StacKAT: Infinite State Network Verification

JULES JACOBS, Cornell University, USA
NATE FOSTER, Cornell University, USA
TOBIAS KAPPÉ, Leiden University, The Netherlands
DEXTER KOZEN, Cornell University, USA
LILY SAADA, Cornell University, USA
ALEXANDRA SILVA, Cornell University, USA
JANA WAGEMAKER, Radboud University, The Netherlands

We develop StacKAT, a network verification language featuring loops, finite state variables, nondeterminism, and—most importantly—access to a stack with accompanying push and pop operations. By viewing the variables and stack as the (parsed) headers and (to-be-parsed) contents of a network packet, StacKAT can express a wide range of network behaviors including parsing, source routing, and telemetry. These behaviors are difficult or impossible to model using existing languages like NetKAT. We develop a decision procedure for StacKAT program equivalence, based on finite automata. This decision procedure provides the theoretical basis for verifying network-wide properties and is able to provide counterexamples for inequivalent programs. Finally, we provide an axiomatization of StacKAT equivalence and establish its completeness.

CCS Concepts: • **Networks** → **Network reliability**; • **Theory of computation** → **Semantics and reasoning**; **Automated reasoning**; **Logic and verification**.

Additional Key Words and Phrases: Network Verification, Equivalence, Decision Procedures, Kleene Algebra

ACM Reference Format:
Jules Jacobs, Nate Foster, Tobias Kappé, Dexter Kozen, Lily Saada, Alexandra Silva, and Jana Wagemaker. 2025. StacKAT: Infinite State Network Verification. 1, 1 (April 2025), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

NetKAT [2, 12, 37] is a domain-specific language for specifying and verifying network behavior. Formally, a NetKAT program is a regular expression over assignments $f \leftarrow v$ and guards $f = v$:[1]

$$e ::= 0 \mid 1 \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e^* \mid f = v \mid f \leftarrow v$$

Expressions like $(f \leftarrow 3 \, + \, f = 3 \cdot f \leftarrow 4)^*$ represent traces similar to the regular expression $(a + b \cdot c)^*$, but with imperative actions $f \leftarrow 3$, $f = 3$, and $f \leftarrow 4$ instead of letters a, b, and c. The action $f \leftarrow v$ sets a variable f to a constant value v, and the guard $f = v$ discards the current trace if variable f is not v. As such, NetKAT can be viewed as an imperative programming language over a simple data model—i.e., network packets represented as finite records from header fields f to values v.

[1] NetKAT also has a special action dup, which we omit here for the sake of simplicity but discuss in a later section.

Authors' Contact Information: Jules Jacobs, Cornell University, Ithaca, USA; Nate Foster, Cornell University, Ithaca, USA; Tobias Kappé, Leiden University, Leiden, The Netherlands; Dexter Kozen, Cornell University, Ithaca, USA; Lily Saada, Cornell University, Ithaca, USA; Alexandra Silva, Cornell University, Ithaca, USA; Jana Wagemaker, Radboud University, Nijmegen, The Netherlands.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Because NetKAT can assign and check equality only with respect to constants, it is too simple to model general-purpose imperative programs written in languages like Python or C. But it turns out to be an ideal model for the kind of computation that arises in high-speed network data planes. Network devices like routers, switches, firewalls, etc., process packets using efficient piplines that classify packets using predicates formulated in terms of constant values ($f = v$) and modify packets using actions that assign packet headers to possibly new constant values ($f \leftarrow v$). Moreover, NetKAT can model not only the behavior of individual devices, but also the collective behavior of different interlinked devices, by designating a variable sw as the packet's current location, and representing the movement of a packet to switch k as an assignment $\mathsf{sw} \leftarrow k$.

Many practical network verification questions can be formulated in terms of equivalence queries—e.g., reachability, waypointing, correct compilation, and more [2]. Fortunately, a NetKAT expression e can mention only finitely many values and local variables, and the values v are drawn from a finite domain of numerical constants; hence, it has finite state. As a result, semantic equivalence of NetKAT programs is decidable, though the procedure is non-trivial [12, 24].

However, NetKAT's restriction to a finite state space precludes modeling a variety of behaviors that arise in practice. First, on real-world devices, packets are not represented as finite records, but as sequences of bytes.[2] When a packet enters a switch it must be parsed to "extract" the relevant headers into local variables. Likewise, when a packet exits a switch, the local variables must be serialized (or "deparsed" to use P4's term [6]) back into bytes. By convention, the unparsed portion of the packet is carried along as the payload. To faithfully model packet parsing and serialization in all the various forms used on different devices, NetKAT's finite records are insufficient.

Second, certain network protocols are not finite state. In *source routing* schemes [38], the packet not only stores its IPv4 destination address, but encodes the entire forwarding path in a stack. Each router pops an element off the stack and forwards the packet to the corresponding next hop. As paths can in principle be arbitrarily long, the stack cannot be encoded using a finite record. Dually, protocols that collect *telemetry* push records onto a stack to log observations about how the packet was processed at each hop on the end-to-end path. In similar fashion, *tunneling* protocols prepend a new set of headers to the packet, encapsulating the original headers in the payload, and use the newly added headers to route through a subnetwork. When the packet ultimately leaves the subnetwork, the new headers are removed and the original headers are restored from the payload. There are also protocols like *Segment Routing* and *MPLS* that combine aspects of tunneling and source routing. NetKAT's finite records are also insufficient for modeling these protocols.

In this paper, we extend NetKAT to StacKAT, enriching its data model so that a packet $\langle h, s \rangle$ comprises both a finite record h mapping fields to values as well as a stack s representing the unparsed portion of the packet. To manipulate the record, we use NetKAT's existing constructs. To manipulate the stack, we add $\mathsf{push}(v)$ and $\mathsf{pop}(v)$ constructs as follows:

$$e ::= \underbrace{0 \mid 1 \mid e_1 + e_2 \mid e1 \cdot e2 \mid e^* \mid f = v \mid f \leftarrow v}_{\text{NetKAT (dup-free)}} \mid \underbrace{\mathsf{push}(v) \mid \mathsf{pop}(v)}_{\text{New StacKAT operations}}$$

The new instruction $\mathsf{push}(v)$ pushes a value v onto the stack, and $\mathsf{pop}(v)$ tests that the top of the stack is v and removes it, dropping the packet if the top of the stack is not v (similar to the guard $f = v$). In many examples, the stack can be thought of as a sequence of bytes that represent the rest of the packet, while the variables hold information extracted from the raw data. Hence, with these

[2]Most devices impose an upper limit on the size of a packet—i.e., the so-called Maximum Transmission Unit (MTU), but mathematically, it is more natural to treat packets as having arbitrary size. In the same way, we usually treat memory on a general-purpose computer as being unbounded, even though physical devices have a finite amount of memory.

simple extensions, StacKAT can model parsing, source routing, telemetry, tunneling, and MPLS, as well as the dup instruction used to generate traces in NetKAT's original semantics.

Various fragments of StacKAT are obviously decidable. For instance, equivalence of StacKAT expressions that do not use push or pop is trivially decidable by a reduction to NetKAT. Also, equivalence of StacKAT expressions that use only push (resp. pop)—and neither local variables nor pop (resp. push)—is also trivially decidable, by reduction to equivalence of regular expressions. However, it is not clear whether equivalence of arbitrary StacKAT programs is decidable, because the addition of a stack results in infinitely many states. Moreover, equivalence of StacKAT programs needs to take into account subtle interactions of $\mathsf{push}(-)$ and $\mathsf{pop}(-)$. For instance, we will want $\mathsf{push}(v) \cdot \mathsf{pop}(v) \equiv 1$, as pushing a value and then immediately popping it does nothing. Similarly, $\mathsf{push}(v) \cdot \mathsf{pop}(w) \equiv 0$ (for $v \neq w$), because pushing a value and then expecting a different value to be at the top of the stack will always drop the packet, just like 0. More subtly, $\mathsf{pop}(v) \cdot \mathsf{push}(v) \not\equiv 1$ because the former rejects packets with v not on top of the stack—although $\mathsf{pop}(v) \cdot \mathsf{push}(v) + 1 \equiv 1$ does hold, because the program on the left has no effect on the stack or packet, just like 1.

A decision procedure for StacKAT must therefore incorporate aspects of NetKAT equivalence (to handle local variables) and regular expression equivalence, but it must additionally handle interactions between $\mathsf{push}(-)$ and $\mathsf{pop}(-)$. To illustrate this subtlety when deciding equivalence, consider the StacKAT programs $e_1 = \mathsf{push}(a)^* \cdot \mathsf{pop}(a)^*$ and $e_2 = \mathsf{push}(a)^* + \mathsf{pop}(a)^*$. The program e_1 first pushes any number of a's onto the stack, and then pops any number of a's off the stack. If there are more pushes than pops, the net effect is to push some number of a's. If there are more pops than pushes, the net effect is to pop some number of a's. This is exactly what e_2 does, and therefore the two are equivalent. In contrast, $e_3 = (\mathsf{push}(a) \cdot \mathsf{push}(a))^* \cdot (\mathsf{pop}(a) \cdot \mathsf{pop}(a))^*$ is not equivalent to e_1, because e_1 can empty the stack with just a on it, while e_3 cannot do this.

Existing formalisms such as Visibly Pushdown Languages (VPLs) [1] handle push-pop interactions, but visibly pushdown automata execute stack actions in response to consuming symbols from an input tape. As the equivalence problem for context-free languages is undecidable, VPLs restrict the possible stack behaviors to a decidable fragment. This is in contrast to StacKAT, where stack behaviors are unrestricted. The reason that StacKAT equivalence nevertheless remains decidable is that unlike VPLs, StacKAT does not have a separate input tape. Rather, StacKAT has *only* a stack, whose initial contents serve as the input packet, without a separate input tape. As a result, the notion of equivalence for StacKAT is distinct from equivalence for VPLs. Notably, adding an input tape to StacKAT makes equivalence undecidable as the stack allows one to recognize context-free languages, and equivalence for context-free languages is undecidable.

Our main technical contribution is a decision procedure for StacKAT equivalence. In a nutshell, our technical development proceeds as follows. We first develop a decision procedure for the "pure" stack fragment of StacKAT (i.e., without local variables). This procedure first converts the StacKAT program to an automaton, and then canonicalizes the automaton in three steps, corresponding to three different kinds of push-pop interactions. The canonicalized automaton has the property that two StacKAT programs are semantically equivalent if and only if the canonicalized automata accept the same language (which is decidable). We subsequently extend our decidability result to arbitrary StacKAT programs, including those with local variables, by reduction to equivalence in the pure stack fragment. Our decision procedure not only decides equivalence, but also produces counterexamples (i.e., input-output pairs) when the given programs are inequivalent.

We furthermore axiomatize StacKAT and prove a completeness result for the fragment without local variables. This requires a fundamentally novel approach, because StacKAT handles equivalences such as $\mathsf{push}(v) \cdot \mathsf{pop}(v) \equiv 1$, and $\mathsf{pop}(v) \cdot \mathsf{push}(v) + 1 \equiv 1$ that were not part of previous frameworks based on Kleene Algebra [25].

Syntax	Meaning	Semantics $[\![\cdot]\!] \subseteq \mathsf{Pk} \times \mathsf{Pk}$
0	*False* – Drops all packets	$\emptyset$
1	*True* – Forwards all packets	$\{(p, p) \mid p \in \mathsf{Pk}\}$
$e_1 + e_2$	*Union* – Send packet through e_1 and e_2	$[\![e_1]\!] \cup [\![e_2]\!]$
$e_1 \cdot e_2$	*Sequencing* – Send packet through e_1 then e_2	$\{(p, p'') \mid (p, p') \in [\![e_1]\!], (p', p'') \in [\![e_2]\!]\}$
e^*	*Iteration* – Perform e any number of times	$\bigcup_{i=0}^{\infty} [\![e^i]\!]$, where $e^0 = 1, e^{n+1} = e \cdot e^n$
$f = v$	*Test equals* – Forwards packets with $f = v$	$\{(\langle h, s\rangle, \langle h, s\rangle) \mid \langle h, s\rangle \in \mathsf{Pk}, h(f) = v\}$
$f \leftarrow v$	*Modification* – Sets field f to v	$\{(\langle h, s\rangle, \langle h, s[f \leftarrow v]\rangle) \mid \langle h, s\rangle \in \mathsf{Pk}\}$
$\mathsf{push}(v)$	*Push* – Push v onto stack	$\{(\langle h, s\rangle, \langle h, v :: s\rangle) \mid \langle h, s\rangle \in \mathsf{Pk}\}$
$\mathsf{pop}(v)$	*Pop* – Test and pop top of stack if it equals v	$\{(\langle h, v :: s\rangle, \langle h, s\rangle) \mid \langle h, s\rangle \in \mathsf{Pk}\}$

Fields $f \in F$ Values $v \in V \subseteq \mathbb{N}$ Header $h \in F \mapsto V$ Stack $s \in \mathsf{list}(V)$ Packet $p \in \mathsf{Pk} ::= \langle h, s\rangle$

Fig. 1. StacKAT syntax, meaning, and semantics.

Contributions. Overall, this paper makes the following contributions.

- The design of StacKAT, a domain specific language for infinite state network verification that can model a wide range of network behaviors, such as packet parsing, source routing, telemetry, tunneling, MPLS, and more (Section 2)
- A decision procedure that verifies the semantic equivalence of two StacKAT programs, or produces a counterexample input-output pair (Sections 3 to 5)
- A complete axiomatization of equivalence for StacKAT without local variables (Section 6)

We include an extensive discussion of related work (Section 7) to highlight the differences between StacKAT and other languages and automata formalisms that go beyond regular languages and share similarities with our work. One of the special things about StacKAT is that it captures the local packet state, while also having access to a stack. Together, these features make its semantics different than the systems that have been studied in the literature, to the best of our knowledge.

For the sake of brevity, proofs appear in the appendix; we provide hyperlinks for easy navigation.

2 A Tour of StacKAT

A StacKAT program operates on packets of the form $\langle h, s\rangle \in \mathsf{Pk}$, comprising a finite record mapping header fields to values $h \in F \to V$, as well as a stack $s \in \mathsf{list}(V)$. Values are drawn from a fixed, finite subset $V \subseteq \mathbb{N}$ of the naturals. The syntax and semantics of the StacKAT language constructs are given in Figure 1. The formal semantics $[\![e]\!] \subseteq \mathsf{Pk} \times \mathsf{Pk}$ of a StacKAT expression is a relation between input and output packets, such that an input packet produces zero or more output packets. A pair of StacKAT programs e and f are equivalent (written $e \equiv f$) if they denote the same relation on input and output packets ($[\![e]\!] = [\![f]\!]$). We have, for instance

$$\mathsf{push}(v) \cdot \mathsf{pop}(v) \equiv 1, \qquad \text{but} \qquad \mathsf{pop}(v) \cdot \mathsf{push}(v) \not\equiv 1,$$

as $(\langle h, s\rangle, \langle h, s\rangle) \in [\![1]\!]$ for all stacks s, but $(\langle h, s\rangle, \langle h, s\rangle) \in [\![\mathsf{pop}(v) \cdot \mathsf{push}(v)]\!]$ only for stacks that start with v on top. Consider now the StacKAT equivalence

$$\mathsf{push}(v)^* \cdot \mathsf{pop}(v)^* \equiv \mathsf{push}(v)^* + \mathsf{pop}(v)^*$$

At first sight, these programs appear different—the first one sequentially executes an arbitrary number of $\mathsf{push}(v)$ followed by an arbitrary number of $\mathsf{pop}(v)$ whereas the second one branches and then either does sequences of only $\mathsf{push}(v)$ or only $\mathsf{pop}(v)$. However, in the StacKAT semantics these are equivalent because doing some number of $\mathsf{push}(v)$'s followed by some other number of $\mathsf{pop}(v)$'s will, on net after canceling out $\mathsf{push}(v) \cdot \mathsf{pop}(v) \equiv 1$, result in either only $\mathsf{push}(v)$'s or only $\mathsf{pop}(v)$'s, depending on whether the number of pushes or pops was larger.

Exercise. Which of the following StacKAT programs are equivalent?

$$e_1 \triangleq \mathsf{pop}(v)^* \cdot \mathsf{push}(v)^* \qquad e_2 \triangleq (\mathsf{push}(v) + \mathsf{pop}(v))^*$$
$$e_3 \triangleq \mathsf{push}(v)^* \cdot (\mathsf{pop}(v) \cdot \mathsf{pop}(v))^* \qquad e_4 \triangleq (\mathsf{pop}(v) \cdot \mathsf{pop}(v))^* \cdot \mathsf{push}(v)^*$$
$$e_5 \triangleq (\mathsf{push}(v) \cdot \mathsf{push}(v))^* \cdot \mathsf{pop}(v)^* \qquad e_6 \triangleq \mathsf{pop}(v)^* \cdot (\mathsf{push}(v) \cdot \mathsf{push}(v))^*$$

The main goal of this paper is to design a decision procedure that enables us to automatically check program equivalence of StacKAT programs, not just for simple examples involving a single value v as above, but for arbitrary StacKAT programs involving multiple different values as well as local variables. Furthermore, if two programs are not equivalent, we would like to find a counterexample input-output pair. As we will see, designing such a procedure in the presence of the stack-related equivalences induced by the semantics of StacKAT requires care and is not a simple reduction to NetKAT or other well-studied automata frameworks.

2.1 Syntactic Sugar

StacKAT is a minimal core calculus, designed for exploring theoretical questions. Nevertheless, in presenting examples, it will be useful to use various derived constructs, which can be defined as syntactic sugar. General Boolean expressions can be supported using + for disjunction and · for conjunction, and negation via De Morgan's laws. For instance, inequality of two fields:

$$f \neq g \;\triangleq\; \neg \sum_{v \in V} f{=}v \cdot g{=}v \;\equiv\; \prod_{v \in V} (f \neq v + g \neq v) \qquad \text{where} \qquad f \neq v \triangleq \sum_{v' \in V \setminus \{v\}} f{=}v'$$

We can use these Boolean expressions inside conditionals and loops [20]:

$$\textbf{if } f{=}v \textbf{ then } e_1 \textbf{ else } e_2 \triangleq\; f{=}v \cdot e_1 + f \neq v \cdot e_2 \qquad \textbf{while } f{=}v \textbf{ do } e \triangleq\; (f{=}v \cdot e)^* \cdot f \neq v$$

We can also define syntactic sugar for assigning one field to another and for comparing fields:

$$f \leftarrow g \triangleq \sum_{v \in V} f{=}v \cdot g \leftarrow v \qquad\qquad f{=}g \triangleq \sum_{v \in V} f{=}v \cdot g{=}v$$

Similarly, we can define syntactic sugar for pushing a field onto the stack, and popping the top off the stack and storing the result in a field:

$$\mathsf{push}(f) \triangleq \sum_{v \in V} f{=}v \cdot \mathsf{push}(v) \qquad\qquad \mathsf{pop}(f) \triangleq \sum_{v \in V} \mathsf{pop}(v) \cdot f \leftarrow v$$

Note that the definitions of many of these derived constructs rely on the finiteness of the value domain V. Of course, enumerating the finite (but presumably very large) domain of values would be totally impractical in an implementation. However, this is not an unreachable approach for a mathematical treatment of the language and its core properties, as with NetKAT.

2.2 What can be expressed in StacKAT

StacKAT can be used to encode the behavior of a wide range of networking applications, including complex behaviors that would be difficult or impossible to model in NetKAT. We first discuss how

network behavior is modeled in both NetKAT and StacKAT, and then discuss StacKAT's increased expressiveness compared to NetKAT.

Modeling the behavior of a single switch. The behavior of a switch is modeled as a program e that takes a packet as input and produces zero or more packets as output. The header fields of the packet are modeled as local variables. One special field sw, called the switch field, is used to track the current switch that the packet resides on. To move the packet to a different switch, we assign $\mathsf{sw} \leftarrow n$ for the target switch n. Typically, the behavior of a switch program e is to inspect some of the packet's header fields (using if $f = v$ then ... else ...), and then either drop the packet (using 0), or send it to a neighboring switch (using $\mathsf{sw} \leftarrow n$), possibly after modifying some of the packet's header fields (using $f \leftarrow v$). A switch can also output more than one packet (using +), such as when a switch forwards a packet to multiple neighbors.

Modeling the behavior of a network of switches. Let $e_1, e_2, \ldots, e_n$ be the programs that model the behavior of the switches in the network. The behavior of the entire network is then given by the program $e \triangleq (\mathsf{sw}=1 \cdot e_1 + \mathsf{sw}=2 \cdot e_2 + \cdots + \mathsf{sw}=n \cdot e_n)^*$. This program first tests which switch the packet is currently on, and then behaves as the program for that switch. The Kleene star is used to encode the fact that a packet can traverse multiple switches.

Decidability of equivalence. Whether two NetKAT programs are equivalent is decidable, because variables only take on finitely many different values. Equivalence of programs can be used to express a wide range of properties about network behavior, e.g.,

- Reachability: is there any packet that can go from switch i to switch j? ($\mathsf{sw}=i \cdot e \cdot \mathsf{sw}=j \equiv 0$)
- Slice isolation: can the network be partitioned into two virtual networks? ($e \equiv e_1 + e_2$)

A key goal of StacKAT is to remain decidable even as we add more expressive constructs.

Increased expressiveness of StacKAT. StacKAT's increased expressiveness comes from the ability to model not only the header fields of the packet, but also the payload or remainder of the packet, which can be manipulated using $\mathsf{push}(v)$ and $\mathsf{pop}(v)$. This allows to model the following features, which are impossible to express in NetKAT:

- **Packet parsing and unparsing:** whereas NetKAT assumes that the header fields of the packet are already parsed, the pop instruction allows StacKAT to model parsing the start of the payload of the packet into header fields, and the push instruction to unparse the header fields back into the payload.
  Example. Parsing headers: $\mathsf{pop}(f_1) \cdot \mathsf{pop}(f_2)$. Serializing headers: $\mathsf{push}(f_1) \cdot \mathsf{push}(f_2)$.
- **Source routing:** in source routing, the packet contains a list of routing instructions that determine the desired path of the packet. StacKAT can model source routing by storing the routing instructions in the packet and then popping them off the stack at each hop.
  Example. Go to new switch based on top of stack: $\mathsf{pop}(\mathsf{sw})$.
- **Tunneling:** tunneling protocols are used to transport packets through sub-networks that use different protocols to carry packets. StacKAT can model tunneling by pushing the new headers onto the stack when the packet enters the sub-network, and stripping them off when exiting the network.
  Example. Enter tunnel: $\mathsf{push}(f_1) \cdot \mathsf{push}(f_2) \cdot f_1 \leftarrow v_1 \cdot f_2 \leftarrow v_2$. Exit tunnel: $\mathsf{pop}(f_1) \cdot \mathsf{pop}(f_2)$.
- **Segment Routing and MPLS:** In segment routing and MPLS, each switch removes a routing instruction as in source routing, but may then also add new instructions onto the packet that cause it to follow the right segment. StacKAT can model this hierarchical behavior found in both Segment Routing and MPLS by pushing and popping values.
  Example. Do stack transformation: $\mathsf{pop}(v_1) \cdot \mathsf{push}(v_2) \cdot \mathsf{push}(v_3) + \mathsf{pop}(w_1) \cdot \mathsf{push}(w_2)$.

- **String matching:** StacKAT can model regular matching over routing instructions by nesting pop instructions in a regular expression.
  Example. Match string: $\mathsf{pop}(a) \cdot (\mathsf{pop}(b) + \mathsf{pop}(c))^*$.
- **Telemetry:** StacKAT can model simple forms of telemetry by pushing tracked values.
  Example. Push telemetry value: $\mathsf{push}(t)$.

NetKAT's *dup* operator. StacKAT can model NetKAT's dup operator, which appends the current packet to a trace for observing internal network behavior. By pushing all fields onto the stack ($\mathsf{dup} \triangleq \prod_{f \in F} \mathsf{push}(f)$), StacKAT simulates NetKAT's trace semantics—e.g., $f \leftarrow 1 \cdot \mathsf{dup} \cdot f \leftarrow 2 \not\equiv f \leftarrow 2$ even though $f \leftarrow 1 \cdot f \leftarrow 2 \equiv f \leftarrow 2$. This encoding preserves NetKAT equivalence: two NetKAT programs are equivalent if and only if their StacKAT translations are equivalent.

Combinations of features. The combination of features above is possible (e.g., parsing, tunneling, MPLS, etc.), provided that only a single stack is used. Use of two multiple stacks makes the equivalence problem undecidable [13]. In fact, this remains true even if one of the stacks is constrained to be read-only or write-only (e.g., MPLS + telemetry on separate stacks), as the problem is then the same as equivalence for context-free grammars, which is known to be undecidable [13].

3 Decidability of equivalence

This section presents our procedure for deciding equivalence of StacKAT programs:

$$\text{Given } e_1 \text{ and } e_2, \text{ decide whether } e_1 \equiv e_2 \text{ or } e_1 \not\equiv e_2.$$

We first present a decision procedure for pure push-pop StacKAT programs without local variables, i.e., without the operations $f = v$ and $f \leftarrow v$. Our decision procedure for full StacKAT programs will invoke this procedure as a subroutine. Programs in the push-pop fragment can be viewed as regular expressions over the alphabet $\Sigma = \{\mathsf{push}(v) \mid v \in V\} \cup \{\mathsf{pop}(v) \mid v \in V\}$. In this view, each word in the language accepted by the regular expression represents a trace of push and pop operations.

We wish to decide equivalence between push-pop programs via this interpretation, and the connection between regular languages and automata. The regular language $L(e)$ of a push-pop program is not the same as $[\![e]\!]$, but a connection can be obtained through a series of canonicalizations that take into account the following non-trivial equivalences induced by the semantics (Figure 1).

First, pushing a value v and then popping v is the identity:

$$\mathsf{push}(v) \cdot \mathsf{pop}(v) \equiv 1 \qquad \text{(push-pop)}$$

Second, pushing v and then popping a different value $v \neq w$ fails:

$$\mathsf{push}(v) \cdot \mathsf{pop}(w) \equiv 0 \qquad \text{if } v \neq w \qquad \text{(filter)}$$

Third, popping v and then pushing v back is *almost* the identity:

$$\mathsf{pop}(v) \cdot \mathsf{push}(v) + 1 \equiv 1 \qquad \text{(pop-push)}$$

The intuition for the third property is that $\mathsf{pop}(v) \cdot \mathsf{push}(v)$ does not alter the stack, provided that the top of the stack is v, and therefore its behaviors are included in those of 1.

Our approach to deciding equivalence proceeds as follows:

(1) We first convert the StacKAT programs e and f to finite automata that accept $L(e)$ and $L(f)$.
(2) We then perform a series of three canonicalization steps corresponding to the three axioms above to obtain canonical automata that accept the canonicalized languages $\overline{L(e)}$ and $\overline{L(f)}$.
(3) Finally, we decide whether $\overline{L(e)} = \overline{L(f)}$ are language equivalent.

$$\frac{x \in L}{x \in \mathsf{pushpop}(L)} \qquad \frac{x \cdot \mathsf{push}(v) \cdot \mathsf{pop}(v) \cdot y \in \mathsf{pushpop}(L)}{x \cdot y \in \mathsf{pushpop}(L)}$$

Fig. 2. Push-pop canonicalization of a language L

We then show that language equivalence of the canonicalized languages implies semantic equivalence of the original StacKAT programs:

$$[\![e]\!] = [\![f]\!] \iff \overline{L(e)} = \overline{L(f)}$$

We proceed to describe the canonicalization steps in detail, and then show how to extend the decision procedure to StacKAT programs with local variables.

3.1 Push-pop canonicalization

Let us assume that we have a StacKAT program e that contains only push and pop operations, and that we have a language $L(e)$ that represents the traces of push and pop operations that e can perform, as well as an NFA that accepts $L(e)$. This NFA can be constructed from regular expression e using standard constructions, such as Antimirov derivatives [3].

The first step of canonicalization reduces matching $\mathsf{push}(v)\cdot\mathsf{pop}(v)$ pairs. We define the operation $\mathsf{pushpop}(L)$ on a language L, which closes L under the reduction $\mathsf{push}(v) \cdot \mathsf{pop}(v) \to \epsilon$. In order to make it possible to perform the canonicalization on automata as well, $\mathsf{pushpop}(L)$ adds reduced traces while keeping original traces in L. This ensures that $\mathsf{pushpop}(L)$ is a regular language whenever L is a regular language. Let us consider an example.

For the singleton language $L = \{\mathsf{push}(1) \cdot \mathsf{push}(2) \cdot \mathsf{pop}(2) \cdot \mathsf{pop}(1) \cdot \mathsf{push}(3) \cdot \mathsf{pop}(3)\}$,

$$\begin{aligned}\mathsf{pushpop}(L) = \{&\mathsf{push}(1) \cdot \mathsf{push}(2) \cdot \mathsf{pop}(2) \cdot \mathsf{pop}(1) \cdot \mathsf{push}(3) \cdot \mathsf{pop}(3),\\ &\mathsf{push}(1) \cdot \mathsf{pop}(1) \cdot \mathsf{push}(3) \cdot \mathsf{pop}(3),\ \ \mathsf{push}(1) \cdot \mathsf{push}(2) \cdot \mathsf{pop}(2) \cdot \mathsf{pop}(1),\\ &\mathsf{push}(3) \cdot \mathsf{pop}(3),\ \ \mathsf{push}(1) \cdot \mathsf{pop}(1),\ \ \epsilon\}\end{aligned}$$

That is, reduced strings are added for all possible reduction orders. The $\mathsf{pushpop}(L)$ canonicalization on sets of strings is defined in Figure 2.

Push-pop canonicalization for the trace automaton. To perform the canonicalization on the automaton, we add shortcut ϵ edges for every $\mathsf{push}(v) \cdot \epsilon^* \cdot \mathsf{pop}(v)$ path in the automaton. In the implementation, to avoid ϵ^* paths, we simultaneously perform ϵ closure.

The rules for adding epsilon edges are shown in Figure 3. First, we add an ϵ self-loop to each state. Then, for each $\mathsf{push}(v) - \epsilon - \mathsf{pop}(v)$ path of length 3, we add an ϵ shortcut edge. Finally, for each $\epsilon - \epsilon$ path of length 2, we add an ϵ shortcut edge. If we perform this closure on an automaton for $L(e)$, the language accepted by the resulting canonicalized automaton is $\mathsf{pushpop}(L(e))$.

LEMMA 3.1. *For any NFA over the alphabet $\Sigma = \{\mathsf{push}(v), \mathsf{pop}(v) \mid v \in V\}$ representing language L, the language accepted by the NFA after performing the push-pop canonicalization is* $\mathsf{pushpop}(L)$.

3.2 Filtering out invalid traces

The next step is to filter out invalid traces that contain $\mathsf{push}(v) \cdot \mathsf{pop}(w)$ with $v \neq w$. It is important that the push-pop canonicalization from the preceding section is performed first, as this can reveal additional $\mathsf{push}(v) \cdot \mathsf{pop}(w)$ combinations by reducing an intervening substring to ϵ, such as $\mathsf{push}(v) \cdot \mathsf{push}(a) \cdot \mathsf{pop}(a) \cdot \mathsf{pop}(w) \to \mathsf{push}(v) \cdot \mathsf{pop}(w)$.

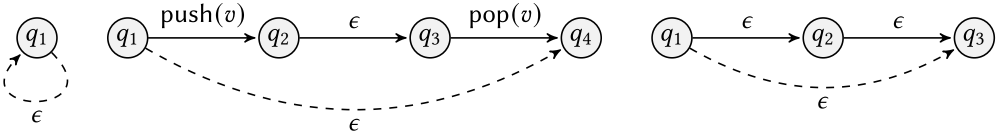


Fig. 3. Push-pop closure rules. Given the presence of the solid edges, we add the dotted edge. For each push(v) – ϵ – pop(v) path, we add an ϵ shortcut edge, and we simultaneously take the reflexive transitive closure of the ϵ edges.

Having performed the push-pop closure first, we also know that we have reduced any push(v) · pop(w) with $v = w$, which makes the original trace with push(v) · pop(w) redundant. Therefore, we filter out *all* traces where a push(v) occurs before a pop(w), regardless of whether $v \neq w$ or $v = w$. In other words, we only keep the traces of the form

$$L_{\text{pop}^*\text{push}^*} \triangleq (\text{pop}(v_1) + \cdots + \text{pop}(v_n))^* \cdot (\text{push}(v_1) + \cdots + \text{push}(v_n))^*$$

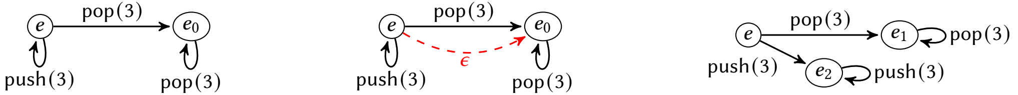
where $V = \{v_1, \ldots, v_n\}$ is the set of values that occur. We therefore define

$$\text{filter}(L) \triangleq L \cap L_{\text{pop}^*\text{push}^*}$$

The intersection of two regular languages is regular, and we can compute an automaton for filter(L) from the automaton for L.

Example. We now show a simple example of the procedure on $e = \text{push}(3)^* \cdot \text{pop}(3)^*$ and $e' = \text{push}(3)^* + \text{pop}(3)^*$ to check that they are equivalent.

In the first step, an automaton is constructed for $e = \text{push}(3)^* \cdot \text{pop}(3)^*$ (left), then we take the push-pop closure (middle), and finally we take the intersection with pop-push (right).



The automaton on the right is also the automaton one would build for $e' = \text{push}(3)^* + \text{pop}(3)^*$ (in the very first step, as in this case taking push-pop closure, intersect with pop-push does not change the automaton). Hence, the automata for the expressions e and e' are bisimilar, and as we shall see, it follows that $[\![\text{push}(3)^* \cdot \text{pop}(3)^*]\!] = [\![\text{push}(3)^* + \text{pop}(3)^*]\!]$.

More generally, we can consider the program $(\text{push}(3)^n)^*(\text{pop}(3)^m)^*$, for any $n, m \in \mathbb{N}$, and show that it is in fact equivalent to $(\text{push}(3)^k)^* + (\text{pop}(3)^k)^*$ where $k = \gcd(n, m)$.

3.3 Pop-push canonicalization

We have now reduced traces where push(v) immediately occurs before pop(v) and limited ourselves to traces where all pop(v) operations happen before push(w) operations. The reader may wonder whether we are now done, as all remaining traces are valid and in reduced form. However, we have not yet taken into account the fact that pop(v) · push(v) is almost the identity, which results in an interaction between different traces. Consider the following trace sets:

$$L_1 \triangleq \{\epsilon,\ \text{pop}(v) \cdot \text{push}(v)\} \qquad\qquad L_2 \triangleq \{\epsilon\}$$

corresponding to the programs $e_1 = 1 + \text{pop}(v) \cdot \text{push}(v)$ and $e_2 = 1$. These trace sets are in reduced form and not equal, but they are semantically equivalent.

At first sight, it may seem attractive to canonicalize the trace sets by removing the redundant trace pop(v) · push(v). However, removing this trace is only valid because the ϵ trace is present. In

$$\frac{x \in L}{x \in \mathsf{poppush}(L)} \qquad \frac{x \cdot y \in \mathsf{poppush}(L) \qquad x \text{ all pop and } y \text{ all push}}{x \cdot \mathsf{pop}(v) \cdot \mathsf{push}(v) \cdot y \in \mathsf{poppush}(L)}$$

Fig. 4. Pop-push canonicalization of a language L

general, we cannot remove traces without considering the full context—i.e., possible interactions between different traces. We conjecture that the resulting operation would be exceedingly difficult to define and compute, particularly on the automaton level.

Instead, we define a canonicalization operation $\mathsf{poppush}(L)$ that *adds* traces by inserting $\mathsf{pop}(v) \cdot \mathsf{push}(v)$ in the middle of a trace in all possible ways. The canonicalization is defined in Figure 4.

This transformation is enough to fully canonicalize the trace language, so that semantic equivalence coincides with language equivalence, as we shall see in the next section. Unfortunately $\mathsf{poppush}(L)$ is not regular, so we cannot compute an automaton for it. In fact, the language is not even visibly pushdown[3] [1], so decidability of language equivalence is not a priori guaranteed. We address this difficulty by shifting perspectives and switching to a different representation.

Zipping the trace language. To address the non-regularity of the language, we switch to a different representation of the trace language. We define the zipped trace language $\mathsf{zip}(L)$, which represents a trace from the middle (i.e., the border between pops and pushes) outwards. For example,

$$\tau = \mathsf{pop}(1) \cdot \mathsf{pop}(2) \cdot \mathsf{push}(3) \cdot \mathsf{push}(4) \qquad \mathsf{zip}(\tau) = (\mathsf{pop}(2), \mathsf{push}(3)) \cdot (\mathsf{pop}(1), \mathsf{push}(4))$$

When there are more pushes than pops or vice versa, we use done as a placeholder. For example,

$$\tau' = \mathsf{pop}(1) \cdot \mathsf{push}(2) \cdot \mathsf{push}(3) \qquad \mathsf{zip}(\tau') = (\mathsf{pop}(1), \mathsf{push}(2)) \cdot (\mathsf{done}, \mathsf{push}(3))$$

We define the zip operation formally in Figure 5. For strings that are not of the form $\mathsf{pop}^*\mathsf{push}^*$, the zipping operation is undefined. For example, the trace $\tau'' = \mathsf{pop}(1) \cdot \mathsf{push}(2) \cdot \mathsf{pop}(3)$ is not zippable, i.e. $\mathsf{zip}(\tau'') = \bot$. We extend zip to sets of traces by applying it to each trace in the set:

$$\mathsf{zip}(L) \triangleq \{\mathsf{zip}(x) \mid x \in L \text{ and } \mathsf{zip}(x) \text{ defined }\}$$

This not only zips the language, but also filters out invalid traces, as the zipping operation is only defined for traces of the form $\mathsf{pop}^*\mathsf{push}^*$.

The key advantage of zip is that whereas $\mathsf{poppush}(L)$ is not regular, $\mathsf{zip}(\mathsf{poppush}(L))$ *is*. In fact, the pop-push canonicalization can easily be performed on the zipped trace language, and the result is manifestly regular. As zipping already filters out invalid traces, and is a bijection on the $L_{\mathsf{pop}^*\mathsf{push}^*}$ fragment, we can check the equivalence of the zipped languages instead of the original languages:

Lemma 3.2. $\mathsf{zip}(\mathsf{filter}(L)) = \mathsf{zip}(L)$

Lemma 3.3. $\mathsf{zip}(L_1) = \mathsf{zip}(L_2) \iff L_1 = L_2$ *for* $L_1, L_2 \subseteq L_{pop^*push^*}$

The following lemma characterizes the zipped trace language of the pop-push canonicalization of a language L. Let $A \triangleq \{(\mathsf{pop}(v), \mathsf{push}(v)) \mid v \in V\}$.

Lemma 3.4. $\mathsf{zip}(\mathsf{poppush}(L)) = A^* \cdot \mathsf{zip}(L)$

$$
\begin{aligned}
\mathrm{zip}(\epsilon) &\triangleq \epsilon \\
\mathrm{zip}(x \cdot \mathrm{pop}(v) \cdot \mathrm{push}(w) \cdot y) &\triangleq (\mathrm{pop}(v), \mathrm{push}(w)) \cdot \mathrm{zip}(x \cdot y) \\
\mathrm{zip}(x \cdot \mathrm{pop}(v)) &\triangleq (\mathrm{pop}(v), \mathrm{done}) \cdot \mathrm{zip}(x) \\
\mathrm{zip}(\mathrm{push}(v) \cdot y) &\triangleq (\mathrm{done}, \mathrm{push}(v)) \cdot \mathrm{zip}(y) \\
\mathrm{zip}(a) &\triangleq \text{undefined otherwise}
\end{aligned}
$$

Fig. 5. Zipping the trace language; in these equations,, $x \in \mathrm{pop}(V)^*$, $y \in \mathrm{push}(V)^*$

$$
\frac{q_1 \xrightarrow{\mathrm{pop}(v)} q_1' \qquad q_2 \xrightarrow{\mathrm{push}(w)} q_2'}{(q_1', q_2) \xrightarrow{(\mathrm{pop}(v),\mathrm{push}(w))} (q_1, q_2')}
\qquad
\frac{q_1 \text{ initial}}{(q_1, q_2) \xrightarrow{\epsilon} (\mathrm{done}, q_2)}
\qquad
\frac{q_2 \text{ final}}{(q_1, q_2) \xrightarrow{\epsilon} (q_1, \mathrm{done})}
\qquad
\frac{}{(q, q) \text{ initial}}
$$

$$
\frac{q_1 \xrightarrow{\mathrm{pop}(v)} q_1'}{(q_1', \mathrm{done}) \xrightarrow{(\mathrm{pop}(v),\mathrm{done})} (q_1, \mathrm{done})}
\qquad
\frac{q_2 \xrightarrow{\mathrm{push}(v)} q_2'}{(\mathrm{done}, q_2) \xrightarrow{(\mathrm{done},\mathrm{push}(v))} (\mathrm{done}, q_2')}
\qquad
\frac{}{(\mathrm{done}, \mathrm{done}) \text{ final}}
$$

$$
\frac{q_2 \xrightarrow{\epsilon} q_2'}{(q_1, q_2) \xrightarrow{\epsilon} (q_1, q_2')}
\qquad
\frac{q_2 \xrightarrow{\epsilon} q_2'}{(\mathrm{done}, q_2) \xrightarrow{\epsilon} (\mathrm{done}, q_2')}
\qquad
\frac{q_1 \xrightarrow{\epsilon} q_1'}{(q_1', q_2) \xrightarrow{\epsilon} (q_1, q_2)}
\qquad
\frac{q_1 \xrightarrow{\epsilon} q_1'}{(q_1', \mathrm{done}) \xrightarrow{\epsilon} (q_1, \mathrm{done})}
$$

Fig. 6. Zipping the trace automaton

Zipping the trace automaton. In order to make use of zip for the decision procedure, we need to define a zipping operation on the automaton level. Given an automaton for a language L, the zipped automaton for $\mathrm{zip}(L)$ should traverse the original automaton "from the middle outwards", taking a backward pop step and a forward push step simultaneously. Given a NFA for language L, we construct an NFA for $\mathrm{zip}(L)$ as follows.

States (q_1, q_2) where q_1 and q_2 are either states of the original automaton, or an extra state done.
Initial states (q, q) where q is any state of the original automaton.
Final states (done, done) only.
Transitions A transition in the zipped automaton corresponds to a simultaneous backward pop and forward push transition in the original automaton. We also add ϵ transitions to done for the initial and final states. If the first or second component of the state is done, we can step in the zipped automaton with a single step in the original automaton. The transitions are defined in Figure 6.

LEMMA 3.5. *For any NFA over the alphabet* $\Sigma = \{\mathrm{push}(v), \mathrm{pop}(v) \mid v \in V\}$ *representing language* L*, the language accepted by the zipped NFA is* $\mathrm{zip}(L)$.

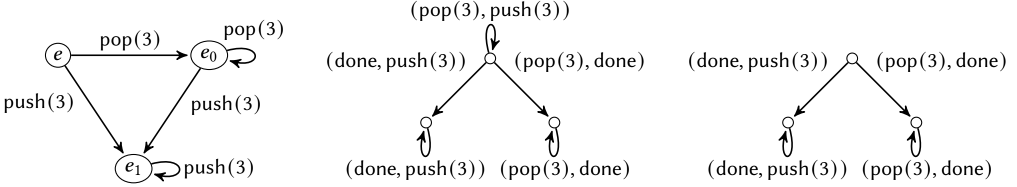
With the zipped trace automaton in Figure 6, we can can now compute the pop-push closure on the automaton level by precomposing the automaton with the A^* language.

[3]This language is not visibly pushdown, because it is not known for each alphabet letter whether it is a call (push), return (pop) or a transition action. For instance a letter $\mathrm{pop}(v)$ in a word $w \in \mathrm{poppush}(L)$ can be both a transition action (in case the letter already appeared in w before the closure) or a return action (in case the letter is a result of the poppush-closure).

Lemma 3.6. *For any NFA over the alphabet* $\Sigma = \{\mathsf{push}(v), \mathsf{pop}(v) \mid v \in V\}$ *representing language* L*, the language accepted by the NFA after performing the push-pop canonicalization, zipping, and precomposing with* A^* *is* $\mathsf{zip}(\mathsf{poppush}(\mathsf{filter}(\mathsf{pushpop}(L))))$.

Example. In the preceding example, the automata for $e = \mathsf{push}(3)^* \cdot \mathsf{pop}(3)^*$ and $e' = \mathsf{push}(3)^* + \mathsf{pop}(3)^*$ already became equivalent after the first two steps of canonicalization, and the final step of zipping and pop-push canonicalization was not necessary. However, in general, the final step is necessary to decide equivalence of push-pop programs as the following example illustrates. Consider the program $e'' = \mathsf{pop}(3)^* \cdot \mathsf{push}(3)^*$ (note that, compared to e, the order of push and pop differs here). An automaton for this program is shown on the left below.

The push-pop closure of this automaton is the same as the original automaton, and the intersection with the pop-push automaton is also the same. The resulting automaton is not bisimilar to the final automata of e and e', yet we know that e'' is semantically equivalent to e and e'. The pop-push closure is needed to decide this equivalence. We therefore construct the zipped automata for e'' and e. These are shown below; in the middle is the zipped automaton for e'', and on the right the zipped automaton for e. We have omitted the ϵ transitions and merged equivalent states for clarity.



In order to then perform the pop-push closure on these automata, we simply prepend the language $(\mathsf{pop}(3), \mathsf{push}(3))^*$ to the automata. This has no effect on the language of the automaton on the left, as it already starts with $(\mathsf{pop}(3), \mathsf{push}(3))^*$. The automaton on the right, however, is extended with a $(\mathsf{pop}(3), \mathsf{push}(3))$ loop on the leftmost state. This crucial last step makes the resulting automata language equivalent, and next we will see that this allows us to conclude that $[\![e'']\!] = [\![e]\!]$.

3.4 Deciding push-pop programs

For a StacKAT program e in the push-pop fragment, let $L(e)$ be the language obtained by viewing e as a regular expression over $\Sigma = \{\mathsf{push}(v), \mathsf{pop}(v) \mid v \in v\}$. Further, define

$$\overline{L} \triangleq \mathsf{poppush}(\mathsf{filter}(\mathsf{pushpop}(L)))$$

Note that the notation $\overline{L}$ does *not* denote the complement of L, as it sometimes does in other work, but rather this specific operation. We have the following theorem:

Theorem 3.7. *For two push-pop programs* e *and* f*, we have* $[\![e]\!] = [\![f]\!] \iff \overline{L(e)} = \overline{L(f)}$.

To prove this theorem, we define a semantic interpretation of a language L as follows:

$$[L] = \bigcup_{x \in L} [\![x]\!]$$

i.e., the semantic interpretation of a language is the union of the semantic interpretations of its strings, viewed as StacKAT programs (a string can be viewed as a program in the obvious way, where the empty string is the program 1). The following lemma connects the semantic interpretation of a push-pop program with the language of its traces.

Lemma 3.8. *For a push-pop program* e*, we have* $[\![e]\!] = [L(e)]$.

i.e., the semantics of a push-pop program agrees with the semantics of all its push-pop traces.
Second, the canonicalization rules are semantically valid w.r.t. $[-]$:

Lemma 3.9. $[L] = [\overline{L}]$

We define an alternative semantic interpretation of a language L as follows:

$$[L]' = \{(\langle \mathsf{emp}, s\rangle, \langle \mathsf{emp}, s'\rangle) \mid \mathsf{pop}(s)\mathsf{push}(s') \in L\}$$

where emp is the empty packet header, and $\mathsf{pop}(s)$ on an entire stack $s = s_1 :: \ldots :: s_n$ (as opposed to a single value) is defined as $\mathsf{pop}(s_1) \cdots \mathsf{pop}(s_n)$, and similarly, $\mathsf{push}(s)$ is defined as $\mathsf{push}(s_n) \cdots \mathsf{push}(s_1)$. On languages of the form $\overline{L}$, these two semantics agree:

Lemma 3.10. $[\overline{L}] = [\overline{L}]'$

The inclusion $\supseteq$ is trivial, and the inclusion $\subseteq$ follows because $[L]'$ differs from $[L]$ only in that $[L]$ allows stepping with stacks that may contain additional values below the strings in L, whereas $[L]'$ requires the stack to exactly match the strings in L. However, $\mathsf{poppush}(L)$ adds additional strings to L with $\mathsf{pop}(v)\mathsf{push}(v)$ pairs in the middle, which precisely match the possible additional values in the stack.

We also have the following (trivial) lemma:

Lemma 3.11. *If strings in L_1, L_2 are of the form* $\mathsf{pop}(s)\mathsf{push}(s')$, *then* $[L_1]' = [L_2]' \iff L_1 = L_2$

We can now return to Theorem 3.7.

Proof (of Theorem 3.7). By Lemma 3.8, it suffices to prove $[L(e)] = [L(f)] \iff \overline{L(e)} = \overline{L(f)}$. By Lemma 3.9, it suffices to prove $[\overline{L(e)}] = [\overline{L(f)}] \iff \overline{L(e)} = \overline{L(f)}$. By Lemma 3.10, it suffices to prove $[\overline{L(e)}]' = [\overline{L(f)}]' \iff \overline{L(e)} = \overline{L(f)}$. We obtain the desired result by Lemma 3.11. □

We conclude that equivalence of StacKAT programs in the push-pop fragment is decidable:

Theorem 3.12. *To decide whether $[\![e]\!] = [\![f]\!]$, convert e and f to NFAs, perform the push-pop closure, zipping, and pop-push closure, obtaining automata for* $\mathsf{zip}(\overline{L(e)})$ *and* $\mathsf{zip}(\overline{L(f)})$, *and check bisimilarity of these resulting automata.*

3.5 Computational complexity

We establish the computational complexity of the decision problem for the push-pop fragment of StacKAT, as well as a matching hardness result. Checking equivalence of StacKAT programs with only push operations and no pop operations (or vice versa) is identical to checking language equivalence of these programs viewed as regular expressions, as no interaction between the two types of operations is possible if only one type of operation is present. Checking language equivalence of regular expressions is known to be PSPACE-complete. We therefore conclude:

Theorem 3.13. *The equivalence problem for the push-pop fragment of StacKAT is PSPACE-hard.*

The problem remains in PSPACE even in the presence of *both* push and pop operation, as our decision procedure builds a polynomially sized automaton from the input, and then performs a NFA equivalence check, which can be done in PSPACE; thus, the problem is PSPACE-complete.

Theorem 3.14. *Our decision procedure for the push-pop fragment of StacKAT is in PSPACE.*

4 Decidability of Equivalence for full StacKAT

We now extend the decision procedure to the full StacKAT language, including tests and modifications, by reducing the problem to the push-pop fragment. For any given input and output packet headers α_1 and α_2, we define the trace language $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$ of a StacKAT program e that can be observed when the input packet header is α_1 and the output packet header is α_2. We will

present a method for constructing an NFA for $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$. We then show that the equivalence problem $[\![e]\!] = [\![f]\!]$ can be reduced to the equivalence problem for the trace languages $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$ and $\mathsf{traces}_{\alpha_1}^{\alpha_2}(f)$ for all α_1 and α_2, and that the latter problem can be solved using the decision procedure for the push-pop fragment.

To start, we define the trace language of a StacKAT program with tests and modifications. We have a trace language for any given input and output packet header values. For example, consider

$$e \triangleq (f{=}1 \cdot \mathsf{push}(1) + f{\leftarrow}2 \cdot \mathsf{push}(2))^*$$

The trace languages $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$ fall into several cases. The trace language is:

- $\mathsf{push}(1)^* \cdot \mathsf{push}(2)^*$ for input packet header $f = 1$, and output packet header $f = 2$.
- $\mathsf{push}(1)^*$ for input packet header $f = 1$, and output packet header $f = 1$.
- $\mathsf{push}(2)^*$ for input packet header $f \neq 1$, and output packet header $f = 2$.
- $\emptyset$ for input packet header $f \neq 1$, and output packet header $f \neq 2$.

Intuitively, $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$ captures an over-approximation of stack manipulations found in e that can take place while e takes input packet header α_1 to output packet header α_2. The trace language is formally defined in Figure 7, by recursion on the structure of the program e.

4.1 The trace automaton

The trace language $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$ for given input and output headers is a regular language, which we prove by showing that it can be represented by a finite automaton. Furthermore, the explicit construction of this automaton shows how to implement a possible decision procedure. We represent states in the automaton as pairs $\langle \alpha, e \rangle$, where α is the current packet header and e is the current expression. Figure 8 defines the transitions of the trace automaton. The automaton takes ϵ transitions for tests and modifications, and to decompose the union and star operations. For a sequence $e_1 \cdot e_2$, the automaton takes a transition on e_1 until e_1 is finished, and then continues with e_2. For push and pop, the automaton takes a transition labeled with the corresponding operation.

To construct the NFA for $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$, we start with the initial state $\langle \alpha_1, e \rangle$ and take transitions according to the rules in Figure 8. The final states are those of the form $\langle \alpha_2, 1 \rangle$.

We also need to show that the resulting automaton is always finite. The α component of $\langle \alpha, e \rangle$ ranges over a finite set, but it may not be clear that the e component only reaches a finite number of distinct expressions, because the star rule $e^* \longrightarrow e \cdot e^*$ can enlarge the expression. However, the automaton steps only to expressions of the form $e_1 \cdot e_2 \cdots e_k$ where each e_i is strictly smaller than e_{i+1}, and e_k smaller than e, where "smaller" means that the expression is a subexpression, possibly with certain occurrences of $f{=}v$, $f{\leftarrow}v$, $\mathsf{push}(v)$, $\mathsf{pop}(v)$ replaced by 1. Therefore, the number of distinct reachable expressions is finite.

Lemma 4.1. *The language accepted by the NFA for* $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$ *is indeed* $\mathsf{traces}_{\alpha_1}^{\alpha_2}(e)$.

4.2 Decidability of equivalence

We have shown how to construct an automaton for the trace language and how to perform the canonicalization steps on the automaton level. We obtain the canonical automaton by putting together steps for the push-pop closure, zipping, and pop-push closure:

$$\overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(e) \triangleq \mathsf{poppush}(\mathsf{zip}(\mathsf{pushpop}(\mathsf{traces}_{\alpha_1}^{\alpha_2}(e))))$$

We can use the canonical automaton to build a decision procedure for equivalence:

Theorem 4.2. $\overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(e_1) = \overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(e_2)$ *for all* α_1, α_2 *if and only if* $[\![e_1]\!] = [\![e_2]\!]$

$$\mathrm{traces}_{\alpha}^{\alpha'}(0) \triangleq \emptyset \qquad \mathrm{traces}_{\alpha}^{\alpha'}(1) \triangleq \begin{cases} \{\epsilon\} & \text{if } \alpha = \alpha' \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{traces}_{\alpha}^{\alpha'}(f{=}v) \triangleq \begin{cases} \{\epsilon\} & \text{if } \alpha = \alpha' \text{ and } \alpha_f = v \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathrm{traces}_{\alpha}^{\alpha'}(f{\leftarrow}v) \triangleq \begin{cases} \{\epsilon\} & \text{if } \alpha' = \alpha[f{\leftarrow}v] \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{traces}_{\alpha}^{\alpha'}(\mathrm{push}(v)) \triangleq \begin{cases} \{\mathrm{push}(v)\} & \text{if } \alpha = \alpha' \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathrm{traces}_{\alpha}^{\alpha'}(\mathrm{pop}(v)) \triangleq \begin{cases} \{\mathrm{pop}(v)\} & \text{if } \alpha = \alpha' \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{traces}_{\alpha}^{\alpha'}(e_1 + e_2) \triangleq \mathrm{traces}_{\alpha}^{\alpha'}(e_1) \cup \mathrm{traces}_{\alpha}^{\alpha'}(e_2)$$

$$\mathrm{traces}_{\alpha}^{\alpha'}(e_1 \cdot e_2) \triangleq \{s_1 \cdot s_2 \mid s_1 \in \mathrm{traces}_{\alpha}^{\beta}(e_1),\ s_2 \in \mathrm{traces}_{\beta}^{\alpha'}(e_2), \beta \in F \longrightarrow V\}$$

$$\mathrm{traces}_{\alpha_1}^{\alpha_2}(e^*) \triangleq \bigcup_{n \in \mathbb{N}} \mathrm{traces}_{\alpha_1}^{\alpha_2}(e^n), \qquad \text{where } e^0 \triangleq 1 \text{ and } e^{n+1} \triangleq e^n \cdot e$$

Fig. 7. The trace language of a StacKAT program.

$$\frac{\alpha_f = v}{\langle \alpha, f{=}v \rangle \xrightarrow{\epsilon} \langle \alpha, 1 \rangle} \qquad \langle \alpha, f{\leftarrow}v \rangle \xrightarrow{\epsilon} \langle \alpha[f{\leftarrow}v], 1 \rangle \qquad \langle \alpha, \mathrm{push}(v) \rangle \xrightarrow{\mathrm{push}(v)} \langle \alpha, 1 \rangle$$

$$\langle \alpha, \mathrm{pop}(v) \rangle \xrightarrow{\mathrm{pop}(v)} \langle \alpha, 1 \rangle \qquad \langle \alpha, e_1 + e_2 \rangle \xrightarrow{\epsilon} \langle \alpha, e_1 \rangle \qquad \langle \alpha, e_1 + e_2 \rangle \xrightarrow{\epsilon} \langle \alpha, e_2 \rangle$$

$$\frac{\langle \alpha, e_1 \rangle \xrightarrow{s} \langle \alpha', e_1' \rangle}{\langle \alpha, e_1 \cdot e_2 \rangle \xrightarrow{s} \langle \alpha', e_1' \cdot e_2 \rangle} \qquad \langle \alpha, 1 \cdot e \rangle \xrightarrow{\epsilon} \langle \alpha, e \rangle \qquad \langle \alpha, e^* \rangle \xrightarrow{\epsilon} \langle \alpha, 1 \rangle \qquad \langle \alpha, e^* \rangle \xrightarrow{\epsilon} \langle \alpha, e \cdot e^* \rangle$$

Fig. 8. The transitions of the trace automaton of a StacKAT program.

The condition on the traces can be checked by language equivalence of the automata, which is decidable. Second, the packet headers α_1, α_2 range over a finite set. Therefore, we have a decision procedure for equivalence of StacKAT programs. The proof of Theorem 4.2 is entirely analogous to Theorem 3.7, except that Lemma 3.8 is replaced by the following lemma:

LEMMA 4.3. *For a StacKAT program e, we have* $[\![e]\!] = \bigcup_{\alpha_1,\alpha_2} [\mathrm{traces}_{\alpha_1}^{\alpha_2}(e)]_{(\alpha_1,\alpha_2)}$

Here $[\mathrm{traces}_{\alpha_1}^{\alpha_2}(e)]_{(\alpha_1,\alpha_2)}$ is defined as adding the packet headers to the pure stack traces:

$$[\mathrm{traces}_{\alpha_1}^{\alpha_2}(e)]_{(\alpha_1,\alpha_2)} \triangleq \{((\alpha_1, s), (\alpha_2, s')) \mid (\langle \mathrm{emp}, s \rangle, \langle \mathrm{emp}, s' \rangle) \in [\mathrm{traces}_{\alpha_1}^{\alpha_2}(e)]\}$$

On the right-hand side of this definition, note that $\mathrm{traces}_{\alpha_1}^{\alpha_2}(e)$ contains words over letters of the form $\mathrm{push}(v)$ and $\mathrm{pop}(v)$ exclusively, and so the pairs in $[\mathrm{traces}_{\alpha_1}^{\alpha_2}(e)]$ are always of the form $(\langle h, s \rangle, \langle h, s' \rangle)$ — i.e., with the output packet the same as the input packet. The rest of the proof proceeds the same way as for Theorem 3.7, but using $[L]_{(\alpha_1,\alpha_2)}$ instead of $[L]$.

4.3 Computational complexity

We establish the computational complexity of the decision procedure for full StacKAT, as well as a matching hardness result. For the hardness result, we reduce equivalence of regular expressions with squaring (which is EXPSPACE-complete [23]) to equivalence of push-only StacKAT programs

with variables. Squaring makes the problem harder because it allows expressions to be encoded more compactly as e^2 instead of $e \cdot e$, which would otherwise lead to an expression that is twice as large. StacKAT variables can compactly encode squaring: e^2 can be modeled by $x \leftarrow 0 \cdot ((x = 0 + x = 1) \cdot e \cdot (x = 0 \cdot x \leftarrow 1 + x = 1 \cdot x \leftarrow 2))^* \cdot x \neq 0 \cdot x \neq 1$, picking x fresh for each square that occurs, initializing $x \leftarrow 0$ at the beginning. In more common parlance, $x \leftarrow 0; \mathsf{while}(x = 0 \vee x = 1)\{e;\ \mathsf{if}\ x = 0\ \mathsf{then}\ x \leftarrow 1\ \mathsf{else\ if}\ x = 1\ \mathsf{then}\ x \leftarrow 2\}$ runs e twice without duplicating e textually. This increases the size of the expression by only a constant factor, and leads to the following result:

Theorem 4.4. *The equivalence problem for full StacKAT is EXPSPACE-hard.*

Our decision procedure matches this bound, as the trace automaton of a StacKAT program is at most exponential in the size of the program, and the subsequent steps are PSPACE:

Theorem 4.5. *Our decision procedure for full StacKAT is in EXPSPACE.*

Note that the encoding of e^2 requires one additional fresh variable per square; the equivalence problem with a fixed set of variables is still PSPACE (as this bounds the size of the trace automaton).

5 Counterexample Extraction and Implementation

The decision procedure for equivalence of StacKAT programs can be used to extract counterexamples. Given two programs e and f that are not equivalent, the procedure will find a packet header pair α_1, α_2 such that $\overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(e) \neq \overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(f)$. The packet headers α_1 and α_2 can be used to construct a counterexample input-output pair for which the programs e and f differ.

Because $\overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(e) \neq \overline{\mathsf{traces}}_{\alpha_1}^{\alpha_2}(f)$, the bisimilarity check will find a difference in the automata. The difference can be used to extract a word that is accepted by one automaton but not the other. Because the final automaton has been zipped, this counterexample word will be of the form $\mathsf{push}(v_1) \cdots \mathsf{push}(v_n)\mathsf{pop}(w_m) \cdots \mathsf{pop}(w_1)$ (after unzipping the word again), where the v_i, w_i are values of the stack symbols. Viewing $\vec{v} = (v_1, \ldots, v_n)$ and $\vec{w} = (w_1, \ldots, w_m)$ as stacks, we can construct a counterexample input-output pair as follows:

$$\text{Input: } \langle \alpha_1, \vec{v} \rangle \quad \text{Output: } \langle \alpha_2, \vec{w} \rangle \quad \text{where} \quad (\langle \alpha_1, \vec{v} \rangle, \langle \alpha_2, \vec{w} \rangle) \in ([\![e]\!] \setminus [\![f]\!]) \cup ([\![f]\!] \setminus [\![e]\!])$$

This input-output pair is a counterexample to the equivalence of e and f, because either e reaches the output packet but f does not, or vice versa. In fact, we can obtain a representation of *all* counterexamples by computing the symmetric difference of the languages of the canonical automata of e and f for each pair of packet headers α_1, α_2. The symmetric difference of the canonical automata of e and f is a regular language, and the corresponding automaton can be used to enumerate all counterexamples, or to find the smallest.

5.1 Implementation

An interactive web-based demonstration of the StacKAT decision procedure is available at https://apndx.org/pub/iqe6/stackatv1.html. The user can input two StacKAT programs, and the decision procedure will output whether the programs are equivalent, and if not, give a counterexample input that produces different outputs for the two programs. The decision procedure relies only on the elementary constructions on finite automata, as described in this paper. The implementation differs from the decision procedure in the paper only in that it supports inequality tests ($f \neq v$) directly, without the need to encode this as a sum over the value space.

5.1.1 Performance Evaluation. The StacKAT equivalence checker was evaluated using five benchmark categories, each testing different aspects of the algorithm's performance:

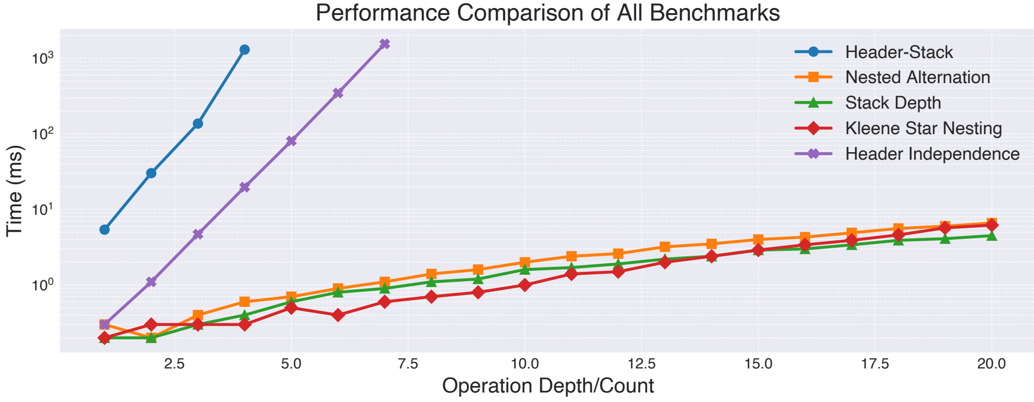


Fig. 9. Performance comparison of all benchmarks. The x-axis shows the number of operations or depth for each benchmark, and the y-axis (logarithmic scale) shows execution time in milliseconds.

- **Header-Stack:** Tests expressions that combine header tests with stack operations, of the form $(f_1{=}0 \cdot \text{push}(0) + f_1{=}1 \cdot \text{push}(1)) \cdot \ldots \cdot (f_n{=}0 \cdot \text{push}(0) + f_n{=}1 \cdot \text{push}(1)) \cdot (\text{pop}(0) + \text{pop}(1)) \cdot \ldots \cdot (\text{pop}(0) + \text{pop}(1))$ compared with the identity 1.
- **Nested Alternation:** Tests how deeply nested choice operations affect performance by comparing left-to-right nesting $((\text{push}(1) + \text{push}(2)) + \text{push}(3))$ with right-to-left nesting $(\text{push}(3) + (\text{push}(2) + \text{push}(1)))$.
- **Stack Depth:** Evaluates performance with expressions that push values onto the stack and then pop them off in reverse order: $\text{push}(1) \cdot \ldots \cdot \text{push}(n) \cdot \text{pop}(n) \cdot \ldots \cdot \text{pop}(1) \equiv 1$.
- **Kleene Star Nesting:** Tests nested repetition operations with expressions of the form $(\ldots((\text{push}(1)^*)^*)^* \ldots)^* \equiv \text{push}(1)^*$.
- **Packet Header Independence:** Tests commutativity of header checks with expressions of the form $(h_1{=}v_1) \cdot (h_2{=}v_2) \cdot \ldots \cdot (h_n{=}v_n) \equiv (h_n{=}v_n) \cdot \ldots \cdot (h_2{=}v_2) \cdot (h_1{=}v_1)$.

As shown in Figure 9, the benchmarks exhibit different scaling behaviors:

- **Exponential Growth:** Header-Stack and Header Independence benchmarks show the steepest increase in execution time, with both growing exponentially in the number of operations. These benchmarks involve state space explosions due to independent variables.
- **Moderate Growth:** Nested Alternation, Stack Depth, and Kleene Star Nesting show more moderate growth rates, making them more tractable for larger inputs.

The results indicate that the StacKAT equivalence checker performance is primarily affected by header space size, which grows exponentially with the number of independent header variables. Operations that primarily affect stack manipulation without expanding the state space show more favorable performance characteristics.

5.1.2 Naïveté of the implementation. The implementation is naïve in the sense that it enumerates over the packet header space and checks equivalence for each input-output packet header pair. For each such pair, it constructs the zipped automata, and checks language equivalence of the two automata. The alphabet of the automata consists of concrete numeric values that are pushed and popped by the program. We leave the design and implementation of a *symbolic* decision procedure for StacKAT as future work. A symbolic decision procedure would not naively enumerate over the packet space, nor would it have concrete numeric values on the edges of the automaton: it would

instead use BDD-like structures, which have been shown to significantly speed up equivalence checking of NetKAT programs [24]. A simple version would represent the alphabet on edges symbolically using a BDD, which would allow operations such as generalized pop ($\mathsf{pop}(V)$, where V is a set of values) to be represented as a single edge. How to best integrate symbolic methods with StacKAT to allow efficient header-to-stack and stack-to-header transfer is a question for future research, and the current implementation should be seen as an interactive playground for exploration of the theory of StacKAT.

6 Axiomatization and Completeness

In this section we propose an axiomatization of the push-pop fragment of StacKAT, and prove its completeness with respect to the equational theory of $[\![-]\!]$.

Our development reuses the language model developed for the decision procedure, and can be divided into two phases:

- First, we axiomatize the equational theory of a simple language model corresponding to push-pop canonicalization and filtering invalid traces—i.e., $L_0(-) = \mathsf{filter}(\mathsf{pushpop}(L(-)))$. For this axiomatization, we extend Kleene Algebra (KA) with axioms $\mathsf{push}(v)\mathsf{pop}(v) = 1$ and $\mathsf{push}(v)\mathsf{pop}(w) = 0$ for $v \neq w$.
- Next, we extend this axiomatization to the equational theory of $\overline{L}$ (resp. $[\![-]\!]$, by Theorem 3.7), using the axiom $\mathsf{pop}(v)\mathsf{push}(v) \leq 1$, and an additional axiom scheme. Because the interaction of push and pop exhibits non-regular, context-free behavior, we introduce a new language operator †, to allow the formation of a limited class of context-free languages.

For brevity, this section highlights the key aspects of our axiomatization and completeness result. For full details, please see the appendix.

6.1 Completeness for L_0

Let L_0 be the language model corresponding to push-pop canonicalization and filtering invalid traces. That is, $L_0(-) = \mathsf{filter}(\mathsf{pushpop}(L(-)))$. To abbreviate notation, let us write $\mathsf{push}(a) \in \Sigma$ simply as a, and $\mathsf{pop}(a) \in \Sigma$ as $\overline{a}$. In this section, we show that the axioms of KA, augmented with the equations $a\overline{a} = 1$ and $a\overline{b} = 0$ for $a \neq b$ are complete for the equational theory of L_0, i.e., $L_0(e_1) = L_0(e_2)$ if $e_1 = e_2$ can be proved from these.

Recall that the push-pop fragment of StacKAT is formed by regular expressions over Σ. Let (E, D) be the usual Brzozowski derivative on regular expressions [8]. Let D' be like D, except it removes symbols from the right instead of the left. The following two versions of the fundamental theorem for regular expressions can be proved from the laws of KA alone: [36]:

$$e = E(e) + \sum_{a\in\Sigma} aD_a(e) \qquad\qquad e = E(e) + \sum_{a\in\Sigma} D'_a(e)a \tag{1}$$

We can straightforwardly extend D and D' to words: for $x \in \Sigma^*$ and $a \in \Sigma$, we define

$$D_\varepsilon(e) = e \qquad D_{xa}(e) = D_a(D_x(e)) \qquad D'_\varepsilon(e) = e \qquad D'_{xa}(e) = D'_a(D'_x(e)).$$

There are only finitely many derivatives $D_x(e)$ and $D'_x(e)$ up to KA equivalence [8].

A *pure push* (respectively, *pure pop*) *expression* is a regular expression over letters of the form $a \in \Sigma$ (respectively, $\overline{a} \in \Sigma$). The expressions 1 and 0 are both pure pop and pure push expressions.

Lemma 6.1. *Every expression $e_1\overline{e}_2$, where e_1 is a pure push expression and $\overline{e}_2$ a pure pop expression, can be provably transformed to a finite sum of pure push expressions of the form $D'_{xa}(e_1)a$, pure pop expressions of the form $\overline{a}D_{xa}(\overline{e}_2)$, or 1, where $x \in \Sigma^*$ and $a \in \Sigma$.*

An expression is in *normal form* if it is a sum of expressions of the form $\overline{e}_1 e_2$, where $\overline{e}_1$ is a pure pop expression and e_2 is a pure push expression. Note that $L(e) = L_0(e)$ if e is in normal form.

THEOREM 6.2. *Every expression can be provably transformed to normal form.*

We now have everything we need to state our completeness result for L_0.

THEOREM 6.3. *The axioms of KA augmented with the equations $a\overline{a} = 1$ and $a\overline{b} = 0$ for $b \neq a$ are sound and complete for the equational theory of the interpretation L_0; that is, for any $e_1, e_2 \in \mathsf{Exp}$, $L_0(e_1) = L_0(e_2)$ iff $e_1 = e_2$ is a deductive consequence of these axioms.*

PROOF. Soundness of the axioms is routine; thus if $e_1 = e_2$ is provable, then $L_0(e_1) = L_0(e_2)$. Conversely, suppose $L_0(e_1) = L_0(e_2)$. Using Theorem 6.2, we can provably transform e_1 and e_2 to normal forms e_1' and e_2', respectively. By soundness, $L_0(e_1') = L_0(e_2')$, and since $L_0(e) = L(e)$ for any e in normal form (by Lemma 6.1), $L(e_1') = L(e_2')$. By the completeness of KA w.r.t. L [19], $e_1' = e_2'$ is provable. In conjunction with the proofs of $e_1 = e_1'$ and $e_2 = e_2'$, we obtain a proof of $e_1 = e_2$. □

6.2 Completeness for StacKAT

We now turn our attention to proving completeness for the push-pop fragment of StacKAT with respect to $\overline{L}$. We continue to assume the axioms for L_0, namely the axioms of KA in addition to the equations $a\overline{a} = 1$ and $a\overline{b} = 0$ for $b \neq a$, but $\overline{L}$ (and therefore $[\![-]\!]$) also satisfy an additional property:

LEMMA 6.4. *The axioms for L_0 plus $\overline{a}a \leq 1$ (equivalently, $1 + \overline{a}a = 1$) are sound for $\overline{L}$.*

Although inspired by this axiom, our axiomatization is rather more complicated, involving an infinite set of rules derived from the fundamental theorem. We do not know whether $\overline{a}a \leq 1$ by itself is enough for completeness, but conjecture that it is not.

To properly describe our axiomatization, we need a new operator. For $B \subseteq \Sigma^*$, define $B^\dagger = \{\overline{x}x \mid x \in B\}$. For example, $(a^*)^\dagger = \{\overline{a}^n a^n \mid n \geq 0\}$. Clearly, † does not preserve regularity, which is not surprising as we are dealing with stacks. However, we will temper the non-regular aspect of this operator by using the two Brzozowski derivatives D and D' in tandem. Still assuming $a\overline{a} = 1$ and $a\overline{b} = 0$ for $b \neq a$, here are some properties of † that are not difficult to derive:

$$\Big(\bigcup_\alpha B_\alpha\Big)^\dagger = \bigcup_\alpha B_\alpha^\dagger \qquad B^\dagger B^\dagger = B^\dagger \qquad (B^\dagger)^* = 1 + B^\dagger \qquad \overline{a}B^\dagger a = (Ba)^\dagger,\ a \in \Sigma \tag{2}$$

It is also clear that if B is regular, then $B^\dagger$ is context-free. We furthermore have the following.

LEMMA 6.5. *For pure pop expressions $\overline{e}_1$ and pure push expressions e_2, $\overline{L}(\overline{e}_1 e_2) = L(\overline{e}_1) \cdot \Sigma^{*\dagger} \cdot L(e_2)$.*

We can also extend † to regular expressions, interpreting $e^\dagger$ as $L(e)^\dagger$. The following then holds.

LEMMA 6.6. *Let $\overline{e}_1$ be a pure pop expression and e, e_2 pure push expressions. The expression $\overline{e}_1 e^\dagger e_2$ is semantically L-equivalent to a sum of expressions of the following forms:*

$$\overline{d}_1\overline{a}d^\dagger bd_2,\ a \neq b \qquad \overline{d}_1\overline{a}d^\dagger \qquad d^\dagger bd_2 \qquad d^\dagger \tag{3}$$

where d_1 is a pure pop expression, d and d_2 are pure push expressions, and $a, b \in \Sigma$, $a \neq b$.

Note that the four forms shown in (3) represent pairwise disjoint sets under L. For example, every string in $L(\overline{d}_1\overline{a}d^\dagger bd_2)$, where $a \neq b$, contains a substring $\overline{a}\overline{x}xb$, whereas every string in $L(\overline{d}_1\overline{a}d^\dagger)$ has a suffix of the form $\overline{a}\overline{x}x$, and no string in normal form can have both.

Let # be a new delimiter symbol and consider the forms

$$ad_1\#d\#bd_2,\ a \neq b \qquad ad_1\#d\# \qquad \#d\#bd_2 \qquad \#d\# \tag{4}$$

corresponding to the four forms of (3). Define the map

$$K : \Sigma^*\#\Sigma^*\#\Sigma^* \to \overline{\Sigma}^*\Sigma^* \qquad\qquad K(x\#y\#z) = \overline{xy}yz.$$

K is not injective; for example, $K(a\#a\#a) = K(aa\#\#aa)$. However, K acts bijectively between strings of the form (3) and corresponding strings of the form (4).

Lemma 6.7. *Let e be one of the expressions in (3) and let e' be the corresponding expression in (4). The map K restricted to $L(e')$ is a bijection $K : L(e') \to L(e)$.*

Let $\sum_i \overline{e}_{i1}e_{i2}$ and $\sum_j \overline{d}_{j1}d_{j2}$ be two expressions in normal form. Using Lemma 6.6, reduce $\sum_i \overline{e}_{i1}\Sigma^{*\dagger}e_{i2}$ and $\sum_j \overline{d}_{j1}\Sigma^{*\dagger}d_{j2}$ to sums of expressions $\sum_i \overline{p}_i q_i^\dagger r_i$ and $\sum_j \overline{u}_j v_j^\dagger w_j$, respectively, of the form (3). Let $\sum_i p_i\#q_i\#r_i$ and $\sum_j u_j\#v_j\#w_j$ be the corresponding sums of expressions of the form (4).

Lemma 6.8. $\overline{L}(\sum_i \overline{e}_{i1}e_{i2}) \subseteq \overline{L}(\sum_j \overline{d}_{j1}d_{j2}) \iff L(\sum_i p_i\#q_i\#r_i) \subseteq L(\sum_j u_j\#v_j\#w_j)$.

Now consider the rule

$$\frac{\sum_i p_i\#q_i\#r_i \leq \sum_j u_j\#v_j\#w_j}{\sum_i \overline{e}_{i1}e_{i2} \leq \sum_j \overline{d}_{j1}d_{j2}}, \tag{5}$$

where the expressions are as described in the paragraph above Lemma 6.8.

Theorem 6.9. *The rule (5), along with $a\overline{a} = 1$, $a\overline{b} = 0$ for $b \neq a$, and the axioms of KA and equational logic, are sound and complete for the equational theory of the push-pop fragment of StacKAT.*

Proof. Suppose we want to prove that $e \leq d$ under the interpretation L_1 or $[\![-]\!]$. By Theorem 6.2, we can provably convert e and d to normal form $\sum_i \overline{e}_{i1}e_{i2}$ and $\sum_j \overline{d}_{j1}d_{j2}$, respectively. By Lemma 6.8 and rule (5), to prove that $\sum_i \overline{e}_{i1}e_{i2} \leq \sum_j \overline{d}_{j1}d_{j2}$ under the interpretation L_1, it suffices to prove $\sum_i p_i\#q_i\#r_i \leq \sum_j u_j\#v_j\#w_j$ under the interpretation R. This is provable, as KA is complete for R. □

7 Related Work

In relating StacKAT to existing extensions of regular languages it is important to make a few observations. Most importantly, a StacKAT program does not have a separate input tape. Hence, it does not capture behaviors that require both processing the input and pushing/popping symbols to/from the stack. Using standard pushdown automata, one can model behaviors like "after every a-transition, write symbol x to the stack" but this is not possible in StacKAT. If StacKAT programs were equivalent to automata with both an input tape and a stack, equivalence of StacKAT languages would clearly be undecidable—e.g., we could then use the stack to parse a context-free grammar on the input tape. Having said that, there are several tractable fragments of context-free languages that are somewhat reminiscent of StacKAT. The reader might wonder if StacKAT is a particular instance of one of these previously studied, tractable fragments. To the best of our knowledge this is not the case. Below, we review below the most closely related work and explain the key differences.

Network Reachability Verification of MPLS Networks. Closely related to our work is the verification of reachability in MPLS networks [15–17]. Since MPLS networks use the packet as a stack, the verification problem can be cast as a reachability problem in a pushdown system. These works can also handle quantitative properties such as failures or latency. Recently, a symbolic decision procedure was developed [4] that can efficiently handle large packet spaces. In contrast with these works, StacKAT handles equivalence queries. An interesting question for future work is whether ideas from the above works can be adapted to StacKAT.

Visibly Pushdown Languages. One important subclass of languages that is particularly relevant for program analysis are visibly pushdown languages (VPLs) [1]. VPLs are accepted by visibly pushdown automata. Whereas in general pushdown automata the stack operations depend on both the input symbol and the current state, in VPAs the input alphabet is partitioned in a way that the type of stack operation (push or pop) is determined by the input symbol itself alone. VPLs are more tractable than general context-free languages as they enjoy several closure and decidability properties. VPLs are closed under union, intersection, and complementation. Moreover, emptiness, membership, and language equality and inclusion are decidable.

We view the relationship between StacKAT and VPLs as follows. Naively, one might hope to use equivalence of pushdown automata as a basis for decision procedures. Unfortunately, that equivalence problem is undecidable. VPLs address this issue by restricting the pushdown automaton to a specific subclass. StacKAT takes a different approach and instead removes the input tape, but keeps the pushdown stack completely unrestricted. Therefore, VPLs and StacKAT are incomparable, being two different ways of taming the expressive power of pushdown automata.

Dyck Languages. VPLs are more general than so-called *parentheses languages*, also often referred to as Dyck languages, which are languages consisting of balanced strings of parentheses, with one and multiple types of parentheses [5, 22]. However, a language $\mathsf{poppush}(L)$ cannot be viewed as such a language, as not all pop/push actions occur parenthesized: a letter $\mathsf{pop}(v)$ in a word w in $\mathsf{poppush}(L)$ can be seen as an opening bracket in case it occurred because of the $\mathsf{poppush}(-)$ closure, but otherwise it is just a letter, and there is not necessarily a closing bracket ($\mathsf{push}(v)$) in w. Thus, the $\mathsf{poppush}(-)$-closure cannot be characterized using Dyck languages.

Another class of related work in which Dyck languages play an important role is language graph reachability analysis, a common way to tackle certain static analyses [27]. Informally, the nodes of a graph represent various program segments, while edges capture relationships between those segments, such as program flow or data dependencies. Analysis is then formulated as a reachability question between nodes in the graph, as witnessed by paths whose labels along the edges produce a string that belongs to a language, which is often a visibly pushdown language. This reachability problem, often referred to as Dyck reachability, has been applied in many contexts including interprocedural data-flow analysis [14], slicing [29], or type-based flow analysis [26]. Dyck reachability has been generalized to bidirected graphs (with applications in pointer analysis) and has been studied in terms of decidability and complexity [18].

Interprocedural Analysis and Pushdown Systems. Precise interprocedural dataflow analyses involve reasoning about the call stack to capture call-return behavior [28, 32]. Pushdown Systems are another formalism used for program analysis and dataflow analysis [7, 11], which model a control state as well as a stack. Dataflow queries are then translated to reachability queries in pushdown systems. This avoids merging data flows from states with the same program point but different calling context, resulting in a more precise analysis. Our work by contrast is focused on program equivalence, in order to enable encoding of verification queries such as slicing, translation validation, loop freedom, etc. [12]. Weigthed versions of pushdown systems have also been developed [30, 31]. It would be interesting to investigate whether techniques for interprocedural dataflow analysis can be harnessed to answer network verification queries, including weighted variants.

Deterministic Context Free Languages. Another relevant subclass of languages are Deterministic Context Free Languages (DCFLs), the class accepted by deterministic pushdown automata (DPDAs). DCFLs are closed under complementation, but not under union. The equivalence problem for DPDAs is decidable [33–35]. Like VPLs, and unlike StacKAT, DCFLs have a separate input tape.

Deterministic One-Counter Automata. Another relevant subclass are the languages accepted by deterministic one-counter automata (DOCA) [39]. A DOCA is a special case of a pushdown automaton, with a single stack that can only hold one type of symbol (often interpreted as a counter), and the stack operations are restricted to either pushing or popping this symbol (or leaving the stack unchanged). However, again because StacKAT becomes undecidable when we have a separate stack and input tape, StacKAT languages are not the languages captured by one-counter automata.

Valence Automata. A generalization of both DOCA and pushdown automata is given by the notion of *valence automaton* [9] where transitions are labeled with elements from a monoid. A valence automaton accepts a string depending not only on reaching a final state but also on the accumulated product of the transition labels equaling the zero of the monoid. Valence automata generalize several models and can be used to study languages and automata under algebraic constraints. In particular, a DOCA is a valence automaton for the monoid of natural numbers under addition. Pushdown automata are valence automata under the monoid of stack operations. Transitions push or pop symbols on the stack, accepting if the stack is empty. Decidability and closure properties of valence automata are dependent on the specific monoid used.

For our setting, it is not clear how to adapt valence automata to StacKAT, because StacKAT programs can transform the stack, and in fact the way they do so is integral to their semantics, whereas (stacks encoded in) valence automata function as an additional constraint on acceptance.

Generalized versions of KA. In [21], the authors investigate an extension of Kleene algebra called omega algebra with domain as an algebra to model pushdown automata. The axioms they present to model the stack are reminiscent of ours. However, they do not discuss issues concerning decidability and completeness of their algebraic theory. Recent advances in completeness techniques led to the development of the Kleene Algebra with Hypotheses framework [10, 25], used to obtain completeness proofs of several Kleene Algebra extensions in a uniform way. However, some of the axioms we consider in StacKAT fail the conditions of Kleene Algebra with Hypotheses and therefore we will not be able to use their framework off-the-shelf. In particular, $\mathsf{pop}(v) \cdot \mathsf{push}(v) \leq 1$ and $\mathsf{push}(v) \cdot \mathsf{pop}(v) \leq 1$ capture behaviors that have been identified as problematic in previous work on hypotheses, because you lose regularity of the associated languages.

8 Future Work

StacKAT is an extension of NetKAT [2, 12, 37], but goes beyond it by adding a stack to the data model. A new symbolic decision procedure was recently developed that can efficiently decide large NetKAT equivalence queries [24]. We would naturally like to integrate StacKAT within this symbolic framework. As we can encode dup in StacKAT, one might wonder if the integration is immediate, which is however not the case. The core of the developments in [24] rely on the fact that during the decision procedure, when trying to determine whether e_1 and e_2 are equivalent, if a dup action is encountered then the packet state is identical at the corresponding states of e_1 and e_2. This is fundamentally different in StacKAT: if equivalent e_1 and e_2 push the same symbol onto the stack, that does not imply that the packet state is also the same at that moment. In other words, StacKAT programs e_1 and e_2 might be equivalent even though the packet state they are in is different. This behavioral difference makes the integration with the framework of [24] challenging, and is left as future work. Another promising direction for future work is to determine the *active domain* of a StacKAT program, similar to APKeep [40], grouping packet values together into equivalence classes, and then reducing the packet space enumeration to the equivalence classes.

References

[1] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *STOC*. https://doi.org/10.1145/1007352.1007390

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. https://doi.org/10.1145/2535838.2535862

[3] Valentin M. Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theor. Comput. Sci.* (1996). https://doi.org/10.1016/0304-3975(95)00182-4

[4] Ryan Beckett and Aarti Gupta. 2022. Katra: Realtime Verification for Multilayer Networks. In *NSDI*. https://www.usenix.org/conference/nsdi22/presentation/beckett

[5] Jean Berstel and Luc Boasson. 2002. Balanced Grammars and Their Languages. In *Formal and Natural Computing - Essays Dedicated to Grzegorz Rozenberg*. https://doi.org/10.1007/3-540-45711-9_1

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM* (07 2014). https://doi.org/10.1145/2656877.2656890

[7] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR*. https://doi.org/10.1007/3-540-63141-0_10

[8] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* (1964). https://doi.org/10.1145/321239.321249

[9] P. Buckheister and Georg Zetzsche. 2013. Semilinearity and Context-Freeness of Languages Accepted by Valence Automata. In *MFCS*. https://doi.org/10.1007/978-3-642-40313-2_22

[10] Amina Doumane, Denis Kuperberg, Damien Pous, and Cécilia Pradic. 2019. Kleene Algebra with Hypotheses. In *FOSSACS*. https://doi.org/10.1007/978-3-030-17127-8_12

[11] Alain Finkel, Bernard Willems, and Pierre Wolper. 1997. A direct symbolic approach to model checking pushdown systems. In *Workshop on Verification of Infinite State Systems*. https://doi.org/10.1016/S1571-0661(05)80426-8

[12] Nate Foster, Dexter Kozen, Mae Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *POPL*. https://doi.org/10.1145/2676726.2677011

[13] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*.

[14] Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *SIGSOFT*. https://doi.org/10.1145/222124.222146

[15] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jirí Srba, and Marc Tom Thorgersen. 2018. P-Rex: fast verification of MPLS networks with multiple link failures. In *CoNEXT*. https://doi.org/10.1145/3281411.3281432

[16] Peter Gjøl Jensen, Dan Kristiansen, Stefan Schmid, Morten Konggaard Schou, Bernhard Clemens Schrenk, and Jirí Srba. 2020. AalWiNes: a fast and quantitative what-if analysis tool for MPLS networks. In *CoNEXT*. https://doi.org/10.1145/3386367.3431308

[17] Peter Gjøl Jensen, Stefan Schmid, Morten Konggaard Schou, Jirí Srba, Juan Vanerio, and Ingo van Duijn. 2021. Faster Pushdown Reachability Analysis with Applications in Network Verification. In *ATVA*. https://doi.org/10.1007/978-3-030-88885-5_12

[18] Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The decidability and complexity of interleaved bidirected Dyck reachability. In *POPL*. https://doi.org/10.1145/3498673

[19] Dexter Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Inf. Comput.* (1994). https://doi.org/10.1006/inco.1994.1037

[20] Dexter Kozen. 1996. Kleene algebra with tests and commutativity conditions. In *TACAS*. https://doi.org/10.1007/3-540-61042-1_35

[21] Vincent Mathieu and Jules Desharnais. 2005. Verification of Pushdown Systems Using Omega Algebra with Domain. In *RelMICS/AKA*. https://doi.org/10.1007/11734673_15

[22] Robert McNaughton. 1967. Parenthesis Grammars. *J. ACM* (1967). https://doi.org/10.1145/321406.321411

[23] Albert R. Meyer and Larry J. Stockmeyer. 1972. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *SWAT*. https://doi.org/10.1109/SWAT.1972.29

[24] Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. 2024. KATch: A Fast Symbolic Verifier for NetKAT. In *PLDI*. https://doi.org/10.1145/3656454

[25] Damien Pous, Jurriaan Rot, and Jana Wagemaker. 2024. On Tools for Completeness of Kleene Algebra with Hypotheses. *LMCS* (2024). https://doi.org/10.46298/LMCS-20(2:8)2024

[26] Jakob Rehof and Manuel Fähndrich. 2001. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*. https://doi.org/10.1145/360204.360208

[27] Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

[28] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. https://doi.org/10.1145/199448.199462

[29] Thomas W. Reps, Susan Horwitz, Shmuel Sagiv, and Genevieve Rosay. 1994. Speeding up Slicing. In *SIGSOFT*. https://doi.org/10.1145/193173.195287

[30] Thomas W. Reps, Stefan Schwoon, and Somesh Jha. 2003. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *SAS*. https://doi.org/10.1007/3-540-44898-5_11

[31] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.* (2005). https://doi.org/10.1016/J.SCICO.2005.02.009

[32] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 1&2 (1996). https://doi.org/10.1016/0304-3975(96)00072-2

[33] Géraud Sénizergues. 1997. The Equivalence Problem for Deterministic Pushdown Automata is Decidable. In *ICALP*. https://doi.org/10.1007/3-540-63165-8_221

[34] Géraud Sénizergues. 2002. L(A) = L(B)? Decidability Results from Complete Formal Systems. In *ICALP*. https://doi.org/10.1007/3-540-45465-9_4

[35] Géraud Sénizergues. 2002. L(A)=L(B)? A simplified decidability proof. *Theor. Comput. Sci.* (2002). https://doi.org/10.1016/S0304-3975(02)00027-0

[36] Alexandra Silva. 2010. *Kleene Coalgebra.* Ph. D. Dissertation. Radboud Universiteit Nijmegen.

[37] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *ICFP*. https://doi.org/10.1145/2784731.2784761

[38] Carl A. Sunshine. 1977. Source routing in computer networks. *Comput. Commun. Rev.* (1977). https://doi.org/10.1145/1024853.1024855

[39] Leslie G. Valiant and Michael S. Paterson. 1975. Deterministic one-counter automata. *J. Comput. Syst. Sci.* (1975). https://doi.org/10.1016/S0022-0000(75)80005-5

[40] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *NSDI*. https://www.usenix.org/conference/nsdi20/presentation/zhang-peng