AUTOMATA LEARNING FOR NETWORK VERIFICATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Mark David Moeller

August 2026

AUTOMATA LEARNING FOR NETWORK VERIFICATION

Mark David Moeller, Ph.D.

Cornell University 2026

As computer networks grow more complicated, tools that verify correctness help avoid costly routing policy errors. Building on classical algorithms for automata, NetKAT is a programming language for defining and verifying network routing policies with an axiomatic proof system as its basis [5]. Unfortunately, while NetKAT is principled and expressive, the performance of NetKAT's decision procedure is worse than state of the art verification styles using other techniques. In addition, NetKAT's approach requires that we have a precise description of a network's forwarding policy, which is not always available. This dissertation bridges both of these gaps: To address performance, we give a new decision procedure for NetKAT program equivalence, bringing automata-based verification up to the state of the art. To facilitate building faithful system models, we develop a theory for *learning* of NetKAT automata based on Angluin's $\mathtt{L}^\star$ algorithm [7]. Our algorithm builds a model of a network by querying example packets to an oracle. After proving the correctness of our learning algorithms, we present an implementation and initial benchmarking results. Finally, we investigate an Angluin-style framework for learning finite automata even when the oracle fails to answer some queries. In this setting, we give a learning algorithm and prove that it still learns a minimum-sized finite automaton despite missing information. We conclude with remarks on next steps towards automatically verifying networks.

BIOGRAPHICAL SKETCH

Mark David Moeller was born in Indianapolis, Indiana, USA, in 1989. He grew up in Valley Forge, Pennsylvania, graduating from Conestoga High School in Berwyn, Pennsylvania. He earned a bachelor of science at the United States Naval Academy (USNA) in Annapolis, Maryland, with a dual major in Mathematics and Computer Science. After graduating from USNA, he completed a Master of Science in Engineering in Computer and Information Science at the University of Pennsylvania in 2013. Following a sea tour as a naval officer, he returned to USNA as a lecturer in Computer Science from April 2018 to June 2020. There, benefiting from guidance and mentorship from his colleagues on the USNA Computer Science faculty, he decided to pursue a PhD and career in academic computer science. Leaving the Navy in 2020, he conducted his PhD work at Cornell in Ithaca, New York from August 2020 to August 2026.

To Prof. Matt Foley

ACKNOWLEDGEMENTS

To Christopher Brown, Adina Crainiceanu, Daniel Roche, and Rick Schlichting, thank you for the mentorship in teaching and preparing me for graduate school. For Chris and Dan in particular, who taught me automata and programming languages in the very first place, thank you so much.

To my advisors Alexandra Silva and Nate Foster, thank you: being coadvised by the two of you has been an incomparable learning experience in choosing, solving, writing about, and speaking about problems involving automata.

In addition to my advisors, I have been so fortunate to have an incredibly "deep bench" of mentors during my time at Cornell. To Dexter Kozen, your textbooks first introduced me to Kleene Algebra and set me down the path leading to Cornell and the work presented here, but beyond that you have been such an important example of what a professor can be. To Adrian Sampson, thank you for all the ways you inspire everyone around you to be better. To Jules Jacobs, our close collaboration on KATch was a special highlight of my graduate school experience; thank you for teaching me about how you think about problems. To David Darais, thank you for two internships and for being such a wonderful mentor over these years. To Hossein Hojjat, your technical wit is inspirational—"let's see your cartoons." To Tiago Ferreira, thank you for sharing your joy and enthusiasm for automata learning. How lucky I've been to have you as both friend and collaborator. To all of my other research collaborators who contributed directly to the work appearing in this dissertation, thank you: Alaia Solko-Breslin, Caleb Koch, Thomas Lu, Olivier Savary Belanger, Cole Schlesinger, Steffen Smolka, and Thomas Wiener.

Thank you to Anke van Zuylen for sharing with me your joy of both teaching and math, and for showing me your teaching process.

Thank you to Elizabeth Estabrook and Becky Stewart for the too-often-thankless hard work that makes Cornell CIS such a nice place to work on a PhD.

For all the rubber duck debugging, running, celebration of successes, and commiseration of setbacks, thank you Eric Campbell. For all the home cooking, squash, preponed lunches, and thoughtful reflections on teaching and research, thank you Anshuman Mohan.

For so much encouragement at every stage, thank you Erin.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

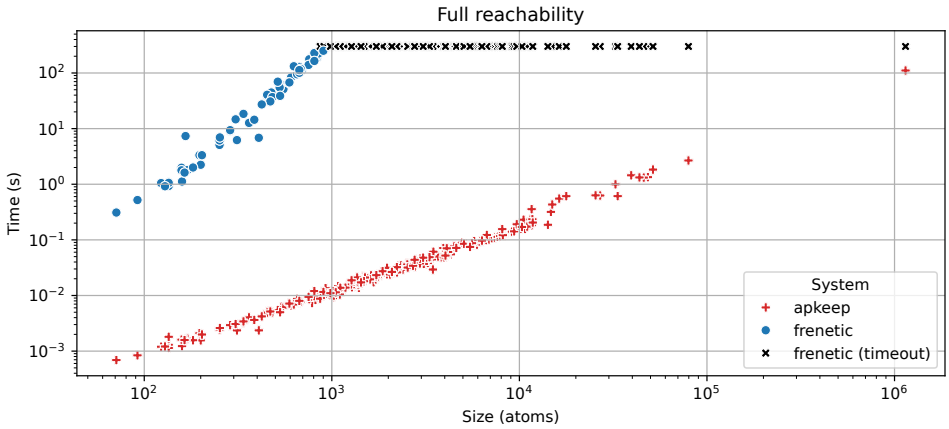
As demands on computer networks have evolved in recent decades, there has been interest in controlling network routing policies in centralized, programmable configurations. This approach, broadly called Software Defined Networking (SDN), involves designing a controller that issues routing updates to the routers in a network. Programming this centralized controller affords greater flexibility to design specialized policies, but this power naturally creates opportunities for more pernicious bugs. This dissertation is about designing automatic verification techniques for network routing policies. We will develop an approach that applies regardless of whether the policies are produced by a control program, a human, a traditional distributed control plane, or artificial intelligence (AI).

NetKAT: Automata-based Network Verification Routers normally use only a small amount of memory in order to run at line rate. As a result, surprisingly many useful policies can be expressed using only finite-state reasoning, which means that such policies can be expressed (and thus verified) using finite automaton models. NetKAT is a domain specific language designed for specifying and verifying network routing policies [5, 55, 101, 103, 104]. In NetKAT, programs denote valid sets of traces (or "packet histories") where a trace is the list of packets observed at each hop in the network. The NetKAT language specifies these traces using a regular expression-like syntax, as follows:

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

NetKAT has a decision procedure for program equivalence, which allows us to perform verification queries via conversion to automata [5, 55]. Unfortunately, existing algorithms for verification queries in NetKAT do not scale as well as state

of the art techniques. Here is a benchmark comparison of time required to check that all hosts can reach all other hosts in a network:



Comparison of Time vs. Network size for all-pairs reachability queries. FRENETIC is the best existing NetKAT implementation. APKEEP is a state of the art tool [120].

The performance gap of automata-based network verification is a problem we seek to address in this dissertation.

Automata Learning for Modeling Systems Setting aside the issue of performance, there are some settings where it is not straightforward to generate a faithful NetKAT expression from our network. Without this, we cannot perform verification *at all*, let alone fast. For instance, suppose we are using proprietary hardware which has behaviors that are not specified by configurations. In such a scenario where a precise model of the network is not readily available, it would be ideal to be able to *learn* such a model by observing traces of network behavior. It turns out that learning models of system behaviors has been well-studied following work by Angluin on *active learning of finite automata* [7]. In her work (and the decades of research it has inspired), one assumes the existence of an oracle for "true system behavior" that can answer two types of queries. In a *membership query*, the learner poses a trace, and the oracle answers whether the trace is a

valid system behavior. In an *equivalence query*, the learner poses a system model as a conjecture. The oracle answers "yes" if the conjecture is correct, otherwise it provides a counterexample trace which has been misclassified by the conjecture. Angluin gave an algorithm for learning a deterministic finite automaton (DFA) model of a system from such an oracle, but to our knowledge, no prior work has applied Angluin-style learning to *network routing policies*. The challenge of building an automaton learner for networks is the second problem addressed by this dissertation.

1.1.1 A Little Broader Context

Some readers may wonder why traditional *software testing* is not adequate to provide the assurances we seek for networks [1]. A short answer is that verification allows us to check policies generated in a controller *before* applying them as an update. We give a longer answer in the rest of this section before stating our thesis.

In software testing, we generate a suite of tests to confirm that the software has correct behavior for a particular collection of inputs. To state another way, for *each bug* that we hope to find using software testing, we must write a test consisting of a sample input and expected behavior that is violated by the system. Since it is normally intractable to test all possible inputs, our testing system is inherently limited by our ability to produce correct tests and by our budget for running them.

On the other hand, in formal program verification, we seek to reason math-

[1] Readers that do *not* wonder this are perfectly justified in skipping to Section 1.2

ematically to ensure the software's behavior is correct on *all possible inputs*. In some setups, a mechanically checkable proof of correctness is produced as an artifact. Of course, failures are still possible (e.g., if you prove the wrong theorem! [10]), but the assurance provided by such verification is much stronger than that of software testing. In essence, we can rule out whole classes of bugs.

Unfortunately, the additional engineering required for verification is often expensive. This work requires engineers with training in formal methods to produce verified production software, and it takes longer. This high cost is sometimes justified, for example in aircraft flight control software [56], or, more recently, cloud services [31], but for routine software development the cost is prohibitive.

In some domains, however, certain properties of the design of a system allow automatic verification techniques to work. For instance, consider a device that reads a finite sequence of symbols and, based on a finite amount of memory, decides whether the input is valid (imagine, e.g., a keypad to a lock). The behavior of a system with these characteristics can be modeled by a DFA. In particular, using automata for verification, called *model-checking*, allows us to achieve the best of both worlds: formal guarantees at low cost. By applying classical algorithms that reason about automata, we can verify properties of system behavior *automatically*. Indeed, model checking has been applied widely in safety-critical systems [12].

In this approach, we first seek to define one automaton describing the *desired set of behaviors* of a system (i.e., the specification). Next, we build another automaton which is a *faithful* model of the engineered system. Finally, adherence to our specification is confirmed by a demonstration that the two automata are equivalent (or sometimes, that the behaviors of the engineered system are a subset of those permitted by the specification).

A particularly compelling line of work following this approach to verification is Kozen's Kleene Algebra with Tests (KAT) [74]. KAT generalizes regular expressions, which are syntax for describing sets of events which can be converted to finite automata. Kozen showed how KAT forms the basis of a reasoning system for equivalence of program control flow structures. For instance, KAT has been used to validate the correctness of compiler passes [76, 35]. To give an intuitive sense, given programs p and q, and a conditional expression b, we can model *if statements* and *while loops* in KAT as follows:

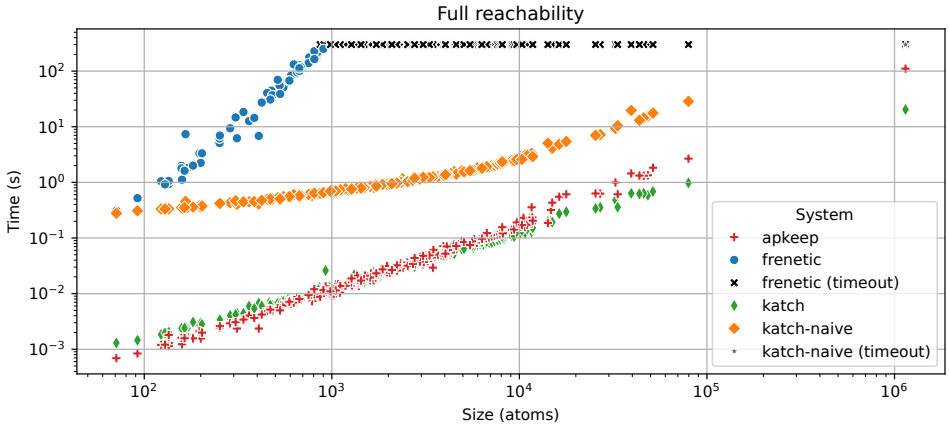
$$\textbf{if } b \textbf{ then } p \textbf{ else } q \triangleq b \cdot p + \neg b \cdot q \qquad\qquad \textbf{while } b \textbf{ do } p \triangleq (b \cdot p)^\star \cdot \neg b.$$

After modeling a program (and any specifications) with a KAT expression, we can then convert it into an automaton, unlocking the power of the algorithms for automata mentioned above. In short, KAT gives us an elegant way to decide questions such as equivalence ("do two programs have the same behavior?"), containment ("are all of the valid behaviors of one program valid for another?"), or emptiness ("does a program have *any* valid behaviors?"). NetKAT, as we have briefly described, is a KAT for reasoning about correctness properties of networks, making it the ideal basis for the task at hand.

1.2 Thesis

The goal of this dissertation is to make fundamental contributions to automata learning and symbolic reasoning to advance the state of the art of network verification. In short, with these techniques, we aim to draw closer to a world where practical network configurations can be *verified automatically*, or with minimal human effort.

Symbolic Equivalence for NetKAT The plan laid out above is promising already, but there is a major problem with applying automata reasoning to networks: if packet histories are our domain of discourse for denoting system behaviors, then the alphabet for the automata we need is *the entire packet space*. This means that any algorithm that produces (or traverses) an explicitly represented automaton is impractical. There is some hope: even though the packet space itself is enormous, it remains true that each individual forwarding decision relies on a small number of packet headers. Therefore, a key requirement of a performant algorithm is a *symbolic* representation for the packet space, i.e., one that stores only the header values and updates that affect behavior. An appropriate symbolic representation allows us to represent (and check properties of) a large number of concrete packet updates by focusing on the particular updated fields. We solve this problem by identifying appropriate data structures for representing sets of packets and relations on packets symbolically. Then, we give a symbolic decision procedure for NetKAT equivalence that we call KATch using these data structures. We prove key results about the central data structures required, and we show empirically that the performance is an asymptotic improvement to prior work on NetKAT. To preview these results, here is the same plot we saw in Section 1.1, with KATch plotted as well:



Comparison of Time vs. Network size for all-pairs reachability queries.

We explain the details of how KATch achieves this competitive performance in Chapter 3.

Learning NetKAT Models Active learning of finite automata has been applied to many settings involving interaction with closed-box systems: Vaandrager has an accessible overview [109]. Of particular interest to us, Angluin-style automata learning has already been applied successfully to implementations of networking protocols [45, 49]. In Chapter 4, we extend the existing work by designing algorithms for conducting Angluin-style learning of NetKAT automata.

Our algorithm, `NKL`*, can learn an *arbitrary* NetKAT expression. This is significant as NetKAT's expressivity is a great strength: we can just as easily model an entire network as we can model an individual switch, subnet, or just the topology, to name some examples. As before, the size of the packet space presents a central challenge to adapting Angluin's framework to the general setting of NetKAT directly. Learning of symbolic automata has received some initial attention [11, 43]. We draw inspiration from this prior work in designing several symbolic learning algorithms for NetKAT automata. In particular, we give and prove correct two algorithms: first, a learner for input-output packet behavior of a closed box network, and second, a learner for full-trace behavior of a network. We prove both algorithms correct and describe an OCaml implementation of each in Chapter 4.

Learning from an Imperfect Oracle While the story so far has allowed us to make significant progress towards building models of networks from observations, there is a serious problem lurking in the reality of any potential practical implementation: the oracles we have assumed in order to conduct the learning process are already a significant extension of real world closed-box systems. One issue is

that we cannot always make direct observations about whether traces are valid. In networking, this means for instance that we do not always have complete telemetry of the packets traversing a network. As a first step in weakening the assumptions required to build a more realistic networking learner, we have tackled this problem in the abstract setting of finite automata. That is, in Chapter 5 we explore the question of how to adapt an Angluin learner to a setting in which membership queries are not always answered with "yes" or "no", but instead might be answered with "don't know" or "don't care." In that context, we prove that with the aid of a solver[2], we can still learn a minimum-size DFA. In addition, we explore a variation on the framework itself in which the equivalence query is replaced by a weaker query we call "distinguish," which can decide whether two input prefixes have a distinguishing suffix, but not resolve full equivalence queries.

1.3 Technical Acknowledgements

The contributions made in this dissertation are based on the results of three published papers featuring work performed with many collaborators. The preliminary material in Chapter 2 is adapted from the early sections of all three papers. The remaining three technical chapters are (lightly) revised versions of the three papers:

- **Chapter 3** is based on the conference paper: Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. KATch: A fast symbolic verifier for NetKAT. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY,

[2]We define this abstractly, but instantiate with a Satisfiability Modulo Theories (SMT) solver.

USA, 2024. Association for Computing Machinery.

- **Chapter 4** is based on the conference paper: Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. Active learning of symbolic NetKAT automata. In *Proceedings of the 46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, June 2025. Association for Computing Machinery.

- **Chapter 5** is based on the conference paper: Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. Automata Learning with an Incomplete Teacher. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Finally, Chapter 6 is original to this dissertation, but it is of course heavily influenced by all of this collaboration. I am indebted to all of my collaborators for their contributions to both the results and presentation contained here. Nevertheless, any remaining errors are mine.

CHAPTER 2

TECHNICAL PRELIMINARIES

To approach our goal of learning models of networks from example traces, we rely on a rich background of theory of both modeling networks and active learning. In this chapter, we review the technical fundamentals essential to the contributions in this dissertation. The chapter is comprised of two sections:

- In Section 2.1, we introduce NetKAT, a domain specific programming language for reasoning about network forwarding planes, which serves as the background for our fast decision procedure for NetKAT equivalence in Chapter 3.
- In Section 2.2, we introduce the fundamentals of automata learning.

We bridge both of these lines of research when we show how to use active learning to build models of networks in Chapter 4. When we finally explore learning in the presence of unknown information in Chapter 5, we rely only on Section 2.2.

2.1 NetKAT and NetKAT Automata

NetKAT is a language for specifying the packet-forwarding behavior of network data planes [5, 55, 101],[1] based on Kozen's Kleene Algebra with Tests (KAT) [73].

2.1.1 NetKAT Syntax and Semantics

The syntax and *compositional* semantics of NetKAT is presented in Table 2.1.

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

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

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

Table 2.1: NetKAT syntax and *compositional* semantics, which are used to develop KATch in Chapter 3.

To a first approximation, NetKAT can be thought of as a simple, imperative language that operates over packets, where a *packet* is a finite record assigning values to fields. We fix a set of packet header fields $F = \{f_1, \ldots, f_n\}$, and a finite set of header values V. We write records (finite maps) as a collection of mappings. For example, a *packet* is a finite record assigning values to fields $\alpha = \{f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\}$. We reference the field f_1 of packet α by $\alpha.f_1 = v_1$. The set of all packets is denoted $\mathsf{Pk} = F \Mapsto V$. The basic primitives in NetKAT are packet tests ($f = v,\ f \neq v$) and packet modifications ($f \leftarrow v$). Program expressions are then compositionally built from tests and packet modifications, using union ($+$), sequencing ($\cdot$), and iteration ($\star$). Conditionals and loops can be encoded in the standard way:

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

In a network, conditionals can be used to model the behavior of the forwarding tables on individual switches while iteration can be used to model the iterated processing performed by the network as a whole; the original paper on NetKAT provides further details [5]. Note that assignments and tests in NetKAT are always

against constant values. This is a key restriction that makes equivalence decidable and aligns with the capabilities of data plane hardware. The dup primitive makes a copy of the current packet and appends it to the trace, which only ever grows as the packet goes through the network. This primitive is crucial for expressing network-wide properties, as it allows the semantics to "see" the intermediate packets processed on internal switches.

NetKAT's compositional semantics, given in Table 2.1, defines the action of a program on an input packet α, producing a set of output traces. The semantics for $\top$ gives a single trace containing the single input packet α, whereas $\bot$ produces no traces. Tests ($f = v$ and $f \neq v$) produce a singleton trace or no traces, depending on whether the test succeeds or fails. Modifications ($f \leftarrow v$) produce a singleton trace with the modified packet. Duplication (dup) produces a single trace with two copies of the input packet. Union ($p + q$) produces the union of the traces produced by p and q. Sequential composition ($p \cdot q$) produces the concatenation of the traces produced by p and q, where the last packet in output traces of p is used as the input to q. Finally, iteration ($p^\star$) produces the union of the traces produced by p iterated zero or more times. Consider the following example:

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

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

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

Consider what happens if we change the program to $(x = 0 \cdot x \leftarrow 1 \cdot \text{dup} \ + \ x \leftarrow 0 \cdot \mathsf{dup})^\star$. Unlike the previous example, where the tests were disjoint, this program can generate multiple outputs for a given input packet, as the right branch of the union can always be taken, leading to sequences $x = 0, x = 0, \ldots$. On the other

hand, if the $x \leftarrow 1$ assignment is performed, then the left branch cannot be taken the next time, because the test $x = 0$ will fail. Hence, the program produces traces with sequences of $x = 0$ packets, with singular $x = 1$ packets between them.

As these simple examples show, despite the fact that assignments and tests in NetKAT programs are always against constant values, the behavior of NetKAT programs can be complicated, particularly when conditionals, iteration, and multiple packet fields are involved.

Traditional presentations of NetKAT distinguish a syntactic test fragment that includes negation:

$$t, r ::= \bot \mid \top \mid f = v \mid t \vee r \mid t \wedge r \mid \neg t \qquad \text{(Test fragment)}$$

$$p, q ::= t \mid f \leftarrow v \mid p + q \mid p \cdot q \mid \text{dup} \mid p^\star \qquad \text{(General expressions)}$$

This distinction is necessary because negation $\neg p$ only makes sense on the test fragment. We replace general negation $\neg t$ in our presentation with only negated field tests $f \neq v$. There is no loss of generality—we translate a test fragment expression t to $[t]_0$ using the DeMorgan rules:

$$[\bot]_0 = \bot \qquad [t \vee r]_0 = [t]_0 + [r]_0 \qquad [\bot]_1 = \top \qquad [t \vee r]_1 = [t]_1 \cdot [r]_1$$

$$[\top]_0 = \top \qquad [t \wedge r]_0 = [t]_0 \cdot [r]_0 \qquad [\top]_1 = \bot \qquad [t \wedge r]_1 = [t]_1 + [r]_1$$

$$[f = v]_0 = (f = v) \qquad [\neg t]_0 = [t]_1 \qquad [f = v]_1 = (f \neq v) \qquad [\neg t]_1 = [t]_0$$

Conditional dup, equivalence, and subsumption An interesting feature of NetKAT is that dup need not appear the same number of times on all paths through the expression. Consider the program $(f \leftarrow v + f \leftarrow w + \mathsf{dup})^\star$, which either sets field f to v or to w, or logs the packet to the trace, and then repeats.

Any number of writes can happen between two dups, and only the effect of the last write before the dup gets logged to the trace.

The use of dup controls the type of equivalence that is checked. If we insert a dup whenever the sw field is updated, we obtain the normal network trace equivalence (sw is a special meta-field used for modeling the packet's location). If we insert a dup after every packet field write, then we obtain the semantics used by Temporal NetKAT [15]. If we insert no dups whatsoever, we obtain input-output equivalence. By inserting dups in some places but not others, we can control the equivalence that is checked in a fine-grained manner.

NetKAT's semantics induce an equivalence $(p \equiv q) \triangleq (\llbracket p \rrbracket = \llbracket q \rrbracket)$ on syntactic programs. In addition to $p \equiv q$, we may want to check inclusion $p \sqsubseteq q$. This can be expressed as $p+q \equiv q$ (or, as we will see in KATch by using a set difference operator, as $p \setminus q \equiv \bot$). As before, the number of dups controls the type of inclusion that is checked: from trace inclusion to input-output inclusion, or something in between. This equivalence is decidable, and can be computed by converting programs to automata and checking for automata equivalence. Before introducing automata, we first describe an alternative view of the semantics of NetKAT that will be useful to us in developing the theory.

2.1.2 Alternate Guarded-String Semantics

There is an alternative view of the semantics of NetKAT which is based on the notion of *guarded strings* from the study of KAT. Of course, these semantics are known to be isomorphic to those we have already seen in Table 2.1, as described in the first paper on NetKAT by Anderson et al [5]. The guarded string semantics

Syntax		**Description**	**Semantics** $[\![p]\!] \subseteq \mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$
$p, q ::=$	$\bot$	*False*	$\emptyset$
$\mid$	$\top$	*True*	$\{\alpha\alpha \mid \alpha \in \mathsf{Pk}\}$
$\mid$	$f = v$	*Test equals*	$\{\alpha\alpha \mid \alpha.f = v\}$
$\mid$	$f \neq v$	*Test not equals*	$\{\alpha\alpha \mid \alpha.f \neq v\}$
$\mid$	$f \leftarrow v$	*Modification*	$\{\alpha\alpha[f \leftarrow v] \mid \alpha \in \mathsf{Pk}\}$
$\mid$	dup	*Duplication*	$\{\alpha \cdot (\alpha \cdot \mathsf{dup}) \cdot \alpha \mid \alpha \in \mathsf{Pk}\}$
$\mid$	$p + q$	*Union*	$[\![p]\!] \cup [\![q]\!]$
$\mid$	$p \cdot q$	*Sequencing*	$[\![p]\!] \diamond [\![q]\!]$
$\mid$	$p^\star$	*Iteration*	$\bigcup_{n \geq 0} [\![p^n]\!]$ s.t. $p^0 \triangleq \top$, $p^{n+1} \triangleq p \cdot p^n$

Table 2.2: NetKAT syntax and *guarded-string* semantics, which are used to develop `NKL`* in Chapter 4.

offer some intuitive flexibility when thinking about automata: we would like to take prefixes of guarded strings and reason about the intermediate state of the automaton that we would reach, and this view allows us to do just that.

These alternate semantics of NetKAT are based on regular sets of *guarded strings* which are elements of the set $\mathsf{GS} = \mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$. Here, dup can be thought of as delimiting "letters" in the string. Equivalently, we can think of guarded strings as elements of the isomorphic set $\mathsf{Pk} \cdot \mathsf{Pk}^\star \cdot \mathsf{Pk}$ which are strings over the alphabet Pk with at least two letters. Given two guarded strings $\alpha x \beta$ and $\gamma y \xi$ their concatenation, denoted by $\diamond$, is defined only when $\beta = \gamma$:

$$\alpha x \beta \diamond \gamma y \xi = \begin{cases} \alpha x y \xi & \beta = \gamma \\ \text{undefined} & \beta \neq \gamma \end{cases} \tag{2.1}$$

This definition lifts to sets of guarded strings $U, V \subseteq \mathsf{GS}$ pointwise: $U \diamond V = \{u \diamond v \mid u \in U \text{ and } v \in V\}$.

NetKAT's guarded string semantics, given in Table 2.2 associates sets of guarded strings to each expression. The semantics for $\top$ gives all traces containing the input/output packet $\alpha\alpha$, whereas $\bot$ produces no traces. Tests ($f = v$ and $f \neq v$) produce filter traces, depending on whether the test succeeds or fails.

Modifications ($f \leftarrow v$) produce a trace with the modified packet. Duplication (dup) produces traces with two copies of the input packet α. Union ($p + q$) produces the union of the traces produced by p and q. Sequential composition ($p \cdot q$) produces the concatenation of the traces produced by p and q, where the last packet in output traces of p is used as the input to q. Finally, iteration ($p^\star$) produces the union of the traces produced by concatenation of p, iterated zero or more times.

2.1.3 NetKAT Automata

Despite the fact that assignments and tests in NetKAT programs are always against constants, the behavior of NetKAT programs can be complicated, particularly when conditionals, iteration, and multiple packet fields are involved. To provide a more operational view of the semantics, NetKAT automata were introduced by Foster et al. [55] as acceptors of guarded strings.

Definition 1. *A NetKAT automaton (NKA) is a four-tuple $(S, s_0, \delta, \varepsilon)$, where S is a finite set of states, $s_0 \in S$ is the start state, δ and ε are the transition and observation functions, respectively*[2]*:*

$$\delta \colon S \to \mathsf{Pk} \to \mathsf{Pk} \to S \qquad\qquad \varepsilon \colon S \to \mathsf{Pk} \to \mathsf{Pk} \to 2$$

We will write $\varepsilon_{\alpha\beta}(s)$ as a shorthand for $\varepsilon(s)(\alpha)(\beta)$ and $\delta_{\alpha\beta}(s)$ as a shorthand for $\delta(s)(\alpha)(\beta)$.

Semantically, an NKA $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ accepts a language $\mathcal{L}(\mathcal{M}) \subseteq \mathsf{GS}$ containing all strings $w \in \mathsf{GS}$ for which $\mathcal{M}(s_0, w) = \top$, where:

$$\mathcal{M}(s, \alpha \cdot \beta) = \varepsilon_{\alpha\beta}(s) \qquad\qquad \mathcal{M}(s, \alpha \cdot \beta \cdot \mathsf{dup} \cdot w') = \mathcal{M}(\delta_{\alpha\beta}(s), \beta \cdot w') \tag{2.2}$$

[2]We use 2 to denote the Booleans, $\{\bot, \top\}$.

Note how in the semantics of state s of NetKAT automata we read two packets $\alpha\beta$ moving to state $\delta_{\alpha\beta}(s)$ *but* we retain β in the string that remains! So, unlike traditional regular expressions and automata, NetKAT is stateful—in effect, the packet β has not been fully consumed, but is used as a *carry-on* packet to remember the last step of the automaton's execution. This peculiarity of NetKAT complicates the semantics (i.e., it is not a straightforward language semantics) and designing equivalence and minimization procedures, which have to account for the carry-on packet. We could fold the packet into the state of the automaton and carry out both equivalence and learning using classical automata, but this results in enormous state blowup and is therefore impractical. This is the key challenge for designing both a decision procedure and a learning algorithm we address in this dissertation.

2.1.4 Bisimulation of NetKAT Automata

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

Given two automata $(S, s_0, \delta, \varepsilon')$ and $(S', s'_0, \delta', \varepsilon')$, we check their equivalence by considering all possible input packets separately. We run the two automata in parallel, starting from an input packet α at their respective start states. We first check that the immediate outputs $\varepsilon_{\alpha\beta}(s_0)$ and $\varepsilon'_{\alpha\beta}(s'_0)$ are equal for all $\beta \in \mathsf{Pk}$. If so, we check that the transitions $\delta(s_0, \alpha)$ and $\delta'(s'_0, \alpha)$ are equivalent, by recursively checking the equivalence of the states $\delta_{\alpha\beta}(s_0)$ and $\delta'_{\alpha\beta}(s'_0)$ for all $\beta \in \mathsf{Pk}$.

We can implement this strategy using a work list algorithm. The work list

Input: A pair of NetKAT automata $(S, s_0, \delta, \varepsilon)$, $(S', s'_0, \delta', \varepsilon')$.
Returns: A Boolean indicating whether the automata are equivalent.
$W \leftarrow \{(s_0, s'_0, \alpha) \mid \alpha \in \mathsf{Pk}\}$;
while W changes **do**
  for $(s, s', \alpha) \in W$ **do**
    if $\varepsilon_{\alpha\beta}(s) \neq \varepsilon'_{\alpha\beta}(s')$ for some $\beta \in \mathsf{Pk}$ **then return false**;
    $W \leftarrow W \cup \{(\delta_{\alpha\beta}(s), \delta'_{\alpha\beta}(s'), \beta) \mid \beta \in \mathsf{Pk}\}$
return true;

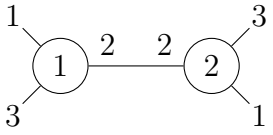
Figure 2.1: Bisimulation algorithm à la Hopcroft & Karp [66]

contains triples (s, s', α), where s and s' are states in the two automata, and α is an input packet. Initially, the work list contains (s_0, s'_0, α) for all packets α. We then repeatedly take triples (s, s', α) from the work list, and check that $\varepsilon_\alpha(s) = \varepsilon'_\alpha(s')$ (as predicates) and add $(\delta_{\alpha\beta}(s), \delta'_{\alpha\beta}(s'), \beta)$ to the work list for all $\beta \in \mathsf{Pk}$. If at any point we find that $\varepsilon_\alpha(s) \neq \varepsilon'_\alpha(t)$, then the automata are not equivalent. If the work list stabilizes, the automata are equivalent. This algorithm is shown in Figure 2.1.

Of course, the issue with this algorithm is that the space of packets is huge. In Chapter 3, we will develop techniques for working with symbolic representations of packets and automata, allowing us to represent and manipulate large sets of packets and to efficiently check equivalence of automata without explicitly enumerating the space of packets.

2.1.5 Encoding Networks in NetKAT

We encode the behavior of networks in NetKAT using a similar approach as in prior work [5, 55, 101]. For example, suppose we have a toy network with two switches with three ports each, connected by a link between both switches' port 2:



We use two special fields sw and pt, which denote a packet's location at a particular switch and port. Using these fields, we can encode the network topology in NetKAT—e.g., the expression,

$$\mathsf{T} \triangleq \mathsf{pt}{=}1 + \mathsf{pt}{=}3 + \mathsf{pt}{=}2 \cdot (\mathsf{sw}{=}1 \cdot \mathsf{sw}{\leftarrow}2 + \mathsf{sw}{=}2 \cdot \mathsf{sw}{\leftarrow}1)$$

Similarly, we can encode routing policies for individual switches in the network above:

$$\mathsf{R} \triangleq \mathsf{pt}{=}1 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{pt}{=}2 \cdot \mathsf{pt}{\leftarrow}1,$$

which encodes the policy: "At either switch, forward packets arriving on port 1 via port 2, and forward packets arriving on port 2 via port 1." This policy drops packets arriving at port 3.

Using these encodings, we can compose the topology (T) and routing policy (R) into one expression capturing the network's *transfer function*: $\mathsf{R} \cdot \mathsf{T}$. We combine this with ingress and egress predicates P_i and P_f to encode the full end-to-end behavior: $\mathsf{P}_i \cdot (\mathsf{R} \cdot \mathsf{T} \cdot \mathsf{dup})^\star \cdot \mathsf{P}_f$. Here we use a dup to denote that we want to log one copy of each packet for each network "hop" (i.e. an update to sw in T). The star indicates that we are interested in all paths allowed by this policy on this topology. The expressions P_i and P_f are predicates which constrain the initial and final packets we are interested in. For example, to investigate all paths by which Switch 1 (Port 1) can reach Switch 2 (Port 1), we might choose $\mathsf{P}_i \triangleq \mathsf{sw}{=}1 \cdot \mathsf{pt}{=}1$, and $\mathsf{P}_f \triangleq \mathsf{sw} = 2 \cdot \mathsf{pt} = 1$. More complicated routing policies may involve other pertinent fields, perhaps dst, encoding a destination switch.

2.2 Active Learning of Finite Automata: The MAT Framework

Dana Angluin established the field of active learning of automata by proposing the *Minimally Adequate Teacher* (MAT) framework [7].

In the MAT framework, a Learner seeks to discover an unknown language by interacting with a Teacher who can answer two types of queries:

Membership: The Learner sends a word u to the Teacher, who answers "yes" if u belongs to the language or "no" if it does not.

Equivalence: The Learner sends a hypothesis automaton $\mathcal{H}$ to the Teacher, who either confirms that $\mathcal{H}$ is correct or provides a counterexample.

The MAT framework has been the basis for active model learning since its introduction in the late 1980s, providing the basis for a variety of algorithms, data structures, and proof techniques (e.g. $\mathrm{L}^\star$ [7], KV [70], TTT [67], $L^\sharp$ [110], etc.). The framework has been applied in practice to learn models for a wide variety of closed-box systems [2, 1]. Angluin's learner discovers a deterministic finite automaton (DFA), but the MAT framework has also been instantiated for other kinds of automata—e.g Mealy machines, weighted automata [20], and nominal automata [93] to name a few.

In a practical instantiation of the MAT framework, we assume that the system can be appropriately modeled by a finite automaton where the symbols of the input language correspond to "input events" for the system, such as button presses, packets received, or functions called. Accordingly, the final states of the automaton

we will learn correspond to the valid states that the system can reach via valid input sequences. Thus, membership queries are carried about by trying an input sequence and observing the behavior. Equivalence queries are often approximated in practice.

After establishing notation, we review Angluin's $\mathtt{L}^\star$ algorithm, which will form the basis for contributions we will make in Chapter 4 and Chapter 5.

2.2.1 Strings, Languages, and Automata

Fix an alphabet of symbols, Σ. We let $a \in \Sigma$ denote an arbitrary symbol and ε the empty string. For strings (also called words) $u, v \in \Sigma^*$, we write uv for the concatenation of u and v (we will occasionally write $u.v$ for emphasis). We use capital letters $U, V \subseteq \Sigma^*$ to denote languages—i.e. sets of strings. Concatenation of languages U, V is written $U \cdot V = \{uv \mid u \in U, v \in V\}$. The set of prefixes for a string s is written $\text{prefixes}(s) = \{u \mid s = uv, u, v \in \Sigma^*\}$. Likewise, $\text{suffixes}(s) = \{v \mid s = uv, u, v \in \Sigma^*\}$. We sometimes even take prefixes of a set $S \subseteq \Sigma^*$, written $\mathsf{prefixes}(S) = \{u \mid u \in \mathsf{prefixes}(s), s \in S\}$. A set $S \subseteq \Sigma^*$ is called *prefix-closed* (*suffix-closed*) if $S = \mathsf{prefixes}(S)$ ($S = \mathsf{suffixes}(S)$). Finally, we denote the symmetric difference of two sets by $S \oplus S' = S - S' \cup S' - S$.

An important class of languages is that of *regular languages*, which are the languages accepted by deterministic finite automata (DFAs). A DFA is a five-tuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the alphabet, $\delta \colon Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ are the accepting states. The transition function is extended to strings inductively by $\hat{\delta} \colon Q \times \Sigma^* \to Q$, where for any $q \in Q$, $\hat{\delta}(q, \varepsilon) = q$ and for any $a \in \Sigma, u \in \Sigma^*$,

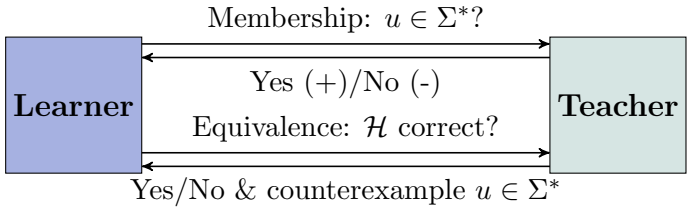


Figure 2.2: The Minimally Adequate Teacher framework.

$\hat{\delta}(q, au) = \hat{\delta}(\delta(q, a), u)$. A string u is accepted by the automaton D iff $\hat{\delta}(q_0, u) \in F$. Moreover, the language accepted by D, written $L(D)$, is the set of all such strings: $L(D) = \{u \in \Sigma^* \mid \hat{\delta}(q_0, u) \in F\}$.

Regular languages have unique minimal representatives: for every regular language, there is a unique DFA with a minimal number of states. There are different ways to build such minimal DFA corresponding to a regular language, but the one that is used to prove correctness of $\mathtt{L}^\star$ is based on so-called Myhill-Nerode equivalence classes. Given a language L and a word $s \in \Sigma^*$, the Myhill-Nerode equivalence class of s, denoted $[s]_L$ is the set of all words s' satisfying: $s \equiv_L s' \overset{\triangle}{\iff} \forall_{e \in \Sigma^*} \cdot (se \in L \iff s'e \in L)$. Myhill-Nerode equivalence classes provide an alternative characterization of regular languages: a language is regular iff it has a finite number of Myhill-Nerode equivalence classes. Furthermore, there is a one-to-one correspondence between the states of the minimal automaton accepting a regular language L and its Myhill-Nerode equivalence classes.

2.2.2 $\mathtt{L}^\star$ Data Structures

$\mathtt{L}^\star$ is an algorithm that implements a Learner according to the interfaces in the MAT framework (see Figure 2.2). The core data structure used in the algorithm is an observation table, which records the information gathered from membership queries and the counterexamples obtained from equivalence queries.

Definition 2 (Observation Table)**.** *Given a language* $L \subseteq \Sigma^*$*, an* observation table *(wrt* L*) is a triple* (S, E, T)*, where:*

- $S \subseteq \Sigma^*$ *is a prefix-closed set of "accessor strings";*
- $E \subseteq \Sigma^*$ *is a suffix-closed set of "distinguishing strings";*
- $T \colon (S \cup S \cdot \Sigma) \cdot E \to \{+, -\}$ *is a map on a finite set of words defined by*

$$T(u) = \begin{cases} + & u \in L \\ - & u \notin L \end{cases}$$

Note that once L is fixed, T is fully determined by S and E, so sometimes we refer to the observation table as simply (S, E). We can think of $L \subseteq \Sigma^*$ as a function $L \colon \Sigma^* \to \{+, -\}$ and then simply write $T(u) = L(u)$. Every u in the domain of T is obtained as a concatenation of a string $s \in S \cup S \cdot \Sigma$ and $e \in E$.

Intuitively, one can think of an observation table as a snapshot of a language L. When L is regular, we will show how the table is organized in a way that its rows can be used to recover the Myhill-Nerode equivalence classes of L, and therefore the minimal deterministic automaton. The use of accessor strings and distinguishing strings to name the elements of S and E will become clear when we explain how to build a DFA from a table.

Consider the regular language $L = \{w \in \{a, b\}^* \mid w \text{ has at least one } a\}$, and sets $S = \{\varepsilon, a\}$, and $E = \{\varepsilon, b\}$. The observation table (S, E) is given on the left in Figure 2.3. When depicting the table, we divide it into an upper part and a lower part. The rows in the upper part are labeled by strings in S, whereas the rows of the lower part are labeled by strings in $S \cdot \Sigma - S = \{sa \mid s \in S, a \in \Sigma,\ sa \notin S\}$. The strings in E label the columns of the table. The entries of the table correspond to the function T.

	ε	b
ε	-	-
a	+	+
b	-	-
aa	+	+
ab	+	+

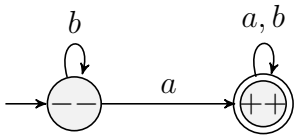


Figure 2.3: An observation table and corresponding automaton.

It is important to note that depicting the finite map $T\colon (S \cup S \cdot \Sigma) \cdot E \to \{+, -\}$ as a table leads to some repetition. For example, the entry in row a, column b must match the entry in row ab, column ε—i.e., since $a.b = ab.\varepsilon$ we also have $T(a.b) = T(ab.\varepsilon)$. It will be convenient to access the rows of the table as follows:

$$\mathsf{row}\colon S \cup S \cdot \Sigma \to E \to \{+, -\} \qquad \mathsf{row}(s)(e) = T(se).$$

We will sometimes abuse notation and apply row to a set of strings to obtain a set of rows. For example, in the table above $\mathsf{row}(S)$ is the set of distinct rows in the upper part of the table—$\mathsf{row}(S) = \{\{\varepsilon \mapsto +, b \mapsto +\}, \{\varepsilon \mapsto -, b \mapsto -\}\}$. Additionally, when E is clear from the context, we will sometimes omit the column indices, writing $\mathsf{row}(S) = \{++, --\}$.

To build a DFA from an observation table, we will use as states the upper rows of the table, but the well-definedness of the construction requires two properties to be satisfied[3].

Definition 3 (Closedness and Distinctness)**.** *A table (S, E, T) is* closed *if for each string $s \in S$ and letter $a \in \Sigma$, we have $\mathsf{row}(sa) \in \mathsf{row}(S)$. It is* distinct *if all the rows labeled by $s \in S$ are distinct: $s, s' \in S \Rightarrow \mathsf{row}(s) \neq \mathsf{row}(s')$.*

Note that the example table above is closed and distinct. We can now make the connection between observation tables and finite automata precise.

[3]Readers familiar with $\mathtt{L}^\star$ will notice we jettisoned consistency for a simpler property—keeping the upper rows of the table distinct. This optimization is due to Maler and Pnueli [89].

1. Initialize $S = \{\varepsilon\}$, and $E = \{\varepsilon\}$
2. While (S, E, T) is not closed, do:
   $S \leftarrow S \cup \{sa\}$,
   where $s \in S, a \in \Sigma$, but $\mathsf{row}(sa) \notin \mathsf{row}(S)$.
3. Conjecture $M = \mathcal{D}(S, E, T)$:
   (a) If M is correct, halt.
   (b) Otherwise, $E \leftarrow E \cup \mathsf{suffixes}(c)$, where c is the provided counterexample. Goto 2.

Figure 2.4: Angluin's $\mathtt{L}^\star$ (Maler-Pnueli version). Whenever S and E change, T is updated using membership queries. Note the extensions of S and E preserve distinctness as an invariant.

Definition 4 (DFA associated with a table)**.** *Given a closed and distinct table (S, E, T), with distinct rows in S, we associate a DFA, $\mathcal{D}(S, E, T) = (Q, \Sigma, \delta, q_0, F)$, where:*

$$Q = \mathit{row}(S) \qquad \delta(\mathit{row}(s), a) = \mathit{row}(sa)$$

$$q_0 = \mathit{row}(\varepsilon) \qquad F = \{\mathit{row}(s) \mid \mathit{row}(s)(\varepsilon) = +\}.$$

The above definition relies on the fact that S is prefix-closed and E is suffix-closed and so both contain ε. Moreover, the transition function δ is well-defined whenever the table is closed and distinct. The first property ensures that $\mathsf{row}(sa) \in Q$, whereas the second ensures the well definedness of $\delta(\mathsf{row}(s), a)$.

The DFA corresponding to the example observation table is depicted on the right in Figure 2.3. The start state is indicated with an incoming arrow, and final state with a double circle. Each state is labeled by its unique row vector.

2.2.3 $\mathtt{L}^\star$ Learner

We are now ready to present $\mathtt{L}^\star$, the algorithm in which a Learner incrementally builds an observation table based on interactions with a Teacher as depicted in Figure 2.2. The $\mathtt{L}^\star$ Learner is shown in Figure 2.4. It starts with $S = E = \{\varepsilon\}$ and then explores potential new rows by examining the rows in the lower part of the table (labeled by strings in $S \cdot \Sigma - S$). If no new rows are found (i.e., as compared to the ones in $\mathsf{row}(S)$), then we conclude that the table is closed. If, on the other hand, a row in the lower part of the table does not appear in the upper part, there is a closedness defect, which can be repaired by adding another string to S. In essence, this repair moves a row from the lower part of the table to the upper part. Accordingly, we generate new rows in the lower part until there is a row in the table for each $s \in S\Sigma$. Once the Learner finds a closed table (distinctness is maintained by construction), it poses an equivalence query to the Teacher. If the hypothesis is wrong the Teacher returns a counterexample which is used to extend the columns of the table.

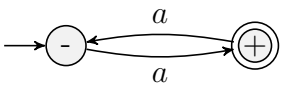
$\mathtt{L}^\star$ can also be understood as incrementally discovering the Myhill-Nerode equivalence classes for the Teacher's language $\mathcal{L}$, which correspond to the states of the minimal DFA. Since the start state always corresponds to the equivalence class for the empty string ε, it starts with the observation table where $S = \{\varepsilon\}$ and $E = \{\varepsilon\}$. Each hypothesis is a refinement of the previous guess, getting closer and closer to the correct minimal automaton. Termination of the algorithm follows as every closedness defect repaired or counterexample processed, results in an automaton with more states than the previous hypothesis.

2.2.4 $\mathtt{L}^\star$ Example

Suppose we choose $\Sigma = \{a\}$ and the Teacher knows the language L described by the regular expression $a(aaa)^*$. The Learner begins by building the observation table below, on the left:

	ε
ε	-
a	+

	ε
ε	-
a	+
aa	-



(2.3)

The table on the left is not closed, because the row for a is not represented in the upper part of the table, hence the Learner extends S and this yields the table above in the middle. This table is closed so the Learner conjectures the corresponding DFA on the right.

The Teacher returns counterexample aaa (which is accepted by the automaton in eq. (2.3) but should be rejected).[4] The Learner processes this counterexample by extending E with its suffixes $\{\varepsilon, a, aa, aaa\}$, yielding the table on the left below:

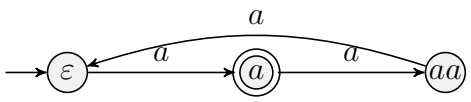
	ε	a	aa	aaa
ε	-	+	-	-
a	+	-	-	+
aa	-	-	+	-

	ε	a	aa	aaa
ε	-	+	-	-
a	+	-	-	+
aa	-	-	+	-
aaa	-	+	-	-

The table on the left above is not closed, as $\mathsf{row}(aa)$ is different from all other rows. To fix this, the Learner adds aa to S yielding the table above, on the right.

[4] In general, there is no requirement for the Teacher to return the shortest counterexample and, indeed, the original complexity analysis of $\mathtt{L}^\star$ takes into account that longer counterexamples might be returned resulting in larger than needed tables. Note that, however, minimality is never compromised as equal rows will be mapped to the same state in the automaton.

This table is closed and distinct, so the Learner makes the following conjecture which the Teacher accepts:



(2.4)

Angluin [7] showed that the $\mathtt{L}^\star$ learner finds the minimal DFA for the target language using a polynomial number of membership queries—results which also apply to the Maler-Pnueli version. However, this property relies on the assumption that the Teacher can answer membership and equivalence queries.

CHAPTER 3

FAST NETWORK VERIFICATION WITH NETKAT

This chapter is adapted from the following published conference paper:

Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. KATch: A fast symbolic verifier for NetKAT. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2024. Association for Computing Machinery.

3.1 Introduction

In the automata-theoretic approach to verification, programs and specifications are each encoded as automata, and verification tasks are reduced to standard questions in formal language theory—e.g., membership, emptiness, containment, etc. [111]. The approach became popular in the mid-1980s, driven by use of temporal logics and model checking in hardware verification, and it remains a powerful tool today. In particular, automata provide natural models of phenomena like transitive closure, which arises often in programs but is impossible to express in pure first-order logic.

NetKAT, a domain-specific language for specifying and verifying the behavior of network data planes. Let us recall the basic syntax of NetKAT, as introduced in Section 2.1:

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

Unlike ordinary regular expressions, which are stateless, NetKAT programs manipulate state in packets. Accordingly, NetKAT's atoms are not letters, but *actions* that either drop or forward the current packet ($\bot$ or $\top$), modify a header field ($f \leftarrow v$), test a header field against a value ($f = v$, $f \neq v$), or append the current packet to the trace (dup). The regular expression operations copy and forward a packet to two different programs ($p + q$), sequence two programs ($p \cdot q$), or loop a program ($p^\star \equiv \top + p + p{\cdot}p + p{\cdot}p{\cdot}p + \cdots$).

A NetKAT program thus gives a declarative specification of the network's global behavior in terms of sets of traces. Specifically, we can model the location of a

packet in the network using a special header field (usually named sw for "switch"), that we can update ($\mathrm{sw} \leftarrow \ell$) to logically move the packet from its current location to a new location ℓ. Given a declarative description of the intended behavior, the NetKAT compiler generates a set of local forwarding tables for individual switches that together realize the global behavior [53, 101].

Moreover, NetKAT not only plays a role as an implementation language, but also as a language for expressing verification queries, analogous to verification tools powered by SMT solvers [83, 14, 108]. In particular, as NetKAT includes a union operator ("+"), containment can be reduced to program equivalence—i.e., $p \sqsubseteq q$ if and only if $p + q \equiv q$. Hence, unlike other contemporary tools, which rely on bespoke encodings and algorithms for checking network properties like reachability, slice isolation, etc. [69, 71, 118, 52], NetKAT allows a wide range of practical verification questions to be answered using the same foundational mechanism, namely program equivalence. For example,

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

NetKAT's dup primitive is a key construct that enables reducing a wide range of network-specific properties to program equivalence. Recall that dup appends the headers of the current packet to the trace. Hence, to verify properties involving the network's internal behavior, we add intermediate packets to the trace using

dup, making them relevant for program equivalence. Conversely, if we only care about the input-output behavior of the entire network, we can omit dup from the specification, so intermediate packets are not considered by the check for program equivalence.

The story so far is appealing, but there is a major fly in the ointment. To decide program equivalence, the automata-theoretic approach relies on the translation from NetKAT programs to automata. But standard NetKAT automata have an enormous space of potential transitions—i.e., the "alphabet" has a "character" for every possible packet. So using textbook algorithms for building automata and checking equivalence would clearly be impractical. In fact, NetKAT equivalence is PSPACE-complete [5]. Instead, what's needed are symbolic techniques for encoding the state space and transition structure of NetKAT automata that avoid combinatorial blowup in the common case.

Prior work on symbolic automata provides a potential solution to the problems that arise when working with large (or even infinite) alphabets. The idea is to describe the transitions in the automata using logical formulas [37, 38] or Binary Decision Diagrams (BDDs) [100] rather than concrete characters. Algorithms such as membership, containment, and equivalence can then be reformulated to work with these symbolic representations. Symbolic automata have been successfully applied to practical problems such as building input sanitizers for Unicode, which has tens of thousands of characters [65]. However, NetKAT's richer semantics precludes the direct adoption of standard notions of symbolic automata. In particular, NetKAT's transitions describe not only predicates but also transformations on the current packet. As Pous writes, it "seems feasible" to generalize his work to NetKAT, but "not straightforward" [100].

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

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

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

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

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

Contribution 1: Efficient symbolic representations (Section 3.2). We design a symbolic data structure called Symbolic Packet Programs (SPPs) for representing both sets of packets and transformations on packets in a symbolic manner. SPPs generalize classic BDDs and are asymptotically more efficient than the representations used in prior work on NetKAT. In particular, SPPs support efficient sequential composition ($e_1 \cdot e_2$) and can be made *canonical* which, with hash consing, allows us to check equivalence of SPPs in constant time.

Contribution 2: Symbolic Brzozowski derivatives (Section 3.3). We introduce a new form of symbolic Brzozowski derivative to produce automata for NetKAT programs. Our approach naturally and efficiently supports the extended "negative" logical operators mentioned above, in contrast to prior approaches that only support "positive" operators like union.

Contribution 3: Symbolic bisimilarity checking (Section 3.4). Building on the foundation provided by SPPs and deterministic NetKAT automata, we develop symbolic algorithms for checking bisimilarity. We present a symbolic version of the standard algorithm that searches in the forward direction through the state space of the automata under consideration. We also present a novel backward algorithm that computes symbolic counter-examples to equivalence—i.e., the precise set of input packets for which equivalence fails.

Contribution 4: KATch implementation (Section 3.5) and evaluation (Section 3.6). Finally, we present a new verification tool, called KATch, implemented in Scala. KATch implements symbolic bisimilarity checking, including an extended set of extended logical operators, as well as symbolic counter-examples. We evaluate the performance of KATch on a variety of real-world topologies, and show that it efficiently answers a variety of verification queries and scales to much larger networks than prior work. Due to its use of our efficient data structures, KATch is several orders of magnitude faster than prior implementations of NetKAT on these realistic examples. Moreover, KATch is faster than prior work on synthetic combinatorial benchmarks by arbitrary large factors.

3.2 Symbolic NetKAT Representations

In this section, we develop symbolic techniques to trace an entire set of packets at once, vastly reducing the number of iterations required to compute a bisimulation in practice. These techniques will allow us to revise the approach of Figure 2.1 with a much more efficient version in Section 3.4.

First, we introduce a representation for symbolic packets, which represent sets of packets, or equivalently, the fragment of NetKAT where all atoms are tests. This representation is essentially a natural n-ary variant of binary decision diagrams (BDDs) [27]. Second, we introduce Symbolic Packet Programs (SPPs), a new representation for symbolic transitions in NetKAT automata, representing the dup-free fragment of NetKAT—i.e., the fragment in which all atoms are tests or assignments. The primary challenge is to design this representation so that it is efficiently closed under the NetKAT operations. Furthermore, we need to be able

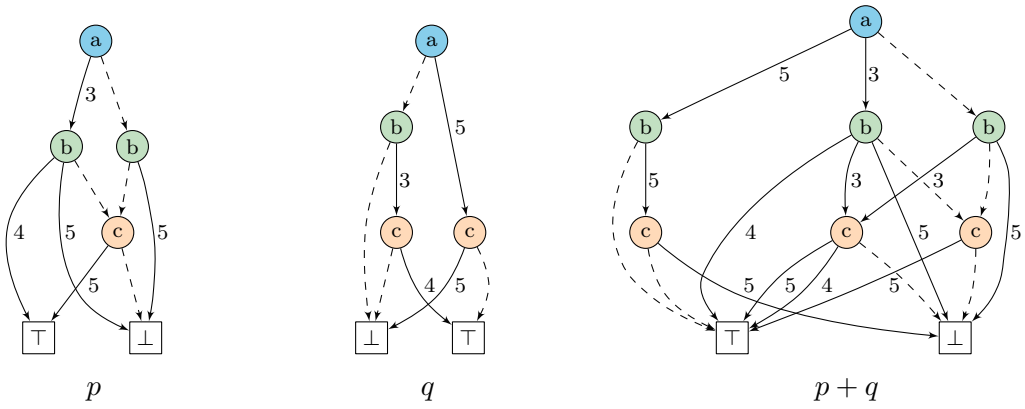


Figure 3.1: Examples of SPs, where $p \triangleq (a = 3 \cdot b = 4 \ + \ b \neq 5 \cdot c = 5)$ and $q \triangleq (b = 3 \cdot c{=}4 + a{=}5 \cdot c{\neq}5)$.

to efficiently compute the transition of a symbolic packet over a SPP, both forward and backward.

3.2.1 Symbolic Packets

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

$$p \in \mathsf{SP} ::= \bot \mid \top \mid \underbrace{\mathsf{SP}(f,\ \{\ldots, v_i \mapsto q_i, \ldots\},\ q)}_{\equiv\ \sum_i f{=}v_i \cdot q_i \ + \ (\prod_i f{\neq}v_i) \cdot q}$$

That is, symbolic packets form an n-ary tree, where each child q_i is labeled with a test of the current field f, and the default case q is labeled with the negation of the other tests. Hence, a given concrete packet has a unique path through the tree.

The following conditions need to be satisfied for a symbolic packet to be in canonical form, inspired by the analogous properties of BDDs:

Reduced If a child q_i is equal to the default case q, it is removed. If only the default case remains, the symbolic packet is reduced to the default case itself.

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

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

Representation As with BDDs, we share nodes in the tree, making the representation of a symbolic packet a directed acyclic graph, as shown in Figure 3.1. Vertices are labeled with the packet field they test. Solid arrows labeled with a number encode the test value, and dashed arrows represent a default case. The sinks of the graph are labeled with $\top$ or $\bot$, indicating membership in the set. On the left, we have a symbolic packet representing all packets where $a = 3$ and $b = 4$, or $b \neq 5$ and $c = 5$. In the middle, we have a symbolic packet representing all packets where $b = 3$ and $c = 4$, or $a = 5$ and $c \neq 5$. On the right, we have the union of these symbolic packets.

Operations Symbolic packets are closed under the NetKAT operations:

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

As defined in Figure 3.2, operations on symbolic packets are computed by traversing the two trees in parallel, recursively taking the operation of the children, making sure to maintain canonical form using the smart constructor. Repetition $p^\star \triangleq \top$ is trivial for symbolic packets. All other operations are defined in Section 3.9.

Primitive tests:

$(f{=}v) \triangleq \mathsf{SP}(f, \{v \mapsto \top\}, \bot) \qquad (f{\neq}v) \triangleq \mathsf{SP}(f, \{v \mapsto \bot\}, \top)$

Operations, base cases:

$$\begin{array}{llllll}
\top \mathbin{\hat{+}} \top \triangleq \top & \top \mathbin{\hat{\cdot}} \top \triangleq \top & \top \mathbin{\hat{\cap}} \top \triangleq \top & \top \mathbin{\hat{\oplus}} \top \triangleq \bot & \top \mathbin{\hat{-}} \top \triangleq \bot \\
\top \mathbin{\hat{+}} \bot \triangleq \top & \top \mathbin{\hat{\cdot}} \bot \triangleq \bot & \top \mathbin{\hat{\cap}} \bot \triangleq \bot & \top \mathbin{\hat{\oplus}} \bot \triangleq \top & \top \mathbin{\hat{-}} \bot \triangleq \top \\
\bot \mathbin{\hat{+}} \top \triangleq \top & \bot \mathbin{\hat{\cdot}} \top \triangleq \bot & \bot \mathbin{\hat{\cap}} \top \triangleq \bot & \bot \mathbin{\hat{\oplus}} \top \triangleq \top & \bot \mathbin{\hat{-}} \top \triangleq \bot \\
\bot \mathbin{\hat{+}} \bot \triangleq \bot & \bot \mathbin{\hat{\cdot}} \bot \triangleq \bot & \bot \mathbin{\hat{\cap}} \bot \triangleq \bot & \bot \mathbin{\hat{\oplus}} \bot \triangleq \bot & \bot \mathbin{\hat{-}} \bot \triangleq \bot
\end{array}$$

Operations, inductive case:

$\mathsf{SP}(f, b_p, d_p) \mathbin{\hat{\pm}} \mathsf{SP}(f, b_q, d_q) \triangleq \mathsf{sp}(f, b', d_p \mathbin{\hat{\pm}} d_q)$
where $b' = \{v \mapsto b_p(v; d_p) \mathbin{\hat{\pm}} b_q(v; d_q) \mid v \in \mathsf{keys}(b_p \cup b_q)\}$

Expansion:

$p \equiv \mathsf{SP}(f, \emptyset, p) \qquad \text{if} \qquad p \in \{\top, \bot, \mathsf{SP}(f', b, d)\}$ where $f \sqsubset f'$

Smart constructor:

$\mathsf{sp}(f, b, d) \triangleq$ if $b' = \emptyset$ then d else $\mathsf{SP}(f, b', d)$
where $b' \triangleq \{v \mapsto p \mid v \mapsto p \in b, p \neq d\}$

Figure 3.2: Definition of the SP operations. The inductive case is identical for all operations (indicated by $\hat{\pm}$), and applies when both SPs test the same field. Expansion inserts a trivial SP node to reduce the remaining cases to the inductive case. The notation $b(v; d)$ means the child $b(v)$ if $v \in \mathsf{keys}(b)$, or the default case d otherwise.

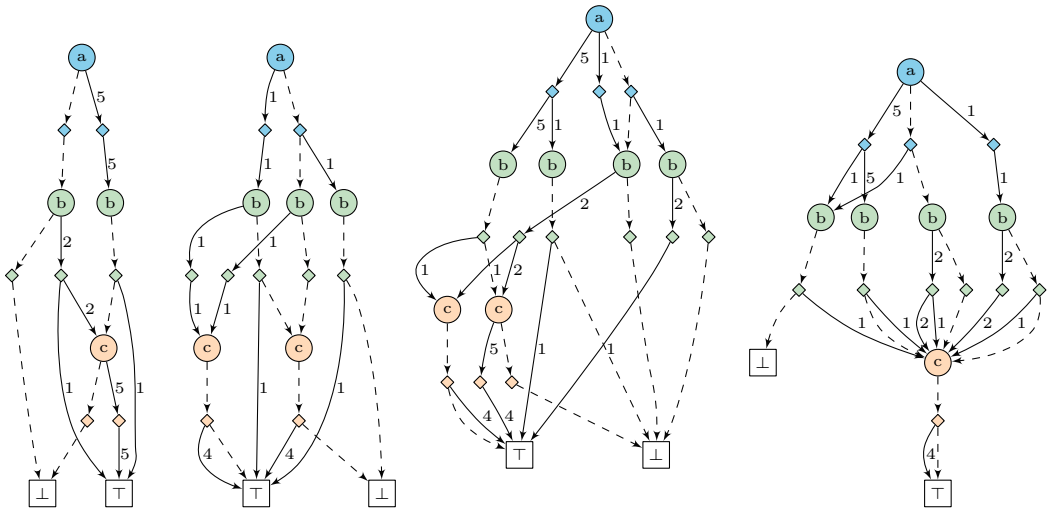
Specifications Each of these operations is uniquely specified by two correctness conditions:

1. They semantically match their counterpart: $[\![p \mathbin{\hat{+}} q]\!] = [\![p{+}q]\!]$ for all $p, q \in \mathsf{SP}$.
2. They maintain canonical form: $p \mathbin{\hat{+}} q$ is reduced and ordered if p, q are.

For further formal treatment of SPs, see Section 3.9.

3.2.2 Symbolic Transitions

We now introduce a representation for symbolic transitions in NetKAT automata. Whereas symbolic packets represent the fragment of NetKAT where all atoms are tests, symbolic transitions correspond to the larger dup-free fragment, where



p q $p \cdot q$ $(p \oplus q)^\star$

Figure 3.3: Examples of SPPs, where $p \triangleq (a = 5 + b = 2) \cdot (b \leftarrow 1 + c = 5)$, and $q \triangleq (b{=}1 + c{\leftarrow}4 + a{\leftarrow}1 \cdot b{\leftarrow}1)$.

all atoms are tests or assignments. This introduces additional challenges for a canonical representation, as well as for the operations. For instance, sequential composition is no longer equivalent to intersection, and the star operator is no longer trivial.

$$p \in \mathsf{SPP} ::= \bot \mid \top \mid \underbrace{\mathsf{SPP}(f, \{\ldots, v_i \mapsto \{\ldots, w_{ij} \mapsto q_{ij}, \ldots\}, \ldots\}, \{\ldots, w_i \mapsto q_i, \ldots\}, q)}_{\equiv \quad \sum_i f{=}v_i \cdot \sum_j f{\leftarrow}w_{ij} \cdot q_{ij} \;+\; (\prod_i f{\neq}v_i) \cdot (\sum_i f{\leftarrow}w_i \;+\; (\prod_i f{\neq}w_i) \cdot q)}$$

Like SPs, SPPs have two base cases, $\top$ and $\bot$. Also like SPs, SPPs test a field f of the input packet against a series of values $v_0, \ldots, v_n$, with a default case $f \neq v_0 \cdots f \neq v_n$. However, instead of continuing recursively after the test, SPPs non-deterministically assign a value w_{ij} to the field f of the output packet, and continue recursively with the corresponding child q_{ij}. This way, SPPs can output more than one packet for a given input packet, and can also output packets with different values for the same field. The default case is further split into two cases, one where the field f is non-deterministically assigned a new value (like in the other cases), and an identity case where the field f keeps the same value as the input packet. However, the packet for the latter case is not produced when

the input packet's field f had a value that could also be produced by the explicit assignments in the default case. This ensures that for a given input packet, a given output packet is produced by a unique path through the SPP.

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

Reduced If a child q_{ij} is equal to $\perp$, it is removed (with the exception of the default-identity case, which is always kept). If one of the default-assignment cases for a value w is $\perp$, it is removed, but to keep the behavior equivalent, an additional test for the same value w is added with a copy of the default-assignments (if a test for the value w was already present, nothing is added)[1]. Further, each of the non-default cases is analyzed, and if we determine that it behaves semantically like the default case for that input value, then it is removed. If only the default case remains, the SPP is reduced to the default case itself.

Ordered A path down the tree always follows the same order of fields, and both the tests and the assignments are ordered by the value v_i or w_{ij}.

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

Representation Like symbolic packets, we represent SPPs as a directed acyclic graph, with duplicate nodes shared. Examples of SPPs are shown in Figure 3.3. Vertices are labeled with the packet field that they test. Solid arrows labeled with a number represent the tests, and dashed arrows represent the default case. Each test is followed by a non-deterministic assignment of a new value to the

[1] see the formal definition in Section 3.10.1

field, indicated by the small diamonds. In the default case, the non-deterministic assignment also has an identity case (keeping the value of the field unchanged), indicated by a dashed arrow emanating from the diamond.

Consider the action of the first SPP in Figure 3.3 on the concrete packet $a = 5 \cdot b = 3 \cdot c = 5$. The first test $a = 5$ succeeds, and the field a is assigned the value 5. The b field is then tested, but the b node only has a default case. The default case does have a non-deterministic assignment, which can set the b field to the new value 1, or it can keep the old value 3. In case the new value of b is 1, the packet is immediately accepted by the $\top$ node, so the packet $a = 5 \cdot b = 1 \cdot c = 5$ is produced. In case the old value of b is kept, the c field is tested, and in our case the value of the c field is 5. In this case, the value of the field is unchanged, and the packet $a = 5 \cdot b = 3 \cdot c = 5$ is produced. In summary, for input packet $a = 5 \cdot b = 3 \cdot c = 5$, the SPP produces packets $a = 5 \cdot b = 1 \cdot c = 5$ and $a = 5 \cdot b = 3 \cdot c = 5$.

When we sequentially compose the two SPPs on the left, we get the third SPP shown. In other words, if we take a concrete packet, and first apply the first SPP, and then apply the second SPP to all of the resulting packets, then we get the same result as if we apply the SPP on the right directly to the concrete packet. Sequential composition is a relatively complex operation, but algorithmically it is a key strength of our representation and reduced/ordered invariant. To understand why, consider taking a concrete packet, and applying the first SPP to it. Once we have applied the a-layer of the SPP to the packet, we already know what the value of the a field in the output packet will be. We can therefore immediately continue with the a layer of the second SPP, without having to consider the other layers of the first SPP. This is in contrast to a naive algorithm, which would have to consider the entire first SPP before being able to apply the second SPP. This

Primitive tests and mutation:

$(f = v) \triangleq \mathsf{SPP}(f, \{v \mapsto \{v \mapsto \top\}\}, \emptyset, \bot)$

$(f \neq v) \triangleq \mathsf{SPP}(f, \{v \mapsto \{v \mapsto \bot\}\}, \emptyset, \top)$

$(f \leftarrow v) \triangleq \mathsf{SPP}(f, \emptyset, \{v \mapsto \top\}, \bot)$

Operations, base cases:

Identical to those in Figure 3.2.

Operations, inductive cases:

$\mathsf{SPP}(f, b_p, m_p, d_p) \mathbin{\hat{+}} \mathsf{SPP}(f, b_q, m_q, d_q) \triangleq \mathsf{spp}(f, b', m_p \mathbin{\vec{+}} m_q, d_p \mathbin{\hat{+}} d_q)$

where:

$b' = \{v \mapsto p(v) \mathbin{\vec{+}} q(v) \mid v \in \mathsf{keys}(b_p \cup b_q \cup m_p \cup m_q)\}$

$\mathsf{SPP}(f, b_p, m_p, d_p) \mathbin{\hat{\cdot}} \mathsf{SPP}(f, b_q, m_q, d_q) \triangleq \mathsf{spp}(f, b', m_A \mathbin{\vec{+}} m_B, d_p \mathbin{\hat{\cdot}} d_q)$

where:

$b' = \{v \mapsto \vec{\sum}_{v' \mapsto p' \in p(v)} \{w' \mapsto p' \mathbin{\hat{\cdot}} q' \mid w' \mapsto q' \in q(v')\} \mid v \in \mathsf{keys}(b_p \cup b_q \cup m_p \cup m_q \cup m_A)\}$

$m_A = \vec{\sum}_{v' \mapsto p' \in m_p} \{w' \mapsto p' \mathbin{\hat{\cdot}} q' \mid w' \mapsto q' \in q(v')\}, \quad m_B = \{w' \mapsto d_p \mathbin{\hat{\cdot}} q' \mid w' \mapsto q' \in m_q\}$

$m_1 \mathbin{\vec{+}} m_2 \triangleq \{v \mapsto m_1(v; \bot) \mathbin{\hat{+}} m_2(v; \bot) \mid v \in \mathsf{keys}(m_1 \cup m_2)\}$,

$\vec{\Sigma} \triangleq n$-ary sum w.r.t. $\vec{+}$

$p(v) \triangleq b_p(v;$ if $v \in m_p \vee d_p = \bot$ then m_p else $m_p \cup \{v \mapsto d_p\})$ (similarly for $q(v)$)

Expansion:

$p \equiv \mathsf{SPP}(f, \emptyset, \emptyset, p)$ if $p \in \{\top, \bot, \mathsf{SPP}(f', b, m, d)\}$ where $f \sqsubset f'$

Repetition:

$p^\star \triangleq \hat{\Sigma}_i\, p^i = \top \mathbin{\hat{+}} p \mathbin{\hat{+}} p \mathbin{\hat{\cdot}} p \mathbin{\hat{+}} p \mathbin{\hat{\cdot}} p \mathbin{\hat{\cdot}} p \mathbin{\hat{+}} \ldots$

(sums in SPP converge in a finite number of steps)

Figure 3.4: Definition of the SPP operations. The inductive case is identical for all operations (indicated by $\hat{+}$), except $\hat{\cdot}$, which is given separately. Expansion inserts a trivial SPP node to reduce the remaining cases to the inductive case. The notation $b(v; d)$ means the child $b(v)$ if $v \in \mathsf{keys}(b)$, or by default d otherwise.

property makes it possible to compute the sequential composition of two SPPs efficiently in practice, and is a key contributor to the performance and scalability of our system.

Operations SPPs are closed not just under sequential composition, but under all of the NetKAT operations. These operations are largely mechanical, but more complex than for SPs, as we need to take assignments into account while respecting the conditions that ensure uniqueness. In particular, sequential composition is no longer equivalent to intersection, as the assignments in the first SPP clearly affect

the tests in the second SPP. Furthermore, the star operator is no longer trivial, as the assignments in the SPP can affect the tests in the SPP itself, and a fixed point needs to be taken. We have the following operations on SPPs:

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

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

$$\mathsf{push} : \ \mathsf{SP} \times \mathsf{SPP} \to \mathsf{SP} \qquad\qquad \mathsf{pull} : \ \mathsf{SPP} \times \mathsf{SP} \to \mathsf{SP}$$

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

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

Specifications Each of these operations is uniquely specified by two correctness conditions:

1. They semantically match their counterpart (e.g. $[\![p \mathbin{\hat{+}} q]\!] \equiv [\![p + q]\!]$ for all $p, q \in \mathsf{SPP}$)
2. They maintain the reduced & ordered invariant (e.g., $p \mathbin{\hat{+}} q$ is reduced & ordered if p, q are)

For push and pull, the correctness condition is as follows:

$$\beta \in [\![\mathsf{push}\ p\ s]\!] \iff \exists \alpha \in [\![p]\!].\ \beta \in [\![s]\!](\alpha), \quad \text{and}$$
$$\beta \in [\![\mathsf{pull}\ s\ p]\!] \iff \exists \alpha \in [\![p]\!].\ \alpha \in [\![s]\!](\beta)$$

For further details, see Section 3.10.

3.3 Symbolic NetKAT Automata via Brzozowski Derivatives

With the symbolic packet and symbolic transition representations in place, we can now define symbolic NetKAT automata. The construction of NetKAT automata shares some similarities with the construction of automata for regular expressions. In prior work [55, 101], NetKAT automata were constructed via Antimirov derivatives [9], which can be extended from regular expressions to NetKAT. The Antimirov derivative constructs a non-deterministic automaton, and as such, is well suited for handling NetKAT's union operator by inserting transitions for the two sub-terms. However, the Antimirov derivative is not well suited for our extended set of logical operators, as we cannot simply insert (non-deterministic) transitions for the operands of intersection, difference, and symmetric difference. Instead, we

extend the Brzozowski derivative [28] to NetKAT, which constructs a deterministic automaton directly, and is better suited for the logical operators.

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

3.3.1 Symbolic NetKAT Automata

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

An automaton for a NetKAT program is similar, but instead of modeling a set of strings, it models the traces that are produced for a given input packet. When a packet travels through the automaton, every state that it traverses acts as a dup operation, appending a copy of the packet to the packet's trace. Therefore, the transition between two states is labeled with a dup-free NetKAT program, represented symbolically as a SPP. That is, each edge in the automaton is not labeled with a single letter, as it would be in a standard DFA or NFA, but with a potentially large SPP. This is necessary because an edge in the automaton intuitively represents "what happens between two dups". This is precisely a dup-free NetKAT program, which can itself have rich behavior, represented canonically as an SPP. This is needed to support fine-grained control over equivalence and inclusion as discussed in Section 2.1—e.g., via conditionally executed dups or assignments be-

tween dups.

Secondly, instead of having a boolean at each state to determine acceptance, NetKAT automata have an additional SPP at each state that determines the set of packets that are produced as output of the automaton when a packet reaches that state. Therefore, a symbolic NetKAT automaton is a tuple $\mathcal{A} = \langle Q, q_o, \delta, \epsilon \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state,

- $\delta : Q \times Q \to \mathsf{SPP}$ is the transition function, and
- $\epsilon : Q \to \mathsf{SPP}$ is the output function.

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

3.3.2 Constructing Automata via Brzozowski Derivatives

We now describe how to construct a symbolic NetKAT automaton for a NetKAT program via the Brzozowski derivative. We take the set of states to be the set

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

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

where $p_i \in \mathsf{SPP}$, and q_i are NetKAT expressions, and the p_i are disjoint ($p_i \mathbin{\hat{\cap}} p_j = \bot$ for $i \neq j$).

Operations Dup is an STS, by taking $r = \top \cdot \mathrm{dup} \cdot \top$. We extend the logical operators to STSs:

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

These operations need to be defined such that the resulting STS is deterministic. For $r_1 \cap r_2$:

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

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

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

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

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

NetKAT expression on the right:

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

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

All of the STS operations are defined in terms of SPP operations. Therefore, an efficient SPP implementation is crucial both for operations on symbolic packets and for operations on STSs.

Specifications Each of these operations is uniquely specified by two correctness conditions:

1. They semantically match their counterpart (e.g. $[\![p \mathbin{\hat{+}} q]\!] \equiv [\![p + q]\!]$ for all $p, q \in \mathsf{STS}$)
2. They maintain the invariant that the transitions are pairwise disjoint.

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

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

$$\delta(p+q) \triangleq \delta(p) \mathbin{\tilde{+}} \delta(q) \qquad \delta(\text{dup}) \triangleq \mathsf{dup}$$

$$\delta(p \cap q) \triangleq \delta(p) \mathbin{\tilde{\cap}} \delta(q) \qquad \delta(f = v) \triangleq \bot$$

$$\delta(p \oplus q) \triangleq \delta(p) \mathbin{\tilde{\oplus}} \delta(q) \qquad \delta(f \neq v) \triangleq \bot$$

$$\delta(p - q) \triangleq \delta(p) \mathbin{\tilde{-}} \delta(q) \qquad \delta(f \leftarrow v) \triangleq \bot$$

$$\delta(p \cdot q) \triangleq \delta(p) \mathbin{\tilde{\cdot}} q \mathbin{\tilde{+}} \epsilon(p) \mathbin{\tilde{\cdot}} \delta(q) \qquad \delta(\top) \triangleq \bot$$

$$\delta(p^\star) \triangleq \epsilon(p)^\star \mathbin{\tilde{\cdot}} \delta(p) \mathbin{\tilde{\cdot}} p^\star \qquad \delta(\bot) \triangleq \bot$$

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

States $Q \triangleq \mathsf{Exp}$.

Initial state $q_0 \triangleq p$.

Transitions take $\delta(q, q')$ to be the SPP of q' in the Brzozowski derivative of $\delta(q)$.

Output $\epsilon : Q \to \mathsf{SPP}$, defined above.

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

The NetKAT automaton for an example program is shown in Figure 3.5. The edges are labeled with the SPPs that represent the transitions. The output SPP of every state is $\top$, i.e., the automaton accepts all packets that reach a state. Note that the SPPs in the figure feature several diamonds without outgoing edges. These diamonds produce no output packet, i.e., the packet is dropped.

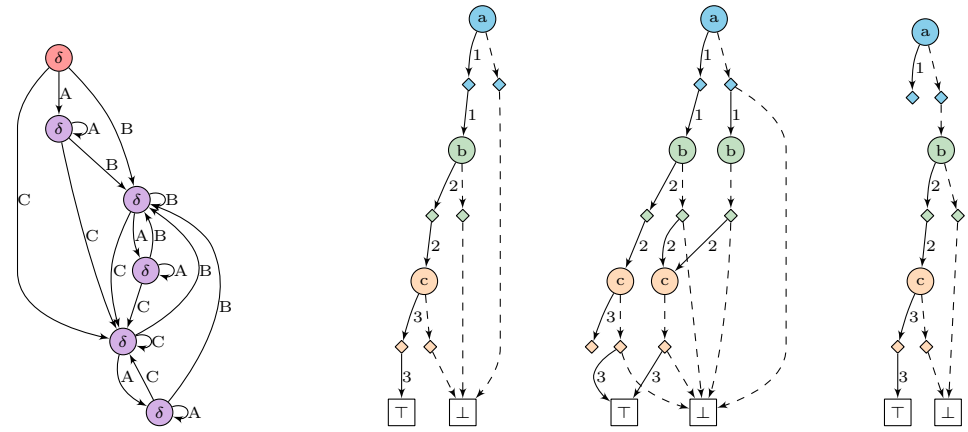


Figure 3.5: Symbolic NetKAT automaton for $p \triangleq ((a{\leftarrow}1 \cdot b{\leftarrow}2 \cdot c{\leftarrow}3 \cdot \mathrm{dup})^\star + (b{=}2 \cdot c{=}3 \cdot \mathrm{dup})^\star)^\star$

For more details, see Section 3.10.

3.4 Bisimilarity and Counter-Example Generation

After converting the NetKAT programs to automata, we can check equivalence of two NetKAT programs by checking whether their automata are bisimilar. Bisimilarity is a well-known notion of equivalence for automata. For NetKAT automata, two states are bisimilar for a given packet α if (1) the states produce the same output packets, and (2) for each possible modified packet α', the two states transition to states that are bisimilar for α'.

Historically, methods for computing bisimulations of automata [23, 42] have been based on either on Moore's algorithm for minimization [94] or Hopcroft and Karp's algorithm [66]. Indeed, the naive approach we saw in Figure 2.1 follows the basic structure of Hopcroft-Karp, in the sense that it relates the start states and proceeds to follow transitions forward in the two automata.

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

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

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

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

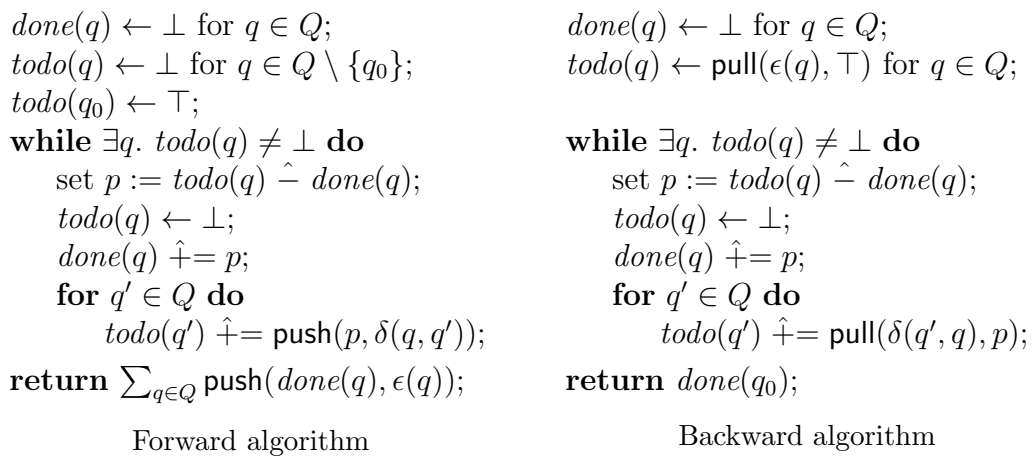


Figure 3.6: Forward and backward algorithms for NetKAT automata

separately.

3.4.1 Forward Algorithm

To check whether a NetKAT automaton drops all input packets, we use an algorithm that works in the forward direction. The forward algorithm computes *all* output packets that the automaton can produce, across all possible input packets. More formally: given a NetKAT automaton for a NetKAT program p, the forward algorithm computes the set of final packets occurring in the traces $[\![p]\!](\alpha)$ across all input packets α. The forward algorithm therefore starts with the complete symbolic packet $\top$ at the input state, and repeatedly applies all outgoing transitions δ to it. In this way, the algorithm iteratively accumulates a symbolic packet at every state, which represents the set of packets that can reach that state from the start state. Once we know the set of packets that can reach a state, we can determine the set of output packets by applying the output function ϵ to the symbolic packet of each state, and taking the union of the results.

The forward algorithm is shown in Figure 3.6. To determine the set of pack-

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

3.4.2 Backward Algorithm

The forward algorithm gives us the set of output packets that a NetKAT automaton can produce, but we are also interested in which input packets cause this output to be produced. For instance, for a given check $A \equiv B$, we want to know *which* input packets cause A and B to produce different sets of output traces. More generally, we want to know which input packets cause a NetKAT automaton to produce a nonempty set of output packets or, formally, given an automaton for a NetKAT program p, what is the set of initial packets α for which $[\![p]\!](\alpha)$ is nonempty.

To answer this question, we developed a *backward algorithm*, shown in Figure 3.6. The backward algorithm computes *all* input packets that can cause the automaton to produce a nonempty set of output packets. The backward algorithm therefore starts with the complete symbolic packet $\top$ at every state, and pulls it backwards through all observation functions ϵ. The algorithm then iteratively accumulates a symbolic packet at every state, which represents the set of packets that will cause the automaton to produce a nonempty set of output packets, when starting from that state. The accumulation is done by pulling the symbolic packet backwards through all transitions until a fixpoint is reached. Once the fixpoint is reached, we return the symbolic packet at the start state, which represents the set of input packets that will cause the automaton to produce a nonempty set of

Statements	**Expressions**
check $e_1 \equiv e_2$	forward e
check $e_1 \not\equiv e_2$	backward e
print e	$e_1 \cap e_2$
$x = e$	$e_1 \oplus e_2$
for $i \in n_1..n_2$ do c	$e_1 - e_2$
	$\hat{\exists}\, f\ e$
	$\hat{\forall}\, f\ e$
	(+ NetKAT, see table 2.1)

Figure 3.7: NKPL syntax.

output packets when starting from the start state.

3.5 Implementation

We have implemented the algorithms in a new system, KATch, comprising 2500 lines of Scala. In this section we explain the system's interface. The implementation provides a surface syntax for expressing queries, which extends the core NetKAT syntax from Table 2.1. The extended syntax is shown in Figure 3.7. The language has statements and expressions, which we describe below.

Statements The check $e_1 \equiv e_2$ command runs the bisimulation algorithm. The system reports success if the expressions are equivalent (when $\equiv$ is used) or inequivalent (when $\not\equiv$) is used, or reports failure otherwise. The other statements behave as expected: The print statement invokes the pretty printer, $x = e$ let-binds names to expressions, and for runs a statement in a loop.

Expressions The forward e expression computes, in forward-flowing mode, the set of output packets resulting from running the symbolic packet $\top$ through the

given expression e (see Figure 3.6). Conversely $\mathsf{backward}\ e$ computes the set of input packets which generate some output packet when run on the given expression (Figure 3.6). These are generally used in conjunction with the $\oplus, -, \cap$ operators to express the desired query. The user may combine these expressions with the $\hat{\exists}$ and $\hat{\forall}$ operators to reason about symbolic packets, and the print operator to pretty print the symbolic packets, or the check operator to assert (in)equivalence of two expressions.

We note that the generality of the language allows us to express some queries in different, equivalent ways. For example, the two checks: $e_1 \equiv e_2$ and $e_1 \oplus e_2 \equiv \bot$ are equivalent. However, the expression on the right lends itself to inspection of counterexample input packets: $\mathsf{print}\ (\mathsf{backward}\ e_1 \oplus e_2)$. This statement pretty prints a symbolic packet having different behavior on e_1 and e_2 (i.e., packets that result in some valid history in one expression and not the other). In other words, it prints the set of all counter-example input packets for $e_1 \equiv e_2$.

Topologies and routing tables While NetKAT can be used to specify routing policies declaratively, it is also possible to import topologies and routing tables into NetKAT. For example, a simple way to define routing tables and topologies in NetKAT (as introduced in Section 2.1.5) is as follows:

$$
\begin{aligned}
\mathsf{R} \triangleq\ & \mathsf{sw}{=}5 \cdot (\mathsf{dst}{=}6 \cdot \mathsf{pt}{\leftarrow}3 + \mathsf{dst}{=}8 \cdot \mathsf{pt}{\leftarrow}4) \\
& \qquad + \ \cdots \ + \\
& \mathsf{sw}{=}7 \cdot (\mathsf{dst}{=}8 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{dst}{=}9 \cdot \mathsf{pt}{\leftarrow}1) \\
\mathsf{T} \triangleq\ & \mathsf{sw}{=}5 \cdot (\mathsf{pt} = 3 \cdot \mathsf{sw}{\leftarrow}6 + \mathsf{pt} = 4 \cdot \mathsf{sw}{\leftarrow}8) \\
& \qquad + \ \cdots \ + \\
& \mathsf{sw}{=}7 \cdot (\mathsf{pt} = 2 \cdot \mathsf{sw}{\leftarrow}8 + \mathsf{pt} = 1 \cdot \mathsf{sw}{\leftarrow}9)
\end{aligned}
$$

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

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

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

Example 5 (All-Pairs Reachability Queries)**.** *A naive way to check reachability of all pairs of hosts in a network is to run the following command for each pair of end hosts n_i, n_j:*

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

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

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

This query does work if sw *is the only field of our packets, because the left hand side contains the packets that can be reached from i via the network, and the right hand side contains packets where the* sw *field is any value in $1..n$. Unfortunately, this does not quite work if the network also operates on other packet fields (say, fields f_1 and f_2), as the packets on the left hand side will have those fields, whereas the packets on the right hand side will only have a* sw *field. The $\hat{\exists}$ and $\hat{\forall}$ operators allow us to manipulate symbolic packets and express all pairs reachability in n queries:*

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

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

$$p \in \hat{\exists}\, f\ e \overset{\triangle}{\iff} \exists v \in V : p[f \leftarrow v] \in [\![e]\!] \quad \textit{and} \quad p \in \hat{\forall}\, f\ e \overset{\triangle}{\iff} \forall v \in V : p[f \leftarrow v] \in [\![e]\!]$$

The implementation does not iterate over V (indeed, V can be unknown). Rather, $\hat{\exists}$ and $\hat{\forall}$ are implemented directly as operations on symbolic packets. In fact, the implementation does not need to fix the sets of fields or values up-front at all. Instead, it operates on a conceptually infinite set of fields and values. This works because SPPs' default cases handle all remaining fields and all values.

Correctness and testing methodology We validated our implementation with the following property-based fuzz testing methodology:

1. We implemented the semantics $[\![p]\!](\alpha)$ in Scala, as described in Table 2.1.
2. We repeatedly pick two SPs/SPPs p and q and a packet α. We enumerated small SPPs up to a bound and generated larger SPPs randomly. We generated packets exhaustively.
3. We check the soundness (e.g., $[\![p \mathbin{\hat{+}} q]\!](\alpha) = [\![p]\!](\alpha) \cup [\![q]\!](\alpha)$) and canonicity (e.g., $[\![p]\!](\alpha) = [\![q]\!](\alpha)$ for all α if and only if $p = q$) of the operations, as well as additional algebraic laws derived from the NetKAT axioms (e.g., $p \mathbin{\hat{+}} q = q \mathbin{\hat{+}} p$).

In addition to property-based testing for SPs and SPPs, we also generated hundreds of thousands of pairs of random NetKAT expressions and checked that KATch's bisimilarity check matches the output of Frenetic. This revealed a subtle bug in Frenetic, which we reported and is now fixed.

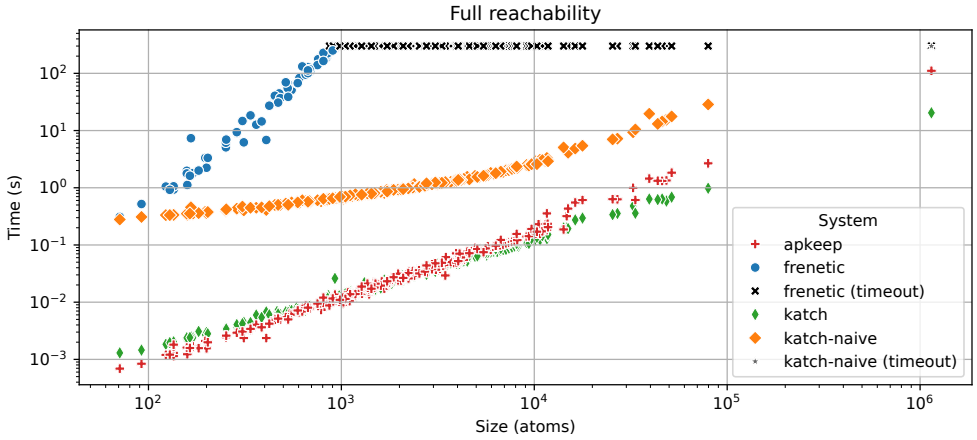


Figure 3.8: Full reachability queries on Topology Zoo. KATch and APKEEP results are averages of 100 runs, preceded by 10 runs of JIT warmup. KATch-naive is without JIT warmup and uses $O(n^2)$ 1-to-1 queries.

3.6 Evaluation

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

1. How does KATch perform compared to the state of the art NetKAT verifier, FRENETIC?
2. How does KATch perform compared to the state of the art specialized network verification tool, APKEEP?
3. How well does KATch scale with the size of topology?
4. When does KATch perform asymptotically better than prior work?

Name (Topology Zoo)	Size (atoms)	1-to-1 reachability KATch	Frenetic	Slicing KATch	Frenetic	Full reachability KATch	APKeep
Layer42	135	0.00	0.04	0.00	0.07	0.00	0.00
Compuserv	539	0.00	0.35	0.01	0.81	0.01	0.01
Airtel	785	0.01	0.82	0.02	1.98	0.01	0.01
Belnet	1388	0.01	3.19	0.03	7.99	0.02	0.01
Shentel	1865	0.01	3.93	0.03	9.76	0.02	0.03
Arpa	1964	0.01	4.23	0.04	10.42	0.03	0.03
Sanet	4100	0.03	23.42	0.07	62.21	0.05	0.07
Uunet	5456	0.04	80.85	0.11	203.80	0.07	0.07
Missouri	9680	0.06	166.87	0.21	441.72	0.13	0.19
Telcove	10720	0.07	441.47	0.21	1121.30	0.12	0.16
Deltacom	27092	0.18	2087.34	0.48	5098.57	0.36	0.63
Cogentco	79682	0.53	18910.82	1.38	54247.73	0.98	2.67
Kdl	1144691	9.87	out of memory	23.67	out of memory	19.44	110.92

Table 3.1: Running time (in seconds) of KATch, Frenetic, and APKeep on Topology Zoo queries. KATch and APKeep results are averages of 100 runs, preceded by 10 runs of JIT warmup.

3.6.1 Topology Zoo

To begin to answer the first three questions, we conducted our experiments using *The Internet Topology Zoo* [72] dataset, a publicly available set of 261 network topologies, ranging in size from just 4 nodes (the original ARPANet) to 754 nodes (KDL, the Kentucky Data Link ISP topology). For each topology, we generated a destination-based routing policy using an all-pairs shortest path scheme that connects every pair of routers to each other.

To demonstrate KATch's scalability, we first ran full (i.e., $O(n^2)$) reachability queries for every topology in the zoo using KATch, Frenetic[2], and APKeep [120]. The results are shown in Figure 3.8 in a log-log plot, so straight lines correspond to polynomials with exponents related to their slope.

The figure shows two different configurations of KATch: one that verifies full reachability using a quadratic number of point-to-point queries and does not use

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

JIT warmup ("katch-naive"), and one that verifies full reachability using a linear number of queries and does use JIT warmup ("katch").

Because of the size of the dataset, we set a timeout of 5 minutes per topology. Under these conditions, FRENETIC was unable to complete for all but the smallest topologies. KATch-naive handles most of the topologies in well under a second, and all but the largest in under 2 minutes. KATch-naive exceeds the timeout on Kentucky Data Link—using a quadratic number of queries to check full reachability produces over 500k individual point-to-point queries in a network with 754 nodes! (However, KATch-naive is able to finish it in just under 20 minutes.)

To avoid combinatorial blowup in the verification query itself, we also used KATch's high-level verification interface, to check full reachability using a *linear* number of queries, as discussed in Theorem 5. FRENETIC does not have an analogous linear mode. APKEEP does have an analogous mode, as it has specialized support for full reachability queries. APKEEP is faster than KATch for the smaller topologies, but slower for the larger ones. Indeed, the slope of the APKEEP line is steeper than the KATch line, indicating that KATch scales slightly better with the size of the network on these queries. It is important to note that APKEEP has support for prefix-matching, ACLs, NAT, incrementality, and other features for which KATch does not have specialized support and which are not tested in this comparison. One should therefore not draw strong conclusions from this comparison; we include it as it is encouraging that reachability via NetKAT equivalence can be competitive with a state-of-the-art specialized tool.

We also randomly sampled a subset of topologies from the Topology Zoo of varying size and generated point-to-point (1-to-1) reachability and slicing queries to be checked against the routing configurations with no timeout. For each query,

we ran KATch and also generated an equivalent query in the syntax of FRENETIC, and ran FRENETIC's bisimulation verifier on those queries. We present a full table of results of these experiments in Table 3.1. The table shows that KATch's relative speedup over FRENETIC is considerable, and increases problem size. This is encouraging, because it shows that KATch is more scalable. FRENETIC was unable to complete KDL within a 200GB memory limit (for comparison, KATch is able to complete the same query with well under 1GB). We also include a selection of full-reachability results from Figure 3.8 in the table. The reader may wonder why full reachability takes only twice as long as 1-to-1 reachability. This is because a significant fraction of the time is spent constructing automata (which are similar for both queries), and much of the work in the full reachability queries is shared due to memoization of SPP operations.

3.6.2 Combinatorial Examples

Finally, we ran experiments to test the hypothesis that SPPs have an asymptotic advantage for certain types of queries. FRENETIC uses forwarding decision diagrams (FDDs), which is a different (and non-canonical) representation for automaton transitions. A key advantage of SPPs over FDDs is that SPPs keep the updates to each field next to the tests of the same field. On the other hand, FDDs keep all updates at the leaves, which can result in combinatorial explosion during sequencing. To test this, we generated the following NetKAT programs:

Inc: Treating the input packet's n boolean fields as a binary number, increment it by one.

Flip: Sequentially flip the value of each of the n boolean fields.

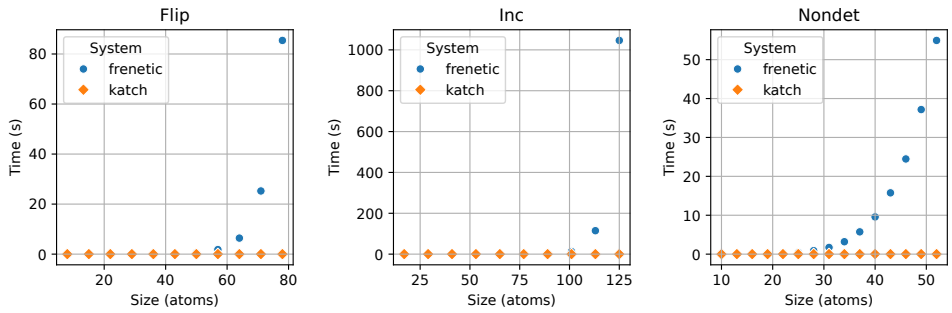


Figure 3.9: Results of running KATch and FRENETIC on combinatorial benchmarks

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

For Inc, we tested that repeatedly incrementing (using the $\star$ operator) can turn packet $00\cdots0$ into $11\cdots1$. For Flip, we tested that flipping all bits twice returns the original packet. For Nondet, we tested that setting the fields nondeterministically twice is the same as doing it once. The results of this experiment are shown in Figure 3.9. Because the available fields are hard-coded in FRENETIC, we only ran the Inc and Flip experiments up to $n = 10$. We ran the non-determinism test up to $n = 15$. KATch finishes all three queries up to $n = 100$ in under a one minute, demonstrating its asymptotic advantage for these queries, while Frenetic shows combinatorial blowup on larger packets.

Source of speedup The speedup of KATch over FRENETIC comes from several sources: (1) the use of SPPs instead of FDDs, which can be exponentially more compact and support more efficient sequential composition, (2) the symbolic bisimulation algorithm operating on SPs, which can handle exponentially large sets of packets at once, and (3) a quadratic-doubling implementation of star, which can handle exponentially long traces quickly. These differences show up most strongly in the combinatorially adversarial examples above, but the speedup is also considerable for the Topology Zoo benchmarks, which are based on real-world topologies

and not designed to be adversarial.

Field order The field order of SPs and SPPs can affect performance, just as for BDDs. In practice, field orders that keep related fields close together are beneficial. Our implementation allows the user to control the field order, but we did not use this for our benchmarks. By default, the fields are in the order in which they first occur in the input file, which turned out to work well enough.

3.7 Related Work

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

NetKAT NetKAT was originally proposed as a semantic foundation for SDN data planes [5]. Indeed, being based on KAT [73], the language provides a sound and complete algebraic reasoning system. Later work on NetKAT developed an automata-theoretic (or coalgebraic) account of the language [55], including a decision procedure based on Brzozowski derivatives and bisimulation, implemented in Frenetic. However, the performance of this approach turns out to be poor, as shown in our experiments, due to the use of ad hoc data structures ("bases") to encode packets and automata. NetKAT's compiler uses a variant of BDDs, called Forwarding Decision Diagrams (FDDs), as well as an algorithm for converting programs to automata using Antimirov derivatives [101], improving on the earlier representation of transitions as sets of bases [55]. The SPPs proposed in this chap-

ter improve on FDDs by ensuring uniqueness and supporting efficient sequential composition in the common case. In addition, the deterministic automata used in KATch support additional "negative" operators that are useful for verification. Other papers based on NetKAT have explored use of the language in other settings such as distributed control planes [19], probabilistic networks [54, 104, 103], and Kleene algebra over composable theories with unbounded state [62]. It would be interesting to extend the techniques developed in this chapter to these richer settings. Another interesting direction for future work is to build a symbolic verifier for the guarded fragment of NetKAT [102].

Network Verification Early work by Xie et al. [116] proposed a unifying mathematical model for Internet routers and developed algorithms for analyzing network-wide reachability. Although the paper did not discuss an implementation, its elegant formal model has been extremely influential in the community and has served as the foundation for many follow-on efforts, including this work. The emergence of software-defined networking (SDN) led to a surge of interest in static data plane verification, including systems such as Header Space Analysis (HSA) [69], Anteater [87], VeriFlow [71], Atomic Predicates (AP) [118], APKeep [120]. These systems all follow a common approach: they build a model of the network-wide forwarding behavior and then check whether given properties hold. However, they vary in the data structures and algorithms used to represent and analyze the network model. For instance, Anteater relies on first-order logic and SAT solvers, while VeriFlow uses prefix trees and custom graph-based algorithms. HSA, AP and APKeep are arguably the most related to our work as they use symbolic representations and BDDs respectively.

The primary difference between KATch and these systems is that the latter

implement specialized algorithms for network-wide analysis queries, while KATch (and Frenetic) solve the more general NetKAT equivalence problem, into which network-wide analysis queries can be encoded. This generic approach is based on principled automata-theoretic foundations, but one might expect it to be less efficient than specialized algorithms. Frenetic was found to be comparable to or faster than HSA [55], so by transitivity we expect KATch to be faster than HSA. However, the current state of the art is APKeep, which is significantly faster than Frenetic and HSA. We include a comparison with APKeep in Section 3.6 on reachability queries. The results show that KATch is competitive with APKeep on these benchmarks, despite solving general NetKAT queries. As discussed, one should not draw strong conclusions from this comparison as APKeep includes additional features; nevertheless, it offers a promising preliminary indication of NetKAT's scalability and performance.

Another line of work has explored how to lift verification from the data plane to the control plane. Batfish [52, 25] proposed using symbolic simulation to analyze distributed control planes—i.e., generating all possible data planes that might be produced starting from a given control-plane configuration. Like KATch, Batfish uses a BDD-based representation for data plane analysis. MineSweeper [16] improves on Batfish using an SMT encoding of the converged states of the control plane that avoids having to explicitly simulate the underlying routing protocols. Recent work has focused on using techniques like modular reasoning [106, 107] abstract interpretation [18], and a form of symmetry reduction [17] to further improve the scalability of control-plane verification.

Automata-Theoretic Approach and Symbolic Automata Our work on KATch builds on the large body of work on the automata-theoretic approach to

verification and symbolic automata. The automata-theoretic approach was pioneered in the 1980s, with applications of temporal logics and model checking to hardware verification [111]. BDDs, originally proposed by Lee [81], were further developed by Bryant [26], and used by McMillan for symbolic model checking [29]. BDD-based techniques were a success story of symbolic model checking of hardware in the 1990s—Chaki and Gurfinkel give an overview [32].

An influential line of work by D'Antoni and Veanus developed techniques for representing and transforming finite automata where the transitions are not labeled with individual characters but with elements of a so-called effective Boolean algebra [37, 38]. Shifting from a concrete to a symbolic representation requires generalizing classical algorithms, such as minimization and equivalence, but facilitates building automata that work over huge alphabets, such as Unicode.

Pous [100] developed symbolic techniques for checking the equivalence of automata where transitions are specified using BDDs. The methods developed in this chapter take Pous's work as a starting point but develop a non-trivial extension for NetKAT. In particular, SPPs provide a compact representation for "carry-on" packets, which is a critical and unique aspect of NetKAT's semantics. Bonchi and Pous [23] explored the use of up-to techniques for checking equivalence of automata. In principle, one can view the invariants enforced by KATch's term representations as a kind of up-to technique, but a full investigation of this idea requires more work.

Our backward algorithm for computing bisimulations can be seen as a variant of Moore's classic algorithm for computing the greatest bisimulation of classic automata. Doenges et al. [42] proposed an analogous approach for checking equivalence of automata that model the behavior of P4 packet parsers [24]. However,

Leapfrog's model is simpler than NetKAT's and is based on classic finite automata. It achieves scalability due to a novel up-to technique that "leaps" over internal buffering transitions rather than symbolic representations.

Acknowledgements

We wish to thank the PLDI reviewers for many helpful comments and suggestions, which improved the paper significantly. We are also grateful to Aaron Gember-Jacobson and Peng Zhang for making AP Keep available as open-source software, to Caleb Koch for early discussions on compact data structures for data plane verification, to the Cornell PLDG for helpful comments on early drafts, and to the EPFL DCSL group for providing a welcoming and supportive environment. This work was supported in part by ONR grant N68335-22-C-0411, DARPA grant W912CG-23-C-0032, ERC grant 101002697, and a Royal Society Wolfson fellowship.

Data Availability

The current version of the implementation is available at `https://github.com/cornell-netlab/KATch`. A snapshot of the code as of artifact evaluation (with a dependencies-included Docker image) is available on Zenodo [3].

[3] `https://doi.org/10.5281/zenodo.10961123`

3.8 Omitted Proofs

In this final section of the chapter, we provide a formal development of SPPs omitted from the description above.

3.8.1 Set of Fields

We assume packet header fields come from a set F, and their values are from a set V. Both F and V must have a total order, $\sqsubset$. The implementation makes frequent use of finite maps, especially with V as keys. We use the symbol ($\Mapsto$) to construct finite map types. For example, a concrete packet α could naively be represented using the type $F \Mapsto V$, which we take to mean that $\alpha = \{f_0 \mapsto v_0, \ldots, f_n \mapsto v_n\}$. All of our maps have only one binding for each key. We often collect the keys as a set with $\mathsf{keys}(\{f_0 \mapsto v_0, \ldots, f_n \mapsto v_n\}) \triangleq \{f_0, \ldots, f_n\}$.

3.8.2 Set of Values

We need one more assumption about the value space: it is larger than the values actually mentioned in any expressions. That we do not assume a fixed finite value space is a boon for compositionality: when KATch determines that two NetKAT expressions are equivalent, then they are equivalent under every value set containing the values in the two expressions. On the other hand, if V is small, say $V = \{0, 1\}$, then the test $f = 0$ is semantically equivalent to $f \neq 1$, but KATch will not equate these two (unless both are preceded by, e.g. $(f = 0 + f = 1)$).

If one wants to reason about a small V using KATch, one can do any *one* of

the following to avoid false negatives:

1. Not use negative tests.
2. Only check equivalences which do not refer to every value.
3. Prefix both sides of equations with tests representing the full set of values, e.g. for $V = \{0, 1\}$, we run the query $(f{=}0 + f{=}1) \cdot e_1 \equiv (f{=}0 + f{=}1) \cdot e_2$. In this case, $f{=}0$ is equivalent to $f{\neq}1$ when appearing in e_1 or e_2.

3.9 Operations on Symbolic Packets

Recall the definition of SP from Section 3.2.1:

$$p \in \mathsf{SP} ::= \bot \mid \top \mid \underbrace{\mathsf{SP}(f,\ \{\ldots, v_i \mapsto q_i, \ldots\},\ q)}_{\equiv\ \sum_i f{=}v_i \cdot q_i\ +\ (\prod_i f{\neq}v_i) \cdot q}$$

We take the semantics of an SP to be the semantics of its associated NetKAT expression, as shown above. For a packet $\alpha \in \mathsf{Pk}$ and $p \in \mathsf{SP}$, note that either $[\![p]\!](\alpha) = \emptyset$ or $[\![p]\!](\alpha) = \{\alpha\}$. Accordingly, in the notation we treat $[\![p]\!]$ as a set of packets and write $\alpha \in [\![p]\!]$ for $\alpha \in [\![p]\!](\alpha)$.

We ensure that every SP ever constructed is canonical (i.e., it is Ordered and Reduced). This invariant is established by ensuring that:

- The SP constructor produces canonical-form SPs if its arguments are canonical-form.
- The other operations only construct SPs using the SP constructor.

Canonicalization The smart constructor for SPs has type

$$\mathsf{sp}\colon (F \times (V \rightharpoonup\!\!\!\!\!\!\in\ \mathsf{SP}) \times \mathsf{SP}) \to \mathsf{SP}$$

and is defined as follows:

$$\mathsf{sp}(f, b, d) \triangleq \text{ if } b' = \emptyset \text{ then } d \text{ else } \mathsf{SP}(f, b', d)$$
$$\text{where } b' \triangleq \{v \mapsto p \mid v \mapsto p \in b, p \neq d\}$$

Operations We have the following 0-ary operations:

$$p \in \mathsf{SP} ::= \top \mid \bot \mid (f = v_0) \mid (f \neq v_0)$$

The $\top$ and $\bot$ are translated directly to the corresponding SP objects. The $f = v_0$ and $f \neq v_0$ are translated as follows:

$$(f = v_0) \triangleq \mathsf{SP}(f, \{v_0 \mapsto \top\}, \bot)$$
$$(f \neq v_0) \triangleq \mathsf{SP}(f, \{v_0 \mapsto \bot\}, \top)$$

Other SPs are built from applying the following operations:

$$\hat{+}, \hat{:}, \hat{\cap}, \hat{\oplus}, \hat{-} \ : \ \mathsf{SP} \times \mathsf{SP} \to \mathsf{SP} \qquad \hat{\neg}, \hat{\star} \ : \ \mathsf{SP} \to \mathsf{SP}$$
$$\hat{\exists}, \hat{\forall} \ : \ F \to \mathsf{SP} \to \mathsf{SP}$$

We define the base cases of the binary operations on SPs using the classic truth table definitions:

$$\top \mathbin{\hat{+}} \top \triangleq \top \quad \top \mathbin{\hat{:}} \top \triangleq \top \quad \top \mathbin{\hat{\cap}} \top \triangleq \top \quad \top \mathbin{\hat{\oplus}} \top \triangleq \bot \quad \top \mathbin{\hat{-}} \top \triangleq \bot$$
$$\top \mathbin{\hat{+}} \bot \triangleq \top \quad \top \mathbin{\hat{:}} \bot \triangleq \bot \quad \top \mathbin{\hat{\cap}} \bot \triangleq \bot \quad \top \mathbin{\hat{\oplus}} \bot \triangleq \top \quad \top \mathbin{\hat{-}} \bot \triangleq \top$$
$$\bot \mathbin{\hat{+}} \top \triangleq \top \quad \bot \mathbin{\hat{:}} \top \triangleq \bot \quad \bot \mathbin{\hat{\cap}} \top \triangleq \bot \quad \bot \mathbin{\hat{\oplus}} \top \triangleq \top \quad \bot \mathbin{\hat{-}} \top \triangleq \bot$$
$$\bot \mathbin{\hat{+}} \bot \triangleq \bot \quad \bot \mathbin{\hat{:}} \bot \triangleq \bot \quad \bot \mathbin{\hat{\cap}} \bot \triangleq \bot \quad \bot \mathbin{\hat{\oplus}} \bot \triangleq \bot \quad \bot \mathbin{\hat{-}} \bot \triangleq \bot$$

The inductive case when the field f matches both SPs can be expressed for all five operations generically, using $\hat{\pm}$ to stand for the operation:

$$\begin{aligned}&\mathsf{SP}(f, b_p, d_p) \mathbin{\hat{\pm}} \mathsf{SP}(f, b_q, d_q) \triangleq \mathsf{sp}(f, b', d_p \mathbin{\hat{\pm}} d_q)\\ &\quad \text{where } \; b' = \{v \mapsto b_p(v; d_p) \mathbin{\hat{\pm}} b_q(v; d_q) \mid v \in \mathsf{keys}(b_p \cup b_q)\}, \text{ and}\\ &\qquad b(v; d) \triangleq \text{if } v \in \mathsf{keys}(b) \text{ then } b[v] \text{ else } d\end{aligned}$$

When the fields do not match, we use a rule we call *expansion* to temporarily create an SP with a matching field.

$$p \equiv \mathsf{SP}(f, \emptyset, p) \qquad \text{if} \qquad p \in \{\top, \bot, \mathsf{SP}(f', b, d)\} \text{ where } f \sqsubset f'$$

Thus when we have different fields in the arguments to an operation, we simply apply expansion to the SP whose field is greater, reducing to the inductive case above.

The remaining operations are defined as follows:

$$p^{\hat{\star}} \triangleq \top$$

$$\hat{\forall}\ f_1\ p \triangleq \begin{cases} p & \text{if } p = \bot \vee p = \top \\ \hat{\bigcap}_{i \le n+1}\, p_i & \text{if } p = \mathsf{SP}(f_1, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1}) \\ \mathsf{sp}(f_2, & \\ \quad \{v_i \mapsto \hat{\forall}\ f_1\ p_i \mid i \le n\}, & \\ \quad \hat{\forall}\ f_1\ p_{n+1}) & \text{if } p = \mathsf{SP}(f_2, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1}), \\ & \quad f_1 \neq f_2 \end{cases}$$

$$\hat{\exists}\ f_1\ p \triangleq \begin{cases} p & \text{if } p = \bot \vee p = \top \\ \hat{\sum}_{i \leq n+1} p_i & \text{if } p = \mathsf{SP}(f_1, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1}) \\ \mathsf{sp}(f_2, & \\ \quad \{v_i \mapsto \hat{\exists}\ f_1\ p_i \mid i \leq n\}, & \\ \quad \hat{\exists}\ f_1\ p_{n+1}) & \text{if } p = \mathsf{SP}(f_2, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1}), \\ & \quad f_1 \neq f_2 \end{cases}$$

3.9.1 Correctness of SP operations

The goal of this section is to show that our representation of symbolic packets is canonical and correct, in the sense that two SPs are semantically equivalent if and only if they are syntactically equal.

First, we formally define a predicate on SPs that they are reduced and ordered (as described in Section 3.2.1). Then we will prove the that this property implies they are canonical in the above sense.

Definition 6 (Reduced and Ordered)**.** *An SP p is* reduced and ordered for a field

$f \in F$ if it satisfies:

$$\mathsf{RO}_f^{\mathsf{SP}} : \mathsf{SP} \to 2$$

$$\mathsf{RO}_f^{\mathsf{SP}}(p) \triangleq \begin{cases} \top & \text{if } p = \top \vee p = \bot \vee \\ & \quad p = \mathsf{SP}(f', \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1}),\ \text{for } n \geq 0 \wedge \\ & \qquad f \sqsubset f' \wedge \\ & \qquad \forall i \in \{0, \ldots, n+1\} : \mathsf{RO}_{f'}^{\mathsf{SP}}(p_i) \wedge \\ & \qquad \forall i \in \{0, \ldots, n\} : p_i \neq p_{n+1} \\ \bot & \text{otherwise} \end{cases}$$

Moreover an SP p is reduced and ordered *if it satisfies:*

$$\mathsf{RO}^{\mathsf{SP}} : \mathsf{SP} \to 2$$

$$\mathsf{RO}^{\mathsf{SP}}\ (p) \triangleq \exists f \in F : \mathsf{RO}_f^{\mathsf{SP}}\ (p)$$

Lemma 7. *For an SP p such that $\mathsf{RO}_f^{\mathsf{SP}}(p)$, then for any concrete packet α, we have $\alpha \in [\![p]\!]$ implies $\forall v \in V : \alpha[f \leftarrow v] \in [\![p]\!]$ and $\alpha \notin [\![p]\!]$ implies $\forall v \in V : \alpha[f \leftarrow v] \notin [\![p]\!]$.*

Proof. By straightforward induction on p. The lemma holds trivially for $p = \top$ and $p = \bot$. For the remaining cases, $\mathsf{RO}_f^{\mathsf{SP}}(p)$ means that p does not test f, so the same path must be taken on $\alpha[f \leftarrow v]$ for each $v \in V$. □

Now we show that our smart constructor only produces SPs which are $\mathsf{RO}^{\mathsf{SP}}$.

Lemma 8. *If $f \sqsubset f'$ and $\mathsf{RO}_{f'}^{\mathsf{SP}}(p_0), \ldots, \mathsf{RO}_{f'}^{\mathsf{SP}}(p_{n+1})$, then $\mathsf{RO}_f^{\mathsf{SP}}$ (sp $(f, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})$).*

Proof. After considering the assumption, we only need to argue that no child p_i in sp $(f, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})$ is equal to the default case. This is true by

assumption in the if-statement true branch of sp and true by construction in the false branch. □

Corollary 9. *If* $\mathsf{RO}^{\mathsf{SP}}(p_1)$ *and* $\mathsf{RO}^{\mathsf{SP}}(p_2)$*, then all of the following hold:*

1. $\mathsf{RO}^{\mathsf{SP}}(p_1 \hat{+} p_2)$,
2. $\mathsf{RO}^{\mathsf{SP}}(p_1 \hat{\cdot} p_2)$,
3. $\mathsf{RO}^{\mathsf{SP}}(p_1 \hat{\cap} p_2)$,
4. $\mathsf{RO}^{\mathsf{SP}}(p_1 \hat{-} p_2)$,
5. $\mathsf{RO}^{\mathsf{SP}}(p_1 \hat{\oplus} p_2)$,
6. $\mathsf{RO}^{\mathsf{SP}}(p_1^{\hat{\star}})$,
7. $\mathsf{RO}^{\mathsf{SP}}(\hat{\forall}\ f\ p_1)$, *and*
8. $\mathsf{RO}^{\mathsf{SP}}(\hat{\exists}\ f\ p_1)$.

Proof. By structural induction on p_1, p_2. If $p_1, p_2 \in \{\bot, \top\}$ the statement holds trivially by the base cases of all the operators. For the inductive case, we consider that the inductive cases use the smart constructor and apply Lemma 8 to conclude the result is $\mathsf{RO}^{\mathsf{SP}}$. We need to be slightly careful in the use of the expansion rule, which temporarily constructs a non-canonical SP (e.g., $\mathsf{SP}(f_1, \emptyset, p_2)$ for p_2, which is not $\mathsf{RO}^{\mathsf{SP}}$ because the test branch map is empty). This construction still works because we apply the inductive case after expansion, which calls the smart constructor—the premises of Lemma 8 are still met for this call by the assumptions for the original p_1, p_2. □

As a result of this corollary and the fact that $\top$, $\bot$, and the SPPs for $f = v$ and $f \neq v$ are all *RO*, all the SPs generated in KATch are *RO*.

Next, the smart constructor produces symbolic packets which are semantically equivalent to the plain datatype constructor:

Lemma 10. $[\![\mathsf{sp}(f, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})]\!] = [\![\mathsf{SP}(f, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})]\!]$

Proof. The case that we return d holds trivially (note $d = p_{n+1}$), since if $p_0 = \ldots = p_n = p_{n+1}$ then also $[\![p_0]\!] = \ldots = [\![p_{n+1}]\!] = [\![\mathsf{SP}(f, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})]\!]$.

In the other case we only delete branches. This is fine because branches we delete are syntactically the same as the default case. Any packets that would have gone to the deleted branch p_i are handled equivalently by the default (i.e., p_{n+1}) case because it is deleted precisely if $p_i = p_{n+1}$. □

Next we will show that two SPs in canonical form are syntactically equal if and only if they are semantically equal.

Theorem 11 (RO form is unique)**.** *For all* $p_1, p_2 \in \mathsf{SP}$*, if* $\mathsf{RO}^{\mathsf{SP}}(p_1)$ *and* $\mathsf{RO}^{\mathsf{SP}}(p_2)$*, then* $p_1 = p_2 \iff [\![p_1]\!] = [\![p_2]\!]$.

Proof. Suppose $\mathsf{RO}^{\mathsf{SP}}(p_1)$ and $\mathsf{RO}^{\mathsf{SP}}(p_2)$. We will show $p_1 = p_2 \iff [\![p_1]\!] = [\![p_2]\!]$.

That $p_1 = p_2 \Rightarrow [\![p_1]\!] = [\![p_2]\!]$ is immediate because $[\![\cdot]\!]$ is a function. For the other direction, we will show $p_1 \neq p_2 \Rightarrow [\![p_1]\!] \neq [\![p_2]\!]$. We show this property by induction on p_1, p_2 structurally. So let $p_1, p_2 \in \mathsf{SP}$ such that $\mathsf{RO}^{\mathsf{SP}}(p_1), \mathsf{RO}^{\mathsf{SP}}(p_2)$, and $p_1 \neq p_2$. Considering symmetry, there is only one base case, i.e. that $p_1 = \top$ and $p_2 = \bot$. Then $[\![p_1]\!] = \mathsf{Pk} \neq \emptyset = [\![p_2]\!]$.

There are four inductive cases after considering symmetry:

1. Let $p_1 = \top$ and $p_2 = \mathsf{SP}(f, \{\ldots, v_i \mapsto p_i, \ldots\}, p_{n+1})$. It suffices to find a packet $\alpha \notin [\![p_2]\!]$ (since $[\![p_1]\!] = \mathsf{Pk}$). From $\mathsf{RO}^{\mathsf{SP}}(p_2)$, there is a $p_i \neq p_{n+1}$ (syntactically!). Apply the induction hypothesis to get that $[\![p_i]\!] \neq [\![p_{n+1}]\!]$. This means that $[\![p_i]\!] \neq \mathsf{Pk}$ or $[\![p_{n+1}]\!] \neq \mathsf{Pk}$. In the first case, $[\![p_i]\!] \neq \mathsf{Pk}$. Pick a packet $\alpha \notin [\![p_i]\!]$, then since $\mathsf{RO}^{\mathsf{SP}}_{f'}(p_i)$ for $f' \sqsupset f$, p_i cannot test f, so it is also the case that $\alpha[f \leftarrow v_i] \notin [\![p_i]\!]$. This in turn means $\alpha \notin [\![p_2]\!]$ and therefore that $[\![p_2]\!] \neq \mathsf{Pk}$. On the other hand if instead $[\![p_{n+1}]\!] \neq \mathsf{Pk}$, then we pick $\alpha \notin [\![p_{n+1}]\!]$. By the assumption in Section 3.8.2, then we can pick a value $v_{n+1} \notin \{v_0, \ldots, v_n\}$. Then by $\mathsf{RO}^{\mathsf{SP}}_{f'}(p_{n+1})$ for $f' \sqsupset f$ and Lemma 7 we have $\alpha[f \leftarrow v_{n+1}] \notin [\![p_{n+1}]\!]$ and thus $\alpha[f \leftarrow v_{n+1}] \notin [\![p_2]\!]$, as needed.

2. Let $p_1 = \bot$ and $p_2 = \mathsf{SP}(f, \{\ldots, v_i \mapsto p_i, \ldots\}, p_{n+1})$. We need to show $[\![p_2]\!] \neq \emptyset$. By $\mathsf{RO}^{\mathsf{SP}}(p_2)$ there is a $p_i \neq p_{n+1}$. Apply the induction hypothesis to get $[\![p_i]\!] \neq [\![p_{n+1}]\!]$. Now this means either $[\![p_i]\!] \neq \emptyset$ or $[\![p_{n+1}]\!] \neq \emptyset$. These two subcases are similar to those in the previous case.

3. Let $p_1 = \mathsf{SP}(f, \{v_{1,0} \mapsto p_{1,0}, \ldots, v_{1,n} \mapsto p_{1,n}\}, p_{1,n+1})$ and $p_2 = \mathsf{SP}(f, \{v_{2,0} \mapsto p_{2,0}, \ldots, v_{2,m} \mapsto p_{2,m}\}, p_{2,m+1})$. We know that $p_1 \neq p_2$. There are two subcases for how this can be true:

   - We have $v_{1,i} \notin \{v_{2,0}, \ldots, v_{2,m}\}$ (w.l.o.g.). From $\mathsf{RO}^{\mathsf{SP}}(p_1)$, we know that $p_{1,i} \neq p_{1,n+1}$. Apply the induction hypothesis to get that $[\![p_{1,i}]\!] \neq [\![p_{1,n+1}]\!]$. In particular, let $\alpha \in [\![p_{1,i} \oplus p_{1,n+1}]\!]$. Then choose $v_{1,n+1} \notin \{v_{1,0}, \ldots, v_{1,n}, v_{2,0}, \ldots, v_{2,m}\}$ and observe that $\alpha[f \leftarrow v_{1,i}] \in [\![p_1]\!] \iff \alpha[f \leftarrow v_{1,n+1}] \notin [\![p_1]\!]$. But from $v_{1,i} \notin \{v_{2,0}, \ldots, v_{2,m}\}$ we know both of these packets will go to the default case of p_2. And $p_{2,m+1}$ is $\mathsf{RO}^{\mathsf{SP}}_{f'}$, which means both packets are in $[\![p_2]\!]$ or not in $[\![p_2]\!]$ because they differ only by f. We conclude $[\![p_1]\!] \neq [\![p_2]\!]$.

   - Otherwise we can renumber p_2 so that $v_{1,i} = v_{2,i}$ for all $i \in \{0, \ldots, n\}$

(noting that $(n = m)$ or else we could resolve in the previous case). Moreover there must be some $i \in \{0, \ldots, n+1\}$ for which $p_{1,i} \neq p_{2,i}$. Applying the inductive hypothesis, we get that $[\![p_{1,i}]\!] \neq [\![p_{2,i}]\!]$. Let $\alpha \in [\![p_{1,i}]\!] \oplus [\![p_{2,i}]\!]$. If $i \leq n$, then $\alpha[f \leftarrow v_i] \in [\![p_1]\!] \oplus [\![p_2]\!]$, and otherwise if $i = n+1$, we get $v_{n+1} \notin \{v_0, \ldots, v_n\}$ from the assumption in Section 3.8.2 and we have $\alpha[f \leftarrow v_{n+1}] \in [\![p_1]\!] \oplus [\![p_2]\!]$. In either case we see that $[\![p_1]\!] \neq [\![p_2]\!]$ as needed.

4. Let $p_1 = \mathsf{SP}(f_1, \{v_{1,0} \mapsto p_{1,0}, \ldots, v_{1,n} \mapsto p_{1,n}\}, p_{1,n+1})$, $p_2 = \mathsf{SP}(f_2, \{v_{2,0} \mapsto p_{2,0}, \ldots, v_{2,m} \mapsto p_{2,m}\}, p_{2,m+1})$, and without loss of generality assume $f_1 \sqsubset f_2$. Assume $\mathsf{RO}^{\mathsf{SP}}(p_1)$. Then there is at least one $i \leq n$ for which $p_{1,i} \neq p_{1,n+1}$. By the induction hypothesis, $[\![p_{1,i}]\!] \neq [\![p_{1,n+1}]\!]$. Let $\alpha[\![p_{1,i}]\!] \oplus [\![p_{1,n+1}]\!]$. Let $v_{n+1} \notin \{v_0, \ldots, v_n\}$. Now consider the two packets $\alpha[f_1 \leftarrow v_i]$ and $\alpha[f_1 \leftarrow v_{n+1}]$. Clearly by construction one is in $[\![p_1]\!]$ and the other is not. But they differ only by f_1 and p_2 cannot test f_1 because $f_1 \sqsubset f_2$ and $\mathsf{RO}^{\mathsf{SP}}(p_2)$, so they must either both be in $[\![p_2]\!]$ or both not be in $[\![p_2]\!]$. Thus one of the two packets witnesses $[\![p_1]\!] \neq [\![p_2]\!]$.

□

Finally, we show that the remaining operations defined above have the expected meanings semantically.

Theorem 12. *For SPs p_1, p_2, all of the following hold.*

1. $[\![p_1 \hat{+} p_2]\!] = [\![p_1]\!] \cup [\![p_2]\!]$.
2. $[\![p_1 \hat{\cdot} p_2]\!] = [\![p_1]\!] \cdot [\![p_2]\!]$.
3. $[\![p_1 \hat{\cap} p_2]\!] = [\![p_1]\!] \cap [\![p_2]\!]$.

4. $[\![p_1 \mathbin{\hat{-}} p_2]\!] = [\![p_1]\!] - [\![p_2]\!]$.

5. $[\![p_1 \mathbin{\hat{\oplus}} p_2]\!] = [\![p_1]\!] \oplus [\![p_2]\!]$.

6. $[\![p^{\hat{\star}}]\!] = [\![p]\!]^\star$

7. $[\![\hat{\forall}\ f\ p]\!] = \{\alpha \in \mathsf{Pk} \mid \forall v \in V \colon \alpha[f \leftarrow v] \in [\![p]\!]\}$

8. $[\![\hat{\exists}\ f\ p]\!] = \{\alpha \in \mathsf{Pk} \mid \exists v \in V \colon \alpha[f \leftarrow v] \in [\![p]\!]\}$

Proof. We again proceed by structural induction on p_1, p_2. We only show the cases for $\hat{+}$ and $\hat{\forall}$ because the other cases are similar or reduce directly to the other operators.

1. The base cases are trivial. For the inductive cases, let $p_1 = \mathsf{SP}(f, \{\ldots, v_{1,j} \mapsto p_{1,j}, \ldots\}, p_{1,n+1})$, and let $p_2 = \mathsf{SP}(f, \{\ldots, v_{2,k} \mapsto p_{2,k}, \ldots\}, p_{2,m+1})$. Let $\alpha \in \mathsf{Pk}$. Considering $\alpha_f = v_j$, we select a p_j: if $v_j \in \{v_{1,0}, \ldots, v_{1,n}\}$, then $p_j = p_{1,j}$, or if $v_j \notin \{v_{1,0}, \ldots, v_{1,n}\}$, then $p_j = p_{1,n+1}$. Similarly (treating the default cases uniformly with nondefault cases), we can select the child p_k from $\{p_{2,0}, \ldots, p_{2,m+1}\}$. From this:

$$\begin{aligned}
\alpha \in [\![p_1 \mathbin{\hat{+}} p_2]\!] &\iff \alpha \in [\![p_j \mathbin{\hat{+}} p_k]\!] && \text{Definition of } \hat{+}\text{; Case analysis for } \alpha_f \\
&\iff \alpha \in [\![p_j]\!] \cup [\![p_k]\!] && \text{Induction hypothesis} \\
&\iff \alpha \in [\![p_1]\!] \cup [\![p_2]\!] && \text{Definitions of } p_1, p_2
\end{aligned}$$

   On the other hand, let $p_1 = \mathsf{SP}(f_1, \{\ldots, v_{1,j} \mapsto p_{1,j}, \ldots\}, p_{1,n+1})$, $p_2 = \mathsf{SP}(f_2, \{\ldots, v_{2,k} \mapsto p_{2,k}, \ldots\}, p_{2,m+1})$, and w.l.o.g., suppose $f_1 \sqsubset f_2$. We see that the result is really by reduction to the previous case, since p_1 and $\mathsf{SP}(f_1, \emptyset, p_2)$ *do* match on their top level field (f_1). Clearly, $[\![p_2]\!] = [\![\mathsf{SP}(f_1, \emptyset, p_2)]\!]$, so that this method produces the correct result by the previous argument.

7. Let $S = \{\alpha \in \mathsf{Pk} \mid \forall v \in V : \alpha[f \leftarrow v] \in [\![p]\!]\}$. We need to show $[\![\hat{\forall}\ f\ p]\!] = S$:

   - Base cases: let $p = \top$ or $p = \bot$. Then trivially $\alpha \in S$ iff $\alpha \in [\![\hat{\forall}\ f\ p]\!] = [\![p]\!]$.
   - Let $p = \mathsf{SP}(f, \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})$. Then $\alpha \in S$ iff $\alpha \in [\![p_i]\!]$ for $i \leq 0 \leq n+1$, since the children (incl. default case) cover all values $v \in V$. But this is the same as $\alpha \in \hat{\bigcap}_{i \leq n+1} p_i$.
   - Let $p = \mathsf{SP}(f', \{v_0 \mapsto p_0, \ldots, v_n \mapsto p_n\}, p_{n+1})$, with $f \neq f'$. Then $\alpha \in [\![p]\!]$ iff $\alpha \in [\![\hat{\forall}\ f\ p_i]\!]$ for some child p_i of p, depending on the value of α_{f_2}. By the induction hypothesis, this is equivalent to $\alpha[f \leftarrow v] \in [\![p_i]\!]$ for all $v \in V$. But then we know $\alpha[f \leftarrow v]_{f'} = \alpha_{f'}$, so this is equivalent to $\alpha[f \leftarrow v] \in [\![p]\!]$ for all $v \in V$, i.e. that $\alpha \in S$, as needed.

□

3.10 Operations on Symbolic Packet Programs

In the implementation, there is also datatype defined for SPPs. We define its operations in this section.

Recall the definition of SPP:

$$p \in \mathsf{SPP} ::= \bot \mid \top \mid \underbrace{\mathsf{SPP}(f, \{\ldots, v_i \mapsto \{\ldots, w_{ij} \mapsto q_{ij}, \ldots\}, \ldots\}, \{\ldots, w_i \mapsto q_i, \ldots\}, q)}_{\equiv\ \sum_i f{=}v_i \cdot \sum_j f{\leftarrow}w_{ij} \cdot q_{ij}\ +\ (\prod_i f{\neq}v_i) \cdot (\sum_i f{\leftarrow}w_i\ +\ (\prod_i f{\neq}w_i) \cdot q)}$$

As with SPs, we take the semantics of an SPP to be the semantics of its associated NetKAT expression, shown here. As before, our SPP tuple here matches the SPP object type in the implementation, and we also only construct SPPs which

are canonical, in the sense of Section 3.2.2. We formalize this canonicalization property in Section 3.10.4.

First of all, there is a straightforward embedding from SP into SPP:

$$\mathsf{SPP}_{\mathsf{id}} \colon \mathsf{SP} \to \mathsf{SPP}$$

$$\mathsf{SPP}_{\mathsf{id}}(\bot) = \bot \qquad \mathsf{SPP}_{\mathsf{id}}(\top) = \top$$

$$\mathsf{SPP}_{\mathsf{id}}(\mathsf{SP}(f, b, d)) = \mathsf{SPP}(f, \{v_i \mapsto \{v_i \mapsto p_i\} \mid v_i \mapsto p_i \in b\}, \emptyset, \mathsf{SPP}_{\mathsf{id}}(d))$$

Observe that by applying the embedding to the SP definition for $f = v$ and $f \neq v$, we obtain the SPP forms for them:

$$(f = v) = \mathsf{SPP}(f, \{v \mapsto \{v \mapsto \top\}\}, \emptyset, \bot)$$

$$(f \neq v) = \mathsf{SPP}(f, \{v \mapsto \emptyset\}, \emptyset, \top)$$

In addition to this embedding, we only need one additional constructor:

$$(f \leftarrow v) \triangleq \mathsf{SPP}(f, \emptyset, \{v \mapsto \top\}, \bot)$$

The remaining SPPs we will construct from the operations given below.

3.10.1 Canonicalization

Once again, we define a smart constructor to help maintain the invariant that all SPPs in memory are canonical:

$$\mathsf{spp} \colon (F \times (V \Rightarrow (V \Rightarrow \mathsf{SPP})) \times (V \Rightarrow \mathsf{SPP}) \times \mathsf{SPP}) \to \mathsf{SPP}$$

Here is the definition:

$$\mathsf{spp}(f, b, m, d) \triangleq \text{ if } m' = \emptyset \wedge b'' = \emptyset \text{ then } d \text{ else } \mathsf{SPP}(f, b'', m', d)$$

where

$$m' = \{v \mapsto p \mid v \mapsto p \in m \wedge p \neq \bot\}$$

$$b' = \{v \mapsto \{w \mapsto p \mid w \mapsto p \in m_i \wedge p \neq \bot\} \mid v \mapsto m_i \in b\} \cup$$
$$\{v \mapsto m' \mid v \mapsto \bot \in m' \wedge v \notin b\}$$

$$b'' = \{v \mapsto m_i \mid v \mapsto m_i \in b' \wedge$$
$$m_i \neq (\text{if } v \in \mathsf{keys}(m') \vee d = \bot \text{ then } m' \text{ else } m' \cup \{v \mapsto d\})\}$$

3.10.2 Operations

In the rest of this section we give the definitions for the remaining operations used to build SPPs.

$$\hat{+}, \hat{:}, \hat{\cap}, \hat{\oplus}, \hat{-} \ : \ \mathsf{SPP} \times \mathsf{SPP} \to \mathsf{SPP} \qquad \hat{\star} \ : \ \mathsf{SPP} \to \mathsf{SPP}$$

The base cases are identical to the corresponding operations on SPs, so we omit them here. Similar to SPs, the inductive case for $\hat{+}, \hat{\cap}, \hat{\oplus}$, and $\hat{-}$ can be expressed at once. We do so here using $\hat{\pm}$. The notation $b(v; d)$ means the child $b(v)$ if $v \in \mathsf{keys}(b)$, or by default d otherwise.

$$\mathsf{SPP}(f, b_p, m_p, d_p) \hat{\pm} \mathsf{SPP}(f, b_q, m_q, d_q) \triangleq \mathsf{spp}(f, b', m_p \vec{\pm} m_q, d_p \hat{\pm} d_q)$$
where $b' = \{v \mapsto p(v) \vec{\pm} q(v) \mid v \in \mathsf{keys}(b_p \cup b_q \cup m_p \cup m_q)\}$

$$m_1 \vec{\pm} m_2 \triangleq \{v \mapsto m_1(v; \bot) \hat{\pm} m_2(v; \bot) \mid v \in \mathsf{keys}(m_1 \cup m_2)\},$$

$$p(v) \triangleq b_p(v; \text{ if } v \in m_q \vee d_p = \bot \text{ then } m_p \text{ else } m_p \cup \{v \mapsto d_p\}) \quad \text{(similarly for } q(v))$$

Sequencing ($\hat{\cdot}$) is more subtle and needs to be defined separately (using $\vec{\Sigma} \triangleq$ n-ary sum w.r.t. $\vec{+}$ as defined above):

$$\mathsf{SPP}(f, b_p, m_p, d_p) \mathbin{\hat{\cdot}} \mathsf{SPP}(f, b_q, m_q, d_q) \triangleq \mathsf{spp}(f, b', m_A \vec{+} m_B, d_p \mathbin{\hat{\cdot}} d_q)$$

$$\text{where} \quad b' = \{v \mapsto \vec{\sum_{v' \mapsto p' \in p(v)}} \{w' \mapsto p' \mathbin{\hat{\cdot}} q' \mid w' \mapsto q' \in q(v')\} \mid v \in \mathsf{keys}(b_p \cup b_q \cup m_p \cup m_q \cup m_A)\}$$

$$m_A = \vec{\sum_{v' \mapsto p' \in m_p}} \{w' \mapsto p' \mathbin{\hat{\cdot}} q' \mid w' \mapsto q' \in q(v')\}, \quad m_B = \{w' \mapsto d_p \mathbin{\hat{\cdot}} q' \mid w' \mapsto q' \in m_q\}$$

The expansion rule works similarly to the SP case. Namely, when the fields do not match, we expand the SPP with the greater field by:

$$p \equiv \mathsf{SPP}(f, \emptyset, \emptyset, p) \qquad \text{if} \qquad p \in \{\top, \bot, \mathsf{SPP}(f', b, m, d)\} \text{ where } f \sqsubset f'$$

Finally star is defined by:

$$p^{\hat{\star}} \triangleq \hat{\Sigma}_{i=0,1,\ldots} p^i$$

$$\text{where} \quad p^0 \triangleq \top$$

$$p^n \triangleq p \mathbin{\hat{\cdot}} p^{n-1} \quad \text{for } n \geq 1$$

In the implementation we perform repeated concatenations until the SPP no longer changes (i.e., the fixpoint is reached).

3.10.3 Push and Pull

First we define two helper functions which "flatten" SPPs into SPs:

$$\mathsf{fwd} : \mathsf{SPP} \to \mathsf{SP} \qquad\qquad \mathsf{bwd} : \mathsf{SPP} \to \mathsf{SP}$$

The idea of fwd is to collect a symbolic packet representing the set of output

packets for an SPP when the input packets can be any packet:

$$\mathsf{fwd}(\top) \triangleq \top$$

$$\mathsf{fwd}(\bot) \triangleq \bot$$

$$\mathsf{fwd}(\mathsf{SPP}(f, b, m, d)) \triangleq \mathsf{sp}(f, b_A \vec{+} b_B \vec{+} b_C \vec{+} b_D, \mathsf{fwd}(d))$$

$$\begin{aligned} \text{where} \quad b_A &= \{v' \mapsto \mathsf{fwd}(p') \mid v' \mapsto p' \in m\} \\ b_B &= \vec{\Sigma}_{v' \mapsto m' \in b} \{w' \mapsto \mathsf{fwd}(p') \mid w' \mapsto p' \in m'\} \\ b_C &= \{v' \mapsto \mathsf{fwd}(d) \mid v' \in \mathsf{keys}(b_B) \setminus (\mathsf{keys}(b) \cup \mathsf{keys}(m))\} \\ b_D &= \{v' \mapsto \bot \mid v' \in \mathsf{keys}(b) \setminus \mathsf{keys}(b_B)\} \end{aligned}$$

Conversely, bwd gives the set of input packets which, when input to the given SPP, produce some nonempty set of packets as output:

$$\mathsf{bwd}(\top) \triangleq \top$$

$$\mathsf{bwd}(\bot) \triangleq \bot$$

$$\mathsf{bwd}(\mathsf{SPP}(f, b, m, d)) \triangleq \mathsf{sp}(f, b_A \vec{+} b_B, d' \hat{+} \mathsf{bwd}(d))$$

$$\begin{aligned} \text{where} \quad d' &= \hat{\Sigma}_{v' \mapsto p' \in m} \mathsf{bwd}(p') \\ b_A &= \{v' \mapsto \hat{\Sigma}_{w' \mapsto p' \in m'} \mathsf{bwd}(p') \mid v' \mapsto m' \in b\} \\ b_B &= \{v \mapsto d' \mid v \in \mathsf{keys}(m) \setminus \mathsf{keys}(b)\} \end{aligned}$$

Recall the push and pull operations which are described in Section 3.2.2:

$$\mathsf{push} : \ \mathsf{SP} \to \mathsf{SPP} \to \mathsf{SP} \qquad \mathsf{pull} : \ \mathsf{SPP} \to \mathsf{SP} \to \mathsf{SP}$$

We conclude the section with their definitions:

$$\mathsf{push}\ p\ s \triangleq \mathsf{fwd}((\mathsf{SPP}_{\mathsf{id}}\ p) \mathbin{\hat{\cdot}} s),$$

$$\mathsf{pull}\ s\ p \triangleq \mathsf{bwd}(s \mathbin{\hat{\cdot}} (\mathsf{SPP}_{\mathsf{id}}\ p)),$$

3.10.4 Correctness of SPP operations

Finally, we give proofs that the SPP operations maintain a canonical form and have the expected semantic meanings.

Definition 13 (Reduced and Ordered)**.** *An SPP p is* reduced and ordered for a field $f \in F$ *if it satisfies:*

$$\mathsf{RO}_f^{\mathsf{SPP}} : \mathsf{SPP} \to 2$$

$$\mathsf{RO}_f^{\mathsf{SPP}}(\bot) \triangleq \top \qquad\qquad \mathsf{RO}_f^{\mathsf{SPP}}(\top) \triangleq \top$$

$\mathsf{RO}_f^{\mathsf{SPP}}(\mathsf{SPP}(f', b, m, d)) \triangleq \top$ *if all of the following hold:*

1. $b \neq \emptyset \vee m \neq \emptyset$
2. $f \sqsubset f'$
3. $\forall (v_i \mapsto m_i) \in b.\ (\forall (v_j \mapsto p_j) \in m_i.\ \mathsf{RO}_{f'}^{\mathsf{SPP}}(p_j) \wedge p_j \neq \bot)$
   $\wedge((m_i \neq m \wedge (v_i \in \mathit{keys}(m) \vee d = \bot)) \vee (m_i \neq (m \cup \{v_i \mapsto d\}) \wedge v_i \notin \mathit{keys}(m) \wedge d \neq \bot))$
4. $(\forall v \mapsto p_i) \in m.\ \mathsf{RO}_{f'}^{\mathsf{SPP}}(p_i) \wedge p_i \neq \bot$
5. $\mathsf{RO}_{f'}^{\mathsf{SPP}}(d)$

$\mathsf{RO}_f^{\mathsf{SPP}}(\mathsf{SPP}(f', b, m, d)) \triangleq \bot$ *otherwise.*

More generally, an SPP p is reduced and ordered *if it satisfies:*

$$\mathsf{RO}^{\mathsf{SPP}} : \mathsf{SPP} \to 2$$

$$\mathsf{RO}^{\mathsf{SPP}}\ (p) \triangleq \exists f \in F : \mathsf{RO}_f^{\mathsf{SPP}}\ (p)$$

Using this definition, we first establish that the smart constructor only produces reduced and ordered SPPs, provided the children in the test and mutation branches are already reduced and ordered.

Lemma 14. *Let* $f \in F, b \in V \Rightarrow (V \Rightarrow \mathsf{SPP}), m \in V \Rightarrow \mathsf{SPP}, d \in \mathsf{SPP}$. *If all of the following hold:*

(a) $f \sqsubset f'$,

(b) $\forall v_i \mapsto m_i \in b.\ (\forall v_j \mapsto p_j \in m_i.\ \mathsf{RO}^{\mathsf{SPP}}_{f'}(p_j))$

(c) $\forall v_i \mapsto p_i \in m.\ \mathsf{RO}^{\mathsf{SPP}}_{f'}(p_i)$

(d) $\mathsf{RO}^{\mathsf{SPP}}_{f'}(d)$

Then $\mathsf{RO}^{\mathsf{SPP}}_{f}\ (\mathsf{spp}(f, b, m, d))$.

Proof. Clearly if $\mathsf{spp}(f, b, m, d) = \bot$ or $\mathsf{spp}(f, b, m, d) = \top$ then the result holds trivially. So suppose $\mathsf{SPP}(f, b'', m', d) = \mathsf{spp}(f, b, m, d)$. Note the variable names are chosen to match those in the definition of spp). We establish the criteria of $\mathsf{RO}^{\mathsf{SPP}}$ one at a time, following the list in the definition.

1. Holds because both b'' and m' being empty is prevented by the outermost if statement in spp.
2. By assumption (a).
3. Let $(v_i \mapsto m_i) \in b''$ and let $v_j \mapsto p_j \in m_i$. We have $\mathsf{RO}^{\mathsf{SPP}}_{f'}(p_j)$ by assumption (b). In addition, $p_j \neq \bot$ because such bindings are filtered out in the construction of b', and the construction of b'' only removes more bindings (not adding any). The remaining condition matches precisely what is checked by the construction of b''.

4. Let $(v_i \mapsto p_i) \in m'$. That we have $\mathsf{RO}^{\mathsf{SPP}}_{f'}(p_i)$ holds by assumption (c). That $p_i \neq \bot$ is true because such bindings are removed by the construction of m' from m.

5. By assumption (d).

□

Next, it follows that all of the operations maintain the property of being reduced and ordered.

Corollary 15. *If for SPPs p_1, p_2, we have $\mathsf{RO}^{\mathsf{SPP}}(p_1)$ and $\mathsf{RO}^{\mathsf{SPP}}(p_2)$, then all of the following hold:*

1. $\mathsf{RO}^{\mathsf{SPP}}(p_1 \mathbin{\hat{+}} p_2)$,
2. $\mathsf{RO}^{\mathsf{SPP}}(p_1 \mathbin{\hat{\cdot}} p_2)$,
3. $\mathsf{RO}^{\mathsf{SPP}}(p_1 \mathbin{\hat{\cap}} p_2)$,
4. $\mathsf{RO}^{\mathsf{SPP}}(p_1 \mathbin{\hat{-}} p_2)$,
5. $\mathsf{RO}^{\mathsf{SPP}}(p_1 \mathbin{\hat{\oplus}} p_2)$,
6. $\mathsf{RO}^{\mathsf{SPP}}(p_1^{\hat{\star}})$,

Proof. Because we only use the smart constructor to create SPPs in the definitions of all of the operations, we only need to show that the premises of Lemma 14 are satisfied when the smart constructor is used. Indeed, these premises are guaranteed by $\mathsf{RO}^{\mathsf{SPP}}(p_1)$, and $\mathsf{RO}^{\mathsf{SPP}}(p_2)$, and comparisons of f, f' in the definitions of the operations. We only need to be slightly careful in the cases where the smart constructor is *not* used (since the expansion form is not RO), but each of these is a direct reduction to a case where the smart constructor is used. Note that the

default case of the expansion form is still RO by assumption, and the expansion rule itself maintains the field order. □

The main benefit of canonicalization is that, after establishing reduced an ordered form, we can check semantic equivalence by checking syntactic equivalence, which we show next.

Theorem 16. *For all* $p_1, p_2 \in \mathsf{SP}$*, if* $\mathsf{RO}^{\mathsf{SPP}}(p_1)$ *and* $\mathsf{RO}^{\mathsf{SPP}}(p_2)$*, then* $p_1 = p_2 \iff [\![p_1]\!] = [\![p_2]\!]$.

Proof. Suppose $\mathsf{RO}^{\mathsf{SPP}}(p_1)$ and $\mathsf{RO}^{\mathsf{SPP}}(p_2)$. ($\Rightarrow$) is immediate because $[\![\cdot]\!]$ is a function. ($\Leftarrow$). Suppose $p_1 \neq p_2$ (syntactically). We will show $[\![p_1]\!] \neq [\![p_2]\!]$ by induction on the structure of p_1, p_2:

1. Let $p_1 = \bot$, and let $p_2 = \top$. Then $[\![p_1]\!] = \emptyset$ and $\alpha \in [\![p_2]\!](\alpha)$ for all $\alpha \in \mathsf{Pk}$.
2. Let $p_1 = \bot$, and let $p_2 = \mathsf{SPP}(f, b, m, d)$. We only need to show that $[\![p_2]\!]$ is nonempty. First of all, if $d \neq \bot$, we are done, since by induction this means $[\![d]\!] \neq \emptyset$, and moreover $\beta \in [\![d]\!](\alpha)$ means that $\beta \in [\![p_2]\!](\alpha[f \leftarrow v])$ for any $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$. So assume $d = \bot$. Either $b \neq \emptyset$ or $m \neq \emptyset$. If $m \neq \emptyset$ we are done, because $v \mapsto p \in m$ means that $p \neq \bot$, and thus by the induction hypothesis that there are $\alpha, \beta \in \mathsf{Pk}$ for which $\beta \in [\![p]\!]$. It follows that $\beta[f \leftarrow v] \in [\![p_2]\!](\alpha)$. So assume $m = \emptyset$. Then there is some $(v_i \mapsto m_i) \in b$. Now because $d = \bot$ and $m = \emptyset$, item (3) of the definition of $\mathsf{RO}_f^{\mathsf{SPP}}$ says that $m_i \neq \emptyset$. Thus there is some $v_j \mapsto p_j \in m_i$ for which $p_j \neq \bot$. So there are packets α, β such that $\beta \in [\![p_j]\!](\alpha)$, and thus $\beta[f \leftarrow v_j] \in [\![p_2]\!](\alpha[f \leftarrow v_i])$.
3. Let $p_1 = \top$, and let $p_2 = \mathsf{SPP}(f, b, m, d)$. We need to show p_2 modifies ($\beta \in [\![p_2]\!](\alpha)$ for $\beta \neq \alpha$) or drops a packet ($[\![p_2]\!](\alpha) = \emptyset$). If $d \neq \top$, then by

the induction hypothesis, either $\beta \in [\![d]\!](\alpha)$ for $\beta \neq \alpha$ or $[\![d]\!](\alpha) = \emptyset$. Then for $\alpha[f \leftarrow v]$ for $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$, we have either $\beta[f \leftarrow v] \in [\![p_2]\!](\alpha[f \leftarrow v])$ with $\beta[f \leftarrow v] \neq \alpha[f \leftarrow v]$ or $[\![p_2]\!](\alpha[f \leftarrow v]) = \emptyset$, respectively. So assume $d = \top$. If $(v_i \mapsto p_i) \in m$, then since $p_i \neq \bot$, we have $\alpha[f \leftarrow v_i] \in [\![p_2]\!](\alpha[f \leftarrow v_j])$ for any $v_j \neq v_i$. So assume $m = \emptyset$. Since then $b \neq \emptyset$, there is some $(v_i \mapsto m_i) \in b$. From item (3) of the definition of $\mathsf{RO}_f^{\mathsf{SPP}}$, we know $m_i \neq \{v_i \mapsto \top\}$. There are several subcases based on what m_i can be:

- If $m_i = \emptyset$, we are done since then $[\![p_2]\!](\alpha[f \leftarrow v_i]) = \emptyset$.
- If $m_i = \{v_j \mapsto \top, \ldots\}$, with $v_j \neq v_i$. Then $\alpha[f \leftarrow v_j] \in [\![p_2]\!]\alpha[f \leftarrow v_i]$ (for any α).
- If $m_i = \{v_j \mapsto p_j, \ldots\}$ with $p_j \neq \top$ (but possibly $v_i = v_j$), then we have by the induction hypothesis either a packet α such that $[\![p_j]\!](\alpha) = \emptyset$ (in which case $[\![p_2]\!](\alpha[f \leftarrow v_i]) = \emptyset$), or some $\beta[f \leftarrow v_j] \neq \alpha[f \leftarrow v_j]$ such that $\beta \in [\![p_j]\!](\alpha)$ (in which case $\beta[v \leftarrow v_j] \in [\![p_2]\!](\alpha[f \leftarrow v_i])$.

4. Let $p_1 = \mathsf{SPP}(f_1, b_1, m_1, d_1)$, and let $p_2 = \mathsf{SPP}(f_2, b_2, m_2, d_2)$. We proceed by the cases in which p_1 and p_2 may differ syntactically, and show why each leads to a difference in semantics:

   - Suppose $f_1 \sqsubset f_2$, and suppose $v_i \mapsto p_i \in m_1$. Then $p_i \neq \bot$, meaning there are packets α, β for which $\beta \in [\![p_i]\!](\alpha)$. Pick $v_j \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1)$. Then $\beta[f_1 \leftarrow v_i] \in [\![p_1]\!](\alpha[f_1 \leftarrow v_j])$, but p_2 cannot modify f_1 at all, so we must have $\beta[f_1 \leftarrow v_i] \notin [\![p_2]\!](\alpha[f_1 \leftarrow v_j])$. On the other hand suppose $m_1 = \emptyset$. Then there must be $v_i \mapsto m_i \in b_1$. Now we have two cases, in item (3) of $\mathsf{RO}^{\mathsf{SPP}}(p_1)$:
     - Let $d_1 = \bot$. Then $m_i \neq \emptyset$. Let $v_j \mapsto p_j \in m_i$. Note $p_j \neq \bot$. Then for α, β such that $\beta \in [\![p_j]\!](\alpha)$, we have $\beta[f_1 \leftarrow v_j] \in [\![p_1]\!](\alpha[f_1 \leftarrow v_i])$.

If $v_i \neq v_j$, we are done, since p_2 cannot modify f_1. Otherwise let $v'_j \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1)$. Then because $d_1 = \bot$, $[\![p_1]\!](\alpha[f_1 \leftarrow v_i]) \neq \emptyset = [\![p_1]\!](\alpha[f_1 \leftarrow v'_j])$. But p_2 cannot test f_1, so we must have $[\![p_2]\!](\alpha[f_1 \leftarrow v_i]) = [\![p_2]\!](\alpha[f_1 \leftarrow v'_j])$.

– Let $d_1 \neq \bot$. Then $m_i \neq \{v_i \mapsto d_1\}$. This means for any $v_j \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1)$, there must be some α, β such that $\beta[f_1 \leftarrow v_i] \in [\![p_1]\!](\alpha[f_1 \leftarrow v_i]) \iff \beta[f_1 \leftarrow v_j] \notin [\![p_1]\!](\alpha[f_1 \leftarrow v_j])$. But p_2 cannot test f_1, so we are done.

For the remaining cases, we let $f = f_1 = f_2$.

- Suppose $d_1 \neq d_2$. Then by the induction hypothesis there is a packet α for which $[\![d_1]\!](\alpha) \neq [\![d_2]\!](\alpha)$. We just pick a value $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2) \cup \mathsf{keys}(b_2)$, and we get $[\![p_1]\!](\alpha[f \leftarrow v]) \neq [\![p_2]\!](\alpha[f \leftarrow v])$.
- Suppose $\mathsf{keys}(m_1) \neq \mathsf{keys}(m_2)$. Then (w.l.o.g.) let $v_i \mapsto p_i \in m_1$ and $v_i \notin \mathsf{keys}(m_2)$, and choose $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_2)$. Then since $p_i \neq \bot$, there are $\alpha, \beta \in \mathsf{Pk}$ such that $\beta \in [\![p_i]\!](\alpha)$. This means $\beta[f \leftarrow v_i] \in [\![p_1]\!](\alpha[f \leftarrow v])$, but $\beta[f \leftarrow v_i] \notin [\![p_2]\!](\alpha[f \leftarrow v])$.
- Suppose $v \mapsto p_i \in m_1$, while $v \mapsto p_j \in m_2$, and $p_i \neq p_j$. By the induction hypothesis, there is a $\alpha \in \mathsf{Pk}$ for which $[\![p_i]\!](\alpha) \neq [\![p_j]\!](\alpha)$. Pick $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)$. Then $[\![p_1]\!](\alpha[f \leftarrow v]) \neq [\![p_2]\!](\alpha[f \leftarrow v])$.
- Assume $d_1 = d_2$ and $m_1 = m_2$, but suppose $\mathsf{keys}(b_1) \neq \mathsf{keys}(b_2)$. Then (w.l.o.g.) let $v_i \mapsto m_i \in b_1$ and $v_i \notin \mathsf{keys}(b_2)$. Now there are two cases depending on whether $v_i \in \mathsf{keys}(m_1) \vee d_1 = \bot$, in which case we know $m_i \neq m_1$. But since $m_1 = m_2$, this means $m_i \neq m_2$. Following the reasoning of the previous two cases (different keys or different SPP children), we find $[\![p_1]\!](\alpha[f \leftarrow v_i]) \neq [\![p_2]\!](\alpha[f \leftarrow v_i])$. On the other hand,

$v_i \notin \mathsf{keys}(m_1) \wedge d_1 \neq \bot$, in which case $m_i \neq (m_1 \cup \{v_i \mapsto d_1\})$. Since $m_1 = m_2$ and $d_1 = d_2$, this means that also $m_i \neq (m_2 \cup \{v_i \mapsto d_2\})$. There are 4 subcases for how this inequality can happen:

  - Suppose $v_i \notin \mathsf{keys}(m_i)$. Then there are α, β for which $\beta \in [\![d_2]\!](\alpha)$ meaning $\beta[f \leftarrow v_i] \in [\![p_2]\!](\alpha[f \leftarrow v_i])$, but $\beta[f \leftarrow v_i] \notin [\![p_1]\!](\alpha[f \leftarrow v_i])$.
  - Otherwise $v_i \mapsto p \in m_i$ for $p \neq d_2$. By the induction hypothesis, there is a $\alpha \in \mathsf{Pk}$ for which $[\![p]\!](\alpha) \neq [\![d_2]\!](\alpha)$. But then $[\![p_1]\!](\alpha[f \leftarrow v_i]) \neq [\![p_2]\!](\alpha[f \leftarrow v_i])$.
  - The other two subcases are different keys or different SPP children for m_i and m_2, but then the counterexample packets are constructed similarly to the previous cases.

- Suppose $v \mapsto m_i \in b_1$, while $v \mapsto m_j \in b_2$, and $\mathsf{keys}(m_i) \neq \mathsf{keys}(m_j)$. Then (w.l.o.g.) let $v' \mapsto p \in m_i$, but $v' \notin \mathsf{keys}(m_j)$. From $\mathsf{RO}^{\mathsf{SPP}}(p_1)$, we know $p \neq \bot$. So there are packets $\alpha, \beta \in \mathsf{Pk}$ such that $\beta \in [\![p]\!](\alpha)$. But then $\beta[f \leftarrow v'] \in [\![p_1]\!](\alpha[f \leftarrow v])$ but $\beta[f \leftarrow v'] \notin [\![p_2]\!](\alpha[f \leftarrow v])$ because p_2 does not have an assignment to v' in the v branch for f.
- Suppose $v \mapsto m_i \in b_1$, while $v \mapsto m_j \in b_2$, and $v' \mapsto p_i \in m_i$, while $v' \mapsto p_j \in m_j$ for $p_i \neq p_j$. By the induction hypothesis, for some packet $\alpha \in \mathsf{Pk}$, we have $[\![p_i]\!](\alpha) \neq [\![p_j]\!](\alpha)$. But then $[\![p_1]\!](\alpha[f \leftarrow v]) \neq [\![p_2]\!](\alpha[f \leftarrow v])$.

□

The rest of the results have to do with the semantics of SPPs. First we justify the "expansion" construction used in several operations to simplify the construction:

Lemma 17. *Suppose* $f_1, f_2 \in F$*, with* $f_1 \sqsubset f_2$*, and* $p = \mathsf{SPP}(f_2, b, m, d)$*. Then* $[\![\mathsf{SPP}(f_1, \emptyset, \emptyset, p))]\!] = [\![p]\!]$.

Proof. For any $\alpha \in \mathsf{Pk}$, clearly $\alpha_{f_1} \notin \mathsf{keys}(\emptyset)$ (i.e., we neither test or set f_1), and it follows that $\beta \in [\![p]\!](\alpha)$ iff $\beta \in [\![\mathsf{SPP}(f_1, \emptyset, \emptyset, p)]\!](\alpha)$. □

Next, we show the SPP smart constructor does not change the semantics, compared to the plain datatype constructor:

Lemma 18 (Semantic equivalence of SPP smart constructor)**.** *For all* $f \in F, b \in (V \Rightarrow (V \Rightarrow \mathsf{SPP})), m \in (V \Rightarrow \mathsf{SPP}), d \in \mathsf{SPP}$*, we have that* $[\![\mathsf{SPP}(f, b, m, d)]\!] = [\![\mathsf{spp}(f, b, m, d)]\!]$.

Proof. We argue in three steps, referring to the local variables in the spp definition:

- First, $[\![\mathsf{SPP}(f, b, m, d)]\!] = [\![\mathsf{SPP}(f, b', m', d)]\!]$

  ($\subseteq$). Let $\alpha, \beta \in \mathsf{Pk}$, and suppose $\beta \in [\![\mathsf{SPP}(f, b, m, d)]\!](\alpha)$. Let $v = \alpha_f$ and let $v' = \beta_f$. Otherwise there are three cases:

  (a) Suppose $v \in \mathsf{keys}(b)$. Clearly since $\beta \in [\![\mathsf{SPP}(f, b, m, d)]\!](\alpha)$, we know $b[v][v'] \neq \bot$. This means that $b'[v][v'] = b[v][v']$, and thus that $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$.

  (b) Suppose $v \in \mathsf{keys}(m) \setminus \mathsf{keys}(b)$. Then $v' \mapsto s \in m$ and $\beta \in [\![s]\!](\alpha[f \leftarrow v'])$. Thus $s \neq \bot$, and we so must have $v' \mapsto s \in m'$, and therefore that $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$.

  (c) Finally if $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$, then $\beta \in [\![d]\!](\alpha)$ (in which case we are done), or $\beta \in [\![s]\!](\alpha[f \leftarrow v'])$ for some $v' \mapsto s \in m$. By the same reason in the previous case, $v' \mapsto s \in m'$, and thus $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$.

($\supseteq$). Let $\alpha, \beta \in \mathsf{Pk}$, and suppose $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$. Let $v = \alpha_f$ and let $v' = \beta_f$. Again there are three cases:

(a) Suppose $v \in \mathsf{keys}(b')$. There are two subcases:

(i) Suppose $v' \in \mathsf{keys}(b'[v])$ and $b'[v][v'] \neq \bot$ (i.e. v is in the left addend of b'), then by construction $b[v][v'] \neq \bot$ and thus $\beta \in [\![\mathsf{SPP}(f, b, m, d)]\!](\alpha)$.

(ii) Otherwise it must be that $v \mapsto \bot \in m'$ and $v' \in \mathsf{keys}(m')$. This, in turn, means $v' \mapsto s \in m$ and $s \neq \bot$. We conclude $\beta \in [\![\mathsf{SPP}(f, b, m, d)]\!](\alpha)$.

(b) Suppose $v \in \mathsf{keys}(m') \setminus \mathsf{keys}(b')$. Then $v' \in \mathsf{keys}(m')$, and the same reasoning as Case (a)(ii) applies.

(c) Finally if $v \notin \mathsf{keys}(b') \cup \mathsf{keys}(m')$, then either $v' \in \mathsf{keys}(m')$, in which case the reasoning of Case (a)(ii) applies, or $\beta \in [\![d]\!](\alpha)$, which also means $\beta \in [\![\mathsf{SPP}(f, b, m, d)]\!](\alpha)$.

- Next, $[\![\mathsf{SPP}(f, b', m', d)]\!] = [\![\mathsf{SPP}(f, b'', m', d)]\!]$

Note that $b'' \subseteq b'$ since b'' is constructed by removing mappings from b'.

($\subseteq$). Let $\alpha, \beta \in \mathsf{Pk}$, and suppose $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$. Let $v = \alpha_f$ and let $v' = \beta_f$. There are two cases:

(a) Let $v \in \mathsf{keys}(b'')$. We know $b''[v][v'] = b'[v][v']$, so $\beta \in [\![\mathsf{SPP}(f, b'', m', d)]\!](\alpha)$.

(b) Let $v \in \mathsf{keys}(b') \setminus \mathsf{keys}(b'')$. Let $m_i = b'[v]$. Note that $\beta \in [\![m_i[v']]\!](\alpha)$. There are two subcases:

(i) Suppose $v \in \mathsf{keys}(m')$ or $d = \bot$. Then from the fact that v was removed, we know $m_i = m'$, which in turn means $v' \in \mathsf{keys}(m')$,

and thus that $\beta \in [\![\mathsf{SPP}(f, b'', m', d)]\!]$ as needed.

(ii) Otherwise, we know $m_i = m' \cup \{v \mapsto d\}$. So either $v' = v$, in which case that means $\beta \in [\![d]\!](\alpha)$, or $v' \neq v$, in which case $\beta \in [\![m'[v']]\!](\alpha[f \leftarrow v'])$. In both cases, we have $\beta \in [\![\mathsf{SPP}(f, b'', m', d)]\!]$.

(c) Let $v \notin \mathsf{keys}(b')$. Then either $\beta \in [\![d]\!](\alpha)$ or $v' \mapsto s \in m'$ and $\beta \in [\![s]\!](\alpha[f \leftarrow v'])$. In either case, we have $\beta \in [\![\mathsf{SPP}(f, b'', m', d)]\!](\alpha)$ as needed.

($\supseteq$). Let $\alpha, \beta \in \mathsf{Pk}$, and suppose $\beta \in [\![\mathsf{SPP}(f, b'', m', d)]\!](\alpha)$. Let $v = \alpha_f$ and let $v' = \beta_f$. There are two cases:

(a) Let $v \in \mathsf{keys}(b'')$. Then $\beta \in [\![b''[v][v']]\!](\alpha)$, and since $b'' \subseteq b'$, we have $\beta \in [\![b'[v][v']]\!](\alpha)$, and thus $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$.

(b) Let $v \in \mathsf{keys}(b') \setminus \mathsf{keys}(b'')$. Then there are 3 subcases:

(i) Suppose $v \in \mathsf{keys}(m')$. So we must have $v' \in \mathsf{keys}(m')$ also ($v \in \mathsf{keys}(m')$ means d is ignored). From $v \in \mathsf{keys}(b') \setminus \mathsf{keys}(b'')$, we know $b'[v] = m'$ and thus that $\beta \in [\![b'[v][v']]\!](\alpha)$, giving $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!]$.

(ii) Suppose $d = \bot$. We know $v' \in \mathsf{keys}(m')$ because $[\![d]\!] = \emptyset$. From here, reasoning follows the previous case that $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!]$.

(iii) Otherwise, we know $b'[v] = m' \cup \{v \mapsto d\}$.

(c) Let $v \notin \mathsf{keys}(b')$. Then either $\beta \in [\![d]\!](\alpha)$ or $v' \mapsto s \in m'$ and $\beta \in [\![s]\!](\alpha[f \leftarrow v'])$. In either case, because $v \notin \mathsf{keys}(b')$, we have $\beta \in [\![\mathsf{SPP}(f, b', m', d)]\!](\alpha)$ as needed.

- Finally, if $b'' = \emptyset$ and $m' = \emptyset$, then $[\![\mathsf{SPP}(f, b'', m', d)]\!] = [\![d]\!]$. This is exactly Lemma 17.

Putting the steps together, we have shown $[\![\mathsf{SPP}(f, b, m, d)]\!] = [\![\mathsf{spp}(f, b, m, d)]\!]$, as required. □

Lemma 19 (Correctess of $\mathsf{SPP}_{\mathsf{id}}$)**.** *Let p be an* SP*. For any $\alpha, \beta \in \mathsf{Pk}$, then $\beta \in [\![\mathsf{SPP}_{\mathsf{id}}\ p]\!](\alpha)$ iff $\beta = \alpha$ and $\beta \in [\![p]\!]$.*

Proof. Straightforward induction on p. □

The next lemma makes the meaning of $p(v)$ more precise:

Lemma 20 (Behavior of $p(v)$)**.** *Let $p = \mathsf{SPP}(f, b, m, d), v \in V$, and $\alpha \in \mathsf{Pk}$. Then:*

$$[\![p]\!](\alpha[f \leftarrow v]) = \bigcup_{(v_i \mapsto p_i) \in p(v)} [\![p_i]\!](\alpha[f \leftarrow v_i])$$

Proof. Let $\alpha, \beta \in \mathsf{Pk}$. We proceed by the four cases of $p(v)$ that corresponding to $p = \mathsf{SPP}(f, b, m, d)$. In each case, we will show $\beta \in [\![p]\!](\alpha[f \leftarrow v])$ iff $\beta \in [\![p_i]\!](\alpha[f \leftarrow v_i])$, observing that the union is actually disjoint.

- Let $v \in \mathsf{keys}(b)$. Then each $(v_i \mapsto p_i)$ is a value and child guarded by $(f = v)$, so we have $\beta \in [\![p]\!](\alpha[f \leftarrow v])$ iff $\beta \in [\![p_i]\!](\alpha[f \leftarrow v_i])$.
- Let $v \in \mathsf{keys}(m) \setminus \mathsf{keys}(b)$. Let $v \mapsto p_i \in m$. Since $v \notin \mathsf{keys}(b)$, $\alpha[f \leftarrow v]$ passes the $(f \neq v)$ that guard the assignments in m. The significance of $v \in \mathsf{keys}(m)$ is that we know the default case d has negative guards $(f \neq v)$ for these values, so d cannot be applied to this packet. Once again $\beta \in [\![p]\!](\alpha[f \leftarrow v])$ iff $\beta \in [\![p_i]\!](\alpha[f \leftarrow v_i])$
- Let $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$, but $d = \bot$. This is similar to the previous case in that the p_i in m are precisely the reachable children, but now $[\![p]\!](\alpha[f \leftarrow v]) = [\![d]\!] = [\![\bot]\!] = \emptyset$.

- Let $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$, and $d \neq \bot$. We apply each child $v_i \mapsto p_i$ from m, but we also have the mapping $v \mapsto d$ (i.e. $v_i = v$ and $p_i = d$). This is appropriate because $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$ ensures we pass all the guards on d, and we have $v_i = v$ because no update to the packet is performed before applying d.

□

We need one more tiny lemma describing the behavior of $\vec{+}$ that we will reuse.

Lemma 21. *For maps $m_1, m_2 \colon V \Rightarrow \mathsf{SPP}$, provided that $[\![p_i \mathbin{\hat{+}} p_j]\!] = [\![p_i]\!] \cup [\![p_j]\!]$ for all p_i, p_j in the values of m_1, m_2, then we have $\beta \in [\![p]\!](\alpha)$ for some binding $v \mapsto p \in m_1 \vec{+} m_2$ iff $\beta \in [\![p_i]\!](\alpha)$ for a binding $v \mapsto p_i \in m_1$ or $\beta \in [\![p_j]\!](\alpha)$ for a binding $v \mapsto p_j \in m_1$.*

Proof. The claim is true by inspection of the definition of $\vec{+}$: the only nontrivial case of that analysis is that there are bindings for v in both m_1 and m_2, but in that case we have $p = p_i \mathbin{\hat{+}} p_j$, and we apply the additional assumption to get the lemma as stated. □

We are ready to show that the operations on SPPs have their expected meanings:

Theorem 22. *For SPPs p_1, p_2, all of the following hold.*

1. $[\![p_1 \mathbin{\hat{+}} p_2]\!] = [\![p_1]\!] \cup [\![p_2]\!]$.
2. $[\![p_1 \mathbin{\hat{\cdot}} p_2]\!](\alpha) = \{\beta \in \mathsf{Pk} \mid \gamma \in [\![p_1]\!](\alpha), \beta \in [\![p_2]\!](\gamma)\}$.
3. $[\![p_1 \mathbin{\hat{\cap}} p_2]\!] = [\![p_1]\!] \cap [\![p_2]\!]$.

4. $[\![p_1 \mathbin{\hat{-}} p_2]\!] = [\![p_1]\!] - [\![p_2]\!]$.

5. $[\![p_1 \mathbin{\hat{\oplus}} p_2]\!] = [\![p_1]\!] \oplus [\![p_2]\!]$.

6. $[\![p^{\hat{\star}}]\!] = \bigcup_{n \geq 0} [\![p^n]\!]$.

Proof. We again proceed by structural induction on p_1, p_2. We only show the cases for $\hat{+}$ and $\hat{\cdot}$ because the other cases are similar or reduce directly to the other operators. In each case, we consider arbitrary packets $\alpha, \beta \in \mathsf{Pk}$.

Now we proceed according to the cases in the theorem statement.

1. We need to show $\beta \in [\![p_1 \mathbin{\hat{+}} p_2]\!](\alpha)$ iff $\beta \in [\![p_1]\!](\alpha)$ or $\beta \in [\![p_2]\!](\alpha)$. The base cases (where both p_1 or p_2 are $\top$ or $\bot$) are trivial. The cases where only one of p_1, p_2 is $\bot$ or $\top$ reduce to the general case by expansion. We start with the case that the two SPPs have the same top level field, i.e., let $p_1 = \mathsf{SPP}(f, b_1, m_1, d_1), p_2 = \mathsf{SPP}(f, b_2, m_2, d_2)$. Note that we rely on Lemma 18 to reason as if plain SPP constructor is called instead of the smart constructor. Evidently $\beta \in [\![p_1 \mathbin{\hat{+}} p_2]\!](\alpha)$ iff β is in one of the children found in the branches, mutation, or default cases. Which case is applicable depends only on α_f. So we handle these three by case, letting $v = \alpha_f$:

   - Now we look at the test branch. In this case, $\beta \in [\![p_1 \mathbin{\hat{+}} p_2]\!](\alpha)$ iff there is a binding $v \mapsto p$ in $\{v \mapsto p_1(v) \mathbin{\vec{+}} p_2(v) \mid v \in \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)\}$. By Lemma 21[4] and Lemma 20, this is the same as saying $\beta \in [\![p_i]\!](\alpha[f \leftarrow v_i])$ for a binding $v_i \mapsto p_i \in p_1(v)$ or $\beta \in [\![p_j]\!](\alpha[f \leftarrow v_j])$, for a binding $v_j \mapsto p_j \in p_2(v)$. Combining these, this is equivalent to $\beta \in [\![p_1]\!](\alpha) \cup [\![p_2]\!](\alpha)$ as needed.

[4]The assumption about the values of m_1, m_2 is satisfied by the induction hypothesis.

- For the mutation branch, $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)$. In this case, $\beta \in [\![p_1 \hat{+} p_2]\!](\alpha)$ whenever there is a binding $(v' \mapsto p) \in (m_1 \vec{\hat{+}} m_2)$ such that $\beta \in [\![p]\!](\alpha)$. By Lemma 21, this is equivalent to saying that either $\beta \in [\![p_i]\!](\alpha)$ for some binding $(v' \mapsto p_i) \in m_1$, or that $\beta \in [\![p_j]\!](\alpha)$ for some $(v' \mapsto p_j) \in m_2$. But now we are done, because $\beta \in [\![p_i]\!](\alpha)$ or $\beta \in [\![p_j]\!](\alpha)$ iff $\beta[f \leftarrow v'] \in [\![p_1]\!](\alpha[f \leftarrow v]) \cup [\![p_j]\!](\alpha[f \leftarrow v])$.
- Now consider the default case, where again $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)$. By the induction hypothesis, $\beta \in [\![d_1 \hat{+} d_2]\!](\alpha)$ iff $\beta \in [\![d_1]\!](\alpha) \cup [\![d_2]\!](\alpha)$. Then since $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1)$ and $v \notin \mathsf{keys}(b_2) \cup \mathsf{keys}(m_2)$, this is true iff $\beta \in [\![p_1]\!] \cup [\![p_2]\!]$.

The other cases, namely that the top level field is different, reduce naturally to the above case by applying Lemma 17.

2. We need to show $\beta \in [\![p_1 \hat{\cdot} p_2]\!](\alpha)$ iff there is a $\gamma \in \mathsf{Pk}$ such that $\gamma \in [\![p_1]\!](\alpha)$ and $\beta \in [\![p_2]\!](\gamma)$. The base cases (where either p_1 or p_2 are $\top$ or $\bot$) are trivial. We proceed by cases, letting $v = \alpha_f$:

   - Test branches case. Thus $v \in \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)$, or $v \in b_2[v']$ for some $v' \in \mathsf{keys}(b_2)$. So there is a binding for v in (what in the definition is called) b'. Applying Lemma 21[5] inductively to this map, we get that $\beta \in [\![p \hat{\cdot} p']\!](\alpha)$ means there must be corresponding bindings $(v_2 \mapsto p) \in p_1(v)$ and $(v_3 \mapsto p') \in p_2(v_2)$. Letting $\gamma = \alpha[f \leftarrow v_2]$, this is equivalent to: $\alpha[f \leftarrow v_2] \in [\![p_1]\!]$ and $\beta \in [\![p_2]\!](\gamma)$, completing the case.
   - For the mutation branch case, $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)$ Then $\beta \in [\![p_1 \hat{\cdot} p_2]\!](\alpha)$ is equivalent to $\beta = \alpha[f \leftarrow v]$ for some binding for $v \in m_A \vec{\hat{+}} m_B$. We apply Lemma 21, and divide into two subcases. Namely $\beta \in [\![p_1 \hat{\cdot} p_2]\!](\alpha)$ iff we come from:

[5] The assumption of the lemma is satisfied by part (1).

(m_A) We apply Lemma 21 again, inductively, to the folded sum in m_A, finding that there are bindings $(v_2 \mapsto p) \in m_1$ and $(v_3 \mapsto p') \in p_2(v_2)$ such that $\beta \in [\![p \mathbin{\hat{\cdot}} p']\!](\alpha)$. Applying the induction hypothesis, there are α, γ, β such that $\gamma \in [\![p]\!](\alpha)$ and $\beta \in [\![p']\!](\gamma)$. This is equivalent to saying $\gamma[f \leftarrow v_2] \in [\![p_1]\!](\alpha)[f \leftarrow v]$ and $\beta[f \leftarrow v_3] \in [\![p_2]\!](\gamma)$ as needed.

(m_B) Then $\beta \in [\![d_1 \mathbin{\hat{\cdot}} p]\!](\alpha)$ for some binding $(v' \mapsto p) \in m_2$. We apply the induction hypothesis to see that this means there is a $\gamma \in [\![d_1]\!](\alpha)$ for which $\beta \in [\![p]\!](\gamma)$. In turn this is equivalent to: $\gamma[f \leftarrow v] \in [\![p_1]\!](\alpha[f \leftarrow v])$ and $\beta[f \leftarrow v'] \in [\![p_2]\!](\gamma[f \leftarrow v])$ as needed.

- We are in the default case when $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(b_2) \cup \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2) \cup \mathsf{keys}(m_A \mathbin{\vec{+}} m_B)$. By the induction hypothesis, $\beta \in [\![d_1 \mathbin{\hat{\cdot}} d_2]\!](\alpha)$ iff there is a γ such that $\gamma \in [\![d_1]\!](\alpha)$ and $\beta \in [\![d_2]\!](\gamma)$, but from $v \notin \mathsf{keys}(b_1) \cup \mathsf{keys}(m_1)$, this is the same as $\gamma \in [\![p_1]\!](\alpha)$ and from $v \notin \mathsf{keys}(b_2) \cup \mathsf{keys}(m_2)$, this is $\beta \in [\![p_2]\!](\gamma)$, completing the proof.

□

Now we justify the definitions of fwd and bwd.

Theorem 23. *For SPPs p_1, p_2, all of the following hold.*

1. $[\![\mathsf{fwd}\ s]\!] = \{\beta \in \mathsf{Pk} \mid \exists \alpha \in \mathsf{Pk}.\ \beta \in [\![s]\!](\alpha)\}$.
2. $[\![\mathsf{bwd}\ s]\!] = \{\beta \in \mathsf{Pk} \mid \exists \alpha \in \mathsf{Pk}.\ \alpha \in [\![s]\!](\beta)\}$.

Proof. We prove each case separately:

1. The base cases, $s = \top$ and $s = \bot$, are trivial. So let $s = \mathsf{SPP}(f, b, m, d)$. Let $\beta \in \mathsf{Pk}$, let $v' = \alpha_f$, and let $B = \bigcup_{v \in \mathsf{keys}(b)} \mathsf{keys}(b[v])$. We will show that $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff there is a packet $\alpha \in \mathsf{Pk}$ such that $\beta \in [\![s]\!](\alpha)$. For each such packet α, we will reserve v to refer to α_f. There are eight cases for the behavior of fwd depending on whether v' is in each of the three sets B, $\mathsf{keys}(b)$, and $\mathsf{keys}(m)$. Note for reference that $\mathsf{keys}(b_B) = B$, and $\mathsf{keys}(b_A) = \mathsf{keys}(m)$.

   (a) Suppose $v' \in B \setminus (\mathsf{keys}(b) \cup \mathsf{keys}(m))$. By applying Lemma 21 to the definition of fwd, we see that $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff $v' \in \mathsf{keys}(b_B)$ corresponding to some binding $v' \mapsto \mathsf{fwd}(p') \in b_B$ for which $\beta \in [\![p']\!]$, or we have a binding $v' \mapsto \mathsf{fwd}(d) \in b_C$. In the first case, $v' \mapsto \mathsf{fwd}(p') \in b_B$ iff there is a binding $v' \mapsto p' \in b[v]$ for some v. Applying the induction hypothesis, this is true iff $\beta \in [\![p']\!]$ for packet $\alpha = \beta[f \leftarrow v]$, and $\beta \in [\![s]\!](\alpha)$ as needed. In the second case, our binding is $v' \mapsto \mathsf{fwd}(d) \in b_C$. By the induction hypothesis, $\beta \in [\![\mathsf{fwd}(d)]\!]$. Because we know that $v' \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$, this also means that for $\alpha = \beta$ we have $\beta \in [\![s]\!](\alpha)$ as needed.

   (b) Suppose $v' \in (B \cap \mathsf{keys}(m)) \setminus \mathsf{keys}(b)$. Applying Lemma 21, we have $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff $v' \in \mathsf{keys}(b_A)$ or $v' \in \mathsf{keys}(b_B)$. In the first case, there is a binding $v' \mapsto \mathsf{fwd}(p') \in m$. By induction hypothesis, that means we can choose $\alpha = \beta$ and have both $\beta \in [\![p']\!]$ and $\beta \in [\![s]\!](\alpha)$. In the other case, then the binding in b_B works just as in the previous case.

   (c) Suppose $v' \in \mathsf{keys}(m) \setminus (B \cup \mathsf{keys}(b))$. In this case, $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff there is an applicable binding in b_A. The argument matches the argument for b_A in the previous case.

   (d) Suppose $v' \in (B \cap \mathsf{keys}(b)) \setminus \mathsf{keys}(m)$. Then $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff there is an applicable binding in b_B. The argument matches (a) and (b).

(e) Suppose $v' \in B \cap \mathsf{keys}(b) \cap \mathsf{keys}(m)$. This case is effectively the same as case (b).

(f) Suppose $v' \in (\mathsf{keys}(b) \cup \mathsf{keys}(m)) \setminus B$. This case is effectively the same as case (c).

(g) Suppose $v' \in \mathsf{keys}(b) \setminus (B \cup \mathsf{keys}(m))$. Observe that it cannot be the case that $\beta \in [\![\mathsf{fwd}\ s]\!]$, since the only bindings available are $v' \mapsto \bot \in b_D$. But we cannot have $\beta \in [\![s]\!](\alpha)$ either, since any packet α which matches a key in b or m will have α_f updated to a value in B or m, but $v' \notin B \cup m$. In addition, a α with $\alpha_f \notin b \cup m$ cannot produce β either, a since in that case no update occurs (so $v' = v$), but we know $v' \in b$.

(h) Suppose $v' \notin \mathsf{keys}(b) \cup \mathsf{keys}(m) \cup B$. Clearly $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff $\beta \in [\![\mathsf{fwd}\ d]\!]$. By the induction hypothesis, that means $\beta \in [\![d]\!](\alpha)$ for some α. In turn this means $\beta \in [\![s]\!](\alpha)$ as needed.

2. The base cases are trivial. Let $s = \mathsf{SPP}(f, b, m, d)$. As in the proof above for fwd, we let $\beta \in \mathsf{Pk}$, let $v' = \alpha_f$, and let $B = \bigcup_{v \in \mathsf{keys}(b)} \mathsf{keys}(b[v])$. We will show that $\beta \in [\![\mathsf{fwd}\ s]\!]$ iff there is a packet $\alpha \in \mathsf{Pk}$ such that $\beta \in [\![s]\!](\alpha)$. For each such packet α, we will reserve v to refer to α_f. Note for reference that $\mathsf{keys}(b_A) = B$ and $\mathsf{keys}(b_B) = \mathsf{keys}(m) \setminus \mathsf{keys}(b)$.

(a) Suppose $v' \in B \setminus (\mathsf{keys}(b) \cup \mathsf{keys}(m))$. Then $\beta \in [\![\mathsf{bwd}\ p\ s]\!]$ iff there is a binding $v' \mapsto m' \in b$ for which there is a binding $v' \mapsto \hat{\sum}_{w' \mapsto p' \in m'} \mathsf{bwd}(p') \in b_A$. By the induction hypothesis (applied inductively to the sum), this is equivalent to there being a packet $\gamma = \beta[f \leftarrow w']$ such that there is an output packet $\alpha \in [\![p']\!](\gamma)$. Now remember that p' is a child of s, so this means that $\alpha \in [\![s]\!](\beta)$ as we wanted.

(b) Suppose $v' \in (B \cap \mathsf{keys}(m)) \setminus \mathsf{keys}(b)$. Then $\beta \in [\![\mathsf{bwd}\ p\ s]\!]$ iff there

is a binding in b_A, as in the previous case, or there is a binding in $v \mapsto d' \in b_B$. This in turn is true iff there is a binding $v'' \mapsto p'$ such that $\beta \in [\![\mathsf{bwd}\ p']\!]$. Indeed, from the fact that $v' \notin \mathsf{keys}(b)$, and by the induction hypothesis, this means that the child p' is reachable for β and moreover produces $\beta[f \leftarrow v''] \in [\![s]\!](\beta)$.

(c) Suppose $v' \in \mathsf{keys}(m) \setminus (B \cup \mathsf{keys}(b))$. Then $\beta \in [\![\mathsf{bwd}\ p\ s]\!]$ iff there is a binding in b_B, and the argument follows the previous case.

(d) Suppose $v' \in (B \cap \mathsf{keys}(b))$. Then $\beta \in [\![\mathsf{bwd}\ p\ s]\!]$ iff there is a binding in b_A, and the argument follows the argument in (a).

(e) Suppose $v' \in (\mathsf{keys}(b) \cup \mathsf{keys}(m)) \setminus B$. In this case v' cannot be in $b_A \vec{+} b_B$, so $\beta \in [\![\mathsf{bwd}\ p\ s]\!]$ iff $\beta \in [\![d' \hat{+} \mathsf{bwd}(d)]\!]$. By the induction hypothesis, this is equivalent to either $\beta \in [\![d']\!]$ or $[\![d]\!](\alpha)$ for some α. In the later case, $\beta \in [\![s]\!](\alpha)$ and we are done. In the former case, there is a binding $v'' \mapsto p' \in d'$ for which $\beta[f \leftarrow v''] \in [\![s]\!](\beta)$, completing the case.

(f) Suppose $v' \notin \mathsf{keys}(m) \cup B$. Then $\beta \in [\![\mathsf{bwd}\ p\ s]\!]$ iff $\beta \in [\![d' \hat{+} \mathsf{bwd}(d)]\!]$, and the argument follows the previous case.

□

We conclude by demonstrating the correctness of push and pull.

Corollary 24. *For* push *and* pull*, the following hold:*

1. $\beta \in [\![\mathsf{push}\ p\ s]\!] \iff \exists \alpha \in [\![p]\!].\ \beta \in [\![s]\!](\alpha)$.
2. $\beta \in [\![\mathsf{pull}\ s\ p]\!] \iff \exists \alpha \in [\![p]\!].\ \alpha \in [\![s]\!](\beta)$.

Proof. Each part follows from the respective definitions, and applying Theorem 23, Lemma 19, and Theorem 22. □

CHAPTER 4

ACTIVE LEARNING OF SYMBOLIC NETKAT AUTOMATA

This chapter is adapted from the following published conference paper:
Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. Active learning of symbolic NetKAT automata. In *Proceedings of the 46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, June 2025. Association for Computing Machinery.

4.1 Introduction

As we have seen already, model-checking is one of the success stories in formal methods, with broad applications in hardware and software. However, model checking requires having accurate models of the systems being verified. These models are usually produced by human experts via a slow and error-prone process.

Active learning is an appealing alternative that can construct an accurate model of a system automatically. Model learning is often used with closed-box systems—i.e., when the source code or inner workings of a system cannot be inspected, but its external behaviors can be observed—and it generates formal models, such as finite automata, that can be used to obtain assurance through testing or verification. In particular, Angluin's $\mathtt{L}^\star$ algorithm, which learns a Deterministic Finite Automaton (DFA) in the so-called Minimally Adequate Teacher (MAT) framework, has inspired nearly 40 years of fruitful research [70, 22, 67, 30, 119].

The $\mathtt{L}^\star$ approach is most applicable when only observations of behavior are available and it is reasonable to assume the system has a finite state model, or can be approximated by such a model. Computer networking is one such domain, which has both of these characteristics. We frequently have only partial information about the policies used to route packets through the network, but we do have the ability to observe packet-level behaviors using command-line tools like `ping` and `traceroute`. At the same time, the programs that execute on network routers are finite state—their behavior is kept intentionally simple to enable them to operate at line rate. Indeed, automata learning has been applied to learning implementations of network protocols [49, 45].

In this chapter, we want to go a step further and learn not just end-host pro-

tocols, but network-wide behaviors. This would be helpful in many scenarios: inferring a model of a single device where the program and configuration are unavailable, to verify that it conforms to a well-understood model; inferring the topology of the network and check that it conforms to the expected structure; or inferring a model an unknown peer network to verify that it is fulfilling a customer agreement.

Indeed, the networking community has recognized that there is great value in having accurate models of individual devices and the network as a whole. For example, Google employs a small team of verification engineers who develop and maintain models of their data center switches in a domain-specific language [3]. They use these models to find bugs (using fuzzers) and verify properties (using symbolic execution tools). However, maintaining the models turns out to be non-trivial—switch pipelines are moderately complex, and their behavior changes as vendors add new features. Using automata learning, rather than having to expend effort developing these models by hand, they could be generated automatically. Learned models also have the advantage of being *ground-truth* in the sense that they are based on actual observations of system behavior.

This chapter presents a framework for automated learning of NetKAT automata. To achieve this goal, we bring together two lines of research: active learning and algebraic approaches to network verification. We develop new automata learning algorithms for the expressive models used in NetKAT, a domain-specific programming language and logic based on Kleene Algebra with Tests [5]. NetKAT allows us to model different aspects of a network at different levels of detail and abstraction, applicable to all the scenarios described above. Moreover, as NetKAT was designed with an automata-theoretic foundation, it is a great fit for active

learning. The key challenges, however, are in designing specialized data structures and algorithms that take advantage of NetKAT's algebraic structure and other details of the networking domain, to scale up to real-world networks.

In particular, while NetKAT has an elegant theory based on using packets as actions in the automaton model, for learning network policies, it has an obvious drawback: the space of packets is exponential in the number of header bits—intractably large for all but toy policies. As we have seen in Chapter 3, symbolic NetKAT automata allowed us to overcome the packet space blow up in the verification problem. Crucially, these automata are symbolic in *both* the transition labels and the state space. Hence, these symbolic NetKAT automata are therefore the ideal target for exploring learning.

Symbolic NetKAT automata are reminiscent of Symbolic Finite Automata (SFAs) [39], a fundamental model introduced to deal with infinite or intractably large alphabets in automata. The key idea in SFAs is that rather than transitions being explicitly defined by the singular element of the alphabet they consume, one can have transitions be labeled by predicates that determine the subset of the alphabet that can take a specific transition. SFAs have enjoyed a range of useful applications [112], largely due to their simple theory and efficient decision procedures, and extensions of classic automata learning algorithms have been proposed for subclasses of symbolic automata [43, 11, 47]. Of particular relevance is recent work on learning a large subclass of SFAs [11].

At first glance, prior work on learning SFAs seems to provide the recipe for dealing with NetKAT automata: just compile the NetKAT automata down to DFAs, and use SFA techniques to deal with the large alphabet. Unfortunately, there are several key differences that make this idea a nonstarter: the semantics of

NetKAT automata allow transitions to be based on a notion of a *carry-on packet* (or current packet), which means that the naive translation of NetKAT automata to DFAs also triggers a blow-up of *state*. This is precisely the issue addressed in symbolic NetKAT automata in Chapter 3, and is the reason why we cannot re-use learning algorithms for SFAs with NetKAT.

This chapter develops active learning for NetKAT automata, making the following contributions:

- **An expanded theory of NetKAT automata**: Angluin-style learning algorithms crucially rely on the classic Myhill-Nerode theorem, which identifies the states of the minimal automaton with equivalence classes of languages. We present a new Myhill-Nerode theorem for NetKAT (Section 4.2) that enable us to characterize a *canonical* form for NetKAT automata.
- **A naive learning algorithm**: We give a "first attempt" solution for learning canonical NetKAT automata, and prove its correctness (Section 4.3). It is naive in that it does not use symbolic techniques, and it therefore has poor performance due to the large "alphabet" and state space.
- **Two algorithms for symbolic NetKAT learning**: To provide scalable learning for NetKAT, we also develop an algorithm for learning symbolic NetKAT automata. We do this in two stages. First, we focus on the subset of NetKAT that does not have the `dup` primitive (i.e., the `dup`-free fragment). We give an algorithm for learning symbolic representations of NetKAT programs in this restricted setting (Section 4.4). Next, using this first symbolic learner as a subroutine for symbolic learning of full-blown NetKAT automata and prove its correctness (Section 4.5).
- **A prototype OCaml implementation**: We have implemented our symbolic

learning algorithms as a prototype to investigate their behavior and properties. We conclude the chapter by describing our implementation and experience learning models for network policies from a standard benchmark suite (Section 4.6).

Omitted proofs can be found in Section 4.10.

4.2 Myhill-Nerode Theorem for NetKAT and Canonical Automata

The essential insight of the $\mathtt{L}^\star$ algorithm is that we can identify Myhill-Nerode equivalence classes for DFAs in an incremental fashion. However, to use this idea in the context of NetKAT, we need to develop a NetKAT-specific notion of Myhill-Nerode. This section develops such a notion, which we will use as a basis for a learning algorithm in Section 4.3. Our approach follows Kozen's more general treatment of Myhill-Nerode for automata on guarded strings [75].

For a language $\mathcal{L}$ over an alphabet Σ, the Myhill-Nerode [95] relation for $\mathcal{L}$ is an equivalence relation on $\Sigma^\star$, written $(\equiv_{\mathcal{L}})$ and defined by:

$$s \equiv_{\mathcal{L}} t \overset{\triangle}{\iff} \forall e \in \Sigma^\star.(se \in \mathcal{L} \iff te \in \mathcal{L})$$

The Myhill-Nerode theorem states that $\mathcal{L}$ is regular iff $\equiv_{\mathcal{L}}$ has finite index. Moreover, the equivalence classes of $\equiv_{\mathcal{L}}$ correspond precisely to the states of the *unique minimal* DFA for $\mathcal{L}$. We would like an analogous characterization for NetKAT, but there are two key challenges: concatenation of guarded strings is not always defined (eq. (2.1)) and NetKAT's semantics process strings by reading two packets at a time–one of which continues as *carry-on* packet for the next step (eq. (2.2)). These differences result in a subtly different characterization for guarded strings.

We first define the set of guarded string prefixes $\mathsf{Pre} = \mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star$ and the set of guarded string suffixes $\mathsf{Suf} = (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$. We also define a helper function $\mathsf{last} \colon \mathsf{Pre} \to \mathsf{Pk}$ and a (partial) concatenation operation $\blacklozenge \colon \mathsf{Pre} \times \mathsf{GS} \to \mathsf{GS}$ for prefixes with guarded strings by:

$$\begin{aligned} \mathsf{last}(\alpha) &= \alpha \\ \mathsf{last}(w \cdot \alpha \cdot \mathsf{dup}) &= \alpha \end{aligned} \qquad\qquad p \blacklozenge \alpha \cdot s = \begin{cases} p \cdot s & \mathsf{last}(p) = \alpha \\ \text{undefined} & \text{otherwise} \end{cases}$$

For a language $\mathcal{L} \subseteq \mathsf{GS}$, we define the relation $\equiv_{\mathcal{L}}$ for $s, t \in \mathsf{Pre}$ by:

$$s \equiv_{\mathcal{L}} t \overset{\triangle}{\iff} (\forall e \in \mathsf{GS}.\ s \blacklozenge e \in \mathcal{L} \iff t \blacklozenge e \in \mathcal{L}) \tag{4.1}$$

The relation $\equiv_{\mathcal{L}}$ is an equivalence on guarded string prefixes.

Lemma 25. *The relation $\equiv_{\mathcal{L}}$ in Equation* (4.1) *is reflexive, symmetric, and transitive.*

The equivalence classes of the classic Myhill-Nerode relation for $\mathcal{L} \subseteq \Sigma^\star$ correspond exactly to the states of the minimal DFA accepting $\mathcal{L}$. However, this result does not immediately transfer to NetKAT. We need to define a simpler form of automata that hides the carry-on packet in the state—this will enable us to precisely capture the fact that the equivalence classes of $\equiv_{\mathcal{L}}$ can only relate prefixes s and t when $\mathsf{last}(s) = \mathsf{last}(t)$.[1]

Definition 26. *A* Packet-state *NetKAT automaton (PNKA) is a four-tuple $(Q, q_0, \partial, \lambda)$, where Q is a finite set of states, $q_0 : \mathsf{Pk} \to Q$ is the start state, $\partial \colon Q \to \mathsf{Pk} \to Q$ is the transition function, and $\lambda \colon Q \to \mathsf{Pk} \to 2$ is the observation function.*

[1]Note that we interpret ($\iff$) to hold when both sides are undefined or both defined and equal, but it does *not* hold when only one side is undefined (regardless of the other side).

The functions ∂ and λ must satisfy the spelling *property: for any states $q, q' \in Q$ and packets $\alpha, \beta \in \mathsf{Pk}$, if $\partial_\alpha(q) = \partial_\beta(q')$, $q_0(\alpha) = q_0(\beta)$, or $q_0(\alpha) = \partial_\beta(q)$, then $\alpha = \beta$.*

The spelling property ensures we cannot arrive at the same state via two different packets. Formally, there is a function $\mathsf{spell}\colon Q \to \mathsf{Pk}$ that outputs the packet of all transitions into each state.

A PNKA $\mathcal{M} = (Q, q_0, \partial, \lambda)$ accepts a language $\mathcal{L}(\mathcal{M}) \subseteq \mathsf{GS}$ consisting of all strings $\alpha \cdot w \in \mathsf{GS}$ satisfying $\mathcal{M}(q_0(\alpha), w) = \top$, where:

$$\mathcal{M}(q, \beta) = \lambda_\beta(q) \qquad \mathcal{M}(q, \beta \cdot \mathsf{dup} \cdot w') = \mathcal{M}(\partial_\beta(q), w')$$

As the name suggests, the distinguishing feature of a PNKA is that the transition function and observation function operate on a single packet: the other packet is "hidden in the state." Although PNKA satisfy additional restrictions compared to standard NetKAT automata, it is important to note that they recognize exactly the same sets of guarded strings.

Theorem 27. *Let $\mathcal{L} \subseteq \mathsf{GS}$. Then $\mathcal{L}$ is accepted by a NKA if and only if it is accepted by a PNKA.*

Now that we have established that PNKA are a suitable representation of NKA we are ready to state the Myhill-Nerode theorem for $NetKAT$. This theorem ensures PNKA have properties that are desirable for automata learning.

Definition 28. *Given a language $\mathcal{L}$ we define $\mathcal{P}_{\equiv_\mathcal{L}} = (Q, q_0, \partial, \lambda)$ by:*

$$Q = \{[s]_{\equiv_\mathcal{L}} \mid s \in \mathsf{Pre}\} \qquad q_0(\alpha) = [\alpha]_{\equiv_\mathcal{L}}$$

$$\partial_\alpha([s]_{\equiv_\mathcal{L}}) = [s \cdot \alpha \cdot \mathsf{dup}]_{\equiv_\mathcal{L}} \qquad \lambda_\alpha([s]_{\equiv_\mathcal{L}}) = s \cdot \alpha \in \mathcal{L}$$

Theorem 29 (Myhill-Nerode for $NetKAT$)**.** *For a given language* $\mathcal{L} \subseteq \mathsf{GS}$*:*

1. $\mathcal{L}$ *is regular if and only if* $\equiv_{\mathcal{L}}$ *has finite index.*
2. *For regular* $\mathcal{L}$*, a minimal* PNKA *accepting* $\mathcal{L}$ *has* $\mid \equiv_{\mathcal{L}} \mid$ *states.*
3. *Any* PNKA *accepting* $\mathcal{L}$ *with* $\mid \equiv_{\mathcal{L}} \mid$ *states is isomorphic to* $\mathcal{P}_{\equiv_{\mathcal{L}}}$*.*

4.3 Learning Canonical NetKAT Automata

In this section, we develop a learning algorithm for *canonical* NetKAT automata (i.e., minimal PNKA). This algorithm is primarily of theoretical interest, as canonical automata will have a large number of states in general. However, the core ideas of this approach will guide us in designing a more practical algorithm in Section 4.5 for learning *symbolic* NetKAT automata.

Our algorithm resembles $\mathsf{L}^\star$, but we cannot use the same state for different "carry-on" packets in a PNKA. Accommodating this restriction requires some additional structure.

4.3.1 The PNKA Teacher

Just as in Angluin's MAT, the MAT framework for NetKAT automata assumes a teacher that can answer two types of queries: membership queries (i.e., whether a given guarded string is part of the target language or not) and equivalence queries (i.e., whether a given hypothesis automaton describes the target language, and returning a counter-example if not).

Definition 30 (MAT Interface)**.** *The MAT interface for Packet-state NetKAT automata is as follows:*

$$\mathsf{mem}_{\mathsf{PNKA}} : \mathsf{GS} \to \mathbb{2} \qquad \mathsf{equiv}_{\mathsf{PNKA}} : \mathit{PNKA} \to 1 + \mathsf{GS}$$

The teacher knows the target language, $\mathcal{L} \subseteq \mathsf{GS}$. We have $\mathsf{mem}_{\mathsf{PNKA}}(w) = \top$ iff $w \in \mathcal{L}$, and $\mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H}) = \top$ if $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. Otherwise, $\mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H}) = w \in \mathcal{L} \oplus \mathcal{L}(\mathcal{H})$. In the rest of this section we develop a naive algorithm for learning a canonical PNKA from this type of teacher.

4.3.2 The Observation Table

The central abstraction used in our learning algorithm is an *observation table*. This data structure keeps track of the knowledge the learner has acquired, and it provides the basis for constructing hypothesis automata. The shape of the table is similar to the classic observation table used in $\mathtt{L}^\star$, but we need to adjust it to the specific nature of NetKAT automata. In particular, we organize the observation table as a collection of smaller tables, one for each packet in Pk.

Definition 31 (Packet Table)**.** *For a packet $\alpha \in \mathsf{Pk}$, a packet table is a triple $(S_\alpha, E_\alpha, T_\alpha)$, where:*

- $S_\alpha \subseteq \{p \mid \mathsf{last}(p) = \alpha, p \in \mathsf{Pre}\}$ *is a set of* access prefixes.
- $E_\alpha \subseteq \mathsf{Suf}$ *is a set of* distinguishing suffixes.
- $T_\alpha \colon (S_\alpha \cup S'_\alpha) \times E_\alpha \to \mathbb{2}$ *is such that* $T_\alpha(s,e) = \top$ *if* $s \cdot e \in \mathcal{L}$*, and* $T_\alpha(s,e) = \bot$ *otherwise.*

We maintain $S'_\alpha \triangleq \bigcup_{\beta \in \mathsf{Pk}} S_\beta \cdot (\alpha \cdot \mathsf{dup})$ *as the set of* $(\alpha \cdot \mathsf{dup})$ extensions *of all packet table sets* S_β*.*

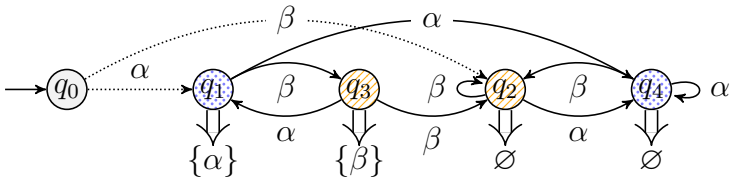
Definition 32 (Observation Table)**.** *An observation table* T *is an* α*-indexed set of Packet Tables,* $(S_\alpha, E_\alpha, T_\alpha)$*, consisting of one packet table per packet in* Pk*.*

Remark 33. *In* $L^\star$*, the extension part of the table is usually written* $S \cdot \Sigma$*, and computes the one letter extension of each string representing a state—i.e., the transitions of an automaton. We require a similar functionality in our case, albeit adjusted to the specific details of packet tables. In particular, for a packet table* $(S_\alpha, E_\alpha, T_\alpha)$*, the set* S'_α *is maintained to have the* $(\alpha \cdot \mathsf{dup})$ *extension of every state in every packet table. This allows* T_α *to hold information on every* α *transition between any pair of states.*

Access Prefixes Access prefixes describe states of the target automaton—the state corresponding to a given access prefix is the one the automaton would land on after processing that prefix. Prefixes comprising a single packet naturally represent the start state, while prefixes comprising two packets (and one dup) represent the states reached by reading a given packet from the start state, and so on.

Distinguishing Sequences Distinguishing sequences, on the other hand, elicit specific behavior from an arbitrary state. Note that because the prefixes come from Pre and distinguishing sequences from Suf, concatenation is always defined and forms a guarded string. Distinguishing sequences can also be understood as the suffixes e in the Myhill-Nerode relation of Section 4.2. In $\mathtt{L}^\star$, the set of distinguishing suffixes is maintained to be suffix-closed by the algorithm. A similar property holds here, but only for the observation table as a whole, not each individual packet table.

Example 34. *Consider the PNKA $\mathcal{M}$ below over the packet space* $\mathsf{Pk} = \{\{f \mapsto 1\}, \{f \mapsto 2\}\}$*, which we abbreviate to* α *and* β*, respectively. States from equivalence classes ending in* α *are colored orange and lined, while states from equivalence classes ending in* β *are colored blue and dotted. The dotted transitions from the start state denote its special type* $q_0 \colon \mathsf{Pk} \to Q$ *in the automaton.*



Note that there are infinitely many possible access prefixes and distinguishing sequences, due to the cyclic structure of the automaton. Some access prefixes for q_1 *are* α *and* $\alpha \cdot \beta \cdot \mathsf{dup} \cdot \alpha$*. An access prefix for* q_3 *is* $\alpha \cdot \beta \cdot \mathsf{dup}$*. Finally,* β *is a distinguishing sequence for* q_1 *and* q_3*, as* $\mathcal{M}(q_0(\alpha), \beta) = \bot$ *but* $\mathcal{M}(q_0(\alpha), \beta \cdot \mathsf{dup} \cdot \beta) = \top$*.*

Closed and Consistent Observation Tables In $\mathtt{L}^\star$, the main loop of the algorithm makes membership queries to construct a closed table, from which a hypothesis can be made. We define an analogous concept here, also in terms of the row function, just as in $\mathtt{L}^\star$. Namely, we define a primitive row function with type $\mathsf{row}_\alpha : S_\alpha \cup S'_\alpha \to E_\alpha \to \mathbb{2}$, defined by $\mathsf{row}_\alpha(s)(e) = T_\alpha(s, e)$. Clearly all we have done here is to curry T_α—but the currying is essential! We will partially apply row_α to a given string s and consider the resulting functions $E_\alpha \to \mathbb{2}$. In this way, we distinguish states by viewing their behavior in terms of the suffixes we have seen for each packet.

Definition 35 (Closed Table)**.** *An observation table* T *is* closed*, if for each* $\alpha \in \mathsf{Pk}$ *and for each* $s' \in S'_\alpha$*, then* $\mathsf{row}_\alpha(s') = \mathsf{row}_\alpha(s)$ *for some* $s \in S_\alpha$*.*

Definition 36 (Consistent Table)**.** *An observation table T is* consistent*, if for each $\alpha, \beta \in \mathsf{Pk}$ and for each $s, s' \in S_\alpha$, then $\mathit{row}_\alpha(s) = \mathit{row}_\alpha(s') \Rightarrow \mathit{row}_\beta(s \cdot \beta \cdot \mathsf{dup}) = \mathit{row}_\beta(s' \cdot \beta \cdot \mathsf{dup})$.*

The closedness property can be checked locally to each packet table (provided that all the required extensions to each prefix by each packet have all already been entered into T), while the consistency property must be checked globally, since successors to rows in one table may be in another table.

We use the observation table in the same way as in $\mathtt{L}^\star$: by making membership queries until the table is closed and consistent, ensuring that the construction of the next PNKA to conjecture is well-defined. In Figure 4.2, we show how to make the table closed and consistent. The function extend performs membership queries as needed so that every T_α is defined for its required domain.

4.3.3 The PNKA Learning Algorithm

Our NKA learning algorithm works by iteratively expanding an observation table, which is then the basis from which a hypothesis is constructed. If the hypothesis is correct then the learner terminates. Otherwise, a counterexample from the equivalence oracle is used to refine the tables. The main routine of the algorithm (Figure 4.1) follows the classic learning loop of MAT-based learners.

$T \leftarrow \{\alpha \mapsto (S_\alpha = \{\alpha\}, E_\alpha = \mathsf{Pk}, T_\alpha = \{\}) \mid \alpha \in \mathsf{Pk}\}$
while $\top$ **do**
  while T not closed and consistent **do**
    $T \leftarrow \mathsf{mkConsistent}(\mathsf{mkClosed}(T))$
  $\mathcal{H} \leftarrow \mathsf{hyp}_{\mathsf{PNKA}}(T)$
  $c \leftarrow \mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H})$
  if $c = \top$ **then**
    return $\mathcal{H}$
  else
    $T \leftarrow \mathsf{update}(T, c)$

Input: Table T, counterex. $c \in \mathsf{GS}$
Output: T updated in place with c.

for $s \in \mathsf{Pre}$, $e \in \mathsf{Suf}$ such that $c = s \cdot e$ **do**
  $S_{\mathsf{last}(s)} \leftarrow S_{\mathsf{last}(s)} \cup \{s\}$
return $\mathsf{extend}(T)$

Figure 4.1: Algorithms for learning canonical automata (above) and subroutine for update (below).

if T_α is not closed **then**
  let $s' \in S'_\alpha$ such that
    $\mathsf{row}_\alpha(s') \notin \{\mathsf{row}_\alpha(s) \mid s \in S_\alpha\}$.
  $S_\alpha \leftarrow S_\alpha \cup \{s'\}$
  return $\mathsf{extend}(T)$

---

if T is not consistent **then**
  let $e \in E_\beta$ such that for some $s, s' \in S_\alpha$, $\mathsf{row}_\alpha(s) = \mathsf{row}_\alpha(s')$,
    but $T_\beta(s \cdot \beta \cdot \mathsf{dup}, e) \neq T_\beta(s' \cdot \beta \cdot \mathsf{dup}, e)$.
  $E_\alpha \leftarrow E_\alpha \cup \{\beta \cdot \mathsf{dup} \cdot e\}$
  return $\mathsf{extend}(T)$

Figure 4.2: Subroutines used in PNKA learning (Figure 4.1): $\mathsf{mkClosed}\colon T \to T$ (above) and $\mathsf{mkConsistent}\colon T \to T$ (below).

For the update operation, we first break the counterexample string at each possible location. Each break results in a prefix ending in some packet α, which we add to S_α.

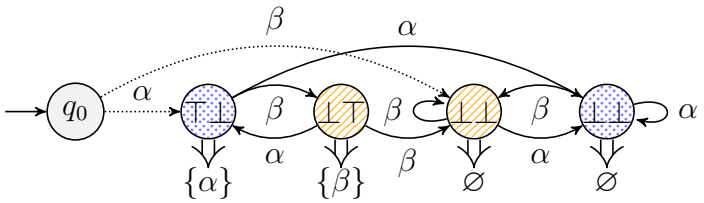
Finally, constructing the hypothesis once the table is closed and consistent is again similar to $\mathtt{L}^\star$, except that we still need to keep the state space associated with each table separate. To make this clear in the formalism, states themselves are a pair of type $\mathsf{Pk} \times (E_\alpha \to 2)$: a packet α, and a row vector of T_α. Given an observation table T, we construct $\mathsf{hyp}_{\mathsf{PNKA}}(T) \triangleq (Q, q_0, \delta, \varepsilon)$, where:

$$Q = \bigcup \{(\alpha, \mathsf{row}_\alpha(s)) \mid s \in S_\alpha, \alpha \in \mathsf{Pk}\} \quad \delta(\alpha, \mathsf{row}_\alpha(s))(\beta) = (\beta, \mathsf{row}_\beta(s \cdot \beta \cdot \mathsf{dup}))$$

$$q_0(\alpha) = (\alpha, \mathsf{row}_\alpha(\alpha)) \qquad \varepsilon(\alpha, \mathsf{row}_\alpha(s))(\beta) = T_\alpha(s, \beta)$$

Example 37. *Were we to select the PNKA of Theorem 34 as the target, we would have the following final packet tables and hypothesis:*

α	α	β
α	$\top$	$\bot$
$\alpha\alpha$	$\bot$	$\bot$
$\alpha\alpha$	$\bot$	$\bot$
$\alpha\alpha\alpha$	$\bot$	$\bot$
$\beta\alpha$	$\bot$	$\bot$
$\alpha\beta\alpha$	$\top$	$\bot$



β	α	β
β	$\bot$	$\bot$
$\alpha\beta$	$\bot$	$\top$
$\alpha\beta$	$\bot$	$\top$
$\alpha\alpha\beta$	$\bot$	$\bot$
$\beta\beta$	$\bot$	$\bot$
$\alpha\beta\beta$	$\bot$	$\bot$

4.3.4 Correctness of the Canonical Learner

Next, we prove that the canonical automaton learner is correct. We start by showing that the learner produces valid conjectures. For succinctness, we will assume the observation tables used to build hypotheses are closed and consistent, as this follows from the definition of the learning algorithm.

Lemma 38. *The hypothesis automaton, $\mathsf{hyp}_{\mathsf{PNKA}}(T)$, is well defined for any Observation Table T.*

The next lemma says that the learner only conjectures minimal (i.e., canonical) automata.

Lemma 39. *For any hypothesis PNKA $\mathcal{H} = (Q, q_0, \partial, \lambda)$ produced by the learner, and for any two prefixes $s, t \in \mathsf{Pre}$, we have that $s \equiv_{\mathcal{H}} t$ iff $s \equiv_{\mathcal{L}(\mathcal{H})} t$.*

The next lemma says that the Myhill-Nerode relation for the target language always refines the access relation for the learner's conjecture. In other words, the learner can only be wrong by failing to distinguish two states that need to be different.

Lemma 40. *Suppose the teacher holds $\mathcal{L}$. Then for each hypothesis PNKA $\mathcal{H}$ of the learner, we have that $(\equiv_{\mathcal{L}})$ refines $(\equiv_{\mathcal{L}(\mathcal{H})})$ in the sense that for prefixes $s, t \in \mathsf{Pre}$, then $s \equiv_{\mathcal{L}} t$ implies $s \equiv_{\mathcal{L}(\mathcal{H})} t$.*

Additionally, hypotheses remain faithful to the data in the observation table they originate from.

Theorem 41. *Given a closed, consistent table T, $\mathcal{H} \triangleq \mathsf{hyp}_{\mathsf{PNKA}}(T)$ agrees on every cell of the packet tables. That is, $T_\gamma(\alpha \cdot s, e) = \mathcal{H}(q_0(\alpha), s \cdot e)$ for any $\alpha \cdot s \in S_\gamma \cup S'_\gamma$, $e \in E_\gamma$, and $\gamma \in \mathsf{Pk}$.*

Finally, each counterexample results in a conjecture $\mathcal{H}$ with at least one more state:

Lemma 42. *Fix the target language $\mathcal{L}$, and suppose a learner conjectures a PNKA $\mathcal{H}$, and we get a counterexample $c = \mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H})$, i.e. $c \in \mathcal{L} \oplus \mathcal{L}(\mathcal{H})$. Then the next conjecture by the learner, $\mathcal{H}'$, will have at least one additional state compared to $\mathcal{H}$.*

Theorem 43. *The algorithm in Figure 4.1 terminates with the correct automaton.*

Proof. It follows that if the learner terminates then the automaton is correct, as we only terminate upon a successful equivalence query. Because the Myhill-Nerode

relation ($\equiv_{\mathcal{L}}$) for the language $\mathcal{L}$ refines those of the hypotheses (Theorem 71), then once we conjecture a hypothesis H for which $\mid \equiv_{\mathcal{L}} \mid = \mid \equiv_{\mathcal{H}} \mid$, we know the automaton must be correct by Theorem 29. By Theorem 75, we make progress toward this bound with every counterexample, completing the proof. □

4.4 Learning Symbolic Packet Programs (SPPs)

Before we present our algorithm for learning symbolic NetKAT automata, we first develop the core techniques needed to handle a restricted case: dup-free NetKAT programs. We then use the learner we develop in this restricted setting as the central component of a learner that handles full NetKAT. In addition to its role in the full automaton learner, however, this algorithm can be used directly whenever input-output packet pairs are enough to capture the relevant behavior of a system, such as a single closed-box device, or the single-step transfer function of a network.

4.4.1 Background

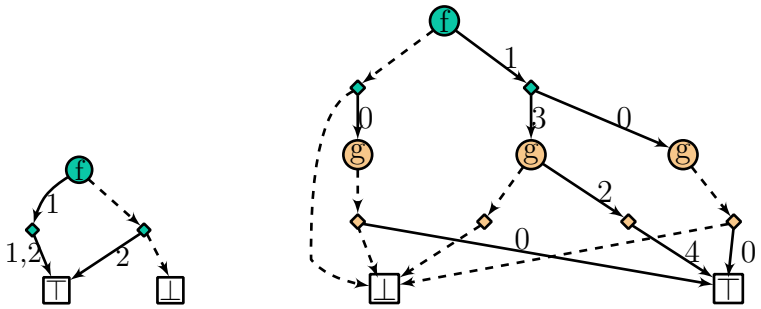
Chapter 3 introduced a set of techniques for reasoning about NetKAT *symbolically*. We briefly recall the Symbolic Packet Program (SPP):

$$p \in \mathsf{SPP} ::= \bot \mid \top \mid \mathsf{SPP}(f, \{\ldots, v_i \mapsto \{\ldots, w_{ij} \mapsto q_{ij}, \ldots\}, \ldots\}, \{\ldots, w_i \mapsto q_i, \ldots\}, q)$$

SPPs have two base cases, $\top$ and $\bot$, which represent the corresponding NetKAT expressions. The third case of an SPP represents testing a field f of an input packet against a series of values $v_0, \ldots, v_i, \ldots, v_n$, with a default case $f \neq v_0 \cdots f \neq v_n$. However, instead of continuing recursively after the test, SPPs nondeterministically assign a value w_{ij} to the field f of the output packet, and continue recursively with

the corresponding child q_{ij}. This way, SPPs can output more than one packet for a given input packet, and can also output packets with different values for the same field. Semantically this third case of an SPP has the semantics of the NetKAT expression: $\sum_i (f = v_i \cdot \sum_j (f \leftarrow w_{ij} \cdot q_{ij})) + (\prod_i f \neq v_i) \cdot (\sum_i f \leftarrow w_i + (\prod_i f \neq w_i) \cdot q)$, where f is a field, v_i, w_{ij}, w_i are values, and q_{ij}, q_j, q are SPPs.

Example 44. *On the left, we draw the SPP corresponding to the NetKAT expression $f = 1 + f \leftarrow 2$, and on the right the one for $(f \leftarrow 0 \cdot g \leftarrow 0) + (f = 1 \cdot g = 2 \cdot f \leftarrow 3 \cdot g \leftarrow 4)$:*



In this chapter, we write the semantics of SPPs as a function $[\![\cdot]\!] : \mathsf{SPP} \to \mathsf{Pk} \times \mathsf{Pk} \to \mathbb{2}$. This is consistent with the semantics given in Table 2.2, since the fragment of $\mathsf{Pk} \cdot (\mathsf{Pk} \cdot \mathsf{dup})^\star \cdot \mathsf{Pk}$ that is reachable without dup is precisely $\mathsf{Pk} \cdot \mathsf{Pk}$. We represent SPPs as directed acyclic graphs, similarly to BDDs. Vertices are labeled with the test field. Solid arrows are labeled with a number encoding the test value, and dashed arrows represent a default case. The sinks of the graph are labeled with $\top$ or $\bot$.

By removing extraneous branches and standardizing between some equivalent forms, SPPs can be made canonical in the sense that for any subset of $\mathsf{Pk} \cdot \mathsf{Pk}$ there is a unique normal-form SPP. In addition, SPPs are closed under all of the NetKAT operations (in Table 2.2), as well as difference, symmetric difference, and

intersection.

4.4.2 The SPP Teacher

We first develop a MAT-based framework (as in Section 4.3) for learning SPPs. The SPP teacher holds a set of guarded strings represented by a dup-free NetKAT expression (i.e., guarded strings of length two, or, pairs of packets). As before, the teacher answers membership and equivalence queries:

$$\mathsf{mem}_{\mathsf{SPP}} : \mathsf{Pk} \times \mathsf{Pk} \to 2 \qquad \mathsf{equiv}_{\mathsf{SPP}} : \mathsf{SPP} \to 1 + \mathsf{Pk} \times \mathsf{Pk}$$

For a membership query, given a pair of packets, the teacher returns $\top$ if the pair is in its language, and $\bot$ otherwise. For an equivalence query, given a SPP h the teacher returns $\top$ if h has the semantics of the target language, otherwise, the teacher returns a pair of packets in the symmetric difference between $[\![h]\!]$ and the target, i.e., a counterexample to the correctness of h.

If the teacher has an SPP for the target program, the implementation of both queries is straightforward: for a membership query we check that the given packet pair is in the semantics of the target (i.e., is a path from root to $\top$), and for the equivalence query, we take the symmetric difference between the target and the hypothesis. If the resulting SPP is $\bot$ we are done; otherwise we choose a counterexample packet pair by choosing a path to $\top$ in the resulting SPP.

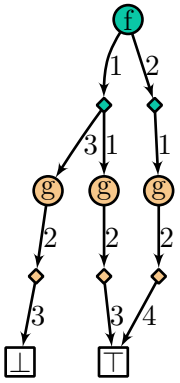
4.4.3 Evidence Packet Programs

The learner we present below proceeds in a similar fashion to $\mathtt{L}^\star$ [7] or its variants (e.g., KV [70]). The learner maintains a data structure with all of the evidence

received from the teacher, and it build successive queries and conjectures guided by this data structure. For learning SPPs, the data structure is called an Evidence Packet Program (EPP):

$$e \in \mathsf{EPP} ::= \top \mid \bot \mid F \times (V \Rightarrow (V \Rightarrow \mathsf{EPP}))$$

As with SPPs, we require the fields to appear in a fixed order as we descend the tree structure of an EPP. Unlike SPPs, we will maintain that every field is present (i.e. the $\top$ and $\bot$ leaves all have the same depth). The other distinction from SPPs is that every edge label in an EPP has a concrete value; i.e., there are no default cases. The EPP is essentially a trie containing example packet pairs, where the values are paired by field, as in SPPs.



An example EPP containing the packet pair $(\{f \mapsto 1, g \mapsto 2\}, \{f \mapsto 3, g \mapsto 3\})$, labeled $\bot$, and the pairs $(\{f \mapsto 1, g \mapsto 2\}, \{f \mapsto 1, g \mapsto 3\})$ and $(\{f \mapsto 2, g \mapsto 2\}, \{f \mapsto 1, g \mapsto 4\})$, labeled $\top$, is shown on the right.

We define the semantics of an EPP as $[\![\cdot]\!] \colon \mathsf{EPP} \to \mathsf{Pk} \times \mathsf{Pk} \rightharpoonup 2$, which, given a packet pair, gives the label of the pair in the EPP.

$$[\![\top]\!](\alpha, \beta) \triangleq \top \qquad [\![\bot]\!](\alpha, \beta) \triangleq \bot$$

$$[\![(f, e)]\!](\alpha, \beta) \triangleq \begin{cases} [\![e[\alpha.f][\beta.f]]\!](\alpha, \beta) & \text{if } \alpha.f \in \mathsf{keys}(e) \wedge \beta.f \in \mathsf{keys}(e[\alpha.f]) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The only pairs that we add to an EPP come from membership queries. Hence, we maintain by construction the invariant that $[\![e]\!]$ is always a restriction of the

semantics of the target SPP, $[\![s]\!]$, in the sense that whenever $[\![e]\!](\alpha, \beta) = \ell$, it is also the case that $[\![e]\!](\alpha, \beta) = \ell$.

4.4.4 The SPP Learning Algorithm

The learner executes the following steps in a loop: (i) convert the EPP to an SPP (ii) conjecture an SPP, and (iii) update the EPP with the counterexample received from the teacher, or terminate if the conjecture was correct. The pseudocode for the learner appears in Figure 4.3. In the rest of this section, we further detail the learner's subroutines.

$e \leftarrow \bot$
$c \leftarrow \mathsf{equiv}_{\mathsf{SPP}}(\bot)$
while $c \neq \top$ **do**
    $e \leftarrow \mathsf{update}(e)(c)(\mathsf{mem}_{\mathsf{SPP}}(c))$
    $h \leftarrow \mathsf{hyp}_{\mathsf{SPP}}(e)$
    $c \leftarrow \mathsf{equiv}_{\mathsf{SPP}}(h)$
return h

Figure 4.3: Algorithm for learning SPPs.

The central function of the learning process the conversion function from EPPs to a hypothesis SPPs: $\mathsf{hyp}_{\mathsf{SPP}} \colon \mathsf{EPP} \to \mathsf{SPP}$, see Figure 4.4. It works in bottom-up fashion, following the structure of the EPP. At each (f_i, e), it selects one of the bindings of e to represent the default case in the SPP. It uses the same value for the default assignment case. Ideally, the selected binding will correspond to a value that classifies the largest fraction of the evidence in the EPP—a default case with this property helps keep the conjectured SPP small. The mechanism for making this choice is to choose the branch that results in the smallest SPP after canonicalization. In the pseudocode below, the function $\mathsf{select}(v)$ builds the SPP that would result from v being chosen as the default case, and $\mathsf{hyp}_{\mathsf{SPP}}$ works by

$\mathsf{hyp}_{\mathsf{SPP}}\ \bot \triangleq \bot \qquad \mathsf{hyp}_{\mathsf{SPP}}\ \top \triangleq \top$

$\mathsf{hyp}_{\mathsf{SPP}}(\mathsf{EPP}(f, e)) \triangleq \min_{v \in \mathsf{keys}(e)} \mathsf{select}(v)$, where:

$\quad \mathsf{select}(v) \triangleq \mathsf{spp}(f, b, m, d)$, where:

$\quad\quad b = \{v' \mapsto \{v'' \mapsto \mathsf{hyp}_{\mathsf{SPP}}\ e' \mid v'' \mapsto e' \in m'\} \mid v' \mapsto m' \in e, v' \neq v\}$

$\quad\quad m = \{v' \mapsto \mathsf{hyp}_{\mathsf{SPP}}\ e' \mid v' \mapsto e' \in e[v], v' \neq v\}$

$$\quad\quad d = \begin{cases} \mathsf{hyp}_{\mathsf{SPP}}\ e[v][v] & \text{if } v \in \mathsf{keys}(e) \text{ and } v \in \mathsf{keys}(e[v]) \\ \bot & \text{otherwise} \end{cases}$$

Figure 4.4: Definition of $\mathsf{hyp}_{\mathsf{SPP}}$, for generalizing EPPs to SPPs.

taking the minimum over $\mathsf{select}(v)$ for all keys v for each level of the EPP. Note this process does not, in general, result in the *smallest possible* SPP consistent with the EPP—it turns out it is NP-hard to find smallest SPPs in general (see Section 4.4.5). To quantify the size of an SPP, we use the following function $\mu \colon \mathsf{SPP} \to \mathbb{N}$:

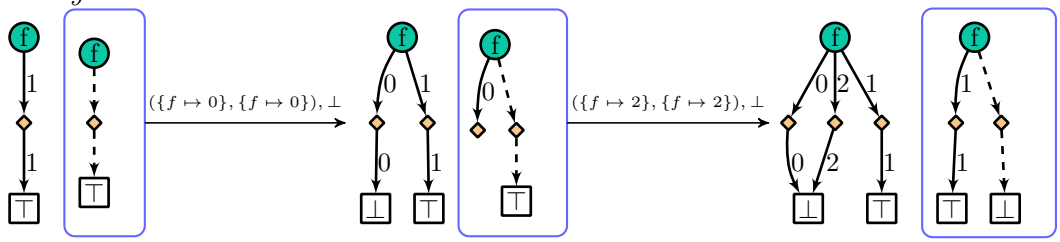
$$\mu(\top) = \mu(\bot) = 0 \qquad \mu(SPP(f, b, m, d)) = \#\text{keys, recursively, in } b, m, d$$

That is, this cost function is simply a sum of the keys in the maps at each level of the SPP, which is a reasonable approximation of the memory footprint of an SPP in a memoized implementation. When we take the minimum of SPPs in the definition below, it is with respect to μ. Note that spp is a "smart constructor" that performs canonicalization of the SPPs. We rely on canonicalization to remove redundant test branches that have the same behavior as the selected default case.

The $\mathsf{update} \colon \mathsf{EPP} \to \mathsf{Pk} \times \mathsf{Pk} \to 2 \to \mathsf{EPP}$ operation used in in Figure 4.3 just adds an example pair to the trie. We assume an implementation that, whenever $e' = \mathsf{update}\ e(\alpha, \beta)\ \ell$, then the label for (α, β) is updated in e' and no other pairs are affected:

$$[\![e']\!](\alpha, \beta) = \ell \wedge \forall (\alpha', \beta') \neq (\alpha, \beta) : [\![e]\!](\alpha', \beta') = \ell' \Rightarrow [\![e']\!](\alpha', \beta') = \ell'.$$

Example 45. *Consider applying the algorithm in Figure 4.3 to the NetKAT expression $f = 1$. The SPPs wrapped in blue boxes below indicate the conjectures h made in each iteration, while the EPPs drawn to the left of the SPPs indicate the set of evidence e according to which the conjectures are made. The labels on the arrows indicate the counterexamples given by the teacher in each iteration. The initial state of the algorithm before the while loop (when $e = \bot$) is omitted for brevity.*



The SPP learner in Figure 4.3 eventually learns the correct SPP held by the teacher. The following lemma shows that SPPs constructed from EPPs are consistent with all the packet pairs in the EPP:

Lemma 46. *Let e be an* EPP*, with the number of fields $|F| \geq 1$. For any $v \in V$, and let $s = \mathsf{SPP}(f, b, m, d) = \mathsf{select}\ v$ for e. Then for all (α, β) such that $[\![e]\!](\alpha, \beta) = \top$, we have $[\![s]\!](\alpha, \beta) = \top$, and for all (α, β) such that $[\![e]\!](\alpha, \beta) = \bot$, we have $[\![s]\!](\alpha, \beta) = \bot$.*

In addition, the learning algorithm terminates ensuring soundness.

Theorem 47. *The algorithm in Figure 4.3 terminates with a correct SPP.*

4.4.5 Finding Small SPPs Is NP-hard

Typically when developing learning algorithms, a learner usually tries to make conjectures as small as possible, as smaller conjectures usually generalize better

and therefore result in more informative counterexamples. However, our $\mathsf{hyp}_{\mathsf{SPP}}$ does not produce SPPs which are minimal in general. It turns out that finding small SPP consistent with a given EPP is NP-hard:

Definition 48. *The Small Consistent SPP Problem,* SmallSPP*, is to determine, given an EPP e and $k \geq 0$, whether there exists an SPP s consistent with e such that $\mu(s) \leq k$.*

Theorem 49. SmallSPP *is NP-hard.*

Proof sketch. By reduction from IndependentSet, which is to determine given an undirected graph $G = (V, E)$ and $k \geq 0$, whether there is a set $V' \subseteq V$ such that $|V'| \geq k$ and for all $v_1, v_2 \in V'$, $(v_1, v_2) \notin E$. Given a graph (V, E), we build an EPP with fields f_1, f_2 and values $V \cup V \times V \cup \{1, 2\}$ as follows. First define two small EPPs:

$$e_\top = \mathsf{EPP}(f_2, \{1 \mapsto \{2 \mapsto \top\}, 2 \mapsto \{1 \mapsto \bot\}\}), \quad \text{and}$$
$$e_\bot = \mathsf{EPP}(f_2, \{1 \mapsto \{2 \mapsto \bot\}, 2 \mapsto \{1 \mapsto \top\}\}).$$

Then we define our EPP by:

$$\mathsf{to\_epp}(G) = \mathsf{EPP}(f_1, \{v_1 \mapsto \{\{v_1, v_2\} \mapsto \mathsf{h}(v_1, v_2) \mid v_2 \in V, v_1 \neq v_2\} \mid v_1 \in V\})$$

where $h(v_1, v_2)$ satisfies that, for any $v_1, v_2 \in V$: $h(v_1, v_2) = h(v_2, v_1) = e_\top$ if $\{v_1, v_2\} \notin E$, and $h(v_1, v_2) = e_\top$ and $h(v_2, v_1) = e_\bot$ otherwise. It can be shown that every SPP consistent with this EPP corresponds to an independent set of G (the independent set is the vertices not chosen for the test branches of the SPP). As a result, the smallest SPP corresponds to the largest independent set. □

4.5 Learning Symbolic NetKAT Automata

The learner in Section 4.3 is appealing for its simplicity, but it has a critical limitation that restricts its applicability in practice. If we require states to be separated by packet, then PNKAs are necessarily much larger than necessary. In addition, the learning algorithm itself must repeatedly iterate through the entire space of packets to update the observation table. To improve on this state of affairs, we now present an algorithm, `NKL*`, for learning *symbolic* NetKAT automata, which uses the SPP learner of Section 4.4 as a subroutine. The advantage to symbolic automata is that they are more compact, as they capture behaviors across sets of packets concisely.

4.5.1 Background

SPPs encode transformations on packets in a symbolic manner. Hence, we can construct automata that use SPPs to represent the transition and observation functions for each state.

Definition 50. *A* symbolic *NetKAT automaton (SNKA) is a four-tuple* $(S, s_0, \delta, \varepsilon)$ *where* S *is a finite set of states,* $s_0 \in S$ *is the start state,* δ *and* ε *are transition and observation functions, respectively:*

$$\delta \colon S \to S \to \mathsf{SPP} \qquad\qquad \varepsilon \colon S \to \mathsf{SPP}$$

There is one additional restriction to ensure that automata are deterministic: for all states $s, s_1, s_2 \in S$ *such that* $s_1 \neq s_2$*, we must have* $\delta(s)(s_1) \mathbin{\hat{\cap}} \delta(s)(s_2) = \bot$*, where* $\hat{\cap}$ *is intersection on SPPs.*

Semantically, an SNKA $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ accepts a language $\mathcal{L}(\mathcal{M}) \subseteq \mathsf{GS}$ containing all strings $w \in \mathsf{GS}$ for which $\mathcal{M}(s_0, w) = \top$, where:

$$\mathcal{M}(s, \alpha \cdot \beta) \triangleq [\![\varepsilon(s)]\!](\alpha, \beta)$$

$$\mathcal{M}(s, \alpha \cdot \beta \cdot \mathsf{dup} \cdot w') \triangleq \begin{cases} \mathcal{M}(s', \beta \cdot w') & \exists s' \in S \text{ such that } [\![\delta(s)(s')]\!](\alpha, \beta) = \top \\ \bot & \text{otherwise} \end{cases}$$

4.5.2 The SNKA Teacher

For the SNKA learning problem, we assume essentially the same teacher as in Section 4.3, except that the teacher supports equivalence queries for symbolic automata instead of PNKA.

Definition 51 (MAT Interface)**.** *The MAT interface for Symbolic NetKAT automata (SNKA) is implemented as a membership and an equivalence oracle:*

$$\mathsf{mem}_{\mathsf{SNKA}} : \mathsf{GS} \to 2 \qquad \mathsf{equiv}_{\mathsf{SNKA}} : \mathit{SNKA} \to 1 + \mathsf{GS}$$

Remark 52. *The concept of using isolated MAT-style learning subroutines for complex transition labels was first explored by Argyros et al. [11] in the context of SFAs. However, using SFA learning for NetKAT automata would only handle the transitions in a symbolic way—i.e., the state space would blow up, as with PNKA learning. Hence, to obtain an efficient algorithm for SNKA learning requires additional insights and data structures, as we show below.*

Our general strategy for the learning algorithm is to follow the canonical learning algorithm of Figure 4.1, but eliminating *all* of the steps where we have to iterate over the entire packet space. To review, here are the steps where the canonical learner (Figure 4.1) iterates over the packet space:

- We initialize a table T_α for every packet α.
- We initialize every E_α to include every individual packet in Pk.
- In two places we apply $\mathsf{extend}(T)$: Since we may think of T_α as already having $|\mathsf{Pk}|$ rows and columns (i.e., of size $|S_\alpha \cup S_\alpha \cdot (\mathsf{Pk} \cdot \mathsf{dup})|$ and $|E_\alpha|$, respectively), then extend requires at least $|\mathsf{Pk}|$ membership queries because we have just added a row or column.

Instead, we will adopt a kind of lazy approach—e.g., rather than eagerly asking $|\mathsf{Pk}|$ many membership queries, we will make incremental guesses and wait to be corrected by a counterexample. In particular, even if we remove all three of these steps, our conjectures will still apply to the entire packet space due to use of the EPP to SPP conversion (Section 4.4). Put another way, we build an automaton with the specific evidence traces that we have seen, and then generalize the automaton from there to be well-defined on all input traces. Before specifying our learning algorithm in pseudocode, we first describe the modifications we make to the core data structures.

4.5.3 Partial Observation Table

We allow the observation table to be partial—i.e., entire packet tables as well as individual entries in packet tables may be missing. Hence, at a given point in the learning process, we will only have a packet tables for the packets we have encountered in some example. In addition, the packet table function T_α does not have information on the extended portion (S'_α), nor is E_α initialized with all of Pk. Putting these ideas together, the type of a partial packet table is as follows:

Definition 53 (Partial Packet Table)**.** *For a packet $\alpha \in \mathsf{Pk}$, a packet table is a tuple $(S_\alpha, E_\alpha, T_\alpha)$, where:*

- $S_\alpha \subseteq \{p \mid \mathsf{last}(p) = \alpha, p \in \mathsf{Pre}\}$ *is a set of* access prefixes.
- $E_\alpha \subseteq \mathsf{Suf}$ *is a set of* distinguishing suffixes.
- $T_\alpha \colon S_\alpha \times E_\alpha \to 2$ *is defined such that $T_\alpha(s,e) = \top$ if $s \cdot e \in \mathcal{L}$, and $T_\alpha(s,e) = \bot$ otherwise.*

Definition 54 (Partial Observation Table)**.** *An observation table P is a set of Partial Packet Tables, $(S_\alpha, E_\alpha, T_\alpha)$. There may or may not be a Partial Packet Table in P for each particular $\alpha \in \mathsf{Pk}$.*

Closedness and Consistency Along with these changes to the table structure, we will omit the closedness check in our algorithm. The reason we can avoid this check is that, since we will not be making membership queries for an extension of every prefix by one packet, we will also not have any corresponding "lower" rows of our packet-tables. We will only add transitions for which we have received evidence. Note that missing transitions are perfectly acceptable in a symbolic automaton—the result is simply to drop certain traces.

We will still check a form of consistency as this is needed to ensure that the hypothesis automaton is deterministic (i.e., consistency will ensure we only allow a transition to one state for a packet pair). The modification to consistency is simple: whenever we have equal rows in a partial packet table $(S_\alpha, E_\alpha, T_\alpha)$, we only require that their successors on, say β, are equal *if* both rows exist in $(S_\beta, E_\beta, T_\beta)$ (the box highlights the difference from Definition 36).

Definition 55 (Consistent Partial Observation Table)**.** *A partial observation table P is* consistent*, if for each $\alpha, \beta \in \mathsf{Pk}$ and for each $s, s' \in S_\alpha$, then*

while P not consistent **do**
  let $e \in E_\beta$ s.t. $\mathsf{row}_\alpha(s) = \mathsf{row}_\alpha(s')$,
    but $T_\beta(s \cdot \beta \cdot \mathsf{dup}, e) \neq T_\beta(s' \cdot \beta \cdot \mathsf{dup}, e)$.
  $E_\alpha \leftarrow E_\alpha \cup \{\beta \cdot \mathsf{dup} \cdot e\}$
return $\mathsf{extend}(P)$

Algorithm for $\mathsf{mkConsistent}\colon T \to T$.

$S \leftarrow \{s_0\}$
$\delta(s_0)(s_0) \leftarrow \bot$
$\varepsilon(s_0) \leftarrow \bot$
return $\mathcal{M}_\varnothing \leftarrow (S, s_0, \delta, \varepsilon)$

$\mathcal{M}_\varnothing$ is the SNKA of the empty language.

$\mathcal{H} \leftarrow \mathcal{M}_\varnothing$; $P \leftarrow \{\}$
while $c \leftarrow \mathsf{equiv}_{\mathsf{SNKA}}(\mathcal{H}) \in \mathsf{GS}$ **do**
  $P \leftarrow \mathsf{update}(P, c)$
  $P \leftarrow \mathsf{mkConsistent}(P)$
  $\mathcal{H} \leftarrow \mathsf{hyp}_{\mathsf{SNKA}}(P)$
return $\mathcal{H}$

Algorithm for learning symbolic automata, `NKL`*.

Input: Partial Observation table P, counterexample string $c \in \mathsf{GS}$
for $s \in \mathsf{Pre}, e \in \mathsf{Suf}$ s.t. $c = s \cdot e$ **do**
  $S_{\mathsf{last}(s)} \leftarrow S_{\mathsf{last}(s)} \cup \{s\}$
  $E_{\mathsf{last}(s)} \leftarrow E_{\mathsf{last}(s)} \cup \{e\}$
return $\mathsf{extend}(P)$

Subroutine update.

Figure 4.5: SNKA learning algorithm and subroutines.

if both $s \cdot \beta \cdot \mathsf{dup}, s' \cdot \beta \cdot \mathsf{dup} \in S_\beta$, *and we have* $\mathsf{row}_\alpha(s) = \mathsf{row}_\alpha(s')$, *then we must have* $\mathsf{row}_\beta(s \cdot \beta \cdot \mathsf{dup}) = \mathsf{row}_\beta(s' \cdot \beta \cdot \mathsf{dup})$.

4.5.4 The SNKA Learning Algorithm

The rest of the SNKA learning algorithm is similar to the one given in Figure 4.1. The update function we defined before (Figure 4.1) almost still works except that we add both *prefixes and suffixes* of each counterexample to P. The updated pseudocode is shown in Figure 4.5.

Due to our dramatic simplification of the outer algorithm, however, the construction of our hypothesis automaton, $\mathsf{hyp}_{\mathsf{SNKA}}$, is more complicated. To separate concerns and tame the complexity somewhat, we construct our automaton in two phases. First, we use the observation table to build a data structure we call an *evidence automaton* (EA), and then we convert the EA to an SNKA.

Constructing the Evidence Automaton

Starting from an observation table, we first construct an EA. An EA is like an SNKA but uses EPPs instead of SPPs.

Definition 56 (Evidence Automaton)**.** *A NetKAT evidence automaton (EA) is a four-tuple* $(Q, q_0, \delta, \varepsilon)$*, with* Q *a finite set of states,* $q_0 \in S$ *the initial state,* $\delta \colon Q \to Q \to \mathsf{EPP}$ *the transition function and* $\varepsilon \colon Q \to \mathsf{EPP}$ *the observation function.*

Now we show how to construct an evidence automaton $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ from a partial observation table P. The first challenge is to create the set of states for the evidence automaton. Here, we are guided by the semantics of the canonical automaton but wish to have as few states as possible.

Packet-specific states For each packet table $(S_\alpha, E_\alpha, T_\alpha) \in P$, we start with a set of states by inspecting the packet table T_α the same way that they are identified in Section 4.3.

So for each α, we define the packet-specific states for each α we have $Q_\alpha \triangleq \{\mathsf{row}_\alpha(s) \mid s \in S_\alpha\}$. This means that for each individual elements $q, q' \in Q_\alpha$ we have prefixes $s, s' \in S$ such that $q = \mathsf{row}_\alpha(s)$ and $q' = \mathsf{row}_\alpha(s')$. Moreover, if $q \neq q'$, this means that there is a suffix $e \in E_\alpha$ for which $\mathsf{row}_\alpha(s)(e) \neq \mathsf{row}_\alpha(s')(e)$, or, equivalently, $T_\alpha(s, e) \neq T_\alpha(s', e)$.

Thus Q_α are just the set of distinct rows of the packet table $(S_\alpha, E_\alpha, T_\alpha)$ that can be distinguished by suffixes from E_α. Next, we merge these states to produce the set of states of the symbolic NKA.

Merged NKA states Now that we have Q_α for each α, we build a set Q, along with a map for each Q_α into Q, which we denote $\eta\colon Q_\alpha \to Q$. There are only two restrictions that guide how we can combine packet-specific states into global states:

1. There is a special $q_0 \in Q$ such that for all Q_α, we have $\eta(\mathsf{row}_\alpha\ \alpha) = q_0$.
2. For each $q, q' \in Q_\alpha$ then $q \neq q'$ implies $\eta(q) \neq \eta(q')$.

The first restriction says that we must map every start state into the same global state. The second restriction says that we cannot combine states from the same Q_α, after all these are already distinguished by a suffix in E_α, and this suffix prevents their combination in the symbolic automaton.

Every combination of packet states into global states that respects these two restrictions is valid. The reason, briefly, is that we may simply guard every behavior by a complete test for each different packet in the worst case (the final SPP might not do this, because it might not be needed). Indeed, different implementations may make different merges here. Different choices for η will result in different sizes for the SPPs in the final automaton (state-minimal SNKAs are not unique).

Although it would be interesting to explore heuristics that result in smaller final SPPs, we defer that question to future work and instead merely insist that the result be minimal with respect to the number of states. To achieve this, we choose Q to have a size matching the largest $|Q_\alpha|$ (clearly any smaller Q will violate restriction 2, above). Otherwise we assume an arbitrary choice for η.

Adding EA Transitions Having created the set of states, Q, we need to build an EPP to label the transitions between each pair of states to build δ. To do this,

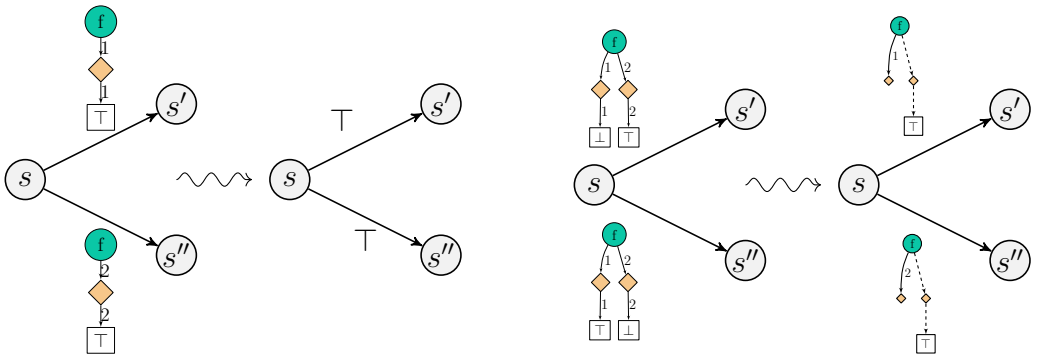


Figure 4.6: Converting EPP transition labels to SPP transition labels.

we start by initializing an empty EPP for each pair of states. Next we iterate over the entire observation table adding a positive example pair to some EPP for each row in each packet table. That is, for each trace $s \cdot \alpha \cdot \mathsf{dup} \cdot \beta \in S_\beta$, we add a pair (α, β) with label $\top$ to the EPP $\delta(\eta(\mathsf{row}_\alpha(s \cdot \alpha)))(\eta(\mathsf{row}_\beta(s \cdot \alpha \cdot \mathsf{dup} \cdot \beta)))$.

Determinizing pairs We would like to perform EPP to SPP conversion on each transition and observation function, but there is a fly in the ointment. Because we will convert each transition to SPP independently, two SPPs leaving the same state might generalize to "overlap" such that the resulting automaton is nondeterministic. For example, suppose we have two EPPs labeling two transitions leaving the same state, s, as shown in Section 4.5.4. If we perform SPP conversion to the EA on the left, we get the symbolic NKA on the right—which is not deterministic! Leaving state s on the packet pair $(\{f \mapsto 1\}, \{f \mapsto 1\})$ we can go to either s' or s'' because $[\![\top]\!](\{f \mapsto 1\}, \{f \mapsto 1\}) = \top$. To prevent this problem, we add negative examples to the EA: for each positive example pair (α, β) such that $[\![\delta(s)(s')]\!](\alpha, \beta) = \top$, we add a negative pair, $(\alpha, \beta, \bot)$ to each EPP $\delta(s)(s'')$ where $s' \neq s''$. This process requires that we do not have two of the same *positive example pairs* leading to different states out of the same state, but this is ensured by the consistency check.

Adding EA Observations Defining the observation function for an SNKA is analogous to determining whether a state in a standard DFA is accepting or not. Recall that in the canonical learner, we are guaranteed to have $\mathsf{Pk} \subseteq E_\alpha$ which allows us to define the observation function on every final packet for each row. We perform the analogous step here. More precisely, for each $q = \mathsf{row}_\alpha(s)$:

$$[\![\varepsilon(\eta(q))]\!](\alpha, \beta) = \ell \iff T_\alpha(s, \beta) \text{ exists and } T_\alpha(s, \beta) = \ell.$$

To summarize the EA construction: for a partial observation table P, we have $\mathsf{ev}(P) \triangleq (Q, q_0, \delta, \varepsilon)$ constructed by the following pseudocode:

Converting the Evidence Automaton to Symbolic NetKAT Automaton

Having constructed an EA from the Observation Table, we are close to having an SNKA to conjecture. Conceptually, we would like to simply map our $\mathsf{hyp}_{\mathsf{SPP}}$ function from Section 4.4 over the transition and observation EPPs in the EA, producing an SNKA. There is one final determinization problem with doing this, which we show by way of an example.

Example 57. *Suppose we have added the negative example pairs to the EPPs, shown in Section 4.5.4. Unfortunately, when we do the SPP conversion, we may still get the automaton on the right, which is still not deterministic! The SPPs for* $f \neq 1$ *and* $f \neq 2$ *overlap on, e.g.,* $(\{f \mapsto 3\}, \{f \mapsto 3\})$.

This overlap illustrated by Theorem 57 can be resolved by taking differences arbitrarily (as SPP operations). The reason is that any pairs remaining in the intersection of the semantics of two same-origin SPPs must *both* not be from example pairs in the EPPs (because otherwise the overlap would have been prevented by

negative pairs, above). We can therefore compute final transition SPPs by subtracting out the other SPPs leaving the same state.

Thus, given an EA $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ we define $\mathsf{sym}(\mathcal{E}) \triangleq (Q, q_0, \delta', \varepsilon')$, where [2]:

$$\delta'(q)(q') \triangleq \mathsf{hyp}_{\mathsf{SPP}}(q)(q') \mathbin{\hat{-}} \hat{\sum_{q'' \neq q'}} \mathsf{hyp}_{\mathsf{SPP}}(q)(q'') \qquad \varepsilon'(q) \triangleq \mathsf{hyp}_{\mathsf{SPP}}(\varepsilon(q))$$

4.5.5 Correctness of the SNKA Learner

We conclude by showing that our symbolic SNKA learner is correct. The first lemma says that each conjecture is a well-defined SNKA.

Lemma 58. *For any partial observation table P, we have that $\mathsf{hyp}_{\mathsf{SNKA}}(P)$ is a well-defined SNKA.*

Lemma 59. *Let $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ be an EA for a consistent partial observation table P (i.e. $\mathcal{E} = \mathsf{ev}(P)$), and let $\mathcal{M} = (Q, q_0, \delta', \varepsilon) = \mathsf{sym}(\mathcal{E})$. For any $q, q' \in Q$ and any pair $(\alpha, \beta) \in \mathsf{Pk} \times \mathsf{Pk}$, then:*

(a) If $[\![\delta(q)(q')]\!](\alpha, \beta) = \top$, then $[\![\delta'(q)(q')]\!](\alpha, \beta) = \top$.

(b) If $[\![\varepsilon(q)]\!](\alpha, \beta) = \top$, then $[\![\varepsilon'(q)]\!](\alpha, \beta) = \top$.

Next we show that hypothesis automata are correct for certain examples in the table.

Lemma 60. *Let P be a consistent partial observation table for target language $\mathcal{L} \subseteq \mathsf{GS}$ and $\mathcal{M} = \mathsf{hyp}_{\mathsf{SNKA}}(P) = (S, s_0, \delta, \varepsilon)$. Let $u \in \mathsf{GS}$. If for every "breaking point" of $u = w \cdot e$ there is a packet table $(S_\alpha, E_\alpha, T_\alpha)$ such that $w \in S_\alpha$ and $e \in E_\alpha$, then we have $\mathcal{M}(\eta(\mathsf{row}_\alpha(w), \alpha \cdot e) = \top \iff T_\alpha(w, e) = \top$.*

[2] Using $\hat{-}$ for semantic difference of SPPs, and $\hat{\sum}$ for sum of SPPs

Corollary 61. *After receiving a counterexample c, any subsequent hypothesis $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ is correct with respect to the target $\mathcal{L}$ on c.*

Theorem 62. *For a regular language $\mathcal{L}$, `MKL`* (in Section 4.5.4) terminates with an SNKA for $\mathcal{L}$.*

Query complexity It is possible to model NetKAT using DFAs, where each packet is a character. Using this representation, the standard $\mathtt{L}^\star$ algorithm would learn a model using polynomial many queries in the size of this DFA. However, this representation of a NetKAT program is in general much larger than the SNKA learned by the algorithm in Section 4.5. Our formal development establishes termination of the SNKA learner, but we leave a more precise analysis to future work. It is important to note that the two algorithms cannot be directly compared because the assumption of equivalence oracle is different for the two models. As the symbolic representations used in SPPs and SNKAs are designed to be compact in the common case, establishing complexity is likely to be more complicated than for $\mathtt{L}^\star$. We note, however, that the PNKA learner (Section 4.3) inherits polynomial query complexity in the size of the PNKA by essentially the same arguments applicable to $\mathtt{L}^\star$.

4.5.6 Networking Example

We return to the network that we encoded in Section 2.1.5. As a reminder, the target expression is:

$$\mathsf{sw}{=}1 \cdot \mathsf{pt}{=}1 \cdot ((\mathsf{pt}{=}1 \cdot \mathsf{pt}{\leftarrow}2 + \mathsf{pt}{=}2 \cdot \mathsf{pt}{\leftarrow}1)\cdot$$
$$(\mathsf{pt}{=}1 + \mathsf{pt}{=}3 + \mathsf{pt}{=}2 \cdot (\mathsf{sw}{=}1 \cdot \mathsf{sw}{\leftarrow}2 + \mathsf{sw}{=}2 \cdot \mathsf{sw}{\leftarrow}1)) \cdot \mathsf{dup})^\star \cdot \mathsf{sw}{=}2 \cdot \mathsf{pt}{=}1$$

Our algorithm succeeds in learning a small automaton for this expression after 6 conjectures and 31 membership queries. For readability, we omit membership queries and observation tables. The SPP labels of transitions are shown using equivalent dup-free expressions.

The learner starts by conjecturing the "drop-everything" automaton, on the right in Figure 4.7. The teacher responds with a positive example (valid trace):

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}]$$

And then the learner conjectures the automaton in the middle in Figure 4.7.

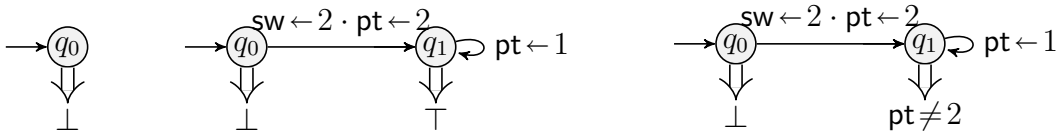


Figure 4.7: The first three conjectures.

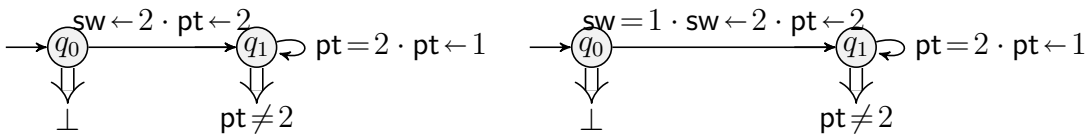
The teacher then provides a negative example (an invalid trace):

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}]$$

and the learner conjectures the automaton on the right in Figure 4.7. The process continues with the teacher giving another negative example:

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\};$$
$$\{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}]$$

and the learner conjecturing the automaton on the left in Section 4.5.6. The



Two further conjectures.

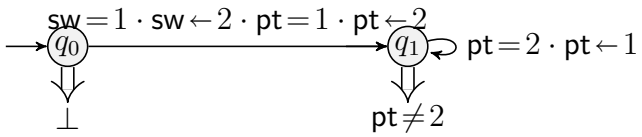
teacher gives yet another negative example— $[\{\mathsf{sw} \mapsto 0, \mathsf{pt} \mapsto 0\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto$

$2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}]$— followed by the learner conjecturing the automaton on the right in Section 4.5.6.

Finally, the teacher gives the negative example:

$$[\{\mathsf{sw} \mapsto 1, \mathsf{pt} \mapsto 0\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 2\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}; \{\mathsf{sw} \mapsto 2, \mathsf{pt} \mapsto 1\}]$$

And the learner conjectures:



The sixth (and last) conjecture.

This automaton is correct and the learning process terminates successfully! Though this is just a toy example, meant to illustrate the steps of the algorithm, it is worth noting that the larger networks that we deal with in the next section follow a similar structure.

4.6 Implementation and Evaluation

We have prototyped our algorithms for learning SPPs (Figure 4.3) and SNKAs (Section 4.5) in OCaml. Although our implementation has not yet been tuned to achieve scalable performance, it is still useful for illustrating the benefits of our approach. We have used it to experiment with learning models of real-world networks drawn from a standard benchmark.

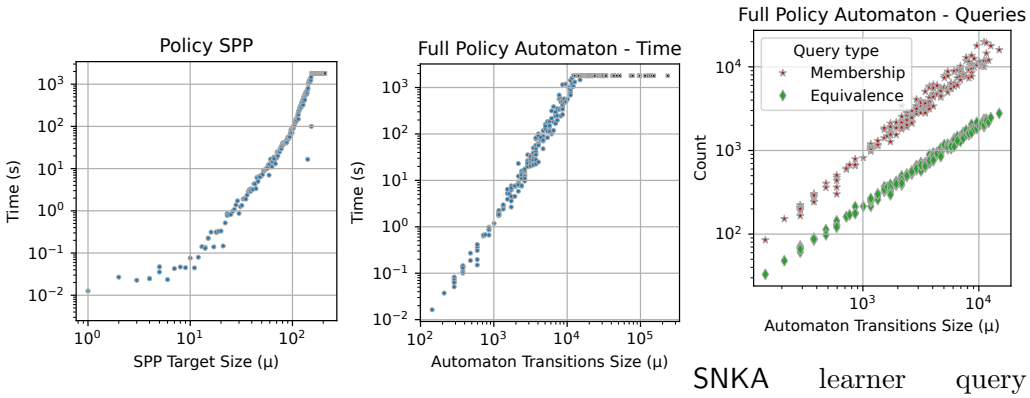
Evaluation To evaluate our approach we sought to answer two research questions:

- How well does the SPP learner scale with topology size for network transfer functions?
- How does the SNKA learner scale with topology size for whole-trace network behavior?

Topology Zoo Policies To explore these questions, we built a set of learning problem instances using the Internet Topology Zoo [72], a dataset for benchmarking network verification tools [55, 91]. For each topology, we have a simple shortest-path routing policy encoded as in Section 2.1.5. The results of our various experiments are presented in Figure 4.8 and described below.

SPP Learning To evaluate the SPP learner, we set each network's "transfer function" ($r \cdot t$, see Section 2.1.5) as the target SPP and ran our SPP learner (that is, membership queries are on pairs of packets representing single-step transformations in the network). The results are shown in Section 4.6 in log-log scales. The learner recovered the correct SPP for 190 out of 261 topologies before reaching a 30 minutes timeout, suggesting its feasibility for learning policies of modest size.

SNKA Learning To evaluate the SNKA learner, we instantiated the teacher with a full network behavior expression, $p_i \cdot (r \cdot t \cdot \mathsf{dup})^\star \cdot p_f$. For our input and output predicates, we choose: $p_i \triangleq \sum_i \mathsf{sw} = i \cdot (\sum_j \mathsf{dst} = j)$, and $p_f \triangleq \sum_i \mathsf{sw} = i \cdot \mathsf{dst} = i$. This says that we are interested in initial packets starting at any origin with any destination, and the final packets which have reached their destination (i.e., membership queries are full-trace behaviors of the policy). The timing results are shown in Section 4.6. The SNKA learner succeeded on learning 190 of the 261 topologies. A plot of the number of membership queries and equivalence versus



SPP learner timings. SNKA learner timings. SNKA learner query counts.

Figure 4.8: Benchmark results for Topology Zoo dataset. Examples that exceeded 30 minutes are depicted with an "x" in the timing plot; the queries plot includes only examples that completed within the time limit. To keep the plot legible, we omit the largest topology, KDL, as it is an outlier.

target size is shown in Section 4.6. The linear shape does not mean linear behavior because the axes are log-log.

Remark 63. *The largest topology solved by the SNKA learner, Uunet, has 48 nodes, with 14 port numbers. Considering the fields* sw, dst, *and* pt *used in the problem, we have* $|\mathsf{Pk}| = 48 \cdot 48 \cdot 14 = 32256$. *Because of the carry-on packet semantics, this means that any approach based on learning a DFA would need to identify many thousands of different* states, *even if the transitions are treated symbolically, since only carry-on packets that will be dropped can be combined. Instead, the SNKA learner here identifies an automaton with only 2 states (excluding a drop state), and even though the transition SPPs are complex, it does so after* only *15,932 membership queries, and 2775 equivalence queries.*

4.7 Practical Concerns

In this section we sketch additional tasks that would be needed to apply our techniques in a practical setting where the oracle is a closed-box network or network

component.

Membership Queries We assume a membership oracle of type $\mathsf{GS} \to 2$ because it allows us to apply the core of MAT style learning to our domain. One might observe, however, that a closed-box network might more naturally be modeled by a type like $\mathsf{Pk} \to \mathsf{GS}$ (i.e., given input packets, traces are produced), and a real system may have nondeterministic behavior. To run our algorithm on a closed-box system, one would need to implement this translation between input packets and input traces. This is not a big issue: we can approximate the $\mathsf{GS} \to 2$ oracle using a "real" oracle by sending the first packet of our desired query trace (perhaps multiple times) and then checking whether the query trace is produced by the system in any of the trials (caching the results for later queries).

Equivalence Queries The assumption of an equivalence oracle in work based on Angluin's $\mathtt{L}^\star$ algorithm, including ours, is indeed a strong assumption. However, in practice, it often suffices to approximate equivalence. There is a large body of prior work (e.g., CacheQuery from PLDI '20 [113]), showing that approximation is effective, so the assumption of an equivalence oracle is less of a practical hurdle than it might seem.

In many systems, equivalence queries are approximated using membership queries. In a nutshell, the idea is to first generate a large set of traces (e.g., all traces up to a given length, or a random sample of such traces) and then execute them on the conjectured model and the system. If no discrepancies are found, then the equivalence query is deemed to be successful. On the other hand, if a discrepancy is found, then the equivalence query is deemed to fail, and the trace that triggered the discrepancy is returned as a counter-example. The Random Wp-

Method [57] and domain-specific approaches [77] offer reasonable approximation guarantees.

4.8 Related Work

Here we have introduced the first attempt at symbolic learning of NetKAT automata, however, a number of similar problems have been the topic of extensive study.

Learning for GKAT. Zetzsche et al. adapt Angluin's $\mathtt{L}^\star$ to the setting of Guarded Kleene Algebra with Tests [119]. Their work bears some similarity to ours in that they also establish a Myhill-Nerode Theorem for GKAT en route to the full presentation of their algorithm. Unfortunately, we cannot apply their results beyond inspiration because of the differences between GKAT and NetKAT: GKAT supports only a "guarded" choice operator, which is deterministic, while NetKAT supports the full nondeterministic choice operation of KAT. Further, NetKAT's carry-on packet semantics is the source of the challenges that we have solved in this chapter, issues that have no analog in GKAT. The incompatibility of NetKAT and GKAT is explored in Wasserstein's Master's thesis [114].

Learning OBDDs. Angluin-style learning of Ordered Binary Decision Diagrams (OBDDs) [58] and of automata with decision diagram transitions [88] have been explored. BDDs also appear in the use of $\mathtt{L}^\star$ to learn Java interface specifications [4] . The structure we use, Symbolic Packet Programs, are reminiscent of decision diagrams, yet have crucial differences which are essential to encode NetKAT policies as discussed in Chapter 3, which require the design of a new learning algorithm.

Learning Networking Protocols. Fiterău-Broştean et al. successfully applied automata learning to learn the state machine of fragments of the TCP protocol, and identified bugs in a widely used TCP implementation [49]. This work was further extended to learning sliding window behavior [48], learning SSH implementations [51], and DTLS implementations [50]. Ferreira et al. introduced the Prognosis tool to apply automata learning to the QUIC protocol, a promising new network protocol aiming to fully replace TCP, TLS/SSL, and HTTP for the modern web [45]. Prognosis was used to identify issues with both the QUIC RFC itself and several implementations. These approaches are prime examples of using automata learning to identify bugs in networking, but they have mostly explored DFAs, which have the drawbacks mentioned above. The contributions of this chapter advance $NetKAT$ automata as a more expressive learning target to the learning toolkit for networking.

4.9 Conclusion

In this chapter, we presented novel techniques for active learning of $NetKAT$ models. We developed the theory of NetKAT with a characterization of canonical automata (akin to the results of Myhill-Nerode for classical automata) and then used this notion to develop a learning algorithm. We designed two symbolic learning algorithms, one for the dup-free fragment of NetKAT (SPP learner) and one for the general case of symbolic $NetKAT$ automata (SNKA learner). We implemented these algorithms in an OCaml prototype and performed an evaluation using existing benchmarks. Our evaluation shows that both the SPP and the SNKA learners scale with topology size.

As a natural next step, we would like to apply our algorithm to practical scenarios, including inferring the model of a device for which no configuration or program is known and analyze potential misconfigurations or malicious behavior. We would also like to explore further improvements to the algorithm via *symbolic packet tables*. Currently, if multiple packets are observed to behave equivalently, we still treat their packet tables independently. This leads to repetition in queries, and a higher memory footprint than potentially needed. We would also like to investigate efficient heuristics for implementations of the state grouping function η. Currently the properties of this function focus on folding the state space, however transition SPP minimality would be as important for the scalability of $NetKAT$ procedures that use the models.

Data Availability

A snapshot of the OCaml code which underwent artifact evaluation is available as a dependencies-included Docker image on Zenodo [3].

4.10 Omitted Proofs

In this final section of the chapter, we complete the proofs that were omitted from the formal development.

[3] `https://doi.org/10.5281/zenodo.15230071`

4.10.1 PNKA Properties & Correctness

Theorem 27. *Let $\mathcal{L} \subseteq \mathsf{GS}$. Then $\mathcal{L}$ is accepted by a NKA if and only if it is accepted by a PNKA.*

Proof.

($\Rightarrow$) Let $\mathcal{N} = (S, s_0, \delta, \varepsilon)$ be an NKA accepting $\mathcal{L}$. We define a PNKA $\mathcal{P} = (Q, q_0, \partial, \lambda)$, where:

$$Q = \mathsf{Pk} \times S \qquad q_0(\alpha) = (\alpha, s_0) \qquad \partial_\beta(\alpha, s) = (\beta, \delta_{\alpha\beta}(s)) \qquad \lambda_\beta(\alpha, s) = \varepsilon_{\alpha\beta}(s)$$

We need to show $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{N})$. We show a stronger result: for any $s \in S$, it holds that $\mathcal{P}((\alpha, s), w) \iff \mathcal{N}(s, \alpha \cdot w)$. The result then follows by taking $s = s_0$. We prove the above result by induction on the length of w: if $w = \beta$, then we have $\mathcal{P}((\alpha, s), \beta) \iff \lambda_\beta(\alpha, s) \iff \varepsilon_{\alpha\beta}(s) \iff \mathcal{N}(s, \alpha \cdot \beta)$. On the other hand if $w = \beta \cdot \mathsf{dup} \cdot w'$, then:

$$\begin{aligned} \mathcal{P}((\alpha, s), w) &\iff \mathcal{P}(\partial_\beta(\alpha, s), w') \iff \mathcal{P}((\beta, \delta_{\alpha\beta}(s)), w') \\ &\overset{\text{IH}}{\iff} \mathcal{N}(\delta_{\alpha\beta}(s), \beta \cdot w') \iff \mathcal{N}(s, \alpha \cdot w). \end{aligned}$$

($\Leftarrow$) Let $\mathcal{P} = (Q, q_0, \partial, \lambda)$ be a PNKA accepting $\mathcal{L}$. We define an NKA $\mathcal{N} = (S, s_0, \delta, \varepsilon)$, where:

- $S = Q \uplus \{s_0, s_\perp\}$
- $\delta_{\alpha\beta}(s) = \begin{cases} \partial_\beta(q_0(\alpha)) & \text{if } s = s_0 \\ \partial_\beta(s) & \mathsf{spell}(s) = \alpha \\ s_\perp & \text{otherwise} \end{cases}$ $\qquad \varepsilon_{\alpha\beta}(s) = \begin{cases} \lambda_\beta(q_0(\alpha)) & \text{if } s = s_0 \\ \lambda_\beta(s) & \mathsf{spell}(s) = \alpha \\ \perp & \text{otherwise} \end{cases}$

Again, we need to show that $\mathcal{L}(\mathcal{N}) = \mathcal{L}(\mathcal{P})$. That is: $\mathcal{N}(s_0, \alpha \cdot w) \iff \mathcal{P}(q_0(\alpha), w)$. We first observe that:

$$\mathcal{N}(s_0, \alpha \cdot \beta) \iff \varepsilon_{\alpha\beta}(s_0) \iff \lambda_\beta(q_0(\alpha)) \iff \mathcal{P}(q_0(\alpha), \beta).$$

and

$$\begin{aligned}\mathcal{N}(s_0, \alpha \cdot \beta \cdot \mathsf{dup} \cdot w') &\iff \mathcal{N}(\delta_{\alpha\beta}(s_0), \beta \cdot w') \iff \mathcal{N}(\partial_\beta(q_0(\alpha)), \beta \cdot w') \\ &\overset{\dagger}{\iff} \mathcal{P}(\partial_\beta(q_0(\alpha)), w') \iff \mathcal{P}(q_0(\alpha), \beta \cdot \mathsf{dup} \cdot w').\end{aligned}$$

We conclude the proof with a proof of †. We show a more general fact: for $q \in Q$ and $\alpha \in \mathsf{Pk}$, with $\mathsf{spell}(q) = \alpha$, it holds that $\mathcal{P}(q, w) \iff \mathcal{N}(q, \alpha \cdot w)$ (to obtain † instantiate this with $q = \partial_\beta(q_0(\alpha))$, $\alpha = \beta$, and $w = w'$). This fact follows by induction on the length of w: if $w = \beta$, then we have $\mathcal{P}(q, \beta) \iff \lambda_\beta(q) \iff \varepsilon_{\alpha\beta}(q) \iff \mathcal{N}(q, \alpha \cdot \beta)$. On the other hand, let $w = \beta \cdot \mathsf{dup} \cdot w'$, and observe that $\mathsf{spell}(q) = \alpha$ means that $\delta_{\alpha\beta}(q) = \partial_\beta(q)$. Then:

$$\mathcal{P}(q, w) \iff \mathcal{P}(\partial_\beta(q), w') \overset{\text{IH}}{\iff} \mathcal{N}(\delta_{\alpha\beta}(q), \beta \cdot w') \iff \mathcal{N}(q, \alpha \cdot w).$$

□

Theorem 65. *The PNKA $\mathcal{P}_{\equiv_\mathcal{L}}$ is a minimal acceptor of the language $\mathcal{L}$.*

Proof. For readability, let $\mathcal{P} \triangleq \mathcal{P}_{\equiv_\mathcal{L}}$. We will prove that $w = \alpha \cdot z \cdot \beta \in \mathsf{GS}$ is accepted by $\mathcal{P}$ if and only if $w \in \mathcal{L}$.

We first show by induction that, for a given $s \in \mathsf{Pre}$ we have that $\mathcal{P}([s]_{\equiv_\mathcal{L}}, z \cdot \beta) = (s \cdot z \cdot \beta \in \mathcal{L})$.

For z with 0 dups, we have $\mathcal{P}([s]_{\equiv_\mathcal{L}}, \beta) = \lambda_\beta([s]_{\equiv_\mathcal{L}}) = s \cdot \beta \in \mathcal{L}$. Now let z have n dups, and assume that $\mathcal{P}([s]_{\equiv_\mathcal{L}}, z \cdot \beta) = s \cdot z \cdot \beta \in \mathcal{L}$.

Then for arbitrary γ we have that:

$$\mathcal{P}([s]_{\equiv_\mathcal{L}}, \gamma{\cdot}\mathsf{dup}{\cdot}z{\cdot}\beta) = \mathcal{P}(\partial_\gamma([s]_{\equiv_\mathcal{L}}), z{\cdot}\beta) = \mathcal{P}([s{\cdot}\gamma{\cdot}\mathsf{dup}]_{\equiv_\mathcal{L}}, z{\cdot}\beta) \overset{\text{IH}}{=} s{\cdot}\gamma{\cdot}\mathsf{dup}{\cdot}z{\cdot}\beta \in \mathcal{L}.$$

We then know that $\mathcal{P}(q_0(\alpha), z \cdot \beta) = \alpha \cdot z \cdot \beta \in \mathcal{L}$, and thus w is accepted by $\mathcal{P}$ iff $w \in \mathcal{L}$.

To see that $\mathcal{P}_{\equiv_\mathcal{L}}$ is minimal, observe that were $\mathcal{P}_{\equiv_\mathcal{L}}$ not minimal, that would mean there exist at least two different states s and s' with the same behavior that could be merged. However, were this the case, the prefixes reaching s and s' would then be in the same equivalence class of $\equiv_\mathcal{L}$. This contradicts the assumption that $s \neq s'$.

Therefore $\mathcal{P}_{\equiv_\mathcal{L}}$ is a minimal acceptor of $\mathcal{L}$ with $|Q| = |\{[s]_{\equiv_\mathcal{L}} \mid s \in \mathsf{Pre}\}| = |\equiv_\mathcal{L}|$ states. □

Definition 66 (State Equivalence Relation)**.** *Given a PNKA $\mathcal{M} = (Q, q_0, \partial, \lambda)$ we say that two prefixes s and t in* Pre *are indistinguishable by $\mathcal{M}$ if and only if the state reached by the automaton on input s is the same as the one reached on input t. Formally we define the relation:*

$$s \equiv_\mathcal{M} t \overset{\triangle}{\iff} \partial^+(s) = \partial^+(t)$$

Where $\partial^+ \colon \mathsf{Pre} \to Q$ is defined by:

$$\partial^+(\alpha) = q_0(\alpha) \qquad \partial^+(\alpha \cdot (\beta \cdot \mathsf{dup}) \cdot w) = \partial^\star(q_0(\alpha), (\beta \cdot \mathsf{dup}) \cdot w)$$

And $\partial^\star \colon Q \times (\mathsf{Pk} \cdot \mathsf{dup})^\star \to Q$ is given by:

$$\partial^\star(q, \epsilon) = q \qquad \partial^\star(q, (\beta \cdot \mathsf{dup}) \cdot w) = \partial^\star(\partial_\beta(q), w)$$

Note that $\equiv_{\mathcal{M}}$ is an equivalence relation and that it can only have a finite number of equivalence classes, one per state.

We now want to show that given a regular language $\mathcal{L} \subseteq \mathsf{GS}$ the relation $\equiv_{\mathcal{L}}$ has finite index. We do this with the help of the following lemma.

Lemma 67. *If $L = \mathcal{L}(\mathcal{M})$ for a PNKA $\mathcal{M}$ then for any $s, t \in \mathsf{Pre}$: if $s \equiv_{\mathcal{M}} t$ then $s \equiv_{\mathcal{L}} t$.*

Proof. First note that a guarded string $w \cdot \beta \in \mathsf{GS}$ satisfies:

$$w \cdot \beta \in \mathcal{L}(\mathcal{M}) \iff \lambda_\beta(\partial^+(w)) = \top$$

Suppose also that $s \equiv_{\mathcal{M}} t$ and let $w = \alpha \cdot e \cdot \beta \in \mathsf{GS}$. It is easy to see that $\partial^+(s \cdot e) = \partial^+(t \cdot e)$. We can then reason:

$$s \blacklozenge w \in \mathcal{L} \iff \lambda_\beta(\partial^+(s \cdot e)) = \top \iff \lambda_\beta(\partial^+(t \cdot e)) = \top \iff t \blacklozenge w \in \mathcal{L}. \quad \square$$

This lemma says that if two prefixes lead to the same state in a PNKA then they are in the same equivalence class of $\equiv_{\mathcal{L}}$. This means that each equivalence class of $\equiv_{\mathcal{L}}$ is a union of equivalence classes of $\equiv_{\mathcal{M}}$. Therefore if $\mathcal{L}$ is regular, then $\equiv_{\mathcal{L}}$ has finite index. In the other direction, if $\equiv_{\mathcal{L}}$ has finite index, then we can build the PNKA $\mathcal{P}_{\equiv_{\mathcal{L}}}$ accepting $\mathcal{L}$ by Theorem 65, so $\mathcal{L}$ is regular.

We finish by isolating claim (3): That the minimality of $(\mathcal{P}_{\equiv_{\mathcal{L}}}) = (Q, q_0, \partial, \lambda)$ is *unique*, and any other PNKA $\mathcal{M} = (Q', q_0', \partial', \lambda')$ with language $\mathcal{L}$ and $| \equiv_{\mathcal{L}} |$ states must be isomorphic to it.

Theorem 68. *Let $\mathcal{P} = (Q, q_0, \partial, \lambda)$ have language $\mathcal{L}$. If a PNKA $\mathcal{M} = (Q', q_0', \partial', \lambda')$ also has language $\mathcal{L}$ and $| \equiv_{\mathcal{L}} |$ states then $\mathcal{M} \cong \mathcal{P}_{\equiv_{\mathcal{L}}}$.*

Proof. For any state $q' \in Q'$ there exists a prefix $s \in \mathsf{Pre}$ that is an access prefix for q'. Formally, $\partial'^{+}(s) = q'$. This is guaranteed as $\mathcal{M}$'s minimality means it cannot have unreachable states.

We now define $h\colon Q \to Q'$ by $h([s]_{\equiv_\mathcal{L}}) = \partial'^{+}(s)$ and show that it is a homomorphism with respect to q_0, ∂, and λ.

The start state q_0 is preserved by h as for any $\alpha \in \mathsf{Pk}$, then $h(q_0(\alpha)) = \partial'^{+}(\alpha) = q'_0(\alpha)$.

The automaton transition structure is preserved as for any $s \in \mathsf{Pre}$, then:

$$h(\partial_\beta([s]_{\equiv_\mathcal{L}})) = h([s \cdot \beta \cdot \mathsf{dup}]_{\equiv_\mathcal{L}}) = \partial'^{+}(s \cdot \beta \cdot \mathsf{dup}) = \partial'_\beta(\partial'^{+}(s)) = \partial'_\beta(h([s]_{\equiv_\mathcal{L}}))$$

Finally, the observation function is preserved by h as for any $\beta \in \mathsf{Pk}$, then:

$$\lambda_\beta([s]_{\equiv_\mathcal{L}}) \iff s \cdot \beta \in L \iff \lambda'_\beta(\partial'^{+}(s)) \iff \lambda'_\beta(h([s]_{\equiv_\mathcal{L}}))$$

We conclude the proof by showing that h is actually an isomorphism. h is injective as:

$$h([s]_{\equiv_\mathcal{L}}) = h([t]_{\equiv_\mathcal{L}}) \implies \partial'^{+}(s) = \partial'^{+}(t) \implies s \equiv_\mathcal{M} t \implies s \equiv_\mathcal{L} t \implies [s]_{\equiv_\mathcal{L}} = [t]_{\equiv_\mathcal{L}}$$

As $|Q| = |Q'|$ and h is injective, h is also surjective, and therefore a bijection. As such, h is an isomorphism, and $\mathcal{M} \cong \mathcal{P}_{\equiv_\mathcal{L}}$. □

4.10.2 PNKA Learning Correctness

Lemma 69. *The hypothesis automaton,* $\mathsf{hyp}_{\mathsf{PNKA}}(T)$*, is well defined for any Observation Table* T*.*

Proof. We need to justify why q_0, δ, and ε are well-defined functions.

- The start state selector q_0 is well defined because α is added to each S_α during initialization, so there is a $\mathsf{row}_\alpha(\alpha)$.
- The transition function is well defined because the rows in each S_α are either distinct or consistent for a given transition label β. This is guaranteed by the table's consistency. Additionally each successor (i.e., $s \cdot (\beta \cdot \mathsf{dup})$ for each s) is guaranteed to exist in S'_β, and is guaranteed to have an equivalent state representative in S_β by the table's closedness.
- Each E_α is initialized to be all of Pk, so all possible $T_\alpha(\cdot, \beta)$ needed to define ε are present.

□

Lemma 70. *For any hypothesis PNKA $\mathcal{H} = (Q, q_0, \partial, \lambda)$ produced by the learner, and for any two prefixes $s, t \in \mathsf{Pre}$, we have that $s \equiv_{\mathcal{H}} t$ iff $s \equiv_{\mathcal{L}(\mathcal{H})} t$.*

Proof. ($\Rightarrow$) Suppose $s \equiv_{\mathcal{H}} t$, and let $e \in \mathsf{Suf}$. Let $s = \alpha \cdot s'$ for some α and s', and $t = \beta \cdot t'$ for some β and t'. Clearly $\mathcal{H}(\partial^+(s), e)$ iff $\mathcal{H}(\partial^+(t), e)$. Then we have $\mathcal{H}(q_0(\alpha), s \cdot e)$ iff $\mathcal{H}(q_0(\beta), t \cdot e)$. Since e is arbitrary, this means $s \equiv_{\mathcal{L}(\mathcal{H})} t$.

($\Leftarrow$) Suppose $s \not\equiv_{\mathcal{H}} t$. Then either s and t end with different packets (in which case clearly $s \not\equiv_{L(\mathcal{H})} t$), or they both end in some packet α. As the access prefixes lead to different states, by definition of the states of $\mathcal{H}$, we must have that $(\alpha, \mathsf{row}_\alpha(s')) \neq (\alpha, \mathsf{row}_\alpha(t'))$, for some equivalent access prefixes s' and t' of s and t, respectively. I.e., $s \equiv_{\mathcal{H}} s'$ and $t \equiv_{\mathcal{H}} t'$. As the rows $\mathsf{row}_\alpha(s')$ and $\mathsf{row}_\alpha(t')$ are different, it must be that there is a distinguishing suffix $e \in E_\alpha$ making them unequal. This suffix witnesses $s' \not\equiv_{\mathcal{L}(\mathcal{H})} t'$, and therefore $s \not\equiv_{\mathcal{L}(\mathcal{H})} t$. □

Lemma 71. *Suppose the teacher holds $\mathcal{L}$. Then for each hypothesis PNKA $\mathcal{H}$ of the learner, we have that $(\equiv_{\mathcal{L}})$ refines $(\equiv_{\mathcal{L}(\mathcal{H})})$ in the sense that for prefixes $s, t \in \mathsf{Pre}$, then $s \equiv_{\mathcal{L}} t$ implies $s \equiv_{\mathcal{L}(\mathcal{H})} t$.*

Proof. (By contrapositive) Suppose $s \not\equiv_{\mathcal{L}(\mathcal{H})} t$. By Theorem 70, this means $s \not\equiv_{\mathcal{H}} t$. If this is because s and t have different final packets, then clearly $s \not\equiv_{\mathcal{L}} t$ already. Otherwise, suppose $\mathsf{last}(s) = \mathsf{last}(t) = \alpha$. Then $\partial^+(s) \neq \partial^+(t)$ means $\mathsf{row}_\alpha(s') \neq \mathsf{row}_\alpha(t')$ for some $s', t' \in \mathsf{Pre}$ such that $s \equiv_{\mathcal{H}} s'$ and $t \equiv_{\mathcal{H}} t'$. This means there is an $e \in E_\alpha$ for which $T_\alpha(s', e) \neq T_\alpha(t', e)$. The table entries come only from membership queries, and thus e is a witness to $s' \not\equiv_{\mathcal{L}} t'$. As s and t are state-equivalent to s' and t', respectively, by construction of the transition function of $\mathcal{H}$ then too $s \not\equiv_{\mathcal{L}} t$ via the same witnessing suffix $e \in E_\alpha$. □

Theorem 72. *Given an observation table T, its hypothesis $\mathcal{H}$, we have that for every prefix $p \in \bigcup_{\alpha \in \mathsf{Pk}}(S_\alpha \cup S'_\alpha)$ we have that $\partial^+(p) = (\mathsf{last}(p), \mathit{row}_{\mathsf{last}(p)}(p))$.*

Proof. We prove this theorem by induction on the length of p. Let p have zero dups. Then $p = \alpha$ for some $\alpha \in \mathsf{Pk}$, and $\partial^+(\alpha) = q_0(\alpha) = (\alpha, \mathsf{row}_\alpha(\alpha))$. Now assume that for any $p \in \bigcup_{\alpha \in \mathsf{Pk}}(S_\alpha \cup S'_\alpha)$ with at most n dups we have that $\partial^+(p) = (\mathsf{last}(p), \mathsf{row}_{\mathsf{last}(p)}(p))$.

Consider a prefix $t \in \bigcup_{\alpha \in \mathsf{Pk}}(S_\alpha \cup S'_\alpha)$ with $n+1$ dups. Then $t = p \cdot (\alpha \cdot \mathsf{dup})$, for some prefix p and packet α. Notice now that p itself is also in the set of table access prefixes, as $\bigcup_{\alpha \in \mathsf{Pk}}(S_\alpha \cup S'_\alpha)$ is prefix-closed by definition. Therefore $\partial^+(t) = \partial(\partial^+(p), \alpha)$. By the inductive hypothesis, this is equal to $\partial((\mathsf{last}(p), \mathsf{row}_{\mathsf{last}(p)}(p)), \alpha) = (\alpha, \mathsf{row}_\alpha(p \cdot (\alpha \cdot \mathsf{dup}))) = (\mathsf{last}(t), \mathsf{row}_{\mathsf{last}(t)}(t))$. □

Lemma 73. *For each closed, consistent observation table T produced by the PNKA learner, and for any $\alpha, \beta \in \mathsf{Pk}$, if $\beta \cdot \mathsf{dup} \cdot e \in E_\alpha$ then $e \in E_\beta$.*

Proof. The claim is initially trivially satisfied because all E_α are initialized to contain dup-free suffixes. We only add to E_β in $\mathsf{mkConsistent}$. By inspection, we see that we only add a trace $\beta \cdot \mathsf{dup} \cdot e$ to E_α if $e \in E_\beta$, maintaining the invariant and proving the claim. □

Theorem 41. *Given a closed, consistent table T, $\mathcal{H} \triangleq \mathsf{hyp}_{\mathsf{PNKA}}(T)$ agrees on every cell of the packet tables. That is, $T_\gamma(\alpha \cdot s, e) = \mathcal{H}(q_0(\alpha), s \cdot e)$ for any $\alpha \cdot s \in S_\gamma \cup S'_\gamma$, $e \in E_\gamma$, and $\gamma \in \mathsf{Pk}$.*

Proof. Let $\alpha \cdot s \in S_\gamma$ and $e \in E_\gamma$. We proceed by induction on the number of dups in e.

For the base case, $e = \beta \in \mathsf{Pk}$. Then $\mathcal{H}(q_0(\alpha), s \cdot e) = \lambda(\partial^+(\alpha \cdot s), \beta)$. By Theorem 72, this equals $\lambda((\gamma, \mathsf{row}_\gamma(\alpha \cdot s)), \beta)$, which by definition of λ equals $T_\gamma(\alpha \cdot s, \beta)$.

Now let $e = (\beta \cdot \mathsf{dup}) \cdot e'$. Since $\alpha \cdot s \in S_\gamma$, we know that $\alpha \cdot s \cdot (\beta \cdot \mathsf{dup}) \in S'_\beta$. By closedness, there is a prefix $\xi \cdot s' \in S_\beta$ such that $\mathsf{row}_\beta(\alpha \cdot s \cdot (\beta \cdot \mathsf{dup})) = \mathsf{row}_\beta(\xi \cdot s')$. By Lemma 73, we know $e' \in E_\beta$. Observing that e' is smaller than e, we can reason:

$$
\begin{aligned}
\mathcal{H}(q_0(\alpha), s \cdot e) &= \mathcal{H}(q_0(\alpha), s \cdot (\beta \cdot \mathsf{dup}) \cdot e') && e = (\beta \cdot \mathsf{dup}) \cdot e' \\
&= \mathcal{H}(q_0(\xi), s' \cdot e') && \alpha \cdot s \cdot (\beta \cdot \mathsf{dup}) \equiv_{\mathcal{H}} \xi \cdot s', \textit{Theorem } 70 \\
&= T_\beta(\xi \cdot s', e') && \text{Induction hypothesis} \\
&= T_\beta(\alpha \cdot s \cdot (\beta \cdot \mathsf{dup}), e') && \mathsf{row}_\beta(\alpha \cdot s \cdot (\beta \cdot \mathsf{dup})) = \mathsf{row}_\beta(\xi \cdot s') \\
&= T_\gamma(\alpha \cdot s, e) && e = (\beta \cdot \mathsf{dup}) \cdot e'.
\end{aligned}
$$

□

Lemma 75. *Fix the target language $\mathcal{L}$, and suppose a learner conjectures a PNKA*

$\mathcal{H}$, and we get a counterexample $c = \mathsf{equiv}_{\mathsf{PNKA}}(\mathcal{H})$, i.e. $c \in \mathcal{L} \oplus \mathcal{L}(\mathcal{H})$. Then the next conjecture by the learner, $\mathcal{H}'$, will have at least one additional state compared to $\mathcal{H}$.

Proof. Let T' be the refined observation table giving rise to $\mathcal{H}'$. As proved by Theorem 41, both $\mathcal{H}$ and $\mathcal{H}'$ correctly represent the observation data in T and T', respectively. As c is a counterexample, $\mathcal{H}$ did not correctly classify it. As such, c must not have been part of the observation data in T. Through the counterexample handling process, c will be incorporated in the observation data of T', and as such, $\mathcal{H}'$ will correctly classify it.

As both $\equiv_{\mathcal{H}}$ and $\equiv_{\mathcal{H}'}$ are refined by $\equiv_{\mathcal{L}}$, but c does not have the same equivalence class in $\equiv_{\mathcal{H}}$ as it does in $\equiv_{\mathcal{H}'}$, and as $\mathcal{H}'$ still correctly classifies the observation data in T, it must be that $\equiv_{\mathcal{H}'}$ is then a strict refinement of $\equiv_{\mathcal{H}}$. As such, it must have at least one more equivalence class than $\equiv_{\mathcal{H}}$, and thus $\mathcal{H}'$ has at least one more state than $\mathcal{H}$. □

4.10.3 SPP Learning Correctness

Lemma 76. *Let e be an* EPP*, with the number of fields $|F| \geq 1$. For any $v \in V$, and let $s = \mathsf{SPP}(f, b, m, d) = \mathsf{select}\ v$ for e. Then for all (α, β) such that $[\![e]\!](\alpha, \beta) = \top$, we have $[\![s]\!](\alpha, \beta) = \top$, and for all (α, β) such that $[\![e]\!](\alpha, \beta) = \bot$, we have $[\![s]\!](\alpha, \beta) = \bot$.*

Proof. We will assume that any canonicalization of SPPs that is performed does not affect the SPP semantics, so we will reason about the *semantics* of a canonical SPP as though canonicalization does not occur (although the sizes and therefore

the choice the algorithm makes will of course depend on canonicalization). We proceed by induction on the number of fields in F (call this number n). For the base case, we consider $n = 1$, i.e. $F = \{f_1\}$. Let $[\![e]\!](\alpha, \beta) = \ell$. We have 3 cases:

- Suppose $\alpha.f_1 = v$ and $\beta.f_1 = v$. Then $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$. Thus $[\![d]\!](\alpha, \beta) = \top$ iff $\ell = \top$, and as a result $[\![s]\!](\alpha, \beta) = \top$ iff $\ell = \top$.
- Suppose $\alpha.f_1 = v$ but $\beta.f_1 \neq v$. Then $v \notin \mathsf{keys}(b)$, but by construction we have $m[\beta.f_1] = \ell$. and thus that $[\![s]\!](\alpha, \beta) = \top$ iff $\ell = \top$.
- Suppose $\alpha.f_1 \neq v$. Then $v \in \mathsf{keys}(b)$, and $\ell = b[\alpha.f_1][\beta.f_1]$ by construction. Thus $[\![s]\!](\alpha, \beta)$ iff $\ell = \top$, as needed.

The inductive cases are similar in structure; we just need to be a bit more careful in dealing with them. Fix $n > 1$.

- Suppose $\alpha.f_n = v$ and $\beta.f_n = v$. Then $v \notin \mathsf{keys}(b) \cup \mathsf{keys}(m)$. By induction, $\beta \in [\![d]\!](\alpha)$ iff $\ell = \top$, and as a result $\beta \in [\![s]\!](\alpha)$ iff $\ell = \top$.
- Suppose $\alpha.f_n = v$ but $\beta.f_n \neq v$. Then $v \notin \mathsf{keys}(b)$. By induction we have $[\![m[\beta.f_n]]\!](\alpha, \beta) = \top$ iff $\ell = \top$, and thus that $[\![s]\!](\alpha, \beta) = \top$ iff $\ell = \top$.
- Suppose $\alpha.f_n \neq v$. Then $\alpha.f_n \in \mathsf{keys}(b)$, and by induction $[\![b[\alpha.f_n][\beta.f_n]]\!](\alpha, \beta) = \top$ iff $\ell = \top$. Thus $[\![s]\!](\alpha, \beta) = \top$ iff $\ell = \top$, as needed.

□

Corollary 77. *Let e be an* EPP*, and let $h = \mathsf{hyp}_{\mathsf{SPP}}\ e$. Then for all (α, β) for which $[\![e]\!](\alpha, \beta)$ is defined, then $[\![e]\!](\alpha, \beta) = [\![\mathsf{hyp}_{\mathsf{SPP}}(e)]\!](\alpha, \beta)$.*

Proof. Observing that $\mathsf{hyp}_{\mathsf{SPP}}$ returns the result of select, the result follows from Lemma 76. □

Theorem 47. *The algorithm in Figure 4.3 terminates with a correct SPP.*

Proof. Let the teacher have SPP s. The learner only adds examples to its EPP (i.e. does not remove) and the labels are only assigned by the teacher, so the EPP maintained is always consistent with e. Because of these facts and the fact that Pk is finite, termination is trivial: the teacher must provide a different counterexample each time, and once the learner has seen all of $\mathsf{Pk} \times \mathsf{Pk}$, it must be correct on the next conjecture because of Corollary 77. □

4.10.4 SNKA Learning Correctness

Lemma 79. *For any partial observation table P, we have that $\mathsf{hyp}_{\mathsf{SNKA}}(P)$ is a well-defined SNKA.*

Proof. The finite set of states and start state are directly from the EA. The EA has only finitely many states because P is finite and so has at most finitely many distinguishing suffixes for each packet space. The transition function δ and observation function ε are well-defined by the definition of $\mathsf{hyp}_{\mathsf{SPP}}$, and moreover the transition function meets the deterministic restriction because this is established directly by the construction of δ' in the definition of sym. □

Lemma 80. *Let $\mathcal{E} = (Q, q_0, \delta, \varepsilon)$ be an EA for a consistent partial observation table P (i.e. $\mathcal{E} = \mathsf{ev}(P)$), and let $\mathcal{M} = (Q, q_0, \delta', \varepsilon) = \mathsf{sym}(\mathcal{E})$. For any $q, q' \in Q$ and any pair $(\alpha, \beta) \in \mathsf{Pk} \times \mathsf{Pk}$, then:*

(a) If $[\![\delta(q)(q')]\!](\alpha,\beta) = \top$*, then* $[\![\delta'(q)(q')]\!](\alpha,\beta) = \top$*.*

(b) If $[\![\varepsilon(q)]\!](\alpha,\beta) = \top$*, then* $[\![\varepsilon'(q)]\!](\alpha,\beta) = \top$*.*

Proof. Let $q, q' \in Q$ be arbitrary states. We prove each part separately:

(a) We note that by the definition of ev, we have $[\![\delta(q)(q')]\!](\alpha,\beta) = \top$ precisely when $q = \eta(\mathsf{row}_\alpha(w))$ for some $w \in \mathsf{Pre}$, and $q' = \eta(\mathsf{row}_\beta(w \cdot \beta \cdot \mathsf{dup}))$. Consistency of P prevents there being a w' for which $\mathsf{row}_\alpha(w') = \mathsf{row}_\alpha(w)$, and thus $\eta(\mathsf{row}_\alpha(w')) = \eta(\mathsf{row}_\alpha(w))$, but that $\mathsf{row}_\beta(w' \cdot \beta \cdot \mathsf{dup}) \neq \mathsf{row}_\beta(w \cdot \beta \cdot \mathsf{dup})$, which would mean $\eta(\mathsf{row}_\beta(w' \cdot \beta \cdot \mathsf{dup})) \neq \eta(\mathsf{row}_\beta(w \cdot \beta \cdot \mathsf{dup}))$ by restriction 2 of η (Section 4.5.4). As a result, there cannot be a $q'' \neq q'$ for which $[\![\delta(q)(q'')]\!](\alpha,\beta) = \top$. In fact, because of the addition of "determinizing pairs": for each $q'' \neq q'$, we have $[\![\delta(q)(q'')]\!](\alpha,\beta) = \bot$. Applying Corollary 77, we have that $[\![\mathsf{hyp}_{\mathsf{SPP}}(\delta'(q)(q'))]\!](\alpha,\beta) = \top$, and $[\![\mathsf{hyp}_{\mathsf{SPP}}(\delta'(q)(q''))]\!](\alpha,\beta) = \bot$ for any $q'' \neq q'$. The last step in finalizing δ' is to take differences to determinize (end of Section 4.5.4). But because we have just shown q to q' is *negative* for this transition, this will not affect our transition from q to q' on (α,β), and we have $[\![\delta'(q)(q')]\!](\alpha,\beta) = \top$ as needed.

(b) Follows immediately from Corollary 77.

□

Lemma 81. *Let P be a consistent partial observation table for target language $\mathcal{L} \subseteq \mathsf{GS}$ and $\mathcal{M} = \mathsf{hyp}_{\mathsf{SNKA}}(P) = (S, s_0, \delta, \varepsilon)$. Let $u \in \mathsf{GS}$. If for every "breaking point" of $u = w \cdot e$ there is a packet table $(S_\alpha, E_\alpha, T_\alpha)$ such that $w \in S_\alpha$ and $e \in E_\alpha$, then we have $\mathcal{M}(\eta(\mathsf{row}_\alpha(w), \alpha \cdot e) = \top \iff T_\alpha(w,e) = \top$.*

Proof. By induction on e. For the base case $e = \beta$ for $\beta \in \mathsf{Pk}$. Then:

$$\begin{aligned}
\mathcal{M}(\eta(\mathsf{row}_\alpha(w), \alpha \cdot e)) &\iff \mathcal{M}(\eta(\mathsf{row}_\alpha(w), \alpha \cdot \beta)) = \top && \text{base case} \\
&\iff [\![\varepsilon'(s)]\!](\alpha, \beta) = \top && \text{definition of } \mathcal{M} \\
&\iff [\![\varepsilon(s)]\!](\alpha, \beta) = \top && \text{by Lemma 80(b)} \\
&\iff T_\alpha(w, \beta) && \text{definition of } \mathsf{ev}
\end{aligned}$$

For the inductive case, $e = \gamma \cdot e'$, and let $s_\alpha = \eta(\mathsf{row}_\alpha(w))$, and $s_\gamma = \eta(\mathsf{row}_\gamma(w \cdot \gamma \cdot \mathsf{dup}))$. Then:

$$\begin{aligned}
\mathcal{M}(s_\alpha, \beta \cdot \gamma \cdot \mathsf{dup} \cdot e') &\iff \mathcal{M}(s_\gamma, \gamma \cdot e') && \text{Lemma 80(a) and} \\
& && [\![\delta'(s_\alpha)(s_\gamma)]\!](\alpha, \gamma) = \top \\
&\iff T_\gamma(w \cdot \gamma \cdot \mathsf{dup}, e') = \top && \text{Induction hypothesis} \\
&\iff T_\alpha(w, e) = \top && \text{definition of } e
\end{aligned}$$

□

Corollary 82. *After receiving a counterexample c, any subsequent hypothesis $\mathcal{M} = (S, s_0, \delta, \varepsilon)$ is correct with respect to the target $\mathcal{L}$ on c.*

Proof. Let $\alpha \cdot e = c$, and note that α, e satisfy the requirements of Lemma 81 by the definition of Section 4.5.4. So:

$$\begin{aligned}
\mathcal{M}(s_0, \alpha \cdot e) &\iff \mathcal{M}(\eta(\mathsf{row}_\alpha(\alpha)), \alpha \cdot e) = \top && \text{restriction 1 of } \eta \text{ (Section 4.5.4} \\
&\iff T_\alpha(\alpha, e) = \top && \text{Lemma 81} \\
&\iff c \in \mathcal{L}
\end{aligned}$$

□

Theorem 62. *For a regular language $\mathcal{L}$, `NKL`* (in Section 4.5.4) terminates with an SNKA for $\mathcal{L}$.*

Proof. Correctness follows from termination because we only terminate upon a successful equivalence query. There are 3 ways examples can be classified as wrong:

(1) The hypothesis merges two of the target's same-packet Myhill-Nerode classes into one state.

(2) The hypothesis has a wrong transition between classes because of missing data in the table.

(3) The hypothesis has a wrong observation of a class because of missing data in the table.

Note that the first of these is the only one possible in the canonical learner (Figure 4.1) because (2) is prevented by maintaining S'_α, and (3) is prevented because $\mathsf{Pk} \subseteq E_\alpha$. We have already shown each counterexample is classified correctly in subsequent hypotheses (Corollary 82). By Theorem 29, we can only correct finitely many errors of type (1). Suppose we have identified n Myhill-Nerode classes in the current partial observation table (i.e., there are a total of n distinct rows across all packet tables). We can only make $n \cdot |\mathsf{Pk}|$ errors of type (2) and $n \cdot |\mathsf{Pk}|$ errors of type (3). We conclude that we can receive only finitely many counterexamples before the hypothesis must be correct on the next conjecture. □

CHAPTER 5

LEARNING FINITE AUTOMATA WITH UNKNOWN INFORMATION

This chapter is adapted from the following published conference paper:
Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. Automata Learning with an Incomplete Teacher. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

5.1 Introduction

Many of the algorithms developed for closed-box inference, including our learners in Chapter 4, are based on the *minimally adequate teacher* (MAT) framework

Contrary to its name, however, in many practical settings, even finding a Minimally Adequate Teacher is challenging. For example, consider learning a model of a closed-box system. How would the Teacher determine whether a given automaton supplied by the Learner captures the behavior of the closed-box system? The best the Teacher can do is check whether they agree on a finite set of tests. Likewise, in settings where the Teacher only has access to a set of positive and negative examples (also known as passive learning) it is unclear how to answer membership queries for inputs that lie outside of the example set.

This chapter presents an alternative to MAT: the *incomplete* Minimally Adequate Teacher (iMAT) framework. An iMAT is allowed to answer "don't care" in response to membership queries and it does not answer equivalence queries at all. We show that Angluin's classic $\mathtt{L}^\star$ algorithm can be instantiated with an incomplete Teacher, by using an SMT solver to help fill in the missing information without needlessly expanding the size of the automaton. More precisely, we show—under modest assumptions—that our algorithm infers the minimal automaton compatible with the information provided by the Teacher. We present an OCaml prototype and demonstrate its behavior on well-known benchmarks [82, 96].

We see the framework developed in this chapter as a first step towards building connections between several different areas. As we explain later, iMAT provides a bridge between active and passive learning. In active learning, additional information can always be gathered (i.e., from the Teacher), whereas in passive

learning, the assumption is that all of the information that will be used for learning has been gathered in advance. With iMAT, one can use the passive information as an incomplete Teacher and proceed to learn the automaton using active techniques like $L^\star$. A lack of perfect information about the language in question also arises in program synthesis, in particular in the "programming by example" paradigm [33, 80, 82, 85, 121]. Here, the goal is to infer a model that is correct for the examples, and hope that it generalizes to other scenarios—i.e., that the examples are somehow representative of a more general pattern. We explore this connection in our use of benchmarks from AlphaRegex [82].

The problem of using finite information to infer a DFA was first studied by Gold in the 1970s [60], and has since been studied by many others [99, 97, 79, 78]. Gold showed that finding a DFA with the minimum number of states is NP-complete [61]. As iMAT subsumes DFA inference from finite data, its scalability is necessarily limited.

Others have studied problems similar to iMAT in prior work, either in their use of incomplete information or SAT solvers [34, 96, 64, 63, 84]. Leucker and Neider's paper [84] includes an "inexperienced teacher" and surveys several learners with similar capabilities as ours. Section 5.9 presents a detailed survey of their paper and other related work. No previous work has studied iMAT in full generality, including: formulating the problem, establishing correctness, building an open-source implementation, and exploring alternate formulations. Hence, compared to prior work, we make the following contributions:

1. We instantiate the iMAT framework in the context of DFAs, using an SMT solver to fill in missing information (Section 5.3).
2. We prove that the Learner terminates and returns a minimal automaton

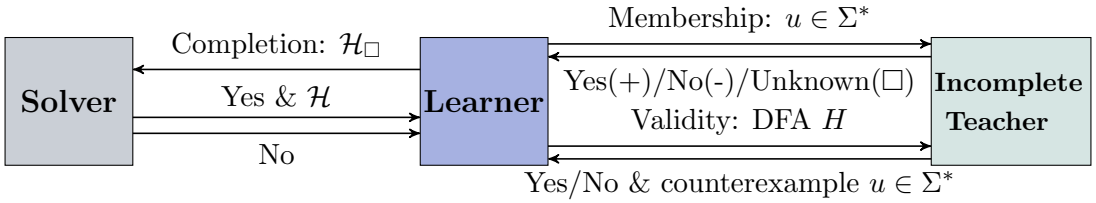


Figure 5.1: The Incomplete Minimally Adequate Teacher framework (iMAT).

compatible with the known information (Section 5.4).

3. We show how to modify our learning algorithm in the presence of the more limited Teacher that uses a simpler interface for compatibility checks (Section 5.5), and prove a (partial) correctness result (Section 5.6).

4. We implement the Learner in OCaml and present optimizations to accelerate the search for a DFA. We evaluate our proposed optimizations with an ablation study, and present performance data on a standard benchmark suite [82, 96].

In the rest of this chapter, we introduce an alternative to the MAT framework, in which the Teacher is incomplete—it might not have answers to all the membership questions—but there is a third agent in the process that can help the Learner guess the missing information. We will then devise, implement, and evaluate an algorithm based on $\mathtt{L}^\star$.

The challenges in devising and implementing such an algorithm are two-fold: on the one hand, since we do not have access to arbitrary membership queries, we will be building a table that has *holes*; on the other hand, the notion of minimality is now with respect to the existing information. Hence, progress towards this automaton is not as simple as in $\mathtt{L}^\star$, requiring the use of heuristics to guess the missing information while still minimizing the size of the final automaton.

5.2 The iMAT Framework

This section introduces iMAT, a new framework for automata learning in which the Teacher is *incomplete* (see Figure 5.1). In iMAT, the Learner still wants to infer a regular language L, but the teacher only has partial information about the language. In particular, instead of holding an explicit language L, the Teacher is assumed to hold disjoint, possibly infinite sets of positive L_+ and L_- negative examples. The Learner seeks to find any language L such that $L_+ \subseteq L$ and $L_- \cap L = \varnothing$. In the literature, such an L is said to *separate* L_+ and L_- [34]. We assume that at least one regular L exists, but there may be several.[1]

The iMAT framework has three components: a Learner, an incomplete Teacher, and a Solver. The Teacher answers two types of queries, like MAT, but with weaker assumptions:

Membership: The Learner sends a word u to the incomplete Teacher, who answers with "yes" (+), "no" (-), or "unknown" ($\square$).

Validity: Given a hypothesis automaton, $\mathcal{H}$, the teacher determines whether $L_+ \subseteq L(\mathcal{H})$ and $L_- \cap L(\mathcal{H}) = \varnothing$. If so, it returns *None*, otherwise, it returns a counterexample string either in $L_+ - L(\mathcal{H})$ or in $L_- \cap L(\mathcal{H})$.

The other component in the iMAT framework is the Solver which the Learner can ask for help in completing the hypothesis construction. The solver answers just one type of query:

Completion: The Learner sends an incomplete hypothesis to the solver and asks

[1]Note that we do *not* require that L_+ or L_- be regular. For example, neither $L_+ = a^n b^n, n > 0$ nor $L_- = b^n a^n, n > 0$ are regular, but they are separated by $L = a^* b^*$.

whether there is a way to complete it into a full automaton. The Solver either says "yes" and returns a complete hypothesis or "no".

iMAT's Validity query is obviously similar to MAT's Equivalence query, but it is different in an important way: an incomplete Teacher cannot return any string not in L_+ or L_- as counterexamples as it lacks information about those strings. Hence, a Validity query returns "yes" whenever there is no evidence against the hypothesis, which is subtly different than confirming the hypothesis directly. So while Validity may seem just as complex as Equivalence, we include it here as a first step toward more practical queries.

Remark 84. *Note that if the incomplete Teacher happens to be a complete oracle, then the iMAT setup can be used for MAT. Consider an iMAT instance where (i) the membership queries always return "yes" or "no" and (ii) the validity query returns a counterexample iff one exists. It follows that the hypothesis $\mathcal{H}$ can be built without ever calling the Solver (and with the same membership query complexity as MAT) and Validity queries correspond precisely to Equivalence queries.*

5.3 $\mathtt{L}^\star_\square$: an iterative iMAT Learner

We now present an iMAT Learner, closely following the approach used in $\mathtt{L}^\star$. In essence, looking back at the schema in Figure 5.1 we want to devise a learner that exploits the loop of membership-validity queries to promote a steady construction of a minimum-size automaton with the minimum amount of information. For the Solver component of iMAT we use an SMT solver (i.e., Z3 [41]).

The Learner holds a (sorted, in increasing size) worklist of candidate observa-

1. $\mathsf{worklist} \leftarrow [(\{\varepsilon\}, \{\varepsilon\})]$
2. Call to SMT solver to fill in $\square$s with $+$ or $-$ in $\mathbf{hd}(\mathsf{worklist})$.
   (a) if UNSAT then:
      i. $\mathsf{worklist} \leftarrow \mathbf{tl}(\mathsf{worklist})@[(S \cup \{s'\}, E) \mid s' \in S \cdot \Sigma - S]$
      ii. Goto step (2).
   (b) if SMT solver returns table (S, E), build the corresponding DFA and make a validity query.
      i. If the validity query succeeds, return the DFA.
      ii. Otherwise, get a counterexample c, set $E' = E \cup \mathsf{suffixes}(c)$, build table with $\square$'s (S, E') and set $\mathsf{worklist} \leftarrow (S, E') :: \mathsf{worklist}$; goto step (2).

Figure 5.2: $\mathtt{L}^\star$ with blanks algorithm, $\mathtt{L}^\star_\square$. We use '@' and '::' to denote standard list operations (i.e., append and cons). As before, we omit membership queries; these occur whenever S and E are changed. Since observation tables are fully determined by S and E, we elide T.

tion tables with $\square$'s. The algorithm works by constructing a worklist of potential hypotheses of increasing size, starting from a table with just ε as row and column labels.

We iteratively take each of these candidate tables and use an SMT solver to attempt filling in the $\square$'s (Completion) so that the table is closed and distinct (see Figure 5.3). Based on the result of the SMT query, we will either pass the completed table to the Teacher (Validity) or try another candidate table. If the Validity query succeeds, we return it. Otherwise, if we get a counterexample c, we refine the current hypothesis table by adding all the suffixes of c to E, and add the table (with $\square$'s) back to the worklist. The resulting iterative algorithm, $\mathtt{L}^\star$ with blanks ($\mathtt{L}^\star_\square$), is shown in Figure 5.2.

When the SMT solver finds that the instance is unsatisfiable, we know that the table cannot be made closed, but we do not know which s from $S \cdot \Sigma - S$ we need to promote to S to fix things as we would in classic $\mathtt{L}^\star$. So we make progress by

Given an observation table, (S, E, T):

1. Declare a Boolean variable b_s for each string $s \in (S \cup S \cdot \Sigma) \cdot E$ for which $T(s) = \square$.
2. For strings in L_+ or L_-, we fix the Boolean constraints:

$$\bigwedge_{s \in L_+} b_s = \text{true} \qquad \text{(Positive Evidence)}$$

$$\bigwedge_{s \in L_-} b_s = \text{false} \qquad \text{(Negative Evidence)}$$

3. Define a predicate to assert rows equal:

$$\text{Eq}(s, s') \triangleq \bigwedge_{e \in E} b_{se} = b_{s'e} \qquad \text{(Rows equal)}$$

4. Assert the table is closed:

$$\bigwedge_{s' \in S \cdot \Sigma - S} \bigvee_{s \in S} \text{Eq}(s, s') \qquad \text{(Closed)}$$

5. Define a predicate to represent the property that a row is unique (more precisely, that it is the first time the row appears):

$$\text{Unique}(s) \triangleq \bigwedge_{s' \in S \colon s' <_{\text{lex}} s} \neg \text{Eq}(s, s') \qquad \text{(Row unique)}$$

6. Declare a Boolean u_s for each string $s \in S$. We set u_s to true if $\mathsf{row}(s)$ is the first time that row appears:

$$u_s = \text{Unique}(s)$$

7. Assert the uniqueness of the rows in S (to maintain distinctness invariant):

$$\bigwedge_{s \in S} \text{Unique}(s)$$

Figure 5.3: SAT encoding for table constraints

extending separate "copies" of S with each string in $S \cdot \Sigma - S$, and adding these tables to the worklist.

	ε	a	aa	baa
ε	$\square$	-	$\square$	+
a	-	$\square$	-	$\square$
b	-	+	+	+
ba	+	+	+	$\square$
aa	$\square$	-	$\square$	$\square$
ab	$\square$	-	$\square$	$\square$
baa	+	+	$\square$	$\square$
bab	-	+	$\square$	$\square$
bb	-	+	+	$\square$

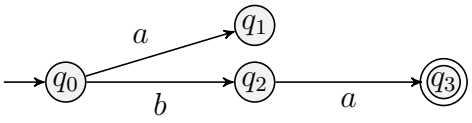


Figure 5.4: A partially filled table and its associated prefix-tree automaton

	ε	a	aa	baa
ε	$\boxminus$	-	$\boxminus$	+
a	-	$\boxminus$	-	$\boxminus$
b	-	+	+	+
ba	+	+	+	$\boxplus$
aa	$\boxminus$	-	$\boxminus$	$\boxplus$
ab	$\boxminus$	-	$\boxminus$	$\boxplus$
baa	+	+	$\boxplus$	$\boxplus$
bab	-	+	$\boxplus$	$\boxplus$
bb	-	+	+	$\boxplus$

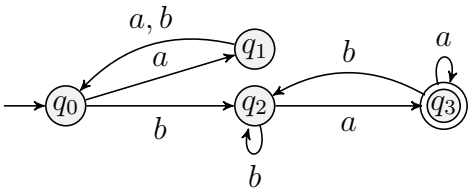


Figure 5.5: Filled-in table representing one way to complete the table shown in Figure 5.4

By adopting the Maler-Pnueli variation of $\mathrm{L}^\star$, the rows in S will always be distinct and correspond to nodes in a prefix tree. The worklist of tables with $\square$'s is providing an enumeration of the search space of prefix trees that correspond to DFAs of increasing size. The SMT solver will essentially be checking for every entry of the worklist whether, for a given prefix tree corresponding to a table, the other transitions not in the tree can be filled in to be consistent with the data. For example, consider the snapshot of a table in the worklist and the corresponding prefix tree, shown in Figure 5.4.

It turns out this table can be made closed and distinct—which indeed means the remaining transitions can be filled in. The resulting filled-in table and corresponding DFA is shown in Figure 5.5.

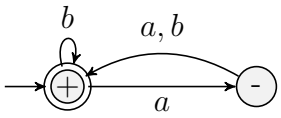
5.3.1 $\mathtt{L}^{\star}_{\square}$ Example

We illustrate the $\mathtt{L}^{\star}_{\square}$ algorithm (from Figure 5.2) with a teacher that starts with the following data: $L_+ = \{\epsilon, aa, ab\}$ and $L_- = \{a, bb, abb\}$. We begin with the following table on the left as the only item in the worklist.

	ε
ε	+
a	-
b	$\square$

	ε
ε	+
a	-
ab	+
aa	+
b	$\square$

	ε
ε	+
a	-
ab	+
aa	+
b	$\boxplus$



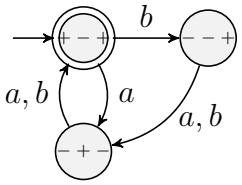
Next, we pop the table on the left off of the worklist and send it to the solver. We can see that even without filling in the blanks that the solver will return UNSAT (observe $\mathsf{row}(a)$ cannot possibly match $\mathsf{row}(\varepsilon)$). So we add a to S and b to S and append the two new tables to the worklist (which was empty). We pop the middle table above off of the worklist and send it to the SMT solver. The solver finds that filling the single blank with a + satisfies the constraints, resulting in the table on the right. We conjecture the corresponding machine on the right and get counterexample bb. We then add the suffixes of bb to E, and enqueue the horizontally extended (S, E, T) to the head the worklist. Thus we again pop the updated table and attempt to fill it in:

	ε	b	bb
ε	+	□	+
a	-	+	-
ab	+	-	□
aa	+	□	□
b	□	-	□

	ε	b	ab
ε	+	-	+
b	-	-	□
a	□	+	□
bb	-	□	□
ba	□	□	□
ab	+	-	□
aa	+	□	□

The table on the left is unsat. So we branch our search out by add ab, aa, and b to S on three new tables and append each of them to the worklist (which currently only holds the table with $S = \{\epsilon, b\}$). For brevity, we fast forward to the table from the last step where b was added, shown above on the right. We take the table off the worklist and the solver successfully fills in the blanks. We then conjecture the machine that corresponds to this table and this is correct on all data in L_+, L_- so the algorithm returns it.

	ε	b	ab
ε	+	-	+
b	-	-	⊞
a	⊟	+	⊟
bb	-	⊞	⊟
ba	⊟	⊞	⊟
ab	+	-	⊞
aa	+	⊟	⊞



The reader may have noticed that several times we naively added all row labels from the lower part of the table, when we could have localized the problem to a single row causing the unsat (we skipped ahead to these tables). This intuition leads

to an optimization based on unsatisfiable cores, which we describe in section 5.7.1.

5.4 Correctness of $\mathtt{L}^\star_\square$

In this section, we show that the Learner $\mathtt{L}^\star_\square$ we introduced in the previous section is correct: the algorithm terminates with the minimum size automaton compatible with L_+, L_- (Theorems 89 and 90). We first show that the search proceeds monotonically in the size of S (and therefore in the size of automata).

Lemma 85 (Size of work lists)**.** *The worklists generated in Figure 5.2 can contain tables of at most 2 sizes, n and $n+1$. When both are present, all the tables of size n are at the front.*

Proof. The initial worklist is $[(\{\varepsilon\}, \{\varepsilon\})]$. So there is one size, $|S| = 1$, and claim is trivially satisfied. As we enter the cycle with a worklist of size n in step (2) note that we add to the worklist in two places. In step $2(a)$, we add a list of items of size $n+1$ so the property is maintained in both cases (it may be that the item we popped was the last one of size n, in which case the list is now a single size). In step $2(b)$, we process the counterexample and add the table back to the front of the worklist—since we do not modify S this is an item of size n and he property is maintained. □

The following definition expresses the notion that an observation table is "on the path" to identifying some target DFA, which will be useful shortly:

Definition 86 (Compatible Observation Table)**.** *An observation table (S, E, T) is* compatible *with a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if:*

1. *The function* $q\colon S \to Q$ *defined by* $q(s) = \hat{\delta}(q_0, s)$ *is injective,*
2. *The blanks can be filled in* by M *to make the rows of* S *distinct, and*
3. *For each* s *such that* $T(s) \neq \square$*, then* $T(s) = +$ *if and only if* $s \in L(M)$*. (i.e.* M *agrees with the non-blank entries of* (S, E, T)*).*

In other words, a table which is compatible with a target DFA can be extended to find the target DFA by adding zero or more strings to S while maintaining the invariant that each row corresponds to a unique state. The next lemma ensures that the worklist will always have some compatible table with a minimum DFA.

Lemma 87. *Let* $M = (Q, \Sigma, \delta, q_0, F)$ *be a minimum-size DFA consistent with* L_+, L_-*. All worklists generated in the algorithm in Figure 5.2 contain at least one table* (S, E, T) *that is compatible with* M*.*

Proof. First observe that M has no unreachable states since it is minimal. Initially, the table $(\{\varepsilon\}, \{\varepsilon\})$ is compatible with M since (a) any function from $S = \{\varepsilon\}$ is injective and (b) there is only one row, so it is distinct. We now check that the algorithm maintains the invariant of having a compatible table in the worklist. As we enter the cycle with a worklist of length n, the only interesting case is when the compatible table (S, E, T) is the head of the list, since in all other cases it will clearly still be on the worklist at the next iteration. We proceed to analyze what happens while processing (S, E, T):

- In Step 2(a), we know that (S, E, T) could not be made made closed by filling in blanks, but by induction hypothesis that the rows of S are distinguished by M. Hence, for at least one $s \in S \cdot \Sigma - S$ we have that s accesses a new state in M (if this were not the case, then SAT instance could be satisfied using M

as an oracle, since then each row in the lower part would match the row in the upper part corresponding to the state accessed). By the same argument, after adding this s to S, we know it will still be distinguishable from the other rows, because if it were not, then using M as an oracle would have satisfied the core. Hence, the table obtained by adding s to S is compatible with M.

- Step 2(b)(i) does not return, so we need not maintain the invariant.
- In Step 2(b)(ii), we do not change S, so that condition (a) of Definition 86 is immediate. Any rows which can be distinguished by M using only the suffixes in E are still distinguished by additional suffixes in E'. So (S, E', T') is compatible with M. □

We now prove a lemma which illustrates that the crux of the search is finding the correct prefix set, S. At this point, we will make a validity query (possibly incorrectly—but the table will be satisfiable) until we are correct. If we have a particular oracle in mind (i.e., any correct automaton), then the lemma says that if S contains accessor prefixes for the states of the oracle (and E is large enough to distinguish states), the table will be satisfiable. To be clear, in the algorithm we course do not use an oracle to fill in the blanks—we use an SMT solver as we do not yet know the automaton. The point is simply that the satisfiability of the SMT instance when a solution exists will be used to establish correctness.

Lemma 88 (Existence of a solution)**.** *Let L_+, L_- be example sets, and let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a minimum state DFA consistent with L_+, L_-. Let (S, E, T) be an observation table that is compatible with $\mathcal{A}$ and such that $|S| = |Q|$. Then there is a closed, distinct, complete observation table (S, E, T') where $T'(s) = T(s)$ for all $T(s) \neq \square$.*

Proof. Use the DFA $\mathcal{A}$ as an oracle to complete the table: let $T'(s) = +$ if $s \in L(\mathcal{A})$, otherwise $T'(s) = -$. Since $\mathcal{A}$ is consistent with the datasets, so $T(s) = T'(s)$ whenever $T(s) \neq \square$.

(closed) Let $s \in S, a \in \Sigma$. If $sa \in S$, we are done, so assume $sa \notin S$. Let $q = \hat{\delta}(q_0, sa)$. By hypothesis, there is a string $s' \in S$ such that $\hat{\delta}(q_0, s') = q$. Then for each $e \in E$, $T'(sae) = \hat{\delta}(q_0, sae) = \hat{\delta}(\hat{\delta}(q_0, sa), e) = \hat{\delta}(q, e) = \hat{\delta}(\hat{\delta}(q_0, s'), e) = \hat{\delta}(q_0, s'e) = T'(s'e)$. Thus $\mathsf{row}(sa) = \mathsf{row}(s')$.

(distinct) By the fact that (S, E, T) is compatible with $\mathcal{A}$, we have that using $\mathcal{A}$ to fill in the blanks of T distinguishes all of the rows labeled by S. $\square$

Theorem 89 (Partial Correctness)**.** *If the algorithm terminates, then it returns a DFA of minimal size consistent with L_+, L_-.*

Proof. Let M be any minimal DFA for L_+, L_+. Then by Lemma 87, at each point there is an observation table on the worklist that is compatible with M. Assume the algorithm eventually terminates. In the worst case, it is when we find the table (S, E, T) with $|S| = n$, for n the number of states of M, (and potentially with $E = \mathsf{suffixes}(L_+ \cup L_-)$). By Lemma 85, we traverse the worklist in nondecreasing order. Hence, if we terminated earlier, it was by conjecturing a correct machine smaller than n, which is a contradiction. $\square$

All that remains to be shown is that we make progress towards this solution and indeed the algorithm terminates.

Theorem 90 (Termination)**.** *The $\mathtt{L}^\star_\square$ algorithm terminates.*

Proof. Let M be a minimal DFA for L_+, L_-, and let (S, E, T) be a table in the worklist compatible with M guaranteed to exist by lemma 87. Step 2(a) makes

progress by increasing the size of S. Moreover, since there are finitely many automata M' whose size is less than or equal to M, there are only finitely many counterexamples we can get in Step 3(b) (because we never again misclassify an example we have seen). Each time we reach the compatible table, we enqueue a table with larger size by 1. When, at latest, we reach the compatible table whose size matches M, then we may need many, but only finitely many more validity queries before we are correct and terminate. □

5.5 Weakening the Teacher: iMAT with Distinguish

Next, we explore another kind of incomplete teacher that is not required to implement Validity queries, but only a weaker set of Distinguish queries. Two strings $s_1, s_2 \in \Sigma^*$ are said to be *distinguished* if there exists $e \in \Sigma^*$ such that,

- $s_1 e \in L_+$ and $s_2 e \notin L_+$ (or vice versa), or
- $s_1 e \in L_-$ and $s_2 e \notin L_-$ (or vice versa).

Membership: As before, the Learner sends a string u and the Teacher responds with "+" if $u \in L_+$, "−" if $u \in L_-$, and "□" otherwise.

Distinguish: The learner sends two strings s_1 and s_2, together with a finite "exclusion set" E, and the teacher responds "yes" with a suffix $e \notin E$ that distinguishes these two strings or "no" if it cannot distinguish the strings.

It should be clear that Distinguish queries are less powerful than Validity queries, as they only concern a pair of strings and not the full language. However, it turns out that when L_+ and L_- are finite, we can adapt the learner to

still have a terminating procedure to learn a correct automaton. We prove these results in Section 5.6.

iMAT with Distinguish has close ties to foundational concepts in formal languages. In particular, the distinguish query is closely related to the Myhill-Nerode equivalence classes $\equiv_L$ of the target language L.

Lemma 91. *Given strings $s_1, s_2 \in \Sigma^*$ with distinguish $\varnothing$ s_1 s_2 $= e$ for some $e \in \Sigma^*$, then there exists some language L that separates L_+ and L_- such that $s_1 \not\equiv_L s_2$. Moreover if, in addition, the result of membership queries to $s_1 e$ and $s_2 e$ are not $\square$, then for* every *L separating L_+ and L_-, we have that $s_1 \not\equiv_L s_2$.*

Proof. For the second claim, suppose $T(s_1 e) \neq T(s_2 e)$ and both are not $\square$. Without loss of generality, $T(s_1 e) = +$ and $T(s_2 e) = -$. Let L be any language separating L_+ and L_-. Clearly $s_1 e \in L$ and $s_2 e \notin L$. That $s_1 \not\equiv_L s_2$ follows from the definition of the Myhill-Nerode equivalence. Now for the first part, assume $T(s_1 e) = -$ and $T(s_2 e) = \square$ (the other cases are similar), and assume L is a regular language that separates L_+ and L_-. Then $L' = L \cup \{s_2 e\}$ is also regular and separates L_+ and L_-. Moreover, for this L we have $s_1 \not\equiv_L s_2$. $\square$

Although iMAT with Distinguish is guaranteed to terminate only when L_+ and L_- are finite, we can show that arbitrary problem instances can be modeled using finite instances. Hence, finite example sets are in a sense complete. More formally, suppose we are given possibly infinite sets of positive and negative examples L_+ and L_-. Then we can find finite subsets of L_+ and L_- that are sufficient for the Learner to recover the minimum-size DFA compatible with L_+ and L_-.

Theorem 92. *Let $L_+, L_- \subseteq \Sigma^*$ be infinite, disjoint example sets (not necessarily regular), and let M be a minimum-size DFA that separates L_+ and L_-. Then*

there are finite *subsets* L_+, L_- *such that* M *is also a minimum-size DFA separating* L_+, L_-.

Proof. Consider the set $\mathcal{M}$ of all DFAs over Σ that have strictly fewer states than M. (Observe that $\mathcal{M}$ is potentially very large, but finite). By the minimality of M on L_+, L_-, each $M' \in \mathcal{M}$ misclassifies a string from either L_+ or L_-. So we can define a function $c\colon \mathcal{M} \to \Sigma^*$ that maps each M' to one such counterexample string. Then the set $S = \{s \mid s = c(M'), M' \in \mathcal{M}\}$ is finite (in fact, $|S| \leq |\mathcal{M}|$). Moreover, we have by construction that for $S_+ = L_+ \cap S$ and $S_- = L_- \cap S$ that each $M' \in \mathcal{M}$ fails to separate S_+, S_- by misclassifying $c(M')$, so that M is a minimum-size separating automaton for S_+, S_- as needed. □

5.6 Correctness of $\mathtt{L}^\star_\square$ with Distinguish

We now show how to adapt the $\mathtt{L}^\star_\square$ to the situation that we only have an iMAT with Distinguish, not Validity. The idea is that we perform $\mathtt{L}^\star_\square$ as before, but when we have a hypothesis machine, we do a series of distinguish queries instead of a validity query. Intuitively, we will ask a distinguish query for each pair of rows which are "made the same" by the hypothesis:

Definition 93 (Distinguish queries for a hypothesis)**.** *Given a closed, distinct, complete table,* (S, E, T) *and associated DFA* M*, we say the* distinguish queries for hypothesis M *are:*

$$\textit{distinguish } E \ s's$$

for each $s' \in S \cdot \Sigma - S$ *and for the unique* s *such that* $\mathsf{row}(s) = \mathsf{row}(s')$.

Observe these distinguish queries are well-defined because each $s' \in S \cdot \Sigma - S$ is guaranteed to have a unique matching row in S.

The first theorem shows that if all of the distinguish queries for a hypothesis return None, then the hypothesis is correct.

Theorem 94 (Validity using Distinguish)**.** *Suppose a learner has a hypothesis DFA $M = (Q, \Sigma, \delta, q_0, F)$ constructed from a closed, distinct, complete observation table (S, E, T). Suppose also that the teacher returns None for all of the distinguish queries for M. Then $L_+ \subseteq L(M)$ and $L(M) \cap L_- = \varnothing$.*

Proof. We show the contrapositive. Suppose there is a counterexample string $c \in L_+$ such that $c \notin L(M)$ (without loss of generality; the other case is similar). It suffices to show that there is a distinguish query that is not None. First of all c cannot be in T since we inherit from $\mathtt{L}^\star$ that hypotheses are correct on all evidence already considered. So we must have $c = s_0' v_0$ for a prefix $s_0' \in S \cdot \Sigma - S$, since $S \cdot \Sigma - S$ includes exactly one prefix of each word not in $S\Sigma$. By closedness of the table, there is an $s_0 \in S$ such that $\mathsf{row}(s_0') = \mathsf{row}(s_0)$. Consider distinguish $E\ s_0\ s_0'$. If this query returns a suffix, we are done, so assume it is None. In particular, this means that $s_0 v_0 \in L_+$, since otherwise v_0 would be a response to the query. Moreover, since $\mathsf{row}(s_0) = \mathsf{row}(s_0')$ we must have that $\hat{\delta}(\mathsf{row}(s_0), v_0) = \hat{\delta}(\mathsf{row}(s_0'), v_0)$, implying that $s_0 v_0 \notin L(M)$ and $s_0 v_0$ is also a counterexample. So, as before, there must be an $s_1' \in S \cdot \Sigma - S$ and suffix v_1 such that $s_0 v_0 = s_1' v_1$. Crucially, s_0 (as $s_0 \in S$) is a proper prefix of s_1', making v_1 a proper suffix of v_0. Since, again, there must be an $s_1 \in S$ for which $\mathsf{row}(s_1) = \mathsf{row}(s_1')$ We proceed by considering the query distinguish $E\ s_1\ s_1'$ and applying the same reasoning, except that we have made progress because v_1 is a suffix of v_0. Clearly we will see a distinguishing suffix after a finite number of these iterations: in particular by the time that $v_i = \varepsilon$, then

$s'_i v_i = s'_i \in S \cdot \Sigma - S$. But that means s'_i is a string we have already seen and is also a counterexample (i.e., $s'_i \in L_+$ but rejected), which is impossible by the construction of the table (which means the distinguishing suffix must be at latest the previous round). □

We modify $\mathtt{L}^\star_\square$ as follows:

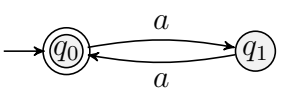
1. Run $\mathtt{L}^\star_\square$ until ready to make a validity query for hypothesis H.
2. In place of the validity query, make the distinguish queries for H.
   (a) If they are all None, stop and return H.
   (b) Otherwise, add the first suffix returned by the teacher to E, and return to Step 1.

Corollary 95. *If the modified* $\mathtt{L}^\star_\square$ *terminates, the learned machine is correct.*

Proof. This follows immediately from theorem 94. □

Of course, this modified procedure is not guaranteed to terminate in general. For some sparse, infinite L_+ and L_-, it is possible for the teacher to have infinitely many suffixes that the learner must consider without making progress. For example, if we have $L_+ = (aa)^*$ and $L_- = (aaaa)^*a$, the learner will soon reach:

	ε
ε	+
a	-
aa	+



At this point, the Learner will ask the query distinguish $E\ \varepsilon\ aa$ and the trouble is that the teacher has an infinite set of suffixes to give in response to distinguish $E\ \varepsilon\ aa$, namely those strings in the set $(aa)^*a$. As a result, the procedure above will not terminate in this case. We are unaware of a query weaker than Validity that still guarantees termination—settling this is an interesting question for future work.

We do have, however, as a special case, guaranteed termination for finite example sets.

Corollary 96. *Let L_+ and L_- be finite, disjoint sets of strings. A learner can still learn a correct automaton for L_+, L_- with only* membership *and* distinguish *queries.*

Proof. Run $\mathtt{L}^\star_\square$ modified for distinguish. Because of theorem 94, we need only show that it terminates. Each time through Step 2 that we do not terminate, the teacher returns a suffix e. But since we add e to E, the teacher can never return this suffix again, and since the examples are finite, they can only have finitely many suffixes. Termination follows. □

5.6.1 A Landscape of Teacher Queries

As we have already presented two variants of iMAT, it is natural to ask how it relates to other formulations of automata learning problems. In this section, we summarize some known results for a variety of queries provided by a Teacher including:

Query name	Type
membership	$\Sigma^* \to 2$
equivalence	$\text{DFA} \to \Sigma^* + 1$
$\text{membership}_\square$	$\Sigma^* \to \{+, -, \square\}$
distinguish	$\mathcal{P}_{\text{fin}}(\Sigma^*) \times \Sigma^* \times \Sigma^* \to \{+, -, \square\}$
validity	$\text{DFA} \to \Sigma^* + 1$

Recall that though equivalence and validity have the same type, they are provided by Teachers with different capabilities: an incomplete Teacher cannot return any string not in L_+ or L_- as counterexamples as it lacks information about those strings. Hence, a Validity query returns "yes" whenever there is no evidence against the hypothesis, which is subtly different than confirming the hypothesis directly, which is what a complete teacher proves through an Equivalence query.

We distinguish two classes of learning problems. For the first class, the Teacher knows a specific regular language and will not return "unknown" in response to Membership queries. Conversely, in the second class, the teacher is incomplete and may return "unknown" in response to Membership queries.

Teacher holds concrete regular language (i.e., no blanks)

Only Membership. Suppose we run $\mathtt{L}^\star$, but replace each equivalence query with a loop that queries all strings looking for a counterexample. It is a consequence of the Myhill-Nerode Theorem that this process will eventually discover the correct automaton. However, the Learner will never be able to determine that it has converged. If the Learner is told the number of states, a terminating procedure is possible [6].

Only Equivalence. We can obtain a trivial terminating procedure with only the equivalence query simply by conjecturing all automata in increasing order of states. Angluin [8] showed that we cannot improve the learner to a polynomial time procedure.

Membership and Equivalence. This is Angluin's MAT: $\mathtt{L}^\star$ is an efficient terminating procedure for determining the correct DFA [7]. See Section 2.2.

Teacher is incomplete (i.e., may return blanks)

Only Membership. Just as in the non-blanks case, a learner can identify a minimum-size correct machine eventually, but cannot determine that it has done so—so there is no terminating procedure. This is Gold's notion of "identification in the limit" [59].

Membership and Validity. Because this problem subsumes all instances of DFA inference from finite data, it is NP-Hard [61] and cannot be approximated in polynomial time (unless $P = NP$) [99]. We give a terminating procedure, $\mathtt{L}^\star_\square$, in section 5.3 and prove it correct in section 5.4. The complexity comments from "Only Equivalence" also hold here.

Membership and Distinguish. We consider a modification of $\mathtt{L}^\star_\square$ which produces a minimum-size automaton if it terminates. We explore this in section 5.5.

5.7 Implementation

We have built an OCaml implementation of the learner (using Validity) presented in this chapter. The goal of our implementation is to provide a reusable imple-

mentation of our framework in a functional language that can be used to explore the iMAT framework, run our algorithm, and implement other algorithms that use similar primitives. Our implementation consists of two packages: (i) `nerode`, a general library for regular languages, with data structures for regular expressions, DFAs, and NFAs, as well as conversions between them, and (ii) `nerode-learn`, which implements Angluin's $L^\star$ algorithm, as well as the iMAT variants developed in this chapter. The `nerode` package is approximately 2,400 lines of code, while the `nerode-learn` package is approximately 3,000 lines of code. We use Z3 as the backend solver for checking all SMT problems.

Data Structures Our OCaml implementation uses the following types.

- Strings are encoded in a module called `Word`, where their representation is a list of Alphabet symbols. Alphabet symbols internally are `int`.[2]

  ```
  type Word.t = Alphabet.symbol list
  ```

  In particular, defining modules for Word and Alphabet allowed for a modular design and allows us to support different alphabets.

- The rows of the observation table (or more precisely, the labels of the rows) are encoded as a set of Words:

  ```
  type RowLabels.t = WordSet.t
  ```

- The columns of the observation table are also encoded as a set of words:

  ```
  type ColLabels.t = WordSet.t
  ```

- Individual entries in the observation table are encoded using the following data type:

  ```
  type entry = True | False | Blank
  ```

[2]We abuse OCaml syntax slightly: `type Word.t` is not a valid type definition, but we keep the module names to show our module boundaries, and concisely differentiate all of our types `t`.

Here, the constructors include `Plus`, `Minus`, and `Blank`.

- Thus we can encode the actual lookup function of the table as a map from words to entries:

```
type TblMap.t = entry WordMap.t
```

Putting these all together, an observation table is encoded as a record s, sa, e, t where s, sa, and e are sets of strings, and $tmap$ is a map from strings to entries:

```
type Table.t = {s : RowLabels.t; sa : RowLabels.t;
                e : ColLabels.t; tmap : TblMap.t}
```

When implementing algorithms on observation tables, there is an interesting data structure choice to be made. An obvious approach is to keep the entire table explicitly as a two-dimensional array. The advantage to this approach is that the row function corresponds precisely to rows in the array. The downside is that to update the entry for a string, one has to visit each location in the table corresponding to how the string can be decomposed into prefix and suffix. To avoid this, instead of keeping the whole table, we keep S, $S \cdot \Sigma - S$, and E as sets and T as a map from strings in $(S \cup S \cdot \Sigma) \cdot E$ to table entries. Thus entries for a string are updated everywhere they appear by a single update to T. The downside is that we must generate the row function "on the fly" by repeatedly accessing T.

Algorithms We now present our OCaml implementation of $\mathtt{L}^\star_\square$. We encode the worklist as a list of tables, which is maintained as an argument in the main recursive loop. A collection of all entries in the observation tables is also maintained globally as an argument in the loop.

```
let wl : Table.t list = ...
let entry_map : entry WordMap.t = ...
```

The main algorithm is a recursive function that searches for the smallest table that is compatible with the given positive and negative examples:

```
let rec lstar_blanks wl (e: ColLabels.t) entry_map: Dfa.t =
  let t, entry_map = List.hd wl |> Table.extend_cols entry_map e
    in
  match fill_blanks_smt t with
  | None →
    let new_ts, entry_map' =
        List.fold
            ~f:(fun (ts_acc, em_acc) sa →
                let new_t, em_acc' = (Table.move_up em_acc sa t)
    in
                new_t::ts_acc, em_acc')
            ~init:([], entry_map) (Table.lower_rows t) in
    lstar_blanks (wl @ new_ts |> List.tl) e entry_map'
  | Some obs →
    let dfa = table_to_dfa obs in
    match conjecture dfa with
    | None → dfa
    | Some cex →
        lstar_blanks wl (ColLabels.union e (suffixes cex))
     entry_map
```

For the most part, the OCaml implementation follows the pseudocode given in Figure 5.2, but there are a few differences. In particular, the columns of the tables (`e`) in the worklist are not updated until right before we attempt to fill them in. We keep a cumulative mapping from strings to table entries (`entry_map`) in the main loop, preventing redundant membership queries, and we also accumulate the suffixes learned from counterexamples (`e`) in the main loop. Intuitively, this approach is sound because it is safe to share counterexamples across tables and thus have the same column labels (`e`) for each (see Lemma 98 below).

Operationally, the `lstar_blanks` function proceeds as follows. First, it removes an item t from the worklist. It adds additional suffixes in e to the table t, using membership queries to determine new column entries in e that cannot be found in *entry_map*, filling in t with `Plus`, `Minus`, or `Blank`; it also updates the entry collection *entry_map* with any new queries. Next, it attempts to fill the blanks in the observation table encoded by t, calling `fill_blanks_smt`, which uses use an SMT solver as discussed below. If the SMT solver cannot find values for the blanks, the algorithm extends the worklist with each new table *new_t* obtained

by adding one row from the bottom part of the table t, using `Table.add`, and recurses. It also updates the *entry_map* with any new query results while filling in new rows. Otherwise, the blanks can be filled, and it makes a conjecture to the teacher. If the conjecture is correct, we have the minimal DFA. Otherwise, the conjecture fails, so we add the suffixes for the new counterexample to e, keep t at the head of `wl`, and recurse.

Efficient SMT Encoding The key to achieving good performance in any solver-aided algorithm, is having an efficient encoding of observation tables. We use the theory of bit-vectors, which is widely supported by SMT solvers, to express the blank-filling constraints. Specifically, given a table with k columns, we encode the rows in the table as bit-vector variables of length k. We add assertions to constrain the bit-vectors so they correspond to the rows in the table[3]. For example, the bit-vector variable for the jth row in the table would be declared as follows,

```
(declare-const s_j (_ BitVec k))
```

and if the entry in the ith column was positive, we would add an assertion:

```
(assert (= ((_ extract i i) s_j) #b1))
```

Alternatively, if the entry in the ith column were a blank, we would declare a bit-vector variable of length 1 to encode the blank,

```
(declare-const b_ij (_ BitVec 1))
```

and add the corresponding constraint:

```
(assert (= ((_ extract i i) s_j) b_ij))
```

Hence, a model of the SMT formula corresponds to a way of filling in the blanks that is consistent with the positive and negative entries in the table.

[3]An earlier version used SAT, with individual entries as boolean variables, but we found that using the bit-vector theory performed better.

Returning to our algorithm, recall that we need the rows in the upper part of the table to be distinct, and we need the table to be closed. For distinctness, we use the built-in `distinct` function from the bit-vector theory. For closedness, we generate assertions stating that each row in the bottom part of the table must be equal to some row in the upper part.

For example, given the observation table on the left:

	ε	a
ε	+	□
a	□	-
b	□	+
aa	-	□
ab	□	□
ba	+	-
bb	□	□

$\Longrightarrow$

	ε	a
ε	+	⊟
a	⊟	-
b	⊞	+
aa	-	⊟
ab	⊞	⊟
ba	+	-
bb	⊞	⊟

Our SMT encoding would have 7 bit-vectors of size 2, one for each row, and 7 bit-vectors of size 1, one for each (unique) blank:

```
(declare-const s1 (_ BitVec 2)) (declare-const b1 (_ BitVec 1))
(declare-const s2 (_ BitVec 2)) (declare-const b2 (_ BitVec 1))
(declare-const s3 (_ BitVec 2)) (declare-const b3 (_ BitVec 1))
(declare-const s4 (_ BitVec 2)) (declare-const b4 (_ BitVec 1))
(declare-const s5 (_ BitVec 2)) (declare-const b5 (_ BitVec 1))
(declare-const s6 (_ BitVec 2)) (declare-const b6 (_ BitVec 1))
(declare-const s7 (_ BitVec 2)) (declare-const b7 (_ BitVec 1))
```

Note that in the table above, the blank in the first row, second column and the blank in the second row, first column must have the same value as they both encode the membership of the string a in the language. Our SMT encoding account for such constraints between blanks. Next, we add assertions to encode the entries in the table:

```
(assert(=((_ extract 1 1) s1) #b1)) (assert(=((_ extract 2 2) s1
   ) b1))
(assert(=((_ extract 1 1) s2) b1))  (assert(=((_ extract 2 2) s2
   ) #b0))
(assert(=((_ extract 1 1) s3) b2))  (assert(=((_ extract 2 2) s3
   ) #b1))
(assert(=((_ extract 1 1) s4) #b0)) (assert(=((_ extract 2 2) s4
   ) b3))
(assert(=((_ extract 1 1) s5) b4))  (assert(=((_ extract 2 2) s5
   ) b5))
(assert(=((_ extract 1 1) s6) #b1)) (assert(=((_ extract 2 2) s6
   ) #b0))
(assert(=((_ extract 1 1) s7) b6))  (assert(=((_ extract 2 2) s7
   ) b7))
```

Note that the assertions reflect the fact the blanks in the first rows must be filled in with the same value, since they are both associated with the string "a". We would also assert distinctness for the upper part,

```
(assert(distinct s1 s2 s3))
```

Finally, we would add assertions for closedness:

```
(assert(or (= s1 s4) (= s2 s4) (= s3 s4)))
(assert(or (= s1 s5) (= s2 s5) (= s3 s5)))
(assert(or (= s1 s6) (= s2 s6) (= s3 s6)))
(assert(or (= s1 s7) (= s2 s7) (= s3 s7)))
```

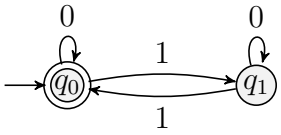
To fill in the rows in table, we can simply read off values assigned to the variables,

```
(define-fun b1 () (_ BitVec 1) #b0)
(define-fun b2 () (_ BitVec 1) #b1)
(define-fun b3 () (_ BitVec 1) #b0)
(define-fun b4 () (_ BitVec 1) #b1)
(define-fun b5 () (_ BitVec 1) #b0)
(define-fun b6 () (_ BitVec 1) #b1)
(define-fun b7 () (_ BitVec 1) #b0)
```

which corresponds to the table on the right above.

Teacher Module Because our design relied on an abstract interface to the teacher, it was straightforward to implement the teacher in different ways. While the benchmarks we used for evaluation (Section 5.8) are lists of labeled examples, we also implemented a feature that allows the user to specify infinite L_+ and

L_- using regular expressions. For example, we could specify $L_+ = (0101)^*$ and $L_- = 10^*$, and the system correctly identifies:



Note that the correct language has a smaller DFA than the one for L_+!

5.7.1 Optimizations

We now look at several optimizations we implemented. The first one will reduce the number of tables generated in step $2(a)$ of Figure 5.2 by making use of unsat-cores; the second prevents the learner from making needlessly similar conjectures by reusing counterexamples that it gets from the teacher, and the third explores the use of a priority queue instead of a list.

Unsat-cores When a table's blanks cannot be filled in by the SMT solver we explore the tables that result by adding each string from $S \cdot \Sigma - S$ to S. In many cases, however, we might be able to identify a row, say $s' \in S \cdot \Sigma - S$ that is already distinguished from every row in S by suffixes in E. In this case, any filling of the blanks would result in an unclosed table *because of this row*, so we can promote *only* this one, as in classic $\mathtt{L}^\star$.

To generalize this, we observe that the closedness assertion corresponding to the row s' above was unsatisfiable on its own. Modern SMT solvers can be configured to compute subsets of constraints that are unsatisfiable alone, called unsat cores,

when returning "unsat" [86]. To justify that we are still guaranteed to find the minimal automaton boils down to extending the argument of Lemma 87:

Lemma 97. *The invariant of Lemma 87 holds, even when* $\mathtt{L}^\star_\square$ *is modified to add only tables resulting from promoting only strings in unsat core.*

Proof. The argument for Lemma 87 applies, but we need to justify that a string in the unsat core accesses a new state of M. This must be true, or else using M as an oracle would fill in the blanks to satisfy the unsat core. □

Suffix-set Sharing The reader may have noticed that the $\mathtt{L}^\star_\square$ Algorithm described above cleverly reuses counterexamples, accumulating the suffix set E. Hence, instead of maintaining a worklist with elements (S, E, T), we only need a worklist that maintains prefix sets S for each individual table. For T, we maintain a cumulative mapping of strings to entries as a parameter in the main loop and simply update missing entries for columns of the popped table on each iteration. For E, we prove that sharing counterexamples is sound:

Lemma 98. *Let* M *be a DFA, and let* (S, E, T) *be compatible with* M *and let* (S, E', T') *be a table such that* $E \subseteq E'$*. Then* (S, E', T') *is also compatible with* M*, where* T' *is the extension of* T *by membership queries for* E'*.*

Proof. Condition (a) of compatibility is immediate since S is unchanged. If M distinguishes the upper part rows with only the suffixes in E, they remain distinguished by E', implying condition (b). Condition (c) holds by hypothesis. □

Corollary 99. *The algorithm in Figure 5.2 is still guaranteed to produce minimal machines when modified to share a single* E *across the worklist.*

Proof. Sharing E means instead of adding a counterexample to the current item's E, traverse the worklist updating all E. By Lemma 98 adding additional strings to E does not affect compatibility, and so the invariant of Lemma 87 still holds and minimality follows. □

Heuristic prioritization Our algorithm as described above uses a list that is sorted only by the size of the prefix set. By instead using a priority queue, we can apply heuristics to investigate tables which are more likely to be compatible with a minimal DFA (in the sense of Definition 86). Because we search monotonically by size, this trick can only save effort on the "last size searched," since all smaller automata are checked first. Still, the number of prefix sets grows rapidly enough that the savings on even this last size justify the optimization.

5.8 Evaluation

To evaluate the performance of our implementation, we timed it on a portion of the benchmark sets created by Oliveira and Silva [96] (for their system BICA) and the benchmarks of Lee, So, and Oh[4] [82]. We present summary data of these runs in Table 5.1 and Figure 5.6. Table 5.1 shows that median total learning times were consistently shorter than the mean total learning times, suggesting that, at each size, the more expensive examples are less common. The mean Z3 time column suggests the system spends around two-thirds of its running time in SMT solving at all sizes. The last column shows how the number of worklist items processed increases with DFA solution size. All benchmarks use the alphabet $\Sigma = \{0, 1\}$. We ran the experiments on a 2.10GHz Intel Xeon Silver 4216 machine with 512GB

[4]We omit benchmark #9, which has a DFA solution of size 16.

Benchmark set	DFA Size	Number of benchmarks	Mean Learn time (s)	Median Learn time (s)	Mean Z3 Time (s)	Mean worklist items
Lee, So, and Oh	2	1	0.0082	0.0082	0.0081	2.0000
	3	10	0.0234	0.0229	0.0226	5.0000
	4	9	0.0600	0.0417	0.0529	7.5556
	5	4	0.1676	0.1360	0.1401	19.0000
Oliveira and Silva	1	80	0.0056	0.0053	0.0055	1.0000
	2	15	0.0100	0.0093	0.0097	2.0667
	3	91	0.0226	0.0216	0.0213	4.0879
	4	84	0.0396	0.0338	0.0346	5.7619
	5	100	0.1237	0.0775	0.0935	8.6200
	6	105	0.3803	0.1684	0.2566	13.6381
	7	79	1.1251	0.6419	0.7583	20.0886
	8	93	10.6307	1.7830	6.1782	44.1613
	9	74	50.1672	7.7361	28.8381	96.8784
	10	25	98.0573	7.3782	65.2680	176.5200
	11	14	1498.4836	101.2503	988.9716	933.2857

Table 5.1: Performance results on established benchmarks. "Learn time" is the total time spent on an individual benchmark, while "Z3 time" is the time spent on a benchmark *within the SMT solver*. "Worklist items" are the number of tables processed from the worklist during learning.

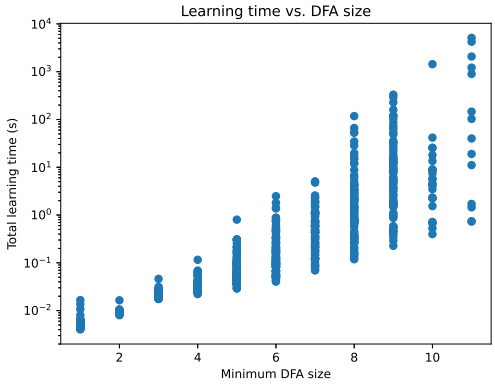


Figure 5.6: Performance on Oliveira and Silva benchmarks generated by automata of sizes 4 to 11.

of memory.

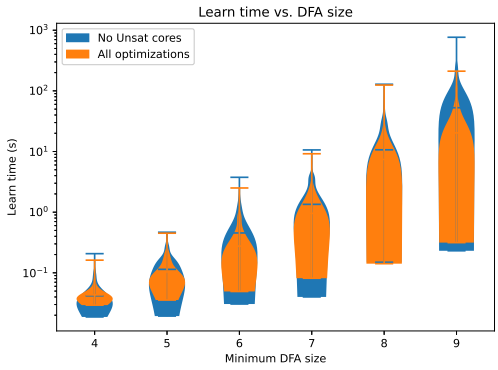
Oliveira and Silva's benchmark set is large: for each size from 4 states to 23 states, they produced 19 random DFAs. For each DFA, there are 5 sets of positive and negative example strings. Each example set is a set of 20 strings of length 30 produced by random walks on the generated DFA. Each problem also contains all the prefixes of those 20 strings. Using this process, they generated a total of 95

problems using each size of random DFA. Importantly, as a result of this generation process, it is very often the case that there is a smaller consistent machine than the one used in generation. In Table 5.1, summary data is presented for benchmarks generated from size 4 to size 11 machines, however, note that for Table 5.1 and Figure 5.7, the size shown is the size of the *learned* (i.e., minimum-size) automaton, not the one used to generate the examples.

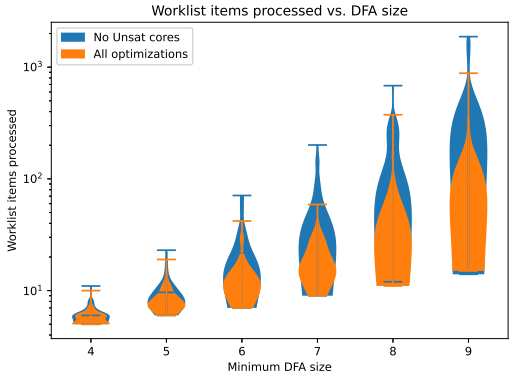
From the results of our experiments, we conclude that the scalability of our system is limited. It is practical in cases where the DFA to be learned is of size 11 or smaller. However, in its current form, learning larger DFAs is prohibitively expensive (see Section 5.10 for a discussion of future work in this direction). In contrast, Heule and Verwer [64] report solving all of the Oliveira and Silva benchmarks for sizes 4 to 21 "within 200 seconds per instance," although their method requires access to all of the finite examples at the start.

5.8.1 Evaluation of optimizations

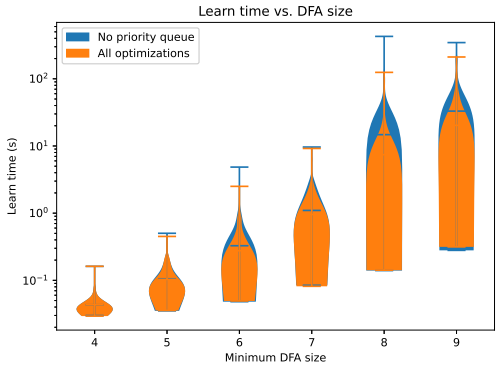
We evaluated the optimizations from Section 5.7.1 by conducting an ablation-style study, presented in Figure 5.7. The ablation study was performed in two stages. First, we ran the system on the Oliveira benchmarks (generated by sizes 4 to 9) with all optimizations turned on. Then, we ran the system on the same benchmarks a separate time for each of the three optimizations, with the optimization turned off. The results show significant time and search space savings for the unsat-cores optimization and heuristic prioritization. The "suffix-set sharing" optimization cannot affect the number of worklist items processed and turned out to result in an entirely negligible time improvement (these figures are omitted).



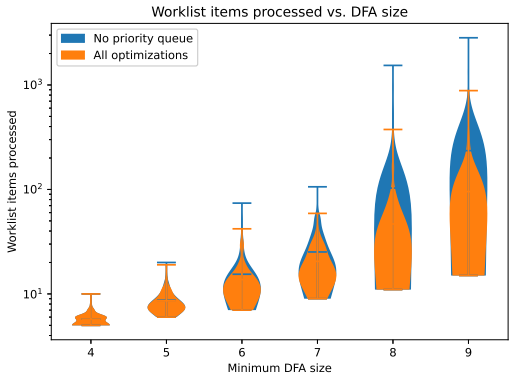
(i) Time reduction from unsat core optimization



(ii) Search space reduction of unsat core



(iii) Time reduction from heuristic prioritization.



(iv) Search space reduction of heuristic prioritization.

Figure 5.7: Ablation study of optimizations proposed in Section 5.7.1. Data for "Suffix-set sharing" is omitted because it produced only negligible speedup.

5.9 Related Work

DFA Inference from finite data. Inference of DFAs from finite data is a long standing problem and different solutions have appeared in the literature (see e.g [40] for an overview). Gold introduced the observation table and considered blank entries ("holes") in the context of passive learning, but his algorithm does not guarantee minimality of the automaton produced [60]. Modern algorithms attack the problem from the perspective of "state merging" [97]. The main idea of these approaches is to build an automaton from the tree of all prefixes of positive examples, called the prefix tree accepter (PTA), and then attempt to merge states, using the

negative examples to validate merges. It is invariant that the positive examples are accepted, but a merge might cause a negative example to be accepted—in which case it is backtracked. Much research has gone into the selection of which merges to prioritize [79, 36]. State merging methods are much faster than the algorithm described in this chapter but in general do not guarantee minimum size. A notable exception to this is the **exbar** Algorithm due to Lang [78], which gives a method for exhaustively exploring potential merges in a way that finding a minimum size DFA is guaranteed at exponential running time cost.

Inference of Separating Automata. Chen et al [34] use a similar notion of blanks in $\mathtt{L}^\star$ (which they call "Don't care"). The problem they solve is closely related, but it requires that the teacher start with two *regular* languages, for which they seek to find a minimal separating DFA. The algorithm they present, L^{sep}, computes the final DFA by first computing a 3-valued automaton (3DFA), which has "Don't care" as output. They then convert this automaton to the minimum-size consistent DFA. Crucially, the interface they define with the teacher requires that the positive and negative example sets be regular languages. Thus in light of this restriction, the problem solved by L^{sep} may be seen as a special case of the iMAT setting.

DFA Inference using SAT solvers. Oliveira and Silva use constraint solvers to find a mapping from the states of a PTA to a particular size [96]. If the search fails, the size is increased until a DFA is found. Their work extends the method of Biermann and Feldman [21], who first explored such mappings from the PTA. Heule and Verwer [64] also use SAT solvers to infer DFAs from examples but their approach uses a SAT formulation closely tied to graph coloring. They first build a compatibility graph for states of the PTA, where states are compatible if their merge is not immediately ruled out by examples. The colors assigned in the

coloring instance thus correspond to states in the smaller DFA, with the edges of the incompatibility graph ruling out incorrect merges.

Active learning of Network Invariants. Grinchtein, Leucker, and Piterman [63] present an algorithm that augments $L^\star$ to handle missing information with a SAT solver for inferring network invariants—i.e., in the sense of Wolper and Lovinfosse's inductive technique verifying for large compositions of finite-state systems [115]. Their work, which improves the earlier method of Pena and Oliveira [98], extends the Angluin notions of table closedness and consistency to tables with blanks (called "weakly closed" and "weakly consistent"). The learner can proceed by performing $L^\star$ corresponding actions on tables while there are still blanks. Once the table is weakly closed and weakly consistent, they produce a series of SAT queries following Biermann and Feldmen's approach ([21], also mentioned below). The result of these SAT queries is a minimum-size automaton consistent with the examples in the table—in their scheme, they do not maintain the invariant that the minimum automaton is equal in size to the upper part of the table as in our approach, so they do not necessarily conjecture hypotheses monotonically.

In follow-on work Leucker and Neider [84] presented a framework which distills the essentials of the algorithm of [63] for learning DFAs. Their formal framework presents the teacher similar to our presentation in Section 5.2, but they do not highlight the solver as a first-class citizen in the framework and think of it as part of the Learner. They do give an overview of potential learners that work with inexperienced teachers. These encompass an approach without membership queries (a naive enumeration of all DFAs of increasing size) and the approach of [63] with a SAT solver, as well as the approach of [34], which we described above. They do not consider any modification similar to our iMAT with Distinguish, and

they do not explore any implementation aspects or benchmarking.

5.10 Discussion

We have presented algorithms that solve the problem of automata inference from an incomplete teacher. The core ideas we applied to produce our algorithm are data structure agnostic and therefore a first direction for future work is whether we can adapt the work we did in this chapter to recent optimizations of $\mathtt{L}^\star$, such as $L^\sharp$.

A particularly interesting optimization is the use of *discrimination trees*, an efficient replacement for observation tables [70, 67]. The first challenge will be to understand how the operations on discrimination trees can be generalized in the setting with $\Box$'s.

Another interesting direction would be to look at $\mathtt{L}^\star$ variants developed for other automata models. We expect that the work in this chapter applies directly to Mealy and Moore machines, with the caveat that the number of options for filling in blanks will be affected by the size of the output sets. But one could also explore how SMTs solvers could help in learning of weighted and symbolic automata, two models with interesting applications for which $\mathtt{L}^\star$-like algorithms were proposed [20, 13, 43, 11].

Finally, there is another $\mathtt{L}^\star$ adaptation, due to Bollig et al., that learns nondeterministic finite automata with access to an Angluin MAT [22]. Another interesting direction for future work would be to investigate the adaptation of their approach to the incomplete setting.

5.11 Another $L^{\star}_{\square}$ Example

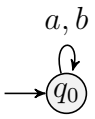
Suppose we start with the following input.

$$L_+ = \{ab, aab, bab, aaab, abab, baab, bbab\}$$
$$L_- = \{aa, ba, bb, aaa, baa, aba, bba, abb, bbb\}$$

We begin with the following table as the only item in the worklist.

	ε
ε	$\square$
a	$\square$
b	$\square$

	ε
ε	$\boxminus$
a	$\boxminus$
b	$\boxminus$



We pop this table off of the worklist, and the SMT solver attempts to fill in its blanks. The solver finds that filling all the blanks with a - satisfies the constraints. We then conjecture the machine that corresponds to the filled-in table.

We get the counterexample *baab*, add its suffixes to E, and push the new (S, E, T), below on the left, onto the top of worklist.

	ε	b	ab	aab	$baab$
ε	$\square$	$\square$	+	+	+
a	$\square$	+	+	+	$\square$
b	$\square$	-	+	+	$\square$

	ε	b	ab	aab	$baab$
ε	$\square$	$\square$	+	+	+
a	$\square$	+	+	+	$\square$
aa	-	+	+	$\square$	$\square$
ab	+	-	+	$\square$	$\square$
b	$\square$	-	+	+	$\square$

Now the table on the left is unsatisfiable. Hence, we add each element in $S\Sigma - S$ to S and add each resulting (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, a\}$ (table depicted on the right above), and the last item has $S = \{\varepsilon, b\}$. Once again, the solver fails because there is no way to fill in the blanks and maintain closedness. Once more, we add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, b\}$, the next item has $S = \{\varepsilon, a, aa\}$, and the last item has $S = \{\varepsilon, a, ab\}$.

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
b	□	-	+	+	□
a	□	+	+	+	□
ba	-	+	+	□	□
bb	-	-	+	□	□

We pop the table above off of the worklist. Again the solver fails because there is no way to fill in the blanks and maintain closedness. We add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, a, aa\}$, the next item has $S = \{\varepsilon, a, ab\}$, the next item has $S = \{\varepsilon, b, ba\}$, and the last item has $S = \{\varepsilon, a, bb\}$.

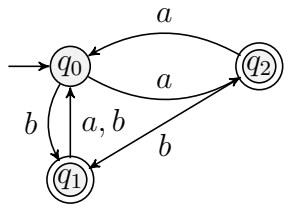
In the next step, we pop the table below on the left off of the worklist. Unsat again. We add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. At the head of the list we now have the table on the right.

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
a	□	+	+	+	□
aa	-	+	+	□	□
aaa	-	+	□	□	□
aab	+	□	□	□	□
ab	+	-	+	□	□
b	□	-	+	+	□

	ε	b	ab	aab	$baab$
ε	□	□	+	+	+
a	□	+	+	+	□
ab	+	-	+	□	□
aa	-	+	+	□	□
aba	-	+	□	□	□
abb	-	□	□	□	□
b	□	-	+	+	□

This time SMT solver successfully fills in the blanks:

	ε	b	ab	aab	$baab$
ε	⊟	⊞	+	+	+
a	⊞	+	+	+	⊞
ab	+	-	+	⊞	⊞
aa	-	+	+	⊞	⊞
aba	-	+	⊞	⊞	⊞
abb	-	⊞	⊞	⊞	⊞
b	⊞	-	+	+	⊞

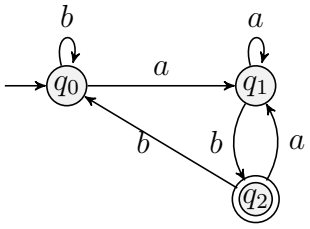


We get counterexample bbb, add its suffixes to E, and the head of the worklist becomes:

	ε	b	ab	aab	$baab$	bb	bbb
ε	□	□	+	+	+	-	-
a	□	+	+	+	□	-	□
ab	+	-	+	□	□	□	□
aa	-	+	+	□	□	□	□
aba	-	+	□	□	□	□	□
abb	-	□	□	□	□	□	□
b	□	-	+	+	□	-	□

Finally the SMT solver successfully fills in the blanks:

	ε	b	ab	aab	$baab$	bb	bbb
ε	⊟	⊟	+	+	+	-	-
a	⊟	+	+	+	⊞	-	⊟
ab	+	-	+	⊞	⊞	⊟	⊟
aa	-	+	+	⊞	⊞	⊟	⊟
aba	-	+	⊞	⊞	⊞	⊟	⊟
abb	-	⊟	⊞	⊞	⊞	⊟	⊟
b	⊟	-	+	+	⊞	-	⊟



The corresponding automaton is consistent with L_+, L_- so we return it as the result.

CHAPTER 6

DISCUSSION

In the preceding chapters, we have made contributions to automata-based verification of networks. We have addressed a performance gap (by introducing symbolic verification algorithms) and a lack of system models (by introducing symbolic learning algorithms). Of course, much work remains to extend the variety settings in which our techniques can be used. To conclude the dissertation, we describe several challenges that we view as the open frontier of improving the state of automata-based verification of networks.

Challenge 1: Further performance improvements. With KATch (Chapter 3), we made asymptotic improvements to the performance of the NetKAT decision procedure by introducing symbolic techniques. Unfortunately, we already started to see the limitations of this approach on the largest of the benchmark topologies (around 800 switches). Modern network engineers control hosts with thousands of devices, and it seems clear that KATch will not scale to this size without additional fundamental insights.

One interesting approach would be to employ methods to exploit symmetry in the configurations. If we can check NetKAT equivalence up to a collection of abstract field values, we may be able to reuse results on large subnets. An idea along this lines is to apply *abstract interpretation*. While KATch is only able to check a snapshot of a network's dataplane, Beckett et. al [18] explored using abstract interpretation to perform verification of the *control plane*. It would be interesting to explore such an approach to scaling dataplane verification.

In KATch, we only studied batch queries where NetKAT automata of each policy were constructed from scratch at query time. Of course, a common use case in practice would involve repeated queries as a network policy is developed

incrementally. From this standpoint, it makes sense to try to take advantage of the fact that many policy updates will be small. Applying this insight might mean that we only have to pay a high up-front cost to verify the initial policy once. Xu et al. have undertaken an interesting extension of NetKAT they call Relational NetKAT [117] pursuant to this idea. They rely on some key insights from KATch for their primary verification technique. However, just as KATch takes advantage of the fact that policies make mostly small updates to packets, Relational NetKAT exploits the fact that *updates to policies themselves* are usually also small. As a result, they derive an incremental verification procedure that runs faster in subsequent runs after small changes are made to a verified program.

While the work in Chapter 4 takes a promising *theoretical* step towards learning NetKAT automata, there is further work required to make it useful in a practical setting. Initial investigation suggests that the engineering effort required to build a basic implementation is modest, but that a performant one will require answering additional research questions. To name a few:

- Part of the engineering process for building a practical implementation of `NKL`* is to leverage the idea of *abstract domains* to reduce the packet space by disregarding unnecessary or unimportant header fields. Can we *learn* the appropriate abstraction?
- What network telemetry is the minimum needed to build accurate dataplane models?
- How can we accommodate gaps in the observed packets to fill in needed information for the models?

The answers to these questions will of course hinge on the particular network

topologies involved and what the learner knows a priori.

Challenge 2: Verify more expressive programs. KATch (Chapter 3) only allows us to answer *qualitative* yes or no questions about network behaviors such as reachability or slice isolation queries. In many practical cases, a network engineer might be interested in *quantitative* properties of their network. For instance, we might want to ask "what is the least cost route between hosts A and B?" or "what is the probability that a packet is routed through a particular waypoint?" These questions require reasoning about NetKAT expressions with *weights*, which cannot be done by KATch. Suárez Acevedo et al. have explored *weighted NetKAT* in a general way in a recent project called wNetKAT[105]. The approach taken by wNetKAT is appealing because it abstracts the quantitative computation by operating on a semiring of weights, which is provided as a parameter to the construction. After choosing this semiring, they decide properties of weighted NetKAT programs based on establishing bounds, such as "Do all paths have weight at most r?" or "Is there a path with weight at least r?" These verification queries are not based on deciding equivalence as in KATch, since equivalence is not guaranteed to be decidable for all semirings. Of note, wNetKAT strictly generalizes NetKAT, in the sense that the boolean semiring can be supplied, and regular NetKAT is recovered. At the same time, the work in their paper is theoretical, and the topic deserves additional study in terms of building efficient, symbolic implementations for answering these verification queries.

Jacobs et al. have explored an extension of NetKAT that allows reasoning about stacks of values associated with each packet, which they call *StacKAT* [68]. This is a natural extension of NetKAT because it means that we can write verification queries that consider parsing of the packets, source routing, and tunneling, to name

several examples. In addition to the standard syntax of NetKAT (Table 2.1), they support two primitives, $\mathsf{push}(v)$ and $\mathsf{pop}(v)$, which push a value onto the stack and pop (and match) a value from the stack, respectively. Remarkably, they show that equivalence is still decidable under these additional operations. Still, their implementation is naive. It remains an interesting question how symbolic techniques similar to those employed in KATch in Chapter 3 can be brought to bear to build a more efficient decision procedure for StacKAT.

Challenge 3: Learn more expressive models. The work on *weighted* NetKAT mentioned under the previous heading provokes a question about whether such weighted models can be learned in the style of `NKL`* from Chapter 4. This is interesting from a practical perspective for similar reasons to those motivating `NKL`*. Namely, if we want to perform quantitative verification checks of a network model, we must have a faithful model at the start. A natural way to develop this model would be to learn one from observations. In fact, for some quantitative properties such as latency, making observations may be the *only* sensible way of determining weight values, since the weights in such cases would not appear in any policy text. There is existing work due to Balle on learning weighted automata that will no doubt be relevant to such an effort [13]. Indeed, this area is active, with work by Ferreira et al. exploring learning of weighted automata with the aid of SMT solvers in a paper that will appear at Computer Aided Verification (CAV) 2026 [44]. Extending insights from this work to wNetKAT will not be straightforward because of the carry-on packet semantics of SNKAs (as was the case for developing `NKL`* in Chapter 4).

Challenge 4: Weakening Assumptions for the Learner The work on iMAT in Chapter 5 provides an initial investigation into learning with incomplete teachers, but it does not address learning NetKAT automata in such a setting. Indeed, learning in networks certainly have problems like this, since we cannot always observe faithfully all of the packets that we might like. Thus, combining the insights of $\mathtt{L}^\star_\Box$ and $\mathtt{NKL}^\star$ might yield a more robust learner in practice. Of course, it seems possible that iMAT might still not be quite the right model for the teacher in the networking domain. For example, how do we distinguish between a drop (i.e., "no" answer) and a blank (i.e., "don't care" answer)? Further investigation of appropriate learning *frameworks* is also worthwhile.

Similar to the way iMAT explored teachers that can answer with blank, it would be valuable to explore learning network models from teachers whose answers can *change*. After all, the policy in place may be modified by a configuration update during the learning process! Ferreira et al. explore a framework for learning classic finite automata in this setting [46]. It is not at all clear that adapting their ideas to $\mathtt{NKL}^\star$ would be straightforward, but it would allow us to learn models under more realistic assumptions.

Conclusion Through adoption of these techniques, we envision a world where essentially all network routes are automatically verified before or shortly after installation. In such a world, we relegate network failures due to misconfiguration to the past. Each of the directions here would be an exciting next step towards realizing this world!

BIBLIOGRAPHY

[1] F. Aarts, J. De Ruiter, and E. Poll. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 461–468, March 2013.

[2] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and Abstraction of the Biometric Passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, Lecture Notes in Computer Science, pages 673–686, Berlin, Heidelberg, 2010. Springer.

[3] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. SwitchV: automated SDN switch validation with P4 models. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM 2022, page 365–379, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2005, page 98–109, New York, NY, USA, 2005. Association for Computing Machinery.

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2014)*, 2014.

[6] Dana Angluin. A note on the number of queries needed to identify regular languages. *Inf. Control.*, 51:76–87, 1981.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[8] Dana Angluin. Negative results for equivalence queries. *Mach. Learn.*, 5(2):121–150, jul 1990.

[9] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, Mar 1996.

[10] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, page 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] George Argyros and Loris D'Antoni. The learnability of symbolic automata. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 427–445, Cham, 2018. Springer International Publishing.

[12] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[13] Borja Balle and Mehryar Mohri. Learning weighted automata. In Andreas Maletti, editor, *Algebraic Informatics*, pages 1–21, Cham, 2015. Springer International Publishing.

[14] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[15] Ryan Beckett, Michael Greenberg, and David Walker. Temporal netkat. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 386–401, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, 2017.

[17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 476–489, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2020)*, 2020.

[19] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, 2016.

[20] Francesco Bergadano and Stefano Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM J. Comput.*, 25(6):1268–1280, dec 1996.

[21] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines

from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, jun 1972.

[22] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of nfa. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 1004–1009, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[23] Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. In *Proceedings of the 40th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2013)*, 2013.

[24] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, Jul 2014.

[25] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd D. Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of the 2023 Conference of the ACM Special Interest Group on Data Communication*, 2023.

[26] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[27] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sep 1992.

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

[29] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *ACM/IEEE Symposium on Logic in Computer Science*, 1990.

[30] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning extended finite state machines. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 250–264, Cham, 2014. Springer International Publishing.

[31] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clement Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. Formally verified cloud-scale authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2508–2521, 2025.

[32] Sagar Chaki and Arie Gurfinkel. Bdd-based symbolic model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 219–245. Springer International Publishing, 2018.

[33] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 487–502, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating DFA's for compositional verification. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[35] Zhang Cheng, Jiyang Wu, Di Wang, and Qinxiang Cao. Denotation-based compositional compiler verification. *ACM Trans. Program. Lang. Syst.*, 48(1), March 2026.

[36] Orlando Cicchello and Stefan C. Kremer. Beyond edsm. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications*, ICGI 2002, page 37–48, Berlin, Heidelberg, 2002. Springer-Verlag.

[37] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *Proceedings of the 41st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2014)*, 2014.

[38] Loris D'Antoni and Margus Veanes. Forward bisimulations for nondeterministic symbolic finite automata. In *Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205*, page 518–534, Berlin, Heidelberg, 2017. Springer-Verlag.

[39] Loris D'Antoni and Margus Veanes. The power of symbolic automata and

transducers. In *Computer Aided Verification*, pages 47–67, Cham, 2017. Springer International Publishing.

[40] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[41] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[42] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. Leapfrog: Certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.

[43] Samuel Drews and Loris D'Antoni. Learning symbolic automata. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 173–189, 2017.

[44] Tiago Ferreira, Kevin Batz, and Alexandra Silva. Smt-based active learning of weighted automata. In *Computer Aided Verification*. Springer International Publishing, 2026.

[45] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM 2021, page 762–774, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Tiago Ferreira, Léo Henry, Raquel Fernandes da Silva, and Alexandra Silva. Conflict-aware active automata learning. *Electronic Proceedings in Theoretical Computer Science*, 390:150–167, 09 2023.

[47] Dana Fisman, Hadar Frenkel, and Sandra Zilles. Inferring symbolic automata. Volume 19, Issue 2:8899, 2023.

[48] Paul Fiterău-Broştean and Falk Howar. Learning-based testing the sliding window behavior of tcp implementations. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification*, pages 185–200, Cham, 2017. Springer International Publishing.

[49] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning fragments of the tcp network protocol. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems*, pages 78–93, Cham, 2014. Springer International Publishing.

[50] Paul Fiterău-Broştean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.

[51] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W.

Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017.

[52] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 469–483, USA, 2015. USENIX Association.

[53] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 279–291, New York, NY, USA, 2011. Association for Computing Machinery.

[54] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, page 282–309, Berlin, Heidelberg, 2016. Springer-Verlag.

[55] Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *Proceedings of the 42nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2015)*, 2015.

[56] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards Formally Verified Optimizing Compilation in Flight Control Software. In Philipp Lucas and Reinhard Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*,

volume 18 of *Open Access Series in Informatics (OASIcs)*, pages 59–68, Dagstuhl, Germany, 2011. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[57] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[58] Ricard Gavaldà and David Guijarro. Learning ordered binary decision diagrams. In *Proceedings of the 6th International Conference on Algorithmic Learning Theory*, ALT 1995, page 228–238, Berlin, Heidelberg, 1995. Springer-Verlag.

[59] E. Mark Gold. Language identification in the limit. *Inf. Control.*, 10:447–474, 1967.

[60] E. Mark Gold. System identification via state characterization. *Automatica*, 8(5):621–636, sep 1972.

[61] E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[62] Michael Greenberg, Ryan Beckett, and Eric Campbell. Kleene Algebra Modulo Theories: A framework for concrete KATs. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.

[63] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 483–497. Springer-Verlag, 2006.

[64] Marijn J. H. Heule and Sicco Verwer. Exact DFA identification using sat solvers. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, pages 66–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[65] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *USENIX Conference on Security*, 2011.

[66] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. 1971.

[67] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734, pages 307–322. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science.

[68] Jules Jacobs, Nate Foster, Tobias Kappé, Dexter Kozen, Lily Saada, Alexandra Silva, and Jana Wagemaker. Stackat: Infinite state network verification. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[69] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 9, USA, 2012. USENIX Association.

[70] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.

[71] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Computer Communications Review*, 42(4):467–472, Sep 2012.

[72] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, Oct 2011.

[73] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 14–33, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[74] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997.

[75] Dexter Kozen. Automata on guarded strings and applications. Technical report, USA, 2001.

[76] Dexter Kozen and Maria-Christina Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proceedings of the First International Conference on Computational Logic*, CL '00, page 568–582, Berlin, Heidelberg, 2000. Springer-Verlag.

[77] Loes Kruger, Sebastian Junges, and Jurriaan Rot. Small test suites for active automata learning. In *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part II*, page 109–129, Berlin, Heidelberg, 2024. Springer-Verlag.

[78] Kevin J. Lang. Faster algorithms for finding minimal consistent DFAs. Technical report, NEC Research Institute, 1999.

[79] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference*, ICGI 1998, page 1–12, Berlin, Heidelberg, 1998. Springer-Verlag.

[80] Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. *SIGPLAN Not.*, 49(6):542–553, jun 2014.

[81] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, Jul 1959.

[82] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, page 70–80, New York, NY, USA, 2016. Association for Computing Machinery.

[83] K. Rustan M. Leino and Valentin Wüstholz. The Dafny Integrated Development Environment. In *F-IDE*, 2014.

[84] M. Leucker and Daniel Neider. Learning minimal deterministic automata from inexperienced teachers. In *Leveraging Applications of Formal Methods*, 2012.

[85] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. TRANSREGEX: Multi-modal regular

expression synthesis by generate-and-repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1210–1222. IEEE, 2021.

[86] Mark Liffiton and Karem Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40:1–33, January 2008.

[87] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. *SIGCOMM Computer Communications Review*, 41(4):290–301, Aug 2011.

[88] Oded Maler and Irini-Eleftheria Mens. *A Generic Algorithm for Learning Symbolic Automata from Membership Queries*, pages 146–169. July 2017.

[89] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. In Manfred K. Warmuth and Leslie G. Valiant, editors, *Proceedings of the Fourth Annual Workshop on Computational Learning Theory, COLT 1991, Santa Cruz, California, USA, August 5-7, 1991*, pages 128–136. Morgan Kaufmann, 1991.

[90] Mark Moeller, Tiago Ferreira, Thomas Lu, Nate Foster, and Alexandra Silva. Active learning of symbolic NetKAT automata. In *Proceedings of the 46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, June 2025. Association for Computing Machinery.

[91] Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. KATch: A

fast symbolic verifier for NetKAT. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2024. Association for Computing Machinery.

[92] Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. Automata Learning with an Incomplete Teacher. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[93] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, POPL 2017, page 613–625, 2017.

[94] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies. (AM-34), Volume 34*. 1956.

[95] Anil Nerode. Linear automaton transformations. 1958.

[96] Arlindo Oliveira and J.P.M. Silva. Efficient algorithms for the inference of minimum size DFAs. *Machine Learning*, 44, July 2001.

[97] Jose Oncina and Pedro García. Inferring regular languages in polynomial update time. *World Scientific*, January 1992.

[98] J.M. Pena and A.L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(11):1619–1632, 1999.

[99] Leonard Pitt and Manfred K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, January 1993.

[100] Damien Pous. Symbolic algorithms for language equivalence and Kleene algebra with tests. In *Proceedings of the 42nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2015)*, 2015.

[101] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 328–341, New York, NY, USA, 2015. Association for Computing Machinery.

[102] Steffen Smolka, Nate Foster, Justin Hsu, Tobias Kappé, Dexter Kozen, and Alexandra Silva. Guarded Kleene algebra with tests: Verification of uninterpreted programs in nearly linear time. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, 2019.

[103] Steffen Smolka, Praveen Kumar, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. Scalable verification of probabilistic networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[104] Steffen Smolka, Praveen Kumar, Nate Foster, Dexter Kozen, and Alexandra Silva. Cantor meets scott: Semantic foundations for probabilistic networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017.

[105] Emmanuel Suárez Acevedo, Tiago Ferreira, Kevin Batz, Oliver Bøving, Nate

Foster, and Alexandra Silva. Weighted NetKAT: A programming language for quantitative network verification. In *Proceedings of the 47th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 10, New York, NY, USA, 2026. Association for Computing Machinery.

[106] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd D. Millstein, and George Varghese. Lightyear: Using modularity to scale BGP control plane verification. In *Proceedings of the 2023 Conference of the ACM Special Interest Group on Data Communication*, 2023.

[107] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Modular control plane verification via temporal invariants. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2023.

[108] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with Rosette. In *Onward! (SPLASH)*, 2013.

[109] Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, jan 2017.

[110] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2022.

[111] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *ACM/IEEE Symposium on Logic in Computer Science*, 1986.

[112] Margus Veanes. Applications of symbolic finite automata. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, volume 7982, pages 16–23. Springer Berlin Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[113] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. Cachequery: learning replacement policies from hardware caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 519–532, New York, NY, USA, 2020. Association for Computing Machinery.

[114] Jacob Wasserstein. Guarded NetKAT: Soundness, partial-completeness, decidability. May 2023. Publisher: Cornell University Library.

[115] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1990.

[116] Geoffry G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOMM*, 2005.

[117] Han Xu, Zachary Kincaid, Ratul Mahajan, and David Walker. Network change validation with Relational NetKAT. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[118] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, Apr 2016.

[119] Stefan Zetzsche, Alexandra Silva, and Matteo Sammartino. Guarded Kleene algebra with tests: Automata learning. *Electronic Notes in Theoretical Informatics and Computer Science*, 2022.

[120] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. APKeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 241–255, Santa Clara, CA, February 2020. USENIX Association.

[121] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 627–648, 2020.