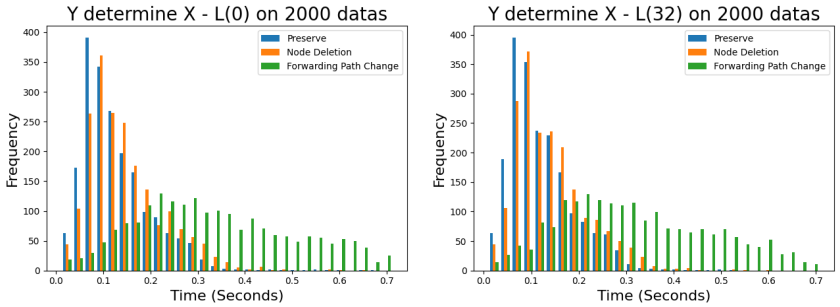
In networks with NAT, it is hard to get the exact change as changes of header also may change the forwarding path, leading to another path which is only able to be described with certain knowledge on the forwarding tables. We handle NAT-related queries by separately verifying the unchanged and affected portions of the network. In `NAT unchanged 1`, we confirm that traces not containing NAT IPs remain unaffected, and in `NAT unchanged 2`, we confirm that traces not going through NAT entries remain unaffected. While `NAT changed 1` checks that without NAT, traces coming from outside cannot reach a private destination with a public NAT ip.

Tunneling queries. The tunneling queries analyze changes from a flat, untunneled network to a tunneled configuration, as discussed in Section 3. Our system encodes symbolic equivalence between encrypted and decrypted behaviors. `Tunneling 1` verifies equivalence when all IP tunnels connect adjacent nodes, which is the original example's IPSec tunneling setting. In contrast, `Tunneling 2` encrypts traffic at randomly selectively nodes, and verifies the correspondence of the decrypted and encrypted network.

5.5 Impact of Optimizations

Our verification engine has two primary optimizations: reachability pruning and a custom splitting algorithm. We now evaluate the performance impact of these optimizations.

Reachability Pruning. This optimization is applied after composing the $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ and before using the splitting algorithm. It prunes unreachable configurations thereby improving the efficiency of splitting. We denote different levels of reachability pruning using $R(\infty)$, $R(1)$, and $R(0)$. $R(\infty)$ computes all reachable pairs exhaustively; $R(0)$ applies no pruning at all, assuming



Task	Base	$L(0)$	$L(32)$	$L(64)$	Naive
Preserve	0.134 (1.00x)	0.136 (1.01x)	0.129 (0.96x)	**TIMEOUT**	**TIMEOUT**
Delete	0.157 (1.00x)	0.153 (0.97x)	0.147 (0.94x)	**TIMEOUT**	**TIMEOUT**
Change	0.355 (1.00x)	0.356 (1.00x)	0.336 (0.95x)	**TIMEOUT**	**TIMEOUT**

Fig. 7. Comparison of verification times under different splitting strategies. TIMEOUT > 1000s.

that all configurations are reachable, and $R(1)$ is the overapproximation obtained from $R(0)$ doing one step of reachability pruning.

As shown in Figure 6, disabling reachability pruning $R(0)$ yields faster performance in the Preserve and Delete tasks due to reduced preprocessing overhead. In these cases, the packet relation is typically simple (e.g., $\overline{1}$ or $\overline{loc = B}$) and can be projected efficiently.

However, for Change tasks that involve nontrivial path changes (e.g., from $A \to C \to B$ to $A \to D \to B$), pruning becomes essential. Without pruning, $R(0)$ loses track of the (x, y) relation after leaving A, and $R(1)$ loses it after passing through C or D on the way to B. Consequently, even if transitions are identity mappings (i.e., $x = x' \wedge y = y'$), the splitting procedure on $((x, y), (x', y'))$ must consider an exponential number of x values due to lack of constraint on (x, y), leading to timeouts (> 1000s).

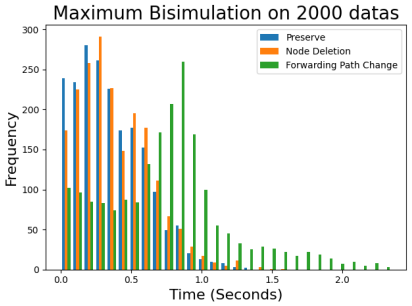
Splitting Algorithm. A key technical contribution of this paper is the novel splitting algorithm we propose for efficient automata projection. We compare its performance against several alternatives. Our design uses the observation that if the y-tape (i.e., output projection) can uniquely determine a group of x-tape values that are indistinguishable with respect to the next output transition, then we can significantly reduce the number of splits required. However, it is also possible to generate a correct projected automaton using either (1) a naive splitting strategy that enumerates all possible x values or (2) a more aggressive strategy where y uniquely determines x.

To evaluate these alternatives, we test each method on automata augmented with unused "hidden" field lengths of 0, 32, and 64 bits. These fields do not affect our current implementation (serving as a baseline, labeled **Base**), but they do impact the strategies that require y to determine x, which we denote as $L(0)$, $L(32)$, and $L(64)$. The naive strategy simply times out (i.e., exceeds 1000 seconds) on all test cases and is labeled **Naive**, as shown in Figure 7.

Our implementation employs hash-consing to memoize BDD splits, avoiding redundant computations when the same BDD structure appears. Even so, this overhead is only small when the hidden length is small (e.g., 0 or 32). However, as the hidden field length increases (e.g., to 64 bits), the splitting overhead becomes exponential, resulting in timeouts (>1000s).

This experiment confirms that our splitting algorithm is both efficient and necessary for practical automata construction in the presence of large packet spaces.

Maximum vs. Localized Bisimulation. As discussed in Section 4, we adopt a *localized* approach that computes the equivalence classes of (x, y) only up to the next transition. This approach is



Task	Average Time (s)
Preserve	0.380146 (2.84x)
Delete	0.407851 (2.60x)
Change	0.783704 (2.21x)

Fig. 8. Runtime of Maximum bisimulation.

significantly more efficient than the *maximum bisimulation* method because it avoids computing the full global bisimilarity relation and allows us to proceed with automata splitting immediately. The complete algorithm for the global approach is provided in the Appendix. Figure 8 presents the experimental results, showing an overall speedup of more than 2.2× across all cases.

5.6 Other Optimizations

We experimented with and implemented several engineering optimizations that yielded significant performance improvements in compiling and manipulating NetKAT transducers. Due to a limit of space and the fact that these optimizations are not at the core of our theory, we put the experiments and verifications for these optimization results in our artifact.

Hash Table Memoization. We memoized results in key procedures such as derivatives, cross-product construction, and BDD splitting, which are often invoked repeatedly with identical inputs. This optimization led to a more than 100× speedup on the Rela benchmark.

Efficient Routing Table Encoding. While different routing encodings ultimately yield the same BDDs, compilation time varies significantly. We replaced the naive sequential insertion of routing entries with a binary tree organization sorted by destination IPs. This change reduced compile time in the Hybrid Cloud Network case from 30+ minutes to under 60 seconds.

Implicit vs. Explicit Location Encoding. We opted to encode location information *within* the BDDs rather than using explicit NetKAT states. This implicit representation yields a more than 4–5× speedup over the explicit alternative on the Rela benchmark.

Set-Based Union Operator. Rather than encoding expressions like $e_1 + e_2 + \ldots + e_n$ as nested binary unions, we implemented a set-based construction $\{e_1, e_2, \ldots, e_n\}$. This ensures that semantically equivalent expressions are also syntactically equivalent, reducing automata state explosion from factorial to linear size in some cases.

Direct Automata Compilation from JSON.. We explored bypassing NetKAT parsing altogether by compiling automata directly from Rela's JSON output. However, due to the need for explicit location encoding (since Antimirov derivatives don't apply to the JSON format), this approach incurred slowdown compared to our current method.

Future Optimizations. The following optimizations could lead to further performance gains but are beyond the current engineering scope of our work:

- **MLBDD Library Enhancements.** Although MLBDD is the best OCaml BDD library we found, it still lacks several key features: 1) Dynamic variable reordering, 2) Garbage collection,

3) Parallelization. These improvements could significantly enhance compilation scalability and runtime.

- **Reimplementation in C/C++.** OCaml, while productive and concise, lacks high-performance transducer libraries and runtime features available in C/C++. Rewriting our core compiler in C/C++ could unlock further performance improvements, especially for low-level data structures and BDD manipulations.

6 Related Work

As mentioned earlier, there has been much past work on single-snapshot network verification [Backes et al. 2019; Beckett et al. 2017; Fogel et al. 2015; Jayaraman et al. 2019; Kakarla et al. 2020; Kazemian et al. 2012; Khurshid et al. 2013; Mai et al. 2011; Zeng et al. 2014] but little work aside from Rela [Xu et al. 2024] (discussed in the introduction) on multi-snapshot change validation.

Algorithms for Automata Construction. There are two main approaches to constructing NetKAT automata. One line of work [Smolka et al. 2015] adopts Antimirov partial derivatives [Antimirov 1996]; the other line of work [Foster et al. 2015; Moeller et al. 2024] is based on Brzozowski derivatives [Brzozowski 1964]. The essential distinction is that Antimirov derivatives yield nondeterministic automata (NFA), while Brzozowski derivatives construct deterministic automata (DFA) directly.

We adopt Antimirov derivatives for three key reasons: 1) The number of Antimirov derivatives is bounded by the syntactic size of the expression, yielding compact NFAs. 2) Even if our $\mathcal{A}(K)$ and $\mathcal{A}(R)$ constructions appear deterministic, their cross-product automaton after synchronization is not guaranteed to remain deterministic. And 3) prior work [Pous 2015] finds Antimirov-based symbolic algorithms to be more efficient than Brzozowski-based alternatives for Kleene Algebra with Tests (KAT). Thus, we construct automata using Antimirov derivatives and then apply determinization in a separate step.

There are also several choices when it comes to deciding equivalence between symbolic automata and/or KAT terms [Bonchi and Pous 2013; D'Antoni and Veanes 2014, 2017; Pous 2015]. Our implementation adopts the latest bisimulation-based symbolic approach used in the KATch system [Moeller et al. 2024]. However, our implementation deviates from KATch in other ways. While KATch uses Forwarding Decision Diagrams (FDDs) to encode transition relations [Moeller et al. 2024; Smolka et al. 2015], we used Binary Decision Diagrams (BDDs) for two reasons. First, while FDDs are optimized for binary transition relations over packet pairs (x, x'), Relational NetKAT uses two-tape transitions of the form $((x, y), (x', y'))$, which we were not able to represent efficiently with FDDs, especially when reordering is needed during splitting. Second, FDDs are designed to handle atomic tests such as $f = v$, and updates like $f \leftarrow v$ efficiently, but have no compact representation for richer constructs like *havoc*, which show up often in our change specifications.

7 Conclusion

Relational NetKAT (RN) is a compositional language for specifying a wide range of network changes, including updates to forwarding paths and modifications to packet-processing behavior such as tunnels, NATs, and firewall rules. These changes are expressed as trace relations, mapping traces in the pre-change network to those in the post-change network. To verify compliance with such specifications, RN compiles trace relations into NetKAT transducers and applies them to NetKAT automatons of both pre- and post-network. To avoid potential blowup in packet space, we design a relational splitting technique that exploits the observation that real-world changes are usually "small." Our OCaml implementation [Xu et al. 2025] and case studies show that RN captures a wide range of realistic network updates and enables efficient verification at industrial scale.

Data-Availability Statement

The companion artifact [Xu et al. 2025] includes the ocaml code.

Acknowledgments

The authors wish to thank the members of Princeton PL group and the anonymous POPL reviewers for their helpful feedback. We are also grateful for the support provided by Bloomberg through the Bloomberg Infrastructure & Security Fellowship, which has supported Han Xu's doctoral studies. This work is also supported in part by NSF awards 2219862, 2219863 and 2312539.

References

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014a. NetKAT: semantic foundations for networks. *SIGPLAN Not.* 49, 1 (Jan. 2014), 113–126. doi:10.1145/2578855.2535862

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014b. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 113–126.

Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science* 155, 2 (1996), 291–319. doi:10.1016/0304-3975(95)00182-4

John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. 2019. *Reachability Analysis for AWS-Based Networks*. 231–241. doi:10.1007/978-3-030-25543-5_14

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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. doi:10.1145/3098822.3098834

Filippo Bonchi and Damien Pous. 2013. Checking NFA equivalence with bisimulations up to congruence. *SIGPLAN Not.* 48, 1 (Jan. 2013), 457–468. doi:10.1145/2480359.2429124

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. doi:10.1145/321239.321249

Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. *SIGPLAN Not.* 49, 1 (Jan. 2014), 541–553. doi:10.1145/2578855.2535849

Loris D'Antoni and Margus Veanes. 2017. Forward Bisimulations for Nondeterministic Symbolic Finite Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 518–534.

Kathi Fisler and Moshe Y. Vardi. 2002. Bisimulation Minimization and Symbolic Model Checking. *Form. Methods Syst. Des.* 21, 1 (July 2002), 39–78. doi:10.1023/A:1016091902809

Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. *SIGPLAN Not.* 50, 1 (Jan. 2015), 343–355. doi:10.1145/2775051.2677011

Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. doi:10.1145/3341302.3342094

Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding Network Misconfigurations by Automatic Template Inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 999–1013. https://www.usenix.org/conference/nsdi20/presentation/kakarla

Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*.

Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. Association for Computing Machinery, New York, NY, USA, 290–301. doi:10.1145/2018436.2018470

Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. 2024. KATch: A Fast Symbolic Verifier for NetKAT. *Proc. ACM Program. Lang.* 8, PLDI, Article 224 (June 2024), 24 pages. doi:10.1145/3656454

Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. *SIGPLAN Not.* 50, 1 (Jan. 2015), 357–368. doi:10.1145/2775051.2677007

Jacques Sakarovitch. 2009. *Elements of Automata Theory*. Cambridge University Press.

Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. *SIGPLAN Not.* 50, 9 (Aug. 2015), 328–341. doi:10.1145/2858949.2784761

Han Xu, David Walker, Ratul Mahajan, and Zachary Kincaid. 2025. *Network Change Validation with Relational NetKAT (Artifact)*. doi:10.5281/zenodo.17650920

Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. 2024. Relational Network Verification. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 213–227. doi:10.1145/3651890.3672238

Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: divide and conquer to verify forwarding tables in huge networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 87–99.

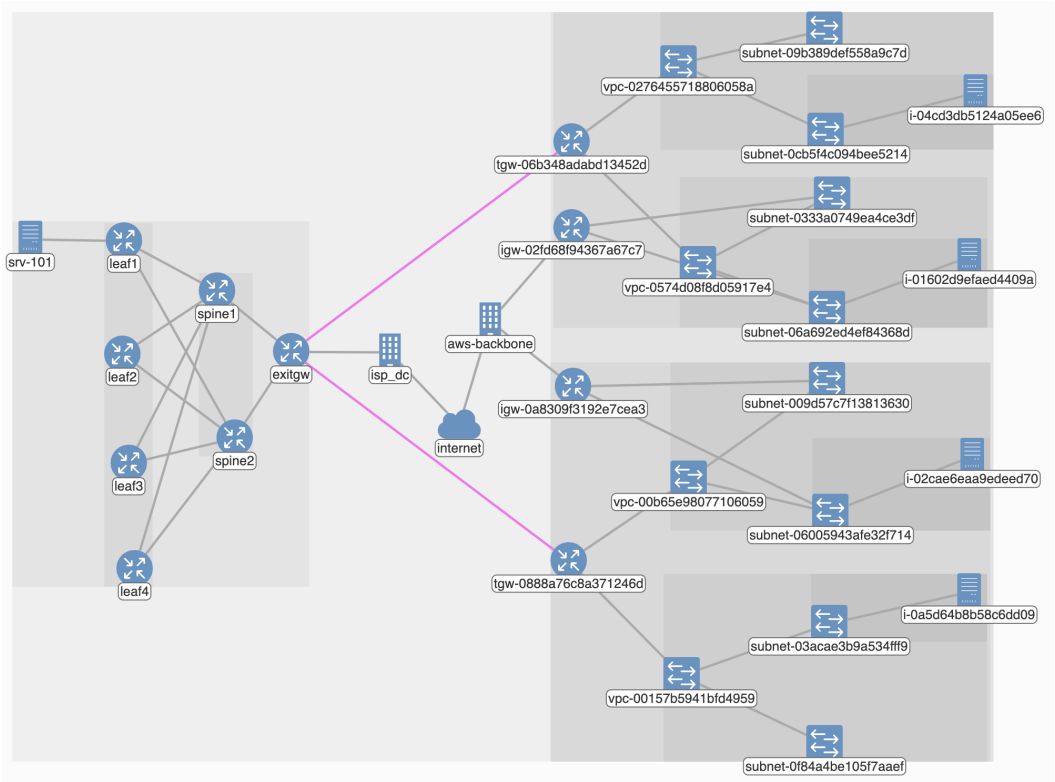


Fig. 9. Topology of Hybrid Cloud Network of Batfish

A Appendix

A.1 Batfish Topology

A.2 Full construction of NetKAT and Relational NeKAT automata

To construct the automaton $\mathcal{A}(K)$ for a NetKAT expression K, we define the automaton as a tuple (S, S_0, S_f, Δ), where:

- $S \subseteq 2^{\mathcal{K} \cup \{\epsilon\}}$: The set of states consists of sets of NetKAT expressions, where K denotes the set of all NetKAT expressions. In practice, only residual expressions generated during the input processing of K are used. The symbol ϵ denotes the empty expression, indicating the end of processing.
- $S_0 = \{K\}$: The initial state consists of the original NetKAT expression.
- $S_f = \{\epsilon\}$: The final state corresponds to the fully consumed expression, represented by the singleton set $\{\epsilon\}$.

Transition Function Compilation. The transition function Δ is computed using Antimirov derivatives, which decompose expressions into manageable components for constructing automata transitions. Specifically:

- $\delta_k : K \to 2^{K \times K}$ splits an expression K into two parts: the first part contains no occurrences of dup, while the second part begins with dup, i.e., it takes the form $dup\ K'$.

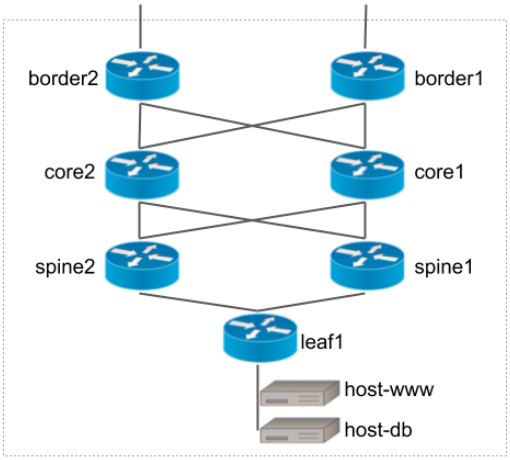


Fig. 10. Topology of Forwarding Change Validation of Batfish

- $\delta_k^\epsilon : K \to 2^K$ extracts the components of K that are free of dup, facilitating transitions where dup does not influence the output.

These derivatives are defined inductively as follows:

- $\delta_k^\epsilon\ Pkr = \{Pkr\}$
- $\delta_k^\epsilon\ (K_1 + K_2) = \delta_k^\epsilon\ K_1 \cup \delta_k^\epsilon\ K_2$
- $\delta_k^\epsilon\ (K_1 \circ K_2) = \delta_k^\epsilon\ K_1 \cdot \delta_k^\epsilon\ K_2$
- $\delta_k^\epsilon\ (K^*) = \bigcup_i \delta_k^{\epsilon(i)}\ (K)$, where

$$\delta_k^{\epsilon(0)}\ (K) = \{1\}, \quad \delta_k^{\epsilon(i+1)}\ (K) = \delta_k^{\epsilon(i)}\ (K) \cdot \delta_k^\epsilon\ (K)$$

- $\delta_k^\epsilon\ dup = \emptyset$

Similarly, the δ derivative is defined as:

- $\delta_k\ Pkr = \emptyset$
- $\delta_k\ (K_1 + K_2) = \delta_k\ K_1 \cup \delta_k\ K_2$
- $\delta_k\ (K_1 \circ K_2) = \{(K_{11}, K_{12} \circ K_2) \mid (K_{11}, K_{12}) \in \delta_k\ K_1\} \cup \{(K_1' \circ K_{21}, K_{22}) \mid K_1' \in \delta_k^\epsilon\ K_1 \wedge (K_{21}, K_{22}) \in \delta_k\ K_2\}$
- $\delta_k\ (K^*) = \{(K_0 \circ K_1, K_2 \circ K^*) \mid K_0 \in \delta_k^\epsilon\ K \wedge (K_1, K_2) \in \delta_k\ K\}$
- $\delta_k\ dup = \{(\overline{1}, dup)\}$

Using these derivatives, the transition function is compiled as:

- $\Delta\ (K_1, K_2) = \bigcup_{(K_1', dup\ K_2) \in \delta_k\ K_1} \{(x, x') | xx' \in [\![K_1']\!]_K\}$
- $\Delta\ (K, \epsilon) = \bigcup_{K' \in \delta_k^\epsilon\ K} \{(x, x') | xx' \in [\![K']\!]_K\}$
- $\Delta\ (\epsilon, K) = \Delta\ \epsilon\ \epsilon = \emptyset$

With these definitions, the automaton $\mathcal{A}(K)$ is fully specified as (S, S_0, S_f, Δ).

A.2.1 NetKAT Transducer Construction. The construction of $\mathcal{A}(R)$ for a relational expression R follows a structured approach inspired by Antimirov derivatives. Let $R_\epsilon = R \cup \{\epsilon \times \epsilon\}$, and define the transducer $(S, S_0, S_f, \Delta_S, \Delta_L, \Delta_R, \Delta_E)$ as follows:

- $S_r \subseteq 2^{\mathcal{R} \cup \{\epsilon \times \epsilon\}}$: States represent residual expressions after processing input.
- $S_0 = \{R\}$: The initial state corresponds to the full relational expression.
- $S_f = \{\epsilon \times \epsilon\}$: The final state indicates that the expression has been fully processed.

Transition Function Compilation. The transition function Δ for relational transducer includes three components:

- Δ_S: Handles transitions of the form $Map(PkR, K)$.
- Δ_L: Handles transitions of the form $Delete(K)$.
- Δ_R: Handles transitions of the form $Insert(K)$.
- Δ_E: Handles transitions of the form $Filter(PkR)$.

To compute these transitions, we use Antimirov derivatives for relational expressions. The derivatives are defined as follows:

- $\delta : R \to 2^{R \times R}$ splits an expression R into two parts: the first part contains the next operation (one of $Filter(PkR)$, $Delete(K)$, $Insert(K)$ or $Map(PkR, K)$), and the second part represents the residual pair of traces.
- $\delta_r^\epsilon : R \to 2^R$ determines whether an expression can evaluate to emptiness.

Definitions of Derivatives. The derivatives are computed as follows. Note that the we have Δ_E as epsilon transition explicitly in our $\mathcal{A}(R)$ components. Therefore, we don't hurry to calculate the transitive closure of Δ_E at this step.

For δ_r^ϵ:

- $\delta_r^\epsilon\ Filter(PkR) = \{Filter(PkR)\}$
- $\delta_r^\epsilon\ Delete(K) = \{Delete(K_1) \mid K_1 \in \delta_k^\epsilon\ K\}$
- $\delta_r^\epsilon\ Insert(K) = \{Insert(K_1) \mid K_1 \in \delta_k^\epsilon\ K\}$
- $\delta_r^\epsilon\ Map(PkR, K) = \{Map(PkR, K_1) \mid K_1 \in \delta_k^\epsilon\ K\}$
- $\delta_r^\epsilon\ (R_1 + R_2) = \delta_r^\epsilon\ R_1 \cup \delta_r^\epsilon\ R_2$
- $\delta_r^\epsilon\ (R_1 \cdot R_2) = \delta_r^\epsilon\ R_1 \cdot \delta_r^\epsilon\ R_2$
- $\delta_r^\epsilon\ (R^*) = \{Filter(havoc)\}$

For δ:

- $\delta_r\ Filter(PkR) = \{\}$
- $\delta_r\ Delete(K) = \{(Delete(K_1), Delete(K_2)) \mid (K_1, K_2) \in \delta_k\ K\}$
- $\delta_r\ Insert(K) = \{(Insert(K_1), Insert(K_2)) \mid (K_1, K_2) \in \delta_k\ K\}$
- $\delta_r\ Map(PkR, K) = \{(Map(PkR, K_1), Map(PkR, K_2)) \mid (K_1, K_2) \in \delta_k\ K\}$
- $\delta_r\ (R_1 + R_2) = \delta_r\ R_1 \cup \delta_r\ R_2$
- $\delta_r\ (R_1 R_2) = \{(R_{11}, R_{12} \cdot R_2) \mid (R_{11}, R_{12}) \in \delta_r\ R_1\} \cup \{(R_1', R_2) \mid R_1' \in \delta_r^\epsilon\ R_1\}$
- $\delta_r\ (R^*) = \{(R_1, R_2 \cdot R^*) \mid (R_1, R_2) \in \delta_r\ R\} \cup \{(R', R^*) \mid R' \in \delta_r^\epsilon\ R\}$

Transition Case Analysis. The transitions for Δ_S, Δ_L, Δ_R, and Δ_E are defined as follows:

- $\Delta_S\ (R_1, R_2) = \bigcup_{(Map(PkR, R_1'), R_2) \in \delta_r\ R_1} \{(x, y), (x', y')) | (xx', yy')) \in [\![Map(PkR, R_1')]\!]_R\}$
- $\Delta_S\ (R, (\epsilon \times \epsilon)) = \bigcup_{Map(PkR, R') \in \delta_r^\epsilon\ R} \{(x, y), (x', y')) | (xx', yy')) \in [\![Map(PkR, R')]\!]_R\}$
- $\Delta_S\ ((\epsilon \times \epsilon), R) = \Delta_S\ ((\epsilon \times \epsilon), (\epsilon \times \epsilon)) = \emptyset$
- $\Delta_L\ (R_1, R_2) = \bigcup_{(Delete(R_1'), R_2) \in \delta_r\ R_1} \{(x, x') | (xx', y)) \in [\![Delete(R_1')]\!]_R\}$

- $\Delta_L\ (R, (\epsilon \times \epsilon)) = \bigcup_{Delete(R') \in \delta_r^\epsilon\ R} \{(x, x') | (xx', y)) \in [\![Delete(R')]\!]_R\}$
- $\Delta_L\ ((\epsilon \times \epsilon), R) = \Delta_S\ ((\epsilon \times \epsilon), (\epsilon \times \epsilon)) = \emptyset$
- $\Delta_R\ (R_1, R_2) = \bigcup_{(Insert(R_1'), R_2) \in \delta_r\ R_1} \{(y, y') | (x, yy')) \in [\![Insert(R_1')]\!]_R\}$
- $\Delta_R\ (R, (\epsilon \times \epsilon)) = \bigcup_{Insert(R') \in \delta_r^\epsilon\ R} \{(y, y') | (x, yy')) \in [\![Insert(R')]\!]_R\}$
- $\Delta_R\ ((\epsilon \times \epsilon), R) = \Delta_R\ ((\epsilon \times \epsilon), (\epsilon \times \epsilon)) = \emptyset$
- $\Delta_E\ (R_1, R_2) = \bigcup_{(Filter(PkR), R_2) \in \delta_r\ R_1} \{(x, y) | (x, y)) \in [\![Filter(PkR)]\!]_R\}$
- $\Delta_E\ (R, (\epsilon \times \epsilon)) = \bigcup_{Filter(PkR) \in \delta_r^\epsilon\ R} \{(x, y) | (x, y) \in [\![Filter(PkR)]\!]_R\}$
- $\Delta_E\ ((\epsilon \times \epsilon), R) = \Delta_E\ ((\epsilon \times \epsilon), (\epsilon \times \epsilon)) = \emptyset$

The transducer $\mathcal{A}(R)$ is thus fully constructed as $(S, S_0, S_f, \Delta_S, \Delta_L, \Delta_R, \Delta_E)$.

A.2.2 Proof of Correctness. Next, we demonstrate that our derivative method is correct.

Theorem A.1. *For the intermediate δ_k and δ_k^ϵ used in the construction of $\mathcal{A}(K)$, we have:*

$$[\![K]\!]_K = \bigcup_{(K_1, K_2) \in \delta_k\ K} [\![K_1]\!]_K \circ [\![K_2]\!]_K \cup \bigcup_{K' \in \delta_k^\epsilon\ K} [\![K']\!]_K$$

Proof. We prove this theorem by induction on the structure of K.

- **Case** Pkr:
  Since $\delta_k\ Pkr = \emptyset$ and $\delta_k^\epsilon\ Pkr = \{Pkr\}$, we have:
  $$[\![Pkr]\!]_K = \emptyset \cup [\![Pkr]\!]_K$$
- **Case** $K_1 + K_2$:
  By induction we have:
  $$[\![K_1]\!]_K = \bigcup_{(K_{11}, K_{12}) \in \delta_k\ K_1} [\![K_{11}]\!]_K \circ [\![K_{12}]\!]_K \cup \bigcup_{K_1' \in \delta_k^\epsilon\ K_1} [\![K_1']\!]_K$$
  $$[\![K_2]\!]_K = \bigcup_{(K_{21}, K_{22}) \in \delta_k\ K_2} [\![K_{21}]\!]_K \circ [\![K_{22}]\!]_K \cup \bigcup_{K_2' \in \delta_k^\epsilon\ K_2} [\![K_2']\!]_K$$
  Since
  $$\delta_k\ (K_1 + K_2) = \delta_k\ K_1 \cup \delta_k\ K_2$$
  $$\delta_k^\epsilon\ (K_1 + K_2) = \delta_k^\epsilon\ K_1 \cup \delta_k^\epsilon\ K_2$$
  Thus
  $$\begin{aligned}
  [\![K_1 + K_2]\!]_K &= [\![K_1]\!]_K \cup [\![K_2]\!]_K \\
  &= \bigcup_{(K_{11}, K_{12}) \in \delta_k\ K_1} [\![K_{11}]\!]_K \circ [\![K_{12}]\!]_K \cup \bigcup_{K_1' \in \delta_k^\epsilon\ K_1} [\![K_1']\!]_K \\
  &\quad \cup \bigcup_{(K_{21}, K_{22}) \in \delta_k\ K_2} [\![K_{21}]\!]_K \circ [\![K_{22}]\!]_K \cup \bigcup_{K_2' \in \delta_k^\epsilon\ K_2} [\![K_2']\!]_K \\
  &= \bigcup_{(K_1', K_2') \in \delta_k\ (K_1 + K_2)} [\![K_1']\!]_K \circ [\![K_2']\!]_K \cup \bigcup_{K' \in \delta_k^\epsilon\ (K_1 + K_2)} [\![K']\!]_K
  \end{aligned}$$

- **Case** $K_1 \circ K_2$:
  By induction we have:

$$[\![K_1]\!]_K = \bigcup_{(K_{11},K_{12})\in\delta_k\ K_1} [\![K_{11}]\!]_K \circ [\![K_{12}]\!]_K \cup \bigcup_{K_1'\in\delta_k^\epsilon\ K_1} [\![K_1']\!]_K$$

$$[\![K_2]\!]_K = \bigcup_{(K_{21},K_{22})\in\delta_k\ K_2} [\![K_{21}]\!]_K \circ [\![K_{22}]\!]_K \cup \bigcup_{K_2'\in\delta_k^\epsilon\ K_2} [\![K_2']\!]_K$$

  Since

$$\begin{aligned}\delta_k\ (K_1 \circ K_2) = &\ \{(K_{11}, K_{12} \circ K_2) \mid (K_{11}, K_{12}) \in \delta_k\ K_1\} \\ &\cup \{(K_1' \circ K_{21}, K_{22}) \mid K_1' \in \delta_k^\epsilon\ K_1 \wedge (K_{21}, K_{22}) \in \delta_k\ K_2\} \\ \delta_k^\epsilon\ (K_1 \circ K_2) = &\ \delta_k^\epsilon\ K_1 \circ \delta_k^\epsilon\ K_2\end{aligned}$$

  Thus

$$\begin{aligned}[\![K_1 \circ K_2]\!]_K = &\ [\![K_1]\!]_K \circ [\![K_2]\!]_K \\ = &\ \bigcup_{(K_{11},K_{12})\in\delta_k\ K_1} ([\![K_{11}]\!]_K \circ [\![K_{12}]\!]_K) \circ [\![K_2]\!]_K \cup \bigcup_{K_1'\in\delta_k^\epsilon\ K_1} ([\![K_1']\!]_K) \circ [\![K_2]\!]_K \\ = &\ \bigcup_{(K_{11},K_{12})\in\delta_k\ K_1} [\![K_{11}]\!]_K \circ [\![K_{12} \circ K_2]\!]_K \\ &\cup \bigcup_{K_1'\in\delta_k^\epsilon\ K_1} ([\![K_1']\!]_K) \circ \Big(\bigcup_{(K_{21},K_{22})\in\delta_k\ K_2} [\![K_{21}]\!]_K \circ [\![K_{22}]\!]_K \cup \bigcup_{K_2'\in\delta_k^\epsilon\ K_2} [\![K_2']\!]_K\Big) \\ = &\ \bigcup_{(K_{11},K_{12})\in\delta_k\ K_1} [\![K_{11}]\!]_K \circ [\![K_{12} \circ K_2]\!]_K \cup \bigcup_{K_1'\in\delta_k^\epsilon\ K_1 \wedge (K_{21},K_{22})\in\delta_k\ K_2} \\ &[\![K_1' \circ K_{21}]\!]_K \circ [\![K_{22}]\!]_K \cup \bigcup_{K_1'\in\delta_k^\epsilon\ K_1 \wedge K_2'\in\delta_k^\epsilon\ K_2} [\![K_1' \circ K_2']\!]_K)\end{aligned}$$

- **Case** K^*: By induction we have

$$[\![K]\!]_K = \bigcup_{(K_1,K_2)\in\delta_k\ K} [\![K_1]\!]_K \circ [\![K_2]\!]_K \cup \bigcup_{K'\in\delta_k^\epsilon\ K} [\![K']\!]_K$$

  Using induction over i, we compute:

$$[\![K^i]\!]_K = \delta_k^{\epsilon(i)}\ (K) \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \Big(\bigcup_{0\le j\le i-1} \Big(\bigcup_{K'\in\delta_k^{\epsilon(j)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^{i-j-1}]\!]_K\Big)\Big)$$

(1) Case $i = 0$.

$$[\![K^0]\!]_K = [\![1]\!]_K = [\![\delta_k^{\epsilon(0)}\ \ (K)]\!]_K \cup \emptyset$$

(2) Case $i = i' + 1$

$$
\begin{aligned}
[\![K^i]\!]_K = & \quad [\![K^{i'}]\!]_K \circ [\![K]\!]_K \\
= & \quad \delta_k^{\epsilon(i')}(K) \circ [\![K]\!]_K \\
& \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \Big(\bigcup_{0\le j\le i'-1} \Big(\bigcup_{K'\in\delta_k^{\epsilon(j)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^{i'-j-1}K]\!]_K \Big)\Big) \\
= & \quad \delta_k^{\epsilon(i')}(K) \circ \Big(\bigcup_{(K_1,K_2)\in\delta_k\ K} [\![K_1]\!]_K \circ [\![K_2]\!]_K \cup \bigcup_{K'\in\delta_k^{\epsilon}\ K} [\![K']\!]_K \Big) \\
& \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \Big(\bigcup_{0\le j\le i-2} \Big(\bigcup_{K'\in\delta_k^{\epsilon(j)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^{i-j-1}]\!]_K \Big)\Big) \\
= & \quad \delta\epsilon(i)_k(K) \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \bigcup_{K'\in\delta_k^{\epsilon(i-1)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2]\!]_K \Big) \\
& \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \Big(\bigcup_{0\le j\le i-2} \Big(\bigcup_{K'\in\delta_k^{\epsilon(j)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^{i-j-1}]\!]_K \Big)\Big) \\
= & \quad \delta_k^{\epsilon(i)}(K) \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \Big(\bigcup_{0\le j\le i-1} \Big(\bigcup_{K'\in\delta_k^{\epsilon(j)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^{i-j-1}]\!]_K \Big)\Big)
\end{aligned}
$$

Since

$$\delta_k(K^*) = \{(K_0 \circ K_1, K_2 \circ K^*) \mid K_0 \in \delta_k^{\epsilon} K \wedge (K_1, K_2) \in \delta_k K\}$$

$$\delta_k^{\epsilon}(K^*) = \bigcup_i \delta_k^{\epsilon(i)}(K) \text{ where } \delta_k^{\epsilon(0)}(K) = \{1\} \text{ and } \delta_k^{\epsilon(i+1)}(K) = \delta_k^{\epsilon(i)}(K) \circ \delta_k^{\epsilon}(K)$$

Thus

$$
\begin{aligned}
[\![K^*]\!]_K = & \quad \bigcup_i [\![K^i]\!]_K \\
= & \quad \bigcup_i (\delta_k^{\epsilon(i)}(K) \cup \bigcup_{(K_1,K_2)\in\delta_k\ K} \Big(\bigcup_{0\le j\le i-1} \Big(\bigcup_{K'\in\delta_k^{\epsilon(j)}\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^{i-j-1}]\!]_K \Big)\Big)) \\
= & \quad \delta_k^{\epsilon}(K^*) \cup \bigcup_{K'\in\delta_k^{\epsilon}\ (K^*)\wedge(K_1,K_2)\in\delta_k\ K} [\![K' \circ K_1]\!]_K \circ [\![K_2 \circ K^*]\!]_K
\end{aligned}
$$

- **Case** *dup*:
  Since $\delta_k\ dup = \{(1, dup)\}$ and $\delta_k^{\epsilon}\ dup = \emptyset$, we have:
  $$[\![dup]\!]_K = [\![dup]\!]_K \cup \emptyset$$

Thus, the theorem is proved. □

Theorem A.2. *For all $K \in K$, suppose $\mathcal{A}(K)$ is the automaton constructed by the algorithm above. Then, for this $\mathcal{A}(K)$, we have:*

$$(K, x_0) \xrightarrow{w} (\epsilon, x_n) \iff x_0 w \in [\![K]\!]_K$$

Proof. We prove the theorem by induction on the length of w.

- **Base case 1:** $w = \epsilon$. Since $K \neq \epsilon$ and $[\![K]\!]_K$ only contains traces with length ≥ 2, thus $w = \epsilon$ is not possible.

- **Base case 2:** $w = x_1$.
  - **Necessary direction:** In this case, we have:

    $$(s_0, x_0) \xrightarrow{x_1} (\epsilon, x_1)$$

    Since $s_0 = K$, we can conclude that $x_0x_1 \in \bigcup_{K' \in \delta_k^\epsilon\ K} [\![K']\!]_K \subseteq [\![K]\!]_K$
  - **Sufficient direction:** If $x_0x_1 \in [\![K]\!]_K$, then $x_0x_1 \in \bigcup_{K' \in \delta_k^\epsilon\ K} [\![K']\!]_K$ for only δ_ϵ generate length 2 traces.
    For this case, since $\Delta\ K\ \epsilon = \bigcup_{K' \in \delta_k^\epsilon\ K} [\![K']\!]_K$, we directly have:

    $$(K, x_0) \xrightarrow{x_1} (\epsilon, x_1)$$

- **Inductive step:** $w = x_1w'$, where x_1 is the first symbol of w and w' is the remaining string. By construction, the transition Δ is defined as:
  - $\Delta\ (K_1, K_2) = \bigcup_{(K_1', dup\ K_2) \in \delta_k\ K_1} [\![K_1']\!]_K$
  - $\Delta\ (K, \epsilon) = \bigcup_{K' \in \delta_k^\epsilon\ K} [\![K']\!]_K$

  From Theorem A.1, we know:

  $$[\![K]\!]_K = \bigcup_{(K_1, K_2) \in \delta_k\ K} [\![K_1]\!]_K \circ [\![K_2]\!]_K \cup \bigcup_{K' \in \delta_k^\epsilon\ K} [\![K']\!]_K$$

  Here, K_1 and K' contain no dup, and $K_2 = dup\ K_2'$ for some K_2'.
  - **Necessary direction:** If $(K, x_0) \xrightarrow{x_1w} (\epsilon, x_n)$, then there exists $K_2 = dup\ K_2'$ such that $(K_2', x_1) \xrightarrow{w} (\epsilon, x_n)$ and $(x_0, x_1) \in \Delta\ (K, K')$.
    For this case, $x_0x_1 \in \bigcup_{(K_1, dup\ K_2') \in \delta_k\ K} [\![K_1]\!]_K$, and the automaton transitions to state K_2' with input x_1. By induction, $x_1w \in [\![K_2']\!]_K$. Thus, $x_0x_1w \in \bigcup_{(K_1, dup\ K_2') \in \delta_k\ K} [\![K_1]\!]_K \circ [\![dup\ \ K_2']\!]_K \subseteq [\![K]\!]_K$, proving the case.
  - **Sufficient direction:** If $x_0x_1w \in [\![K]\!]_K$, then $x_0x_1 \in [\![K_1]\!]_K, x_1w \in [\![K_2']\!]_K$, and $(K_1, dup\, K_2') \in \delta_k\ K$ by trace length classification.
    By induction, we have: $(K_2', x_1) \xrightarrow{w} (\epsilon, x_n)$.
    By the definition of Δ, we have:

    $$(K, x_0) \xrightarrow{x_1} (K_2', x_1)$$

    Thus

    $$(K, x_0) \xrightarrow{x_1w} (\epsilon, x_n)$$

    Hence, the case is proven.

□

Theorem A.3. *For the intermediate δ_r and δ_r^ϵ used in the construction of $\mathcal{A}(R)$, we have:*

$$[\![R]\!]_R = \bigcup_{(R_1, R_2) \in \delta_r\ R} [\![R_1]\!]_R \cdot [\![R_2]\!]_R \cup \bigcup_{R' \in \delta_r^\epsilon\ R} [\![R']\!]_R$$

Proof. We prove this theorem by induction on the structure of R.

- **Case** $Filter(PkR)$:
  Since $\delta_r\ Filter(PkR) = \{\}$ and $\delta_r^\epsilon\ Filter(PkR) = \{Filter(PkR)\}$, we have:
  $$[\![Filter(PkR)]\!]_R = \emptyset \cup [\![Filter(PkR)]\!]_R.$$
- **Case** $Map(PkR, K)$, $Delete(K)$, and $Insert(K)$:
  Since $\delta_r\ R = \emptyset$ and $\delta_r^\epsilon\ R = \{R\}$, we have:
  $$[\![R]\!]_R = [\![R]\!]_R \cup \emptyset.$$
- **Case** $R_1 + R_2$, and $R_1 \cdot R_2$:
  Follows the same proof as Theorem A.1.
- **Case** R^*:
  Note that $[\![R^*]\!]_R = \bigcup_{n\geq 0} [\![R^n]\!]_R$, where $R^0 = Filter(havoc)$, $R^{n+1} = R^n \cdot R$
  And that
  – $\delta_r^\epsilon\ (R^*) = \{Filter(havoc)\}$
  – $\delta_r\ (R^*) = \{(R_1, R_2 \cdot R^*) \mid (R_1, R_2) \in \delta_r\ R\} \cup \{(R', R^*) \mid R' \in \delta_r^\epsilon\ R\}$
  Follows the same induction process in Theorem A.1, it is easy to show
  $$[\![R^+]\!]_R = \bigcup_{(R_1,R_2)\in\delta_r\ R} [\![R_1]\!]_R \cdot [\![R_2]\!]_R$$
  while $[\![R^0]\!]_R = [\![Filter(\mathsf{havoc})]\!]_R$. Therefore, this case is proved.

Thus, the theorem is proved. □

Theorem A.4. *For all $R \in R$, suppose $\mathcal{A}(R)$ is the transducer constructed by the algorithm above. Then, for this $\mathcal{A}(R)$, we have:*

$$(R, (x, y)) \xrightarrow{(w,z)} (\epsilon \times \epsilon, (x', y')) \iff (xw, yz) \in [\![R]\!]_R.$$

Proof. We prove the theorem by induction.

- **Necessary direction:** If $(R, (x, y)) \xrightarrow{(w,z)} (\epsilon \times \epsilon, (x', y'))$, then there are four cases to consider:
  (1) **Case 1:**The inductive step involves a Δ_S transition. Besides, we have $w = x_1 w'$ and $z = y_1 z'$ such that either:
  $$\Delta_S\ (R_1, R_2) = \bigcup_{(Map(PkR,R_1'),R_2)\in\delta_r\ R_1} \{(x, y), (x', y'))|(xx', yy')) \in [\![Map(PkR, R_1')]\!]_R\}$$
  or
  $$\Delta_S\ (R, (\epsilon \times \epsilon)) = \bigcup_{Map(PkR,R')\in\delta_r^\epsilon\ R} \{(x, y), (x', y'))|(xx', yy')) \in [\![Map(PkR, R')]\!]_R\}$$
  – If the next state is R_2, by induction we have there exists x_1, y_1, w', z' such that:
  $$(x_1 w', y_1 z') \in [\![R_2]\!]_R.$$
  $$(x_0 x_1, y_0 y_1) \in [\![Map(PkR, R')]\!]_R.$$
  Thus:
  $$\begin{aligned} (x_0 x_1 w', y_0 y_1 z') &= (x_0 x_1, y_0 y_1) \cdot (x_1 w', y_1 z') \\ &\in [\![Map(PkR, R')]\!]_R \cdot [\![R_2]\!]_R \\ &\subseteq [\![R]\!]_R \end{aligned}$$

– If the next state is $\epsilon \times \epsilon$, then:

$$(x_0x_1, y_0y_1) \in \bigcup_{R' \in \delta_r^\epsilon\ R} [\![R']\!]_R \subseteq [\![R]\!]_R$$

proving the case.

(2) **Case 2,3 and 4:** The inductive step involves a Δ_E , Δ_L or Δ_R transition. The reasoning is identical to **Case 1**.

- **Sufficient direction:** If $(xw, yz) \in [\![R]\!]_R$, we prove this by induction on the sum of lengths of w and z. Since:

$$[\![R]\!]_R = \bigcup_{(R_1,R_2)\in\delta_r\ R} [\![R_1]\!]_R \cdot [\![R_2]\!]_R \cup \bigcup_{R' \in \delta_r^\epsilon\ R} [\![R']\!]_R$$

we analyze the following cases:

(1) **Case 1:** $(xw, yz) \in \bigcup_{R' \in \delta_r^\epsilon\ R} [\![R']\!]_R$.

then we have $(R, (x, y)) \xrightarrow{(w,z)} (\epsilon \times \epsilon, (x', y'))$ for all the Δ_S, Δ_E , Δ_L or Δ_R.

(2) **Case 2:** $(xw, yz) \in [\![R_1]\!]_R \cdot [\![R_2]\!]_R$, where $R_1 = Map(PkR, R_1')$ and $(R_1, R_2) \in \delta_r\ R$. Suppose $w = x_1w'$ and $z = y_1z'$. Then:

$$(xx_1, yy_1) \in [\![R_1]\!]_R, \quad (x_1w', y_1z') \in [\![R_2]\!]_R.$$

We directly have

$$(R_1, (x, y)) \xrightarrow{(x_1,y_1)} (R_2, (x_1, y_1)).$$

By induction we have:

$$(R_2, (x_1, y_1)) \xrightarrow{(w',z')} (\epsilon \times \epsilon, (x', y')).$$

Hence:

$$(R_2, (x, y)) \xrightarrow{(w,z)} (\epsilon \times \epsilon, (x', y')).$$

(3) **Case 3:** $(xw, yz) \in [\![R_1]\!]_R \cdot [\![R_2]\!]_R$, where $R_1 = Insert(K)$, $R_1 = Delete(K)$ or $R_1 = Filter(PkR)$. The reasoning is similar to **Case 2**.

□

Theorem A.5 (Soundness of Automata). *For $\mathcal{A}(K)$ and $\mathcal{A}(R)$ induced by K and R, we have:*

$$L(\mathcal{A}(K)) = [\![K]\!]_K \quad \textit{and} \quad L(\mathcal{A}(R)) = [\![R]\!]_R.$$

Proof. This follows directly from the theorems above. □

A.3 Number of States

We begin by defining the length functions $l(K)$ and $l(R)$, which serve as upper bounds on the number of distinct states in the automata generated from NetKAT and Relational NetKAT expressions using the derivative-based construction.

NetKAT Expression Length $l(K)$: We define a syntactic length function $l(K)$ on NetKAT expressions to provide a loose upper bound on the number of states generated during automaton construction. The definition is as follows:

- $l(\mathsf{PkR}) = 1$
- $l(K_1 + K_2) = l(K_1) + l(K_2) + 1$
- $l(K_1 \cdot K_2) = l(K_1) + l(K_2) + 1$
- $l(K^*) = l(K) + 1$

- $l(dup) = 2$

Note that we assign $l(dup) = 2$ by convention, since in our derivative-based construction, we treat dup as syntactic sugar for the expression $dup \cdot \overline{1}$. Although this convention may be unconventional, it suffices for the purpose of showing that the number of automaton states is bounded by the structural size of the expression.

Relational NetKAT Expression Length $l(R)$:

- $l(Filter(PkR)) = 2$
- $l(Delete(K)) = l(K) + 1$
- $l(Insert(K)) = l(K) + 1$
- $l(Map(PkR, K)) = l(K) + 2$
- $l(R_1 + R_2) = l(R_1) + l(R_2) + 1$
- $l(R_1 \cdot R_2) = l(R_1) + l(R_2) + 1$
- $l(R^*) = l(R) + 1$

We write $E(K)$ and $E(R)$ to denote the sets of derivative expressions (i.e., states) generated by the derivative construction for K and R, respectively.

We also define the following normalization conventions:

$$\epsilon \circ K = K, \quad \epsilon \times \epsilon \cdot R = R$$

We now prove bounds on the size of the state space generated.

Theorem A.6. *For all NetKAT expressions K, we have:*

$$|E(K)| \leq l(K) + 1$$

Proof. By structural induction on K:

- **Case** PkR: $E(K) = \{\mathsf{PkR}, \epsilon\}$, so the bound holds: $2 \leq 1 + 1$.
- **Case** dup: $E(K) = \{dup, \overline{1}, \epsilon\}$, also satisfying the bound.
- **Case** $K = K_1 + K_2$: By inductive hypothesis,

$$|E(K)| \leq |E(K_1)| + |E(K_2)| \leq l(K_1) + 1 + l(K_2) + 1 = l(K) + 1$$

- **Case** $K = K_1 \cdot K_2$: From the derivative semantics,

$$E(K) \subseteq E(K_1) \cdot \{K_2\} \cup E(K_2)$$

  Hence,

$$|E(K)| \leq |E(K_1)| + |E(K_2)| \leq l(K_1) + l(K_2) + 2 = l(K) + 1$$

- **Case** $K = K'^*$:

$$E(K) \subseteq E(e') \cdot \{K'^*\} \cup \{\epsilon\}$$

  Thus,

$$|E(K)| \leq |E(K')| + 1 \leq l(K') + 2 = l(K) + 1$$

This completes the inductive proof. □

Theorem A.7. *For all relational NetKAT expressions R, we have:*

$$|E(R)| \leq l(R) + 1$$

Proof. We proceed by structural induction on R.

- **Case** $R = Filter(PkR)$: The derivative yields at most two expressions: the filter itself and ϵ. So $|E(R)| \leq 2 = l(R)$, which satisfies the bound.

- **Case** $R = \mathit{Delete}(K)$: The set of derivatives satisfies:

$$E(R) \subseteq \{\mathit{Delete}(K') \mid K' \in E(K)\} \cup \{\epsilon \times \epsilon\}$$

  Thus,

$$|E(R)| \leq |E(K)| + 1 \leq l(K) + 1 + 1 = (l(K) + 1) + 1$$

  Since $l(R) = l(K) + 1$, thus we get the proof.
- **Case** $R = \mathit{Insert}(K)$ **or** $R = \mathit{Map}(PkR, K)$: Same argument as for delete.
- **Case** $R = R_1 + R_2$, $R = R_1 \cdot R_2$ **or** $R = R'^*$: Same as Theorem A.6

Hence the bound $|E(R)| \leq l(R) + 1$ holds in all cases. □

A.4 Cross Product and Synchronization Construction

The construction of the intermediate cross-product transducer $\mathcal{A}(R)|_{\mathcal{A}(K)} = (S_{kr}, S_{kr0}, S_{krf}, \Delta'_S, \Delta'_L, \Delta'_R, \Delta'_E)$, derived from the NetKAT automaton $\mathcal{A}(K) = (S_k, S_{k0}, S_{kf}, \Delta)$ and the NetKAT transducer $\mathcal{A}(R) = (S_r, S_{r0}, S_{rf}, \Delta_S, \Delta_L, \Delta_R, \Delta_E)$, has been described in Section 4. We now describe the synchronization phase in detail and explain how we eliminate unsynchronized transitions to obtain a standard two-tape transducer.

Transition Functions. The goal is to construct a synchronized transition function Δ^*_S that simulates the effects of Δ'_R, Δ'_L, and Δ'_E using only transitions that advance both tapes. To achieve this, we define two auxiliary transition sets:

- **Right-only transition simulation:**

$$\Delta_{SR}(s_1, s_2) = \left\{((x_1, y_1), (x_1, y_2)) \mid (y_1, y_2) \in \Delta'_R(s_1, s_2)\right\}$$

  This set simulates second-tape-only transitions by fixing the first-tape packet and moving the second-tape packet. This yields a transition that is synchronized syntactically, even though the first tape logically remains unchanged.
  We then define the extended synchronized transition set as:

$$\Delta'_{SR} = \Delta'_S \cup \Delta_{SR}$$

- **Left-only and ϵ transition simulation:**

$$\Delta_{LE}(s_1, s_2) = \left\{ \begin{array}{l} ((x_1, y_1), (x_2, y_1)) \mid (x_1, x_2) \in \Delta'_L(s_1, s_2) \\ \cup\ ((x_1, y_1), (x_1, y_1)) \mid (x_1, y_1) \in \Delta'_E(s_1, s_2) \end{array} \right\}$$

  This captures both first-tape-only transitions and ϵ transitions. Since neither advances the second tape, we fix the second-tape component while applying transformations or identity steps to the first tape.

Explanation.

- The set Δ'_{SR} augments the base synchronized transition set Δ'_S by incorporating simulated right-only moves. These transitions effectively allow the second tape to move forward while syntactically advancing both tapes.
- The set Δ_{LE} models silent transitions where only the first tape moves or neither tape moves. These transitions must be eliminated through standard ϵ-closure procedures.

Transitive Closure of Δ'_E and Δ'_L Transitions. To properly simulate sequences of Δ_{LE} transitions, we compute their transitive closure:

$$\Delta^0_{LE}(s, s) = \{((x, y), (x, y))\}$$

$$\Delta^{i+1}_{LE}(s_1, s_3) = \left\{((x_1, y), (x_3, y)) \mid \exists s_2.\ ((x_1, y), (x_2, y)) \in \Delta^i_{LE}(s_1, s_2) \wedge ((x_2, y), (x_3, y)) \in \Delta_{LE}(s_2, s_3)\right\}$$

$$\Delta^*_{LE}(s_1, s_2) = \bigcup_{i \geq 0} \Delta^i_{LE}(s_1, s_2)$$

Composite Transition Function. Using Δ'_{SR} and Δ^*_{LE}, we define the final synchronized transition function Δ^*_S as follows:

- For an initial state $s_0 \in S_{kr0}$:

$$\Delta^*_S(s_0, s_3) = \left\{ ((x_0, y_0), (x_3, y_3)) \;\middle|\; \begin{array}{l} \exists s_1, s_2.\ ((x_0, y_0), (x_1, y_1)) \in \Delta^*_{LE}(s_0, s_1) \\ \wedge\ ((x_1, y_1), (x_2, y_2)) \in \Delta'_{SR}(s_1, s_2) \\ \wedge\ ((x_2, y_2), (x_3, y_3)) \in \Delta^*_{LE}(s_2, s_3) \end{array} \right\}$$

- For all other states $s_0 \notin S_{kr0}$:

$$\Delta^*_S(s_0, s_2) = \left\{ ((x_0, y_0), (x_2, y_2)) \;\middle|\; \begin{array}{l} \exists s_1.\ ((x_0, y_0), (x_1, y_1)) \in \Delta'_{SR}(s_0, s_1) \\ \wedge\ ((x_1, y_1), (x_2, y_2)) \in \Delta^*_{LE}(s_1, s_2) \end{array} \right\}$$

This composite transition function replaces all unsynchronized transitions with semantically equivalent synchronized ones. As shown in the synchronization theorem, this ensures that the projected trace language on the second tape is preserved exactly, which suffices for relational verification.

A.5 Proof of Correctness of Composition

In this section, we prove the correctness of the composition and synchronization.

Theorem A.8 (Correctness of Δ'_{SR}). *$((x_1, y_1), (x_2, y_2)) \in \Delta'_{SR}\ ((s_{k1}, s_{r1}), (s_{k2}, s_{r2}))$ if and only if one of the following conditions holds:*

(1) $(s_{k1}, x_1) \xrightarrow{x_2} (s_{k2}, x_2)$ *and* $(s_{r1}, (x_1, y_1)) \xrightarrow{(x_2, y_2)} (s_{r2}, (x_2, y_2))$

(2) $s_{k1} = s_{k2}$ *and* $(s_{r1}, (x_1, y_1)) \xrightarrow{(\epsilon, y_2)} (s_{r2}, (x_2, y_2))$

Proof. From the definition of Δ_{kr0}:

$$\Delta_{kr0}\ ((s_k, s_r), (s'_k, s'_r)) = \{((x_1, y_1), (x_2, y_2)) \mid (x_1, x_2) \in \Delta\ (s_k, s'_k) \wedge ((x_1, x_2), (y_1, y_2)) \in \Delta_S\ (s_r, s'_r)\}$$
$$\cup \{((x_1, y_1), (x_2, y_2)) \mid s_k = s'_k \wedge ((x_1, y_1), (x_2, y_2)) \in \Delta_R\ (s_r, s'_r)\}.$$

By case analysis of the above definition, we can directly verify the two conditions stated in the theorem. Thus, the proof is complete. □

Theorem A.9 (Correctness of Δ^*_{LE}). *$((x_1, y_1), (x_{n+1}, y_{n+1})) \in \Delta^*_{LE}\ ((s_{k1}, s_{r1}), (s_{k(n+1)}, s_{r(n+1)}))$, if and only if there exists w, such that:*

$$(s_{k1}, x_1) \xrightarrow{w} (s_{k(n+1)}, x_{n+1}) \quad \textit{and} \quad (s_{r1}, (x_1, y_1)) \xrightarrow{(w, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y_{n+1})).$$

Proof. We only need to show that
$((x_n, y), (x_{n+1}, y)) \in \Delta^n_{LE}\ ((s_{k1}, s_{r1}), (s_{k(n+1)}, s_{r(n+1)}))$ if and only if there exists w such that:

$$(s_{k1}, x_1) \xrightarrow{w} (s_{k(n+1)}, x_{n+1}) \quad \text{and} \quad (s_{r1}, (x_1, y)) \xrightarrow{(w, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y)).$$

- **Necessary Direction:** We prove this claim by induction on $(x_1, x_{n+1}) \in \Delta^n_{LE}\ ((s_{k1}, s_{r1}), (s_{k(n+1)}, s_{r(n+1)}))$.
  - **Base Case:** $n = 0$.
    For $n = 0$, we have:

$$((x_1, y), (x_1, y)) \in \Delta^0_{LE}\ ((s_{k1}, s_{r1}), (s_{k1}, s_{r1})).$$

    On the other hand:

$$(s_{k1}, x_1) \xrightarrow{\epsilon} (s_{k1}, x_1) \quad \text{and} \quad (s_{r1}, (x_1, y)) \xrightarrow{(\epsilon, \epsilon)} (s_{r1}, (x_1, y)).$$

Thus the claim holds for the base case.

– **Inductive Step:** Assume the claim holds for n'. We prove it for $n = n' + 1$.
By definition:

$$\Delta_{LE}^{n+1}(s_1, s_3) = \{((x_1, y), (x_3, y)) \mid \exists s_2, x_2, ((x_1, y), (x_2, y)) \in \Delta_{LE}^{n}(s_1, s_2) \wedge ((x_2, y), (x_3, y)) \in \Delta_{LE}(s_2, s_3)\}$$

Suppose:

$$((x_1, y), (x_{n+1}, y)) \in \Delta_{LE}^{n+1}((s_{k1}, s_{r1}), (s_{k(n+1)}, s_{r(n+1)})).$$

Then there exist $s_{kn'}, s_{rn'}, x_{n'}$ such that:

$$((x_1, y), (x_n, y)) \in \Delta_{LE}^{n}((s_{k1}, s_{r1}), (s_{kn}, s_{rn}))$$

And

$$((x_n, y), (x_{n+1}, y)) \in \Delta_{LE}((s_{kn}, s_{rn}), (s_{k(n+1)}, s_{r(n+1)})).$$

By the induction hypothesis, there exists w' such that:

$$(s_{k1}, x_1) \xrightarrow{w'} (s_{kn}, x_n) \quad \text{and} \quad (s_{r1}, (x_1, y)) \xrightarrow{(w', \epsilon)} (s_{rn}, (x_n, y)).$$

From $((x_n, y), (x_{n+1}, y)) \in \Delta_{LE}((s_{kn}, s_{rn}), (s_{k(n+1)}, s_{r(n+1)}))$, we know either:

$$(s_{kn}, x_n) \xrightarrow{x_{n+1}} (s_{k(n+1)}, x_{n+1}) \quad \text{and} \quad (s_{rn}, (x_n, y)) \xrightarrow{(x_{n+1}, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y)).$$

or

$$s_{kn} = s_{k(n+1)} \quad \text{and} \quad x_n = x_{n+1} \quad \text{and} \quad (s_{rn}, (x_n, y)) \xrightarrow{(\epsilon, \epsilon)} (s_{r(n+1)}, (x_n, y)).$$

Combining these, let $w = w' x_{n+1}$ in the first case and $w = w'$ in the second case. Then:

$$(s_{k1}, x_1) \xrightarrow{w} (s_{k(n+1)}, x_{n+1}) \quad \text{and} \quad (s_{r1}, (x_1, y)) \xrightarrow{(w, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y)).$$

Thus, the claim holds for the necessary direction.

- **Sufficient Direction:**
We prove this claim by induction on $(s_{r1}, (x_1, y)) \xrightarrow{(w, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y))$.

– **Base Case:** $(s_{r1}, (x_1, y)) \xrightarrow{(\epsilon, \epsilon)} (s_{r1}, (x_1, y))$.
For $w = \epsilon$, we have:

$$(s_{k1}, x_1) \xrightarrow{\epsilon} (s_{k1}, x_1)$$

On the other hand:

$$((x_1, y), (x_1, y)) \in \Delta_{LE}^{0}((s_{k1}, s_{r1}), (s_{k1}, s_{r1})).$$

Thus the claim holds for the base case.

– **Inductive Step:**
Suppose there exists w such that:

$$(s_{k1}, x_1) \xrightarrow{w} (s_{k(n+1)}, x_{n+1}) \quad \text{and} \quad (s_{r1}, (x_1, y)) \xrightarrow{(w, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y)).$$

We do induction on $(s_{r1}, (x_1, y)) \xrightarrow{(w, \epsilon)} (s_{r(n+1)}, (x_{n+1}, y))$. Then there are two cases:

$$(s_{kn}, x_n) \xrightarrow{x_{n+1}} (s_{k(n+1)}, x_{(n+1)}) \quad \text{and} \quad (s_{rn}, (x_n, y)) \xrightarrow{(x_{n+1}, \epsilon)} (s_{r(n+1)}, (x_{(n+1)}, y)).$$

or

$$(s_{kn}, x_n) \xrightarrow{\epsilon} (s_{kn}, x_n) \quad \text{and} \quad (s_{rn}, (x_n, y)) \xrightarrow{(\epsilon, \epsilon)} (s_{r(n+1)}, (x_n, y)).$$

By the induction hypothesis:

$$((x_1, y), (x_n, y)) \in \Delta_{LE}^{n}((s_{k1}, s_{r1}), (s_{kn}, s_{rn})).$$

From both two cases, we know:

$$((x_n, y), (x_{n+1}, y)) \in \Delta_{LE}\ ((s_{kn}, s_{rn}), (s_{k(n+1)}, s_{r(n+1)})).$$

Hence:

$$((x_n, y), (x_{n+1}, y)) \in \Delta_{LE}^{n+1}\ ((s_{k1}, s_{r1}), (s_{k(n+1)}, s_{r(n+1)})).$$

This completes the sufficient direction.

Thus, the theorem is proven. □

Theorem A.10 (Composition). *Let $\mathcal{A}(K) = (S_k, S_{k0}, S_{kf}, \Delta), \mathcal{A}(R) = (S_r, S_{r0}, S_{rf}, \Delta_S, \Delta_L, \Delta_R, \Delta_E)$, and generates a synchronized transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s = (S_{kr}, S_{kr0}, S_{krf}, \Delta_S^*, \emptyset, \emptyset, \emptyset)$. then for start state $s_{k0} \in S_{k0}$ and $s_{r0} \in S_{r0}$, and all tape string z with length $n \geq 1$,*
we have for $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$:

$$\exists w', ((s_{k0}, s_{r0}), (x_0, y_0)) \xrightarrow{(w', z)} ((s_{kn}, s_{rn}), (x_n, y_n))$$

if and only if for $\mathcal{A}(K)$ and $\mathcal{A}(R)$:

$$\exists w, \quad (s_{k0}, x_0) \xrightarrow{w} (s_{kn}, x_n) \quad \textit{and} \quad (s_{r0}, (x_0, y_0)) \xrightarrow{(w,z)} (s_{rn}, (x_n, y_n)).$$

Proof. We prove both directions of the equivalence.

- **Necessary Direction:**

  We prove by induction on the sequence $((s_{k0}, s_{r0}), (x_0, y_0)) \xrightarrow{(w', z)} ((s_{kn}, s_{rn}), (x_n, y_n))$.
  - **Base Case:** $n = 1$.
    For the transition:

    $$(s_{kr0}, (x_0, y_0)) \xrightarrow{(x_1, y_1)} ((s_{k1}, s_{r1}), (x_1, y_1)),$$

    by the definition of Δ_{kr}:

    $$\Delta_S^*\ ((s_{k0}, s_{r0}), (s_{k1}, s_{r1})) = \{((x_0, y_0), (x_1, y_1)) \mid \exists s'_{k0}, s'_{r0}, s'_{k1}, s'_{r1}, \text{ such that:}$$
    $$(x_0, x'_0) \in \Delta_{kr1}^*\ ((s_{k0}, s_{r0}), (s'_{k0}, s'_{r0})), \quad ((x'_0, y_0), (x'_1, y_1)) \in \Delta_{kr0}\ ((s'_{k0}, s'_{r0}), (s'_{k1}, s'_{r1})),$$
    $$(x'_1, x_1) \in \Delta_{kr1}^*\ ((s'_{k1}, s'_{r1}), (s_{k1}, s_{r1}))\}.$$

    Applying Theorem A.8 for Δ_{kr0} and Theorem A.9 for Δ_{kr1}^*, there exist w_1, w_2, w_3 such that:
    (1) $(s_{k0}, x_0) \xrightarrow{w_1} (s'_{k0}, x'_0)$,
    (2) $(s_{r0}, (x_0, y_0)) \xrightarrow{(w_1, \epsilon)} (s'_{r0}, (x'_0, y'_0))$,
    (3) $(s'_{k0}, x'_0) \xrightarrow{w_2} (s'_{k1}, x'_1)$,
    (4) $(s'_{r0}, (x'_0, y'_0)) \xrightarrow{(w_2, y_1)} (s'_{r1}, (x'_1, y'_1))$,
    (5) $(s'_{k1}, x'_1) \xrightarrow{w_3} (s_{k1}, x_1)$,
    (6) $(s'_{r1}, (x'_1, y'_1)) \xrightarrow{(w_3, \epsilon)} (s_{r1}, (x_1, y_1))$.
    Thus, the concatenated string $w = w_1 w_2 w_3$ satisfies:

    $$(s_{k0}, x_0) \xrightarrow{w} (s_{k1}, x_1) \quad \text{and} \quad (s_{r0}, (x_0, y_0)) \xrightarrow{(w, y_1)} (s_{r1}, (x_1, y_1)).$$

  - **Inductive Step:** Suppose the claim holds for $n = m$. We prove it for $n = m + 1$.
    For the transition:

    $$((s_{kn}, s_{rn}), (x_n, y_n)) \xrightarrow{(x_{n+1}, y_{n+1})} ((s_{k(n+1)}, s_{r(n+1)}), (x_{n+1}, y_{n+1})),$$

by the same reasoning as the base case, there exists w such that:

$$(s_{kn}, x_n) \xrightarrow{w} (s_{k(n+1)}, x_{n+1}) \quad \text{and} \quad (s_{rn}, (x_n, y_n)) \xrightarrow{(w, y_{n+1})} (s_{r(n+1)}, (x_{n+1}, y_{n+1})).$$

Concatenating this result with the inductive hypothesis for $((s_{k0}, s_{r0}), (x_0, y_0)) \xrightarrow{(x_1,y_1)\cdots(x_n,y_n)} ((s_{kn}, s_{rn}), (x_n, y_n))$, we obtain the desired result ww' for $n = m + 1$.

- **Sufficient Direction:** We prove this by induction on n.
  - **Base Case:** $n = 1$.
    Suppose there exists w such that:

$$(s_{k0}, x_0) \xrightarrow{w} (s_{k1}, x_1) \quad \text{and} \quad (s_{r0}, (x_0, y_0)) \xrightarrow{(w, y_1)} (s_{r1}, (x_1, y_1)).$$

    Decompose w into w_1, w_2, w_3 as in the necessary direction, then $(s_{kr0}, (x_0, y_0)) \xrightarrow{(x_1,y_1)} ((s_{k1}, s_{r1}), (x_1, y_1))$.
  - **Inductive Step:** Assume $n = m + 1$. By the inductive hypothesis, we know:
    (1) $(s_{k0}, x_0) \xrightarrow{w} (s_{km}, x_m)$,
    (2) $(s_{r0}, (x_0, y_0)) \xrightarrow{(w, y_1 \cdots y_m)} (s_{rm}, (x_m, y_m))$.
    Now decompose w into three parts w_1, w_2, and w_3, corresponding to the intermediate transitions:
    (1) $(s_{k0}, x_0) \xrightarrow{w_1} (s_{km}, x_m)$,
    (2) $(s_{r0}, (x_0, y_0)) \xrightarrow{(w_1, y_1 \cdots y_m)} (s_{rm}, (x_m, y_m))$,
    (3) $(s_{km}, x_m) \xrightarrow{w_2} (s_{km'}, x_{m'})$,
    (4) $(s_{rm}, (x_m, y_m)) \xrightarrow{(w_2, y_n)} (s_{rm'}, (x'_m, y_n))$,
    (5) $(s_{km'}, x_{m'}) \xrightarrow{w_3} (s_{kn}, x_n)$,
    (6) $(s_{rm'}, (x_{m'}, y_n)) \xrightarrow{(w_3, \epsilon)} (s_{rn}, (x_n, y_n))$.
    Here, w_2 corresponds to a Δ^*_{LE} transition, and w_3 corresponds to a Δ'_{SR} transition. By the inductive hypothesis, for w_1, there exists:

$$(s_{kr0}, (x_0, y_0)) \xrightarrow{w_m, z_m} ((s_{km}, s_{rm}), (x_m, y_m)).$$

    Combining this result with the transitions for w_2 and w_3, we conclude that:

$$(s_{kr0}, (x_0, y_0)) \xrightarrow{w,z} ((s_{kn}, s_{rn}), (x_n, y_n)).$$

□

Theorem A.11. *Let $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ be the synchronized transducer constructed from the cross-product and synchronization procedures. Then:*

$$\left\{v \;\middle|\; (u, v) \in L((\mathcal{A}(R)|_{\mathcal{A}(K)})^s)\right\} = [\![K \triangleright R]\!]_K.$$

Proof. This follows directly from Theorem A.10. □

A.6 Reachability Optimization

We define a reachability optimization over $T = (S_{kr}, S_{kr0}, S_{krf}, \Delta^*_S, \emptyset, \emptyset, \emptyset)$. Let *Reach* be a function of the type $S \to 2^{Pk \times Pk}$, defined as follows:

- $Reach_0(s) = \{(x, y) \mid s = s_0\}$
- $Reach_{i+1}(s) = \{(x, y) \mid \exists x', y', s', ((x', y'), (x, y)) \in \Delta\ s'\ s \wedge (x', y') \in Reach_i(s')\}$
- $Reach(s) = \bigcup_i Reach_i(s)$

Theorem A.12. *For an transducer* $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$*, let:*

$$\Delta_S^{reach}(s, s') = \{((x, y), (x', y')) \mid (x, y) \in Reach(s) \wedge ((x, y), (x', y')) \in \Delta_S^*(s, s')\}.$$

Then, for the transducer $T^{reach} = (S, S_0, S_f, \Delta_S^{reach}, \emptyset, \emptyset, \emptyset)$*, we have* $L(T) = L(T^{reach})$.

Proof. We prove the equivalence of $L(T)$ and $L(T^{reach})$.

- **Necessary Direction:** $L(T^{reach}) \subseteq L(T)$ By definition, $\Delta'(s, s') \subseteq \Delta(s, s')$, so every trace in T^{reach} is also a valid trace in T. Hence, $L(T^{reach}) \subseteq L(T)$.
- **Sufficient Direction:** $L(T) \subseteq L(T^{reach})$ Suppose in T, from the start state s_0, there is a trace:

  $$(s_0, (x_0, y_0)) \xrightarrow{(x_1 \ldots x_n, y_1 \ldots y_n)} (s_n, (x_n, y_n)).$$

  By definition of *Reach*, $\forall i, (x_i, y_i) \in Reach_i(s_i)$. Thus:

  $$((x_i, y_i), (x_{i+1}, y_{i+1})) \in \Delta_S^{reach}(s_i, s_{i+1}),$$

  which means the same trace exists in T^{reach}. Therefore, $L(T) \subseteq L(T^{reach})$.

Combining both directions, we conclude that $L(T) = L(T^{reach})$. □

A.7 Projection

Theorem A.13 (Correctness of Projection). *Let* $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ *be a synchronous NetKAT transducer. If there exists a bisimulation witness B, then T is* efficiently projectable.

Proof. Suppose the bisimulation witness function we found for T is B of the type $S \to Pk \times Pk$. Then we prove the theorem by induction on the length n of the string u and v (their length are guaranteed to be equal since the transducer is synchronized).

- **Base Case:** $n = 1$. By definition:

  $$|\Delta_S|\ (s, s') = \{(y, y') \mid ((x, y), (x', y')) \in \Delta_S\ (s, s')\}.$$

  Thus, $(y_0, y_1) \in |\Delta_S|\ (s, s')$ if and only if there exist x_0, x_1 such that:

  $$((x_0, y_0), (x_1, y_1)) \in \Delta_S\ (s, s').$$

- **Inductive Step:** Assume the result holds for $n = k$, and prove it for $n = k + 1$. For $n = k$, there exist $x_0, x_1, \ldots, x_k$ such that:

  $$(s_0, (x_0, y_0)) \xrightarrow{(x_1 \ldots x_k, y_1 \ldots y_k)} (s_k, (x_k, y_k)),$$

  if and only if:

  $$(s_0, y_0) \xrightarrow{y_1 y_2 \ldots y_k} (s_k, y_k).$$

  – **Sufficient Direction:** If there exists x_{k+1} and s_{k+1} such that:

  $$((x_k, y_k), (x_{k+1}, y_{k+1})) \in \Delta_S\ (s_k, s_{k+1})$$

  then by definition:

  $$(y_k, y_{k+1}) \in |\Delta_S|\ (s_k, s_{k+1}).$$

  Thus

  $$(s_0, y_0) \xrightarrow{y_1 y_2 \ldots y_{k+1}} (s_{k+1}, y_{k+1}).$$

– **Necessary Direction:** If:

$$(y_k, y_{k+1}) \in |\Delta_S|\ (s_k, s_{k+1}).$$

then there exist x'_k, x_{k+1} such that:

$$((x'_k, y_k), (x_{k+1}, y_{k+1})) \in \Delta_S\ (s_k, s_{k+1}).$$

By condition (2), we have bisimulation witness B summarizing the input:

$$(x'_k, y_k) \in B(s_k).$$

By the inductive hypothesis and condition (3), we have bisimulation witness B summarizing the output:

$$(x_k, y_k) \in B(s_k)$$

Finally, by condition (1), we deduce there exists x'_{k+1} such that:

$$((x_k, y_k), (x'_{k+1}, y_{k+1})) \in \Delta_S\ (s_k, s_{k+1}),$$

Thus:

$$(s_0, (x_0, y_0)) \xrightarrow{(x_1 \ldots x_k x'_{k+1}, y_1 \ldots y_k y_{k+1})} (s_{k+1}, (x'_{k+1}, y_{k+1})),$$

This completes the proof. □

A.8 Partition

Before we prove the correctness of our partitioning strategy, we begin with a structural property of the transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ constructed from a NetKAT expression K and relation R.

Theorem A.14. *For the transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ constructed from K and R, its unique final state is $(\epsilon, \epsilon \times \epsilon)$, and there are no outgoing transitions from this state.*

Proof. The synchronized transducer $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ has the form

$$(\mathcal{A}(R)|_{\mathcal{A}(K)})^s = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset),$$

where the final state set is

$$S_f = S_K \times S_R = \{(\epsilon, \epsilon \times \epsilon)\}.$$

By construction of $\mathcal{A}(K)$ and $\mathcal{A}(R)$, both the state ϵ in $\mathcal{A}(K)$ and the state $\epsilon \times \epsilon$ in $\mathcal{A}(R)$ have no outgoing transitions. Therefore, the product state $(\epsilon, \epsilon \times \epsilon)$ in $(\mathcal{A}(R)|_{\mathcal{A}(K)})^s$ has no outgoing transitions either. This establishes the result. With or without the reachability optimization doesn't affect this proof. □

Now are ready for the proof of partition.

Theorem A.15 (Correctness of Splitting). *Let $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ be a synchronous NetKAT transducer. Without loss of generality, we suppose that final states have no outgoing transitions. Then the split automaton $T^p = (S^p, S_0^p, S_f^p, \Delta_S^p, \emptyset, \emptyset, \emptyset)$ constructed as above satisfies:*

Proof. We first prove that $L(T^p) = L(T)$.

Sufficient Direction: ($L(T^p) \subseteq L(T)$) This follows directly from construction. For all r_1, r_2, we have:

$$\Delta_S^p((s_1, b_1), (s_2, b_2)) \subseteq \Delta_S(s_1, s_2),$$

so every trace in T^p is also a trace in T.

Necessary Direction: ($L(T) \subseteq L(T^p)$) By definition of our witness function, all of the states that have no outgoing transition will be witnessed by $\{Pk \times Pk\}$. Therefore, after splitting, all $s_f \in S_f$ will be resulting in the state $(s_f, \{Pk \times Pk\})$.

We proceed by induction on the length n of all traces ends up in a final state $s_f \in S_f$.

- **Base case (n = 1):** The trace has the form:

$$(s_0, (x_0, y_0)) \xrightarrow{(x_1, y_1)} (s_1, (x_1, y_1)) \text{ and } s_1 \in S_f$$

Since the partition covers the entire relation:

$$\bigcup_{r' \in P(s_0, s_1)} r' = \Delta_S(s_0, s_1),$$

there exists some b_0 such that $(x_0, y_0) \in b_0$ and $(s_0, b_0) \in S_p$. Also, since the final state has no outgoing transitions, it is tagged as $Pk \times Pk$, which does not prune the output. Therefore, we have:

$$((s_0, b_0), (x_0, y_0)) \xrightarrow{(x_1, y_1)} ((s_1, Pk \times Pk), (x_1, y_1)).$$

- **Inductive step (n = m + 1):** By induction, suppose the trace

$$((s_1, b_1), (x_1, y_1)) \xrightarrow{(x_2 \ldots x_{m+1}, y_2 \ldots y_{m+1})} ((s_{m+1}, Pk \times Pk), (x_{m+1}, y_{m+1})) \text{ and } s_{m+1} \in S_f$$

exists in $((\mathcal{A}(R)|_{\mathcal{A}(K)})^s)^p$, with $(x_1, y_1) \in b_1$. Consider the transition

$$(s_0, (x_0, y_0)) \xrightarrow{(x_1, y_1)} (s_1, (x_1, y_1)).$$

By construction, there exists r_0 such that $(x_0, y_0) \in r_0$ and:

$$((s_0, b_0), (x_0, y_0)) \xrightarrow{(x_1, y_1)} ((s_1, b_1), (x_1, y_1)).$$

By chaining, the entire trace exists in $L(T^p)$. Thus, $L(T) \subseteq L(T^p)$.

Bisimiulation Witness: Each state in T^p is explicitly tagged with a bisimiulation witness. Define:

$$B(s, b) = b.$$

We now verify that this satisfies the projection properties:

- **Property (2) and Property (3)— Coverage:** For each transition $((x, y), (x', y')) \in \Delta_{pS}((s, b), (s', b'))$, the input is drawn from b (by construction) and the output is truncated to lie in b', ensuring $(x, y) \in b$ and $(x', y') \in b'$.
- **Property (1) — Bisimulation Witness:** By the partitioning algorithm, suppose the original transition is r while its bisimulation witness is b, then for all $(x_1, y), (x_2, y) \in b$, we have:

$$((x_1, y), (x', y')) \in r \iff ((x_2, y), (x', y')) \in r$$

Therefore

$$((x_1, y), (x', y')) \in \{((x, y), (x', y')) \in r : (x', y') \in b'\}$$

$$\iff ((x_2, y), (x', y')) \in \{((x, y), (x', y')) \in r : (x', y') \in b'\}$$

Thus after the truncation

$$((x_1, y), (x', y')) \in \Delta_S^p((s, b), (s', b')) \iff ((x_2, y), (x', y')) \in \Delta_S^p((s, b), (s', b'))$$

Thus, T^p is efficiently projectable, and its language is equal to that of T, completing the proof. □

A.9 Determinization

The determinization procedure for NetKAT automata has been described in detail by Smolka et al. [Smolka et al. 2015]; Here, we first define the notion of determinism relevant for NetKAT automata. Given an automaton $M = (S, S_0, S_f, \Delta)$, we say it is *deterministic* if:

(1) **Transition disjointness:** For all $s \in S$ and distinct states $s_1, s_2 \in S$,

$$\Delta(s, s_1) \cap \Delta(s, s_2) = \emptyset.$$

(2) **Transition totality:** For every $s \in S$, the set of all outgoing transitions from s is exhaustive:

$$\bigcup_{s' \in S} \Delta(s, s') = Pk \times Pk.$$

(3) **Unique initial state:** The automaton has exactly one initial state:

$$|S_0| = 1.$$

We adopt a two-step strategy to ensure these properties. First, we make the initial state unique and ensure that all outgoing transitions from each state are pairwise disjoint. Then, we extend the transition relation to guarantee totality.

To begin, suppose the original automaton is given by $M = (S, S_0, S_f, \Delta)$. We first construct its non-total determinized version:

$$M_{\text{det}} = (S_{\text{det}}, S_{\text{det0}}, S_{\text{detf}}, \Delta_{\text{det}}),$$

The construction is as follows:

- $S_{det} = 2^S$
- $S_{det0} = \{S_0\}$
- $S_{detf} = \{S \mid S \cap S_f \neq \emptyset\}$
- Transition function:

$$\Delta_{det}(s_{det1}, s_{det2}) = \left(\bigcap_{s_2 \in s_{det2}} \bigcup_{s_1 \in s_{det1}} \Delta(s_1, s_2) \right) \setminus \left(\bigcup_{s_2 \notin s_{det2}} \bigcup_{s_1 \in s_{det1}} \Delta(s_1, s_2) \right).$$

Then we prove the soundness and completeness of this automata

Theorem A.16 (Transition Disjointness). *Let $M = (S, S_0, S_f, \Delta)$ and its determinized version $M_{\text{det}} = (S_{det}, S_{det0}, S_{detf}, \Delta_{det})$. Then:*

$$L(M) = L(M_{\text{det}}),$$

and M_{det} satisfies the transition disjointness property.

Proof. To prove $L(M) = L(M_{\text{det}})$, we need to show that:

$$(s_1, x_1) \xrightarrow{w} (s_n, x_n) \iff \forall s_{det1}, s_1 \in s_{det1}, \exists s_{detn}, s_n \in s_{detn}, (s_{det1}, x_1) \xrightarrow{w}_{det} (s_{detn}, x_n).$$

We proceed by induction on w.

Base Case ($w = \epsilon$): Trivially, $(s_1, x_1) \xrightarrow{\epsilon} (s_1, x_1)$ and $(s_{det1}, x_1) \xrightarrow{\epsilon} (s_{det1}, x_1)$.

Inductive Step ($w = w'x_{n+1}$): By the inductive hypothesis:

$$(s_1, x_1) \xrightarrow{w'} (s_n, x_n) \iff \forall s_{det1}, s_1 \in s_{det1}, \exists s_{detn}, s_n \in s_{detn}, (s_{det1}, x_1) \xrightarrow{w'}_{det} (s_{detn}, x_n).$$

Now, suppose $(x_n, x_{n+1}) \in \Delta(s_n, s_{n+1})$. Using the definition of Δ_{det}, we calculate:

$$\begin{aligned}
\bigcup_{s \in s_{det2}} \Delta_{det}\ s_{det1}\ s_{det2} &= \bigcup_{s \in s_{det2}} \Big(\bigcap_{s_2 \in s_{det2}} \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2 \Big) \setminus \Big(\bigcup_{s_2 \notin s_{det2}} \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2 \Big) \\
&= \bigcup_{s \in s_{det2}} \bigcap_{s_2 \in s_{det2}} \Big(\bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2 \setminus \bigcup_{s_2' \notin s_{det2}} \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2' \Big) \\
&= \bigcup_{s \in s_{det2}} \bigcap_{s_2 \in s_{det2}} \bigcap_{s_2' \notin s_{det2}} \Big(\bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2 \setminus \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2' \Big) \\
&= \bigcup_{s \in s_{det2}} \{(x, x') \mid \forall\ s_2, (x, x') \in \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s_2 \Leftrightarrow s_2 \in s_{det2}\} \\
&= \{(x, x') \mid (x, x') \in \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s\} \\
&= \bigcup_{s_1 \in s_{det1}} \Delta\ s_1\ s
\end{aligned}$$

Taking $s = s_{n+1}$, we conclude that $(x_n, x_{n+1}) \in \Delta(s_n, s_{n+1}) \subseteq \bigcup_{s \in s_{det(n+1)}} \Delta_{det}(s_{detn}, s_{det(n+1)})$. Hence, there exists $s_{det(n+1)}$ satisfying the property.

For the reverse direction, we have if $(x_n, x_{n+1}) \in \Delta_{det}(s_{detn}, s_{det(n+1)})$, by definition, $(x_n, x_{n+1}) \in \Delta(s_n, s_{n+1})$. This completes the proof for language equivalence.

Transition Disjointness Property: Suppose s_{det2} and s'_{det2} differ in s'. Assume $s' \in s_{det2}$ and $s' \notin s'_{det2}$. Then:

$$\Delta_{det}(s_{det1}, s_{det2}) \subseteq \bigcup_{s_1 \in s_{det1}} \Delta(s_1, s'),$$

and:

$$\Delta_{det}(s_{det1}, s'_{det2}) \cap \bigcup_{s_1 \in s_{det1}} \Delta(s_1, s') = \emptyset.$$

Thus, M_{det} is disjoint in transition. □

A.9.1 Complete Automata. The next step is to make the deterministic automaton complete. We add a single state $s_\perp$ to the determinized automaton $M = (S, S_0, s_f, \Delta)$, transforming it into $M' = (S', S_0, S_f, \Delta')$.

The construction of the complete automaton is as follows:

- $S' = S \cup \{s_\perp\}$, where $s_\perp$ is a fresh state.
- S_0, and S_f remain unchanged.
- The transition function Δ' is defined as:
  - If $s, s' \in S$, then $\Delta'(s, s') = \Delta(s, s')$.
  - If $s \in S$ and $s' = s_\perp$, then:

$$\Delta'(s, s_\perp) = Pk \times Pk \setminus \bigcup_{s' \in S} \Delta(s, s').$$

  - If $s = s_\perp$ and $s' \in S$, then:

$$\Delta'(s_\perp, s') = \emptyset.$$

– If $s = s_\perp$ and $s' = s_\perp$, then:

$$\Delta'(s_\perp, s_\perp) = Pk \times Pk.$$

Theorem A.17. *The automaton M' satisfies $L(M') = L(M)$, and is deterministic.*

Proof. By construction it is easy to show the determinism.

Moreover, since $s_\perp$ is not a final state and has no outgoing transitions to the original states S, any path reaching $s_\perp$ cannot contribute to the language. Therefore, $L(M') = L(M)$. □

A.10 Bisimulation

The final step is to check bisimulation between two automata. Suppose we have two determinized and complete automata $M_1 = (S_1, \{s_{10}\}, S_{1f}, \Delta_1)$ and $M_2 = (S_2, \{s_{10}\}, S_{2f}, \Delta_2)$. We aim to verify if $L(M_1) = L(M_2)$. We adopt the method from the NetKAT automata equivalence check algorithm.

Algorithm 1: NetKAT Automata Equivalence Check

Input: Two deterministic automata $M_1 = (S_1, \{s_{10}\}, S_{1f}, \Delta_1)$, $M_2 = (S_2, \{s_{20}\}, S_{2f}, \Delta_2)$.
Output: Boolean indicating whether M_1 and M_2 accept the same language.

(1) Initialize $W \leftarrow \{(s_{10}, s_{20}, \alpha) \mid \alpha \in Pk\}$.
(2) **While** W changes:
  (a) For each $(s_1, s_2, \alpha) \in W$:
    (i) **If** $s_1 \in S_{1f}$ and $s_2 \notin S_{2f}$, or vice versa, **then return false**.
    (ii) **Otherwise**, for all s_1', s_2' and α' such that

$$(\alpha, \alpha') \in \Delta_1(s_1, s_1') \cap \Delta_2(s_2, s_2'),$$

    add (s_1', s_2', α') to W.
(3) **Return true**.

Soundness and Completeness of the Algorithm. To prove that the algorithm correctly determines whether two automata are bisimilar, we establish the following properties:

- **Soundness:** If the algorithm returns **true**, then M_1 and M_2 are bisimilar.
- **Completeness:** If M_1 and M_2 are bisimilar, then the algorithm returns **true**.

Definitions.

- Two automata are bisimilar if there exists a relation $R \subseteq S_1 \times S_2 \times Pk$ such that:

(1) $(s_{10}, s_{20}, \alpha) \in R$ for all $\alpha \in Pk$.
(2) For each $(s_1, s_2, \alpha) \in R$, if $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$, then $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$ for some s_2', and $(s_1', s_2', \alpha') \in R$.
(3) Similarly, for $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$, there exists s_1' such that $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$, and $(s_1', s_2', \alpha') \in R$.
(4) $s_1 \in S_{1f}$ if and only if $s_2 \in S_{2f}$.

We want to prove that two complete and deterministic automata $M_1 = (S_1, \{s_{10}\}, S_{1f}, \Delta_1)$ and $M_2 = (S_2, \{s_{20}\}, S_{2f}, \Delta_2)$ are equivalent if and only if they are bisimilar.

Theorem A.18. *Let $M_1 = (S_1, \{s_{10}\}, S_{1f}, \Delta_1)$ and $M_2 = (S_2, \{s_{20}\}, S_{2f}, \Delta_2)$ be complete and deterministic automata. Then:*

$$L(M_1) = L(M_2) \iff M_1 \text{ and } M_2 \text{ are bisimilar.}$$

Proof. We prove both directions of the equivalence.

(1) If M_1 and M_2 are bisimilar, then $L(M_1) = L(M_2)$: Assume M_1 and M_2 are bisimilar. Let $R \subseteq S_1 \times S_2 \times Pk$ be a bisimulation relation such that:

(1) $(s_{10}, s_{20}, \alpha) \in R$ for all $\alpha \in Pk$.
(2) If $(s_1, s_2, \alpha) \in R$ and $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$ in M_1, then there exists $s_2' \in S_2$ such that $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$ in M_2, and $(s_1', s_2', \alpha') \in R$.
(3) Similarly, for $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$ in M_2, there exists $s_1' \in S_1$ such that $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$ in M_1, and $(s_1', s_2', \alpha') \in R$.
(4) $s_1 \in S_{1f} \iff s_2 \in S_{2f}$.

To show $L(M_1) = L(M_2)$, let $w \in L(M_1)$. By definition, there exists a sequence of transitions:

$$(s_{10}, x_0) \xrightarrow{x_1} \ldots \xrightarrow{x_n} (s_{1n}, x_n),$$

where $s_{1n} \in S_{1f}$.

Since $(s_{10}, s_{20}, \alpha) \in R$ for all α, and R is preserved across transitions, there exists a corresponding sequence of transitions in M_2:

$$(s_{20}, x_0) \xrightarrow{x_1} \ldots \xrightarrow{x_n} (s_{2n}, x_n),$$

where $s_{2n} \in S_{2f}$. Thus $w \in L(M_2)$.

By symmetry, if $w \in L(M_2)$, then $w \in L(M_1)$. Therefore, $L(M_1) = L(M_2)$.

(2) If $L(M_1) = L(M_2)$, then M_1 and M_2 are bisimilar: Assume $L(M_1) = L(M_2)$. We construct a bisimulation relation $R \subseteq S_1 \times S_2 \times Pk$ as follows:

$$R = \{(s_1, s_2, \alpha) \mid \text{for all } w, (s_1, x) \xrightarrow{w} (s_1', x') \text{ in } M_1 \wedge s_1' \in S_{1f} \iff (s_2, x) \xrightarrow{w} (s_2', x') \text{ in } M_2 \wedge s_2' \in S_{2f}\}.$$

We need to verify that R satisfies the conditions of bisimulation:

(1) **Base Case:** $(s_{10}, s_{20}, \alpha) \in R$ for all α. This follows because $L(M_1) = L(M_2)$, and for the initial states s_{10} and s_{20}, their final states after processing any word w must match.
(2) **Inductive Step:** Suppose $(s_1, s_2, \alpha) \in R$. If $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$ in M_1, then:
   - By definition of R, for all w, if $s_1' \in S_{1f}$, then $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$ in M_2, and $s_2' \in S_{2f}$.
   - Similarly, if $s_1' \notin S_{1f}$, then $s_2' \notin S_{2f}$. Thus, $(s_1', s_2', \alpha') \in R$.

   Conversely, if $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$ in M_2, we can similarly argue that there exists $s_1' \in S_1$ such that $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$ in M_1, and $(s_1', s_2', \alpha') \in R$.
(3) **Final States:** For $(s_1, s_2, \alpha) \in R$, the condition $s_1 \in S_{1f} \iff s_2 \in S_{2f}$ is explicitly part of the construction of R. Thus, the bisimulation relation preserves final state equivalence.

Therefore, M_1 and M_2 are bisimilar. □

Theorem A.19. *The equivalence checking algorithm returns **true** if and only if M_1 and M_2 are bisimilar.*

Proof. **Soundness:** Assume the algorithm returns **true**. Define the relation

$$R = \{(s_1, s_2, \alpha) \mid (s_1, s_2, \alpha) \in W\},$$

where W is the working set computed by the algorithm. By construction and the iterative update steps of W, the following conditions hold:

(1) If $(s_1, s_2, \alpha) \in R$, then $s_1 \in S_{1f}$ if and only if $s_2 \in S_{2f}$ (enforced by step 2(a)(i)).
(2) If $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$ in M_1, then step 2(a)(ii) ensures that there exists a matching transition $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$ in M_2 such that $(s_1', s_2', \alpha') \in R$.

(3) Symmetrically, any transition in M_2 is matched by a corresponding transition in M_1.

Hence, R is a bisimulation relation, and M_1 and M_2 are bisimilar.

Completeness: Assume M_1 and M_2 are bisimilar. Let $R \subseteq S_1 \times S_2 \times A$ be a bisimulation relation. We show that the algorithm returns **true**.

(1) The initial working set W contains all triples (s_{10}, s_{20}, α) for $\alpha \in A$, ensuring the base cases of $R \subseteq W$.

(2) At each iteration, if $(s_1, s_2, \alpha) \in R$ and there exists a transition $(s_1, \alpha) \xrightarrow{\alpha'} (s_1', \alpha')$, then by the definition of bisimulation, there exists a matching transition $(s_2, \alpha) \xrightarrow{\alpha'} (s_2', \alpha')$, and the algorithm includes (s_1', s_2', α') in W.

Since the relation R is finite and the algorithm terminates when W stabilizes, all bisimulation conditions are eventually verified, and the algorithm returns **true**. □

A.11 Maximum Bisimulation

In the paper, we proposed a localized bisimulation approach that focuses on computing the equivalence class of (x, y) based on whether they lead to the same (x', y') in the next state. However, a larger equivalence class may also be computed using the classical notion of bisimilarity, which considers agreement over the y-tape across all subsequent states.

Definition 6 (x-bisimulation). *For a synchronous NetKAT transducer $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$, a relation $x_1 \equiv_{s,y} x_2$ is an x-bisimulation if:*

- *For all $((x_1, y), (x_3, y')) \in \Delta_S(s, s')$, there exists x_4 such that $((x_2, y), (x_4, y')) \in \Delta_S(s, s')$ and $x_3 \equiv_{s',y'} x_4$.*
- *For all $((x_2, y), (x_4, y')) \in \Delta_S(s, s')$, there exists x_3 such that $((x_1, y), (x_3, y')) \in \Delta_S(s, s')$ and $x_3 \equiv_{s',y'} x_4$.*

We define the maximum x-bisimulation relation $R_{\max}$ to be the union of all x-bisimulations.

This definition follows the tradition of general bisimulation, with the difference that the equivalence relation $x_1 \equiv_{s,y} x_2$ is parameterized by both the current state s and the y-tape. This is sufficient for performing projection on transducers where only the X-tape is eliminated, while the state s and Y-tape remain unchanged. Identifying equivalence classes on the X-tape can significantly reduce the number of states that must be considered during projection.

We now show how to transform a general synchronous transducer into one that is efficiently projectable. Our procedure involves two steps:

(1) Compute the bisimulation witness set for each state.
(2) Split each state according to its bisimulation witness.

For the first step, suppose we have a partition function P that partitions the (x, y) pairs based on whether they agree on the resulting y' values at the next state.

Definition 7 (Bisimulation Witness Partition Function). *Let $P : (Pk \times Pk) \times (Pk \times Pk) \to 2^{Pk \times Pk}$ be a partition function if the following hold:*

(1) *(Disjointness) For all $r_1, r_2 \in P(R)$,*

$$\{(x, y) \in r_1\} \cap \{(x, y) \in r_2\} = \emptyset.$$

(2) *(Agreement on y) For all $r \in P(R)$ and all $(x_1, y), (x_2, y) \in r$,*

$$\{y' \mid ((x_1, y), (x_3, y')) \in R\} = \{y' \mid ((x_2, y), (x_3, y')) \in R\}.$$

(3) *(Coverage)*

$$\bigcup_{r \in P(R)} r = \{(x, y) \mid ((x, y), (x', y')) \in R\}\,.$$

This partition function defines the equivalence classes of (x, y) pairs based on whether they agree on the y' values reachable in the next state. With this function, we can now compute the bisimulation witness set for each state.

Algorithm 2: Bisimulation Witness Generation

1 **Input:** A synchronous NetKAT transducer $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$, Partition function $P : (Pk \times Pk) \times (Pk \times Pk) \to 2^{Pk \times Pk}$

2 **Output:** A target function $\mathcal{B} : S \to 2^{2^{Pk \times Pk}}$, producing a set of bisimulation witnesses for each state.

(1) Initialize $\mathcal{B}(s) \leftarrow \{\{(pk_1, pk_2) \mid pk_1, pk_2 \in Pk\}\}$ for all states s.

(2) **While** $\mathcal{B}$ changes for any state s_2 (initialization counts as a change):

(a) For each $s_1 \in S$:

(i) For each current bisimulation witness $B(s_2) \in \mathcal{B}(s_2)$, compute the new partition set P with respect to predecessor s_1:

$$\mathcal{B}_{\text{update}}(s_1) = \bigcup_{B(s_2) \in \mathcal{B}(s_2)} P\left(\{((x, y), (x', y')) \in \Delta_S(s_1, s_2) \mid (x', y') \in B(s_2)\}\right)$$

(ii) Update the witness set for s_1 by intersecting with the new partitions:

$$\mathcal{B}(s_1) \leftarrow \left\{B(s_1) \cap B_{\text{update}}(s_1) \mid B(s_1) \in \mathcal{B}(s_1),\ B_{\text{update}}(s_1) \in \mathcal{B}_{\text{update}}(s_1)\right\}$$

(3) **Return** $\mathcal{B}$.

Then we simply perform state splitting based on the bisimulation witness set. The procedure is as follows.

We define the split transducer:

$$T^p \triangleq (S^p, S_0^p, S_f^p, \Delta_S^p, \emptyset, \emptyset, \emptyset)$$

as:

- $S^p \triangleq \{(s, b) \mid s \in S, b \in \mathcal{B}(s)\}$.
- $S_0^p \triangleq \{(s_0, b) \mid s_0 \in S_0, b \in \mathcal{B}(s_0)\}$.
- $S_f^p \triangleq \{(s_f, b) \mid s_f \in S_f, b \in \mathcal{B}(s_f)\}$.
- Transition relation:

$$\Delta_S^p((s_1, b_1), (s_2, b_2)) \triangleq \{((x, y), (x', y')) \in \Delta_S(s_1, s_2) \mid (x, y) \in b_1, (x', y') \in b_2\}$$

Theorem A.20 (Correctness of Projection). *Let $T = (S, S_0, S_f, \Delta_S, \emptyset, \emptyset, \emptyset)$ be a synchronous NetKAT transducer. Then $L(T) = L(T^p)$, and T^p is efficiently projectable.*

Proof. The language equivalence $L(T) = L(T^p)$ follows from a similar argument used in the proof of correctness for the localized approach.

To prove efficient projectability, assume for contradiction that T^p is not efficiently projectable. Then there must exist states s_1, s_2 and packets x_1, x_2, y such that $((x_1, y), (x_3, y')) \in \Delta(s_1, s_2)$, but there is no x_4 such that $((x_2, y), (x_4, y')) \in \Delta(s_1, s_2)$. However, the algorithm checks and back-propagates all partitions to the predecessor states. Therefore, such a case would have triggered a partition refinement for state s_1, contradicting the assumption. Hence, T^p is efficiently projectable. □

Received 2025-07-10; accepted 2025-11-06