A Constraint Solving Approach to Parikh Images of Regular Languages

AMANDA STJERNA, Uppsala University, Sweden
PHILIPP RÜMMER, University of Regensburg, Germany and Uppsala University, Sweden

A common problem in string constraint solvers is computing the Parikh image, a linear arithmetic formula that describes all possible combinations of character counts in strings of a given language. Automata-based string solvers frequently need to compute the Parikh image of products (or intersections) of finite-state automata, in particular when solving string constraints that also include the integer data-type due to operations like string length and indexing. In this context, the computation of Parikh images often turns out to be both prohibitively slow and memory-intensive. This paper contributes a new understanding of how the reasoning about Parikh images can be cast as a constraint solving problem, and questions about Parikh images be answered without explicitly computing the product automaton or the exact Parikh image. The paper shows how this formulation can be efficiently implemented as a calculus, PC*, embedded in an automated theorem prover supporting Presburger logic. The resulting standalone tool Catra is evaluate on constraints produced by the Ostrich+ string solver when solving standard string constraint benchmarks involving integer operations. The experiments show that PC* strictly outperforms the standard approach by Verma et al. to extract Parikh images from finite-state automata, as well as the over-approximating method recently described by Janků and Turoňová by a wide margin, and for realistic timeouts (under 60 s) also the nuXmv model checker. When added as the Parikh image backend of Ostrich+ to the Ostrich string constraint solver's portfolio, it boosts its results on the quantifier-free strings with linear integer algebra track of SMT-COMP 2023 (QF_SLIA) enough to solve the most Unsat instances in that track of all competitors.

CCS Concepts: • **Theory of computation** → **Quantitative automata**; *Regular languages*; **Automated reasoning**; *Program verification*; *Verification by model checking*.

Additional Key Words and Phrases: Parikh images, string solvers, model checking

ACM Reference Format:
Amanda Stjerna and Philipp Rümmer. 2024. A Constraint Solving Approach to Parikh Images of Regular Languages. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 138 (April 2024), 29 pages. https://doi.org/10.1145/3649855

1 INTRODUCTION

Extending automated theorem provers and SMT solvers with support for rich string constraints is important for program analysis, particularly to detect cross-site scripting vulnerabilities and other string manipulation bugs [Bultan et al. 2017]. To check the satisfiability of a formula like $x \in \mathcal{L}_1 \wedge y \in \mathcal{L}_2 \wedge |x| > |y|$, with string variables x, y and regular languages $\mathcal{L}_1, \mathcal{L}_2$, it is necessary to reason about the possible lengths of x, y that are admitted by $\mathcal{L}_1, \mathcal{L}_2$, respectively. The set of word lengths $\{|w| \mid w \in \mathcal{L}\}$ in a language $\mathcal{L}$ is a special case of the Parikh image of a regular

Authors' addresses: Amanda Stjerna, Department of Information Technology, Uppsala University, Lägerhyddsvägen 1, hus 10, Uppsala, Sweden, amanda.stjerna@it.uu.se; Philipp Rümmer, Faculty of Informatics and Data Science, University of Regensburg, Bajuwarenstraße 4, Regensburg, 93053, Germany, philipp.ruemmer@ur.de, Department of Information Technology and Uppsala University, Lägerhyddsvägen 1, hus 10, Uppsala, Sweden, philipp.ruemmer@it.uu.se.

ACM 2475-1421/2024/4-ART138
https://doi.org/10.1145/3649855

language. This required combined reasoning about strings and string length has long been identified as a major bottleneck in string solvers [Abdulla et al. 2015; Berzish et al. 2017, 2021; Janků and Turoňová 2020]. Other string solvers make use of Parikh automata [Klaedtke and Rueß 2002], and thus Parikh images in the general case, to handle operations that combine strings and integers (including `str.substr` and `str.at`), which comes at an even higher price in terms of computational complexity [Chen et al. 2020].

The Parikh image, more broadly, is a characterisation of formal languages in terms of their character counts. Given a language $\mathcal{L}$ over an alphabet $\{a_1, \ldots, a_k\}$, the Parikh image is a set of k-dimensional vectors that contains some vector $[m_1, \ldots, m_k]$ if and only if the language $\mathcal{L}$ contains a word in which each a_i occurs m_i times. It is a classical result that the Parikh image of every context-free language (and, thus, also of every regular language) is a semilinear set, i.e., Presburger-definable [Parikh 1966]. In fact, it is possible to compute an existential Presburger arithmetic formula describing the Parikh image of any *context-free* language in linear time [Verma et al. 2005] in the size of the grammar describing the language. For the special case of regular languages, this result was also stated in [Seidl et al. 2004].

In applications, it is often necessary to consider the Parikh image not only of a single regular language, but of the intersection of multiple languages. This happens in string solvers in particular, as conjunctions of string constraints lead to the computation of length images of intersections of regular languages represented as finite-state automata. On the other hand, we are often not interested in a closed-form description of the complete Parikh image, but rather in checking whether the Parikh image contains vectors satisfying some given properties. *The main problem considered in this paper is the following:*

Joint satisfiability of Parikh images modulo Presburger arithmetic constraints

Given regular languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$ over a common alphabet $\{a_1, \ldots, a_k\}$, and a Presburger arithmetic formula $\phi(x_1, \ldots, x_k)$, decide if there is a word $w \in \mathcal{L}_1 \cap \cdots \cap \mathcal{L}_n$ whose Parikh vector $[m_1, \ldots, m_k]$ satisfies ϕ.

The problem can be solved using the classical construction of Parikh images [Seidl et al. 2004; Verma et al. 2005] by first computing the product of finite-state automata accepting the languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$, then extracting a Presburger arithmetic formula ρ describing the Parikh image of the product automaton, and finally checking the satisfiability of $\phi \wedge \rho$.

While theoretically elegant, this construction has several disadvantages that easily turn into bottlenecks when reasoning about Parikh images in algorithms or applications.

Firstly, computing the product automaton is often prohibitively expensive in both memory and CPU time. In several instances we have observed while solving real-world string constraints, the computation of the product of automata exhausts the memory of any machine available due to the exponential blow-up in size of the product, quickly becoming intractable as the number of automata in the product increases.

Secondly, the constructed Presburger arithmetic formula ρ contains a linear number of existential quantifiers in the size of the product automaton, as well as complex Boolean structure, which is needed to express the connectedness of paths considered in the construction. Solving formulas of this kind generally tends to be taxing for solvers [Chen et al. 2020]; in the present case, since the product automaton itself is exponentially big, also formula ρ has exponential size in the number of considered regular languages, and can easily become too complex for today's Presburger arithmetic solvers to handle.

The current best published mitigation for this problem relies on the observation that the Parikh image of an intersection of languages can be over-approximated by the conjunction of the Parikh images of the individual languages. Such over-approximation can be used to derive that no word exists whose Parikh vector satisfies a formula ϕ, falling back to computing the intersection of the languages (or of some subset of the languages) for the satisfiable cases [Janků and Turoňová 2020].

Addressing these concerns, we have developed a calculus for Parikh images of intersections of regular languages that we call PC*. It allows us to interleave the computation of product automata with the derivation of the Parikh image. This enables the two calculations to inform each other, eliminating unnecessary work, and pruning the size of the partial products considered in the computation for a smaller memory footprint. Moreover, our approach allows us to do this precisely, without the expensive fallbacks when approximations fail as in the case of [Janků and Turoňová 2020]. Our calculus is applicable not only to Parikh images, but can be used for reasoning about the image of regular languages under arbitrary homomorphisms into commutative monoids. Besides Parikh images, this includes the length abstraction of regular languages commonly used in string solvers.

Our primary influence for this work is the operation of (finite-domain) constraint programming (CP) solvers [Marriott and Stuckey 1998]. CP solvers typically employ a combination of lazy enforcement of constraints through *propagation*, pruning values from decision variable domains that do not satisfy the constraints during solving, and *branching*, splitting proofs by cases in a way that will encourage maximum propagation.

For our problem, the constraints are automata connectivity and linear equations (Presburger arithmetic formulas). We use propagation to lazily enforce connectivity of automata, removing transitions as they become unreachable during solving or are eliminated by constraints in the Presburger formula. We also branch on the presence or absence of transitions to encourage propagation of the connectivity constraint. This allows us to whittle down the automata (including intermittent products) before moving on with potentially explosive product construction. The underlying observation is that though the domain of our integer variables is infinite, the number of simple paths through the automata (and their product) is finite. *In summary, the key ideas are:*

Key paradigms in PC*

- Enforce automata connectivity constraints lazily;
- Use *propagation* of constraints on the Parikh image to prune transitions from the automata that are incompatible with constraints;
- Intelligently *branch* on the presence or absence of key transitions to drive propagation of the connectivity constraint;
- Compute products of automata lazily, after pruning transitions that violate constraints using propagation.

We implement PC* as a plug-in theory for the Princess automated theorem prover [Rümmer 2008], and additionally wrap the Parikh image solver as a stand-alone tool, Catra. Catra also supports a variant of the approximate method of [Janků and Turoňová 2020], its fall-back variant adapted from [Verma et al. 2005], and an adapter for the nuXmv model checker [Cavada et al. 2014]. Using Catra, we compare PC* to the other back-ends on 37 497 distinct Parikh automata intersection problems generated by Ostrich+ when solving the PyEx string constraint benchmark suite involving string length constraints [Reynolds et al. 2017], finding that PC* outperforms both nuXmv and the baseline method, allowing PC* to solve every problem solved by baseline within 30 s

in under 5 s. PC* in particular outperforms nuXmv on unsatisfiable instances, more than doubling the number of unsatisfiable results found within a 30 s timeout.

We also extend the Ostrich string solver [Chen et al. 2019] with a new back-end based on Catra as the Parikh image intersection solver of the Ostrich+ algorithm, obtaining improvements in both Sat and Unsat performance in Ostrich' results from SMT-COMP 2023, crucially allowing Ostrich to win the unsatisfiable category of the track of quantifier-free, linear constraints over string logic (QF_SLIA) and increasing the number of Sat results by 3% on the same track.

In summary, we contribute:

- The PC* calculus to efficiently check the satisfiability of the Parikh image of an intersection of regular languages modulo Presburger arithmetic side conditions.
- Techniques to efficiently implement PC* in a modern automated theorem prover, including strategies for case splitting, clause learning, and constraint propagation for connectedness.
- The Catra tool for solving such instances, containing an implementation of PC*, the over-approximation described in [Janků and Turoňová 2020], and an adapter for the nuXmv model checker [Cavada et al. 2014].
- Experiments illustrating the performance of PC* on real-world examples from string solving, including 37 497 instances in a standardised format made available for future study.

1.1 Related Work

The problem of satisfying Parikh images over products of regular languages, modulo Presburger arithmetic side conditions, amounts to checking emptiness of products of Parikh automata. Parikh automata are regular automata extended with integer counters with given increments and decrements for each transition, where we allow checking a set of linear constraints on the final values of the counters (but not their intermittent values) [Klaedtke and Rueß 2002]. Parikh automata without constraints on the final values on their registers are also sometimes called cost-enriched automata, weighted automata, or counter automata, depending on exact definitions and side constraints. The decision problem tackled in this paper, determining the emptiness of an intersection of Parikh automata, was shown to be PSPACE-complete [Figueira and Libkin 2015].

Parikh image computations, as well as Parikh automata, feature extensively in string solvers, including as mentioned above Ostrich and Ostrich+ [Chen et al. 2020, 2019], but also forms the basis of Trau [Abdulla et al. 2017], and occurs in Sloth [Holík et al. 2017]. Parikh images frequently appear when introducing cardinality constraints like length or string indexing. While relying on Parikh images (or on being able to check the emptiness of Parikh images under given side constraints), the mentioned papers do not propose any techniques to compute Parikh images.

Seidl et al. and Verma et al. define a closed-form description of the Parikh image of any regular language as an existential Presburger arithmetic formula. An approach to optimise the construction is to over-approximate the Parikh image of a product of k automata $\psi(\mathcal{L}(\mathcal{A}_1) \cap \ldots \cap \mathcal{L}(\mathcal{A}_k))$ with the conjunction of the Parikh images, $\bigwedge_{i=1}^{k} \psi(\mathcal{L}(\mathcal{A}_i))$ [Janků and Turoňová 2020]. Due to the over-approximation, this approach is primarily useful for unsatisfiable instances, and requires falling back to computing the product of the automata before using the standard approach for finding its image originally presented in [Verma et al. 2005]. Our calculus PC*, in a somewhat similar but more fine-grained manner, utilises laziness to postpone or avoid the most expensive steps in the computation of Parikh images of the intersection of regular languages.

Our calculus PC* is also similar in spirit to the work of Stanford et al., who tackle the exponential blow-up resulting from Boolean combinations of finite-state automata in SMT string solvers through the use of symbolic derivatives [Stanford et al. 2021]. The research by Stanford et al. does not consider Parikh images, however. In addition to presenting a decision procedure for lazily

dispatching constraints, we similarly also allow for symbolic labelling of automata to handle large alphabets.

Beyond the field of string solving, Parikh image computation is used as an elementary building block in a variety of areas. For instance, Parikh automata have been proposed as the basis of queries in graph databases [Figueira and Libkin 2015]; Parikh images are used for handling cardinalities in parameterised model checking for epistemic logic [Stan and Lin 2021]; or for handling summation constraints in expressive array logics [Raya 2023]. Our calculus PC*, and the stand-alone solver Catra, are potentially useful in all such applications.

Other generalisations of the Parikh image than the projections we use here have been studied. Prominent examples include generalising the Parikh map to segments of a fixed length [Karhumäki 1980] and the more general Parikh matrix, which contains not only the Parikh vector, but also information about the order of letters. Another notable generalisation is the p-vector, introduced in [Siromoney and Rajkumar Dare 1985], which denotes the position of each letter in the word rather than the number of their occurrences and allows for generalisations into infinite alphabets. All of these in some sense extend the Parikh map. By contrast, the main utility of the formulation introduced here is to reason about Parikh images *lazily,* thereby potentially obtaining answers more quickly. We expect that our calculus PC* can be generalised to the mentioned functions on formal languages as well, but leave such investigations to future work.

2 AN INTUITION FOR OUR APPROACH

In this section, we will introduce an intuition for how a string constraint problem in an automata-based solver like Ostrich+ is translated to a Parikh automata intersection problem, and solved using PC*.

We use PCRE regular expression notation here and throughout the paper, writing them `like this`. This means that `|` is alternation, `*` the Kleene star, and `.` matches any single character. For the length of a string s, we write $|s|$. All number variables in this example are in $\mathbb{N}$ with 0.

Example 2.1. Consider the following set of string constraints over string variables s_1, s_2 and integer variables n, i:

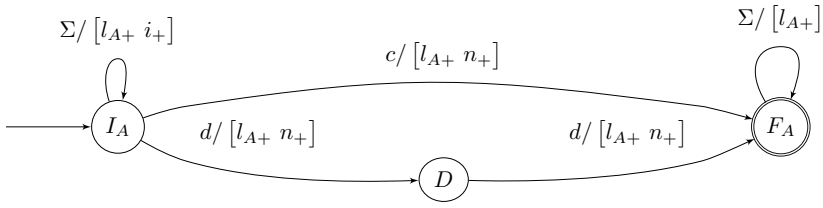
(i): $s_1 \in \mathcal{L}(\texttt{c|dd})$.
(ii): $s_2 \in \mathcal{L}(\mathcal{B})$, the language accepted by automaton $\mathcal{B}$ of Fig. 1b (ignoring the registers l_B).
(iii): $s_1 = \texttt{substring}(s_2, i, n)$, that is s_1 is an n-length substring of s_2 starting at offset i.
(iv): $n > i$, that is what comes before s_1 in s_2 is shorter than s_1.
(v): $i > 0 \wedge |s_2| - i - n > 0$, that is there is at least one character in s_2 before and after s_1.

Although the constraints are simple, it should be noted that decidability results for string constraints involving integers are notoriously hard to obtain; many state-of-the-art string solvers, for instance Z3 [de Moura and Bjørner 2008], different versions of Z3-str [Mora et al. 2021; Zheng et al. 2013], or cvc5 [Barbosa et al. 2022] are not guaranteed to be complete or to terminate on such constraints (but, of course, they often achieve very good performance in practice).

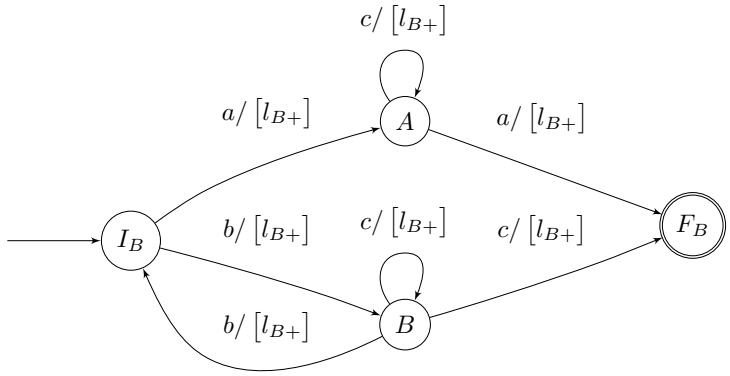
A decision procedure for a fragment of string constraints with integers, covering our constraints is defined in [Chen et al. 2020]. We will first perform translation into Parikh automata, then break down how the constraints will be handled by PC*. This example will recur in a more formal fashion in Section 4, which can be read in parallel if (even) more detail is desired.

2.1 Translating the Constraints into a Parikh Automata Intersection Problem

By a Parikh automaton we mean a standard non-deterministic finite automaton (NFA) with a set of integer registers (Parikh registers) that are incremented (or decremented) at each transition. A formal definition can be found in Definition 4.1.



(a) The automaton A', whose Parikh registers count the length of the string accepted by the automaton (register l_A), the start offset of the substring (i) and length (n) of the substring matching c|dd. Note the symbolic transitions in the starting and accepting states matching any symbol in the alphabet!



(b) The automaton B, where the Parikh registers count the length of the string in register l_B.

Fig. 1. The collection of automata we use as running examples, both derived from Example 2.1.

To solve Example 2.1 using PC* in a theorem proving context, Constraint (i) to Constraint (iii) can be translated to the product of Parikh automata $\mathcal{B} \times \mathcal{A}'$. Automaton $\mathcal{B}$ is given in Fig. 1b. Automaton $\mathcal{A}'$ (Fig. 1a) is an automaton defining the pre-image of $\mathcal{L}(\texttt{c|dd})$ under the `substring` function, and obtained by applying the construction described in [Chen et al. 2020]. Intuitively, $\mathcal{A}'$ describes all tuples (s_2, i, n) such that $\texttt{substring}(s_2, i, n)$ is in $\mathcal{L}(\texttt{c|dd})$. To this end, $\mathcal{A}'$ contains registers l_A, i, n, which counts the length of the overall string read, the length of the prefix before the extracted substring, and the length of the substring, respectively. Registers are assumed to start from zero in each automaton run.

In the automata of Fig. 1, we use the notation $a/\left[x_+\right]$ to mean that a transition reads an input character a and increments register x by one. We will omit zero-valued increments and assume that all registers are scoped to their automaton. The increments are usually represented as a vector (hence the brackets), but as it is mostly sparse here, we use the symbolic notation $\left[x_{1+}\right]$ rather than the more cumbersome $[0, 1, 0, \ldots, 0]$.

Note that the labels of transitions can be ranges of characters with Σ representing any character in the alphabet.

Intuitively, Example 2.1 is unsatisfiable. By inspecting the automata, we realise that the path using dd in $\mathcal{A}'$ is unusable since are no d-labelled transitions in $\mathcal{B}$. This means that $s_1 = \texttt{c}$, and thus $n = 1$. That means that Constraint (iv) implies that $i = 0$. However, no path through B passes by a c

without a preceding character. Therefore, already Constraint (i), Constraint (iii) and Constraint (iv) together are unsatisfiable.

An eager approach to finding the Parikh image, as described in [Verma et al. 2005], would start by computing the product $\mathcal{A}' \times \mathcal{B}$, translating it to a Presburger formula with i, n, etc., as free variables, and then adding Constraint (iv) and Constraint (v) to the resulting set of linear inequalitites. Scalability of this construction is limited due to the size of the product automata to be computed, and the complexity of the resulting Presburger formula.

2.2 Solving the Parikh Automata Intersection Problem Using PC*

We will now proceed to show how our calculus PC* can lazily prove the unsatisfiability of the running example. The calculus interleaves several reasoning principles, which we will later define precisely as calculus rules: (i) Similarly, as in [Verma et al. 2005], we first describe each automaton as a flow network, counting how often each transition is taken in a run of the automaton. The flow constraints are an over-approximation of the possible accepting runs of an automaton. (ii) We use a linear integer solver to simplify the flow constraints and prune away paths through the automata that are not feasible. (iii) We use a tailor-made propagation algorithm to identify disconnected transitions of the automata that can never be taken and lazily add the corresponding constraints. (iv) When propagation cannot infer further constraints, we use case splitting to subdivide the problem into smaller parts. (v) Once the constituent automata of a product have become sufficiently small, we compute the precise product automaton.

2.2.1 Approximating Paths Through Automata Using Flow Analysis. We associate each transition t of $\mathcal{A}'$ and $\mathcal{B}$ with a fresh variable x_t ranging over natural numbers (i.e., non-negative integers). These variables represent how many times each transition is taken. Hence, the final value for registers $r_1, \ldots, r_k$ of each automaton is the element-wise sum:

$$\begin{bmatrix} r_1 \\ \vdots \\ r_k \end{bmatrix} = \sum_t x_t \cdot \bar{K}_t \text{ where } \bar{K}_t \text{ are the increments of transition } t. \tag{1}$$

We proceed by adding linear constraints requiring transition variables to represent a flow through their automaton by adding linear constraints stating that the number of incoming transitions is equal to the number of outgoing ones. E.g., state B of automaton $\mathcal{B}$ would have the sum

$$\cancel{x_{\langle B,\mathrm{c},B\rangle}} + x_{\langle I_B,\mathrm{b},B\rangle} = \cancel{x_{\langle B,\mathrm{c},B\rangle}} + x_{\langle B,\mathrm{c},F_B\rangle} + x_{\langle B,\mathrm{b},I_B\rangle} \tag{2}$$

where we let $x_{\langle q,l,q'\rangle}$ refer to the integer variable associated with a transition from state q to state q' with label l. The initial state of the automaton receives an additional inflow of 1, and accepting states have an additional common outflow of 1. Note that self-loops cancels themselves out. Therefore, when a loop can become unreachable, additional constraints are required to ensure consistency.

2.2.2 Arithmetic Flow Simplification. We then use linear arithmetic reasoning on the register equations (1) and flow equations (2) to simplify the counters associated with transitions. As an example, we start with $\mathcal{A}'$. Substituting back solutions to the equations for $\mathcal{A}'$ to the automaton, we obtain the automaton in Fig. 2, where the notation a/t now denotes a transition that accepts letter a and is taken t times. In this case, the transitions of the automaton can be specified purely in terms of the free variables we care about representing: the length of the accepted word (l_A), the start of the substring (i), and the length of the substring (n).

Having obtained this representation, we conclude that $1 < n \leq 2$ from Constraint (iv) and Constraint (v). The lower bound directly follows from Constraint (iv) and Constraint (v), whereas the upper bound is obtained by the following reasoning. Since all transition variables are non-negative

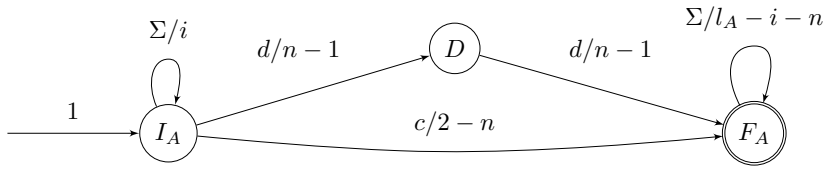
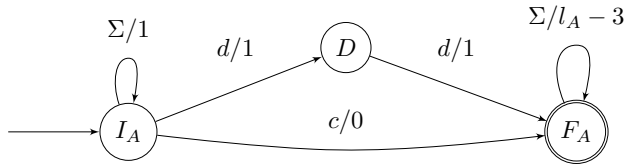
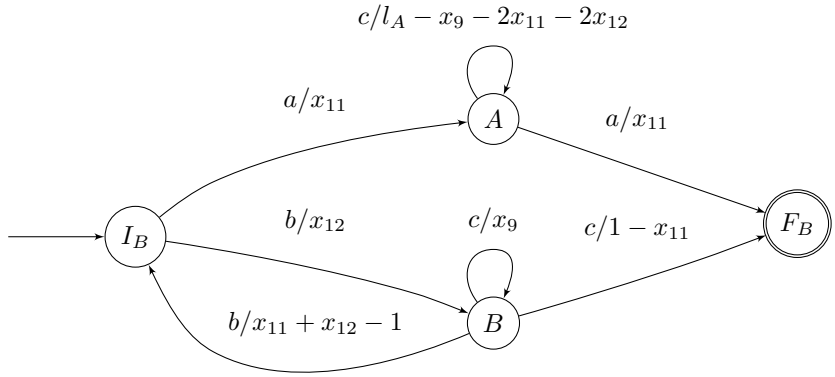


Fig. 2. $\mathcal{A}'$ after even more simplification using linear algebra, with most transitions expressed in terms of n. Intuitively, this version captures the fact that n is given by distributing the incoming flow of 1 across the two outgoing transitions from $I_{A'}$, the initial state.



(a) $\mathcal{A}'$ with its associated transition variables in symbolic form.



(b) $\mathcal{B}$ with its associated transition variables in symbolic forms. The large number of implicitly existentially quantified variables x_i on transitions suggest that this automaton has a more complex structure with relation to its (free) target variable representing the string length than $\mathcal{A}'$.

Fig. 3. The automata after arithmetic simplification.

integers (a transition cannot be used a negative amount of times), $n \leq 2$ from the I_A to F_A transition. Therefore, it follows that $n = 2$, which implies that the I_A to F_A transition can never be used under these constraints and that we must use the two d-labelled transitions, both now taken $n - 1 = 1$ times. Substituting the derived value of n and applying similar reasoning to automaton $\mathcal{B}$, we arrive at the simplified automata in Fig. 3.

Note that the per-automata length counting registers have been assigned the same solver variable l_A. Since they have to be equal, either one can be used in both automata through similar applications of equality elimination rules.

2.2.3 Case Splitting and Connectivity Propagation. We now have a choice of two paths through automaton $\mathcal{B}$; the upper through state A or the lower through state B. Since arithmetic reasoning

and propagation are not able to resolve this choice, we perform a case split by selecting a transition variable that would disconnect some strongly connected component of automaton $\mathcal{B}$, in this case the transition from I_B to state B guarded by variable x_{12}. We split the reasoning into the cases $x_{12} > 0$ (transition used) and $x_{12} = 0$ (transition unused). For presentation, we focus on the latter case, as the former can be handled in a similar way.

In the case $x_{12} = 0$, we can conclude that the state B is now unreachable, which means that its outgoing transitions are now unusable. Propagation can therefore infer the additional equations $x_9 = 0$ and $x_{11} = 1$.

2.2.4 Computing Products. After discounting all transitions that can no longer be taken, we are left with two (small) flat automata, and can compute their product with relative ease. By putting off computing the product $\mathcal{B} \times \mathcal{A}'$ until after performing linear reasoning, and using that to prune transitions, we have computed a smaller product than we would have with an eager approach. Computing the product, we will immediately notice that the d-labelled transition of automaton $\mathcal{A}'$ has no correspondence in $\mathcal{B}$, leading to an empty product. We can close the proof goal and backtrack, and we will eventually derive that the imposed constraints are unsatisfiable by repeating the same process on the other branch.

3 PRELIMINARIES

We first survey some of the required background on finite-state automata and the Parikh image. In addition, we assume basic familiarity with first-order logic, Presburger arithmetic, and the classical sequent calculus; for reference, see e.g. [Fitting 1996].

3.1 Monoids

A monoid $M = \langle X; \oplus; 0_M \rangle$ is an algebraic structure consisting of the non-empty carrier set of elements, X; an associative binary operation $X \times X \to X$ denoted as $\oplus$, that is where for all $a, b, c \in X$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$; and an identity element $0_M \in X$ such that $0_M \oplus a = a \oplus 0_M = a$ for every $a \in X$. We sometimes use integer multiplication to represent a repeated application of $\oplus$, e.g. $3a = a \oplus a \oplus a$, for $a \in X$. M is called *commutative* if $\oplus$ also commutes, that is if $a \oplus b = b \oplus a$ for all $a, b \in X$.

A *homomorphism* is a structure-preserving map between two monoids $M = \langle X; \oplus; 0_M \rangle$ and $M' = \langle X'; \oplus'; 0_{M'} \rangle$, that is a map $h : X_1 \to X_2$ such that $h(a \oplus b) = h(a) \oplus' h(b)$ and $h(0_M) = 0_{M'}$.

3.2 Languages, Finite-State Automata and Their Products

We define an alphabet as a finite set of symbols Σ with words Σ^*, and the concatenation operation as $s_1 \circ s_2$ over two strings s_1, s_2. Note that $\Sigma^* = \langle \Sigma; \circ; \epsilon \rangle$, is a non-commutative monoid, referred to as the free monoid on Σ. The string length function, $|s|$ is an example of a homomorphism between Σ^* and $\mathbb{Z}$.

A *finite-state automaton* $\mathcal{A}$ with alphabet Σ is a tuple $\langle Q, q_I, F, \delta \rangle$, where Q is the set of states, q_I the initial state, F the set of accepting states, and $\delta \subseteq Q \times \Sigma \times Q$ the transition relation. We write a transition $t = \langle q, l, q' \rangle \in \delta$ as $t = q \xrightarrow{l} q'$. Similarly, we use the notation $q \to$ to refer to the set of transitions starting in q, and $\to q$ to refer to the set of transitions coming into q, whenever the automaton is clear from the context.

We will let variables $t, t', t_1, \ldots, t_n$, etc., denote transitions, $q, \ldots, q_n$ states, and $\mathcal{A}, \ldots, \mathcal{A}_n$ automata, and use subscript indexing ($\delta_{\mathcal{A}}$) to refer to the transitions, states, etc., of a given automaton.

A *path* $p = \langle q_0 l_1 q_1 \ldots q_n \rangle$ of an automaton $\mathcal{A}$ is a sequence of states $q_0, \ldots, q_n$ interleaved with letters $l_1, \ldots, l_n$ such that $q_0 = q_I$ and $q_{i-1} \xrightarrow{l_i} q_i$ for $i \in \{1, \ldots, n\}$. The path is *accepting* if the end

state is accepting, $q_n \in F$. The *set of paths* of $\mathcal{A}$ is denoted by $\text{Paths}(\mathcal{A})$. Additionally, we use the notation $\text{Paths}(\mathcal{A}, q)$ to mean all paths ending in state q. We write $t \in p$ to express that transition t occurs on path p. The *states* of a path $p = \langle q_0 l_1 q_1 \dots q_n \rangle$, denoted $\text{States}(p) = \{q_0, \cdots, q_n\}$ are the states visited along p. Note that $q_I \in \text{States}(p)$ for every path since all paths start in the initial state. The *word* of a path $p = \langle q_0 l_1 q_1 \dots q_n \rangle$ is the word $w(p) = l_1 \circ \cdots \circ l_n \in \Sigma^*$ formed by the labels on the path. Finally, the set of words accepted by an automaton $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$, is the set of words of accepting paths.

The *product* of two automata $\mathcal{A}_1 = \left\langle Q^1, q_I^1, F^1, \delta^1 \right\rangle$ and $\mathcal{A}_2 = \left\langle Q^2, q_I^2, F^2, \delta^2 \right\rangle$ is the automaton

$$\mathcal{A}_1 \times \mathcal{A}_2 = \left\langle Q_1 \times Q_2, \langle q_I^1, q_I^2 \rangle, F^1 \times F^2, \delta \right\rangle$$
$$\text{with} \quad \delta = \{\langle \langle q, q'' \rangle, l, \langle q', q''' \rangle \rangle \mid \langle q, l, q' \rangle \in \delta^1, \langle q'', l, q''' \rangle \in \delta^2\} \,.$$

The product automaton runs $\mathcal{A}_1$ and $\mathcal{A}_2$ in parallel on an input and only accepts the input if both automata would do so; we have $\mathcal{L}(\mathcal{A}_1 \times \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

3.3 The Parikh Map and Its Image

Formally, the *Parikh map* over an alphabet $\Sigma = \{a_1, \dots, a_k\}$ is defined as in [Kozen 1997]:

$$\psi : \Sigma^* \to \mathbb{N}^k$$
$$\psi(s) = [\#a_1(s), \#a_2(s), \dots, \#a_k(s)]$$

That is, $\psi(s)$ is a vector of the number of occurrences of each character in the language for a given string s. For example, for $\Sigma = \{a, b\}$, we would have $\psi(abb) = [1, 2]$.

We define the image of this map, the *Parikh image*, of some language $\mathcal{L} \subseteq \Sigma^*$ as:

$$\psi(\mathcal{L}) = \{\psi(x) \mid x \in \mathcal{L}\}$$

Thus, we would have $\psi(\{ab, abb\}) = \{[1, 1], [1, 2]\}$. We also sometimes use the standard notation $\#l(w)$ to talk about an individual letter l in a word w. For example, for the Parikh vector above, we would have $\#a(abb) = 1$.

Parikh's theorem states that any context-free language has a Parikh-equivalent regular language (c.f. [Esparza et al. 2011] for a construction of such automata from context-free grammars and [Lavado et al. 2013] for bounds on its size). The Parikh image is therefore a semi-linear set and Presburger-definable. While Parikh's theorem applies to arbitrary context-free languages, in this paper we focus only on regular languages.

3.4 The Parikh Image of a Regular Language Expressed in Presburger Arithmetic

It is known that the Parikh image of any context-free language can be described by a linear-size existential Presburger formula [Verma et al. 2005]. This representation can be straightforwardly adapted for use with a product of regular languages. For an intuition, the approach consists of first computing the product, then assigning each state and transition an existentially quantified non-negative integer variable, and then describing all paths through the automaton through two sets of constraints: flow equations relating the inflow and outflow of each automaton state, and constraints that enforce connectedness by ordering states by distance in a spanning tree rooted in the initial state.

We refer to this model as the baseline approach, though we also apply optimisations as described in Section 7.1. The calculus introduced in this paper, by contrast, lazily enforces the connectedness constraint while also interleaving the computation of products of automata and propagating information between the steps to reduce the amount of work that needs to be done.

4 PROJECTIONS ON PARIKH IMAGES

The Parikh map ψ represents a homomorphism from the (free) non-commutative monoid Σ^* to the (free) commutative monoid $\mathbb{N}^k$. We are, however, often interested in projections of the Parikh map, rather than the full image. We therefore consider *arbitrary* homomorphisms $h : \Sigma^* \to M$, where $M = (X; \oplus; 0_M)$ is a commutative monoid. We give several examples of such projections on Parikh images later in this section.

Observe that every homomorphism $h : \Sigma^* \to M$ can be represented as the composition $h' \circ \psi$, for some homomorphism $h' : \mathbb{N}^k \to M$. One of the insights underlying our approach is that it is more efficient to directly compute a projection $h(\mathcal{L})$ on the Parikh image, than to first compute the standard image $\psi(\mathcal{L})$ followed by projection to some property of interest.

Example 4.1 (String Length)**.** One such simplifying homomorphism can express string length, the problem that originally motivated our study of the Parikh map. This mapping is relevant when solving constraints that combine language membership with string length, for instance the constraint given in the introduction:

$$x \in \mathcal{L}_1 \wedge y \in \mathcal{L}_2 \wedge |x| > |y| \tag{3}$$

To solve this formula, let $M = \mathbb{N}$, and define the homomorphism L by $L(a) = 1$ for all characters $a \in \Sigma$. The length of a string $s = s_1 \circ \cdots \circ s_n$ is given by $L(s) = \sum L(s_i) = 1 + \cdots + 1 = n$, and to solve (3) we can instead solve the equi-satisfiable formula $\alpha \in L(\mathcal{L}_1) \wedge \beta \in L(\mathcal{L}_2) \wedge \alpha > \beta$. This paper proposes efficient native procedures to reason about membership constraints like $\alpha \in L(\mathcal{L}_1)$, avoiding the computation of the complete image $L(\mathcal{L}_1)$. This encoding can be seen applied to automaton B in Example 2.1 (see Fig. 1b).

4.1 Integer Constraints on Strings

Parikh images are also applicable for deciding more general classes of string constraints [Chen et al. 2020]. Consider the substring constraint Constraint (iii) of Example 2.1: $s_1 = \texttt{substring}(s_2, i, n)$, that is s_1 is an n-length substring of s_2 starting at offset i. That constraint belongs to an expressive fragment of string logic that cannot be decided by most state-of-the-art string solvers.

In Section 2 we modelled Constraint (iii) (and the other constraints) using *Parikh automata* [Cadilhac et al. 2011; Klaedtke and Rueß 2002]. A Parikh automaton is a finite-state automaton extended such that transitions are additionally labelled with offset vectors defining the increments of a finite number of counters. This means that Parikh automata recognise words over an extended alphabet $\Sigma \times D$, where $D \subseteq \mathbb{N}^d$ is a finite set of increment vectors (notation as in [Cadilhac et al. 2011]) and Σ is the alphabet of the original automaton.

We use the symbols π_Σ, π_D to denote projections to the first and the second component of a composite letter $(a, \bar{v})$, respectively, and extend those projections to words:

$$\pi_\Sigma((a_1, \bar{v}_1) \circ \cdots \circ (a_k, \bar{v}_k)) \;=\; a_1 \circ \cdots \circ a_k, \qquad \pi_D((a_1, \bar{v}_1) \circ \cdots \circ (a_k, \bar{v}_k)) \;=\; \bar{v}_1 + \cdots + \bar{v}_k.$$

Definition 4.1. A Parikh automaton of dimension $d \geq 0$ is a pair $\langle \mathcal{A}, C \rangle$, where $C \subseteq \mathbb{N}^d$ is a semi-linear set (or, equivalently, a Presburger formula), and $\mathcal{A}$ is a finite automaton with the alphabet $\Sigma \times D$, where $D \subseteq \mathbb{N}^d$. We say that $\langle \mathcal{A}, \varphi \rangle$ recognises a word $w \in \Sigma^*$ if and only if the automaton has a run accepting an extended word $w' \in (\Sigma \times D)^*$ such that $\pi_\Sigma(w') = w$ and $\pi_D(w') \in C$.

Applied to Constraint (iii), the decision procedure in [Chen et al. 2020] will construct a pre-image of s_1 under the substring operation, and check whether this pre-image is consistent with the constraint $s_2 \in \mathcal{L}(B)$, corresponding to the full product seen in Example 2.1. Because substring depends on the values of the integer variables i, n, a Parikh automaton, shown in Fig. 1a, models the pre-image. The Parikh automaton has dimension 3, as it includes also the length register l_A,

besides variables i, n. Intuitively, this construction accumulates any prefix symbol, incrementing i to mark the start of the substring; followed by the automaton representing the substring itself, modified to add a counter to increment n at each transition; followed by a state that accepts any suffix after the matched substring.

Denoting the language described by Fig. 1a as $\mathcal{L}_{pre}$, we can then replace $s_1 = \texttt{substring}(s_2, i, n)$ and $s_2 \in \mathcal{L}(B)$ (of Constraint (i)) with an equi-satisfiable formula that no longer contains any explicit substring operation:

$$p \in \mathcal{L}_{pre} \wedge \pi_\Sigma(p) \in \mathcal{L}(B) \wedge \ \pi_D(p) = \begin{bmatrix} l_A \\ i \\ n \end{bmatrix} \ \wedge 0 \leq i \leq i + n \leq l_A \tag{4}$$

To check the satisfiability of Eq. (4), we need a decision procedure that can process intersections of regular languages (in this case, of $\mathcal{L}_{pre}$ and $\mathcal{L}(B)$, synchronising on Σ), while imposing the side condition $0 \leq n \leq m \leq |p|$ on the increment sum. In [Chen et al. 2020], this decision procedure turned out to be main bottleneck of the string solver, which was one of the motivations to develop the lazy algorithm proposed in this paper.

5 A CALCULUS FOR PROJECTIONS ON PARIKH IMAGES

We start by defining our calculus, PC*, for one automaton, and only extend it to products of automata in Section 6. Assume an automaton $\mathcal{A} = \langle Q, q_I, F, \delta \rangle$ with k transitions $\delta = \{t_1, \cdots, t_k\}$. We use the notations introduced in Section 3.2. For convenience, we introduce the following additional notions:

Definition 5.1. The *transition count*, $\#(t, p)$ is the number of times a transition $t = q \xrightarrow{l} q' \in \delta$ appears on a path p. A *transition selection function* is a function $\sigma : \ \delta \to \mathbb{N}$ labelling every transition $t \in \delta$ with a non-negative number.

We introduce the two predicates that will be used by our calculus, with the following definitions:

Definition 5.2. The Parikh predicate, $\textsc{Im}_{\mathcal{A},h}(\sigma, m)$ holds for some automaton $\mathcal{A} = \langle Q, q_I, F, \delta \rangle$, some homomorphism $h : \Sigma^* \to M$ to a commutative monoid M, some transition selection function $\sigma : \ \delta \to \mathbb{N}$, and some monoid element $m \in M$ if m is an element of the Parikh image of $\mathcal{L}(\mathcal{A})$ modulo h, or more formally, when there is an accepting path $p = \langle q_0 l_1 q_1 \dots q_n \rangle \in \text{Paths}(\mathcal{A})$ such that $\sigma(t) = \#(t, p)$ for all $t \in \delta$, and $m = h(w(p))$.

Definition 5.3. $\textsc{Conn}(\mathcal{A}, \sigma)$ holds for some automaton $\mathcal{A} = \langle Q, q_I, F, \delta \rangle$ and transition selection function $\sigma : \ \delta \to \mathbb{N}$ if for every $t = q \to q' \in \delta$, if $\sigma(t) > 0$ there is some σ-selected accepting path that visits t's starting state q. More formally, if $\sigma(t) > 0$ then there is some path $p \in \text{Paths}(\mathcal{A})$ with $\sigma(t') > 0$ for every $t' \in p$, and $q \in \text{States}(p)$ such that p ends in an accepting state $q' \in F$. The predicate $\textsc{Conn}(\mathcal{A}, \sigma)$ represents the condition that $\mathcal{A}$ is connected under the selection function σ for every transition, and is implied by $\textsc{Im}_{\mathcal{A},h}(\sigma, m)$.

The rules of PC* for one automaton are given in Table 1. The rules operate on sets of formulas and can be interpreted as rules of a one-sided sequent calculus, in which all formulas are located in the antecedent [Fitting 1996]. The rules relate premises $\Gamma_1, \dots, \Gamma_k$ with some conclusion Γ. When constructing a proof, we start with some root Γ, and then apply proof rules to the goals of the proof in bottom-up direction until all goals are closed, or no more rules are applicable.

A proof in which all proof goals are closed shows that the formulas in the root Γ of the proof are inconsistent (have no solutions). An unclosable goal to which no rules are applicable gives rise to a solution of the formulas in the root Γ. Such a goal will only contain formulas in Presburger arithmetic, allowing a solution to be computed using standard algorithms [Harrison 2009].

Table 1. Derivation rules for one automaton.

Name	Rule	Side conditions
Expand	$\dfrac{\textsc{Conn}(\mathcal{A},\sigma),\ \textsc{Flow}(\mathcal{A},\sigma),\ m = \sum_{t\in\delta_\mathcal{A}} \sigma(t)\cdot h(t), E, \Gamma}{\textsc{Im}_{\mathcal{A},h}(\sigma, m), E, \Gamma}$	None
Split	$\dfrac{\textsc{Conn}(\mathcal{A},\sigma), E, \Gamma, \sigma(t) = 0 \mid \textsc{Conn}(\mathcal{A},\sigma), E, \Gamma, \sigma(t) > 0}{\textsc{Conn}(\mathcal{A},\sigma), E, \Gamma}$	if $t \in \delta_\mathcal{A}$
Prop	$\dfrac{\textsc{Conn}(\mathcal{A},\sigma), \{\sigma(t') = 0 \mid t' \in C\}, E, \Gamma, \sigma(t) = 0}{\textsc{Conn}(\mathcal{A},\sigma), \{\sigma(t') = 0 \mid t' \in C\}, E, \Gamma}$	if $t \in \delta_\mathcal{A}$ and $\textsc{Dom}(C, \mathcal{A}, t)$
Subsume	$\dfrac{E, \Gamma}{\textsc{Conn}(\mathcal{A},\sigma), E, \Gamma}$	Split and Prop cannot be applied

We use the convention of splitting the formulas in proof goals into linear (in-) equalities (E) and other formulas (Γ), and assume that predicates Im and Conn only occur positively. The transition selection function σ is represented symbolically and can, in practice, be read as a function from transitions to $\mathbb{N}$-valued terms (e.g. `t` or `t+1`). In our implementation Catra, described in Section 7, σ is a vector of fresh variables with the same size as δ.

To ensure termination, rules can only be applied when they add new formulas on every created branch (the notion of *regularity* of a proof is required [Fitting 1996]). For example, this means that Split can only be applied to proof goals that contain neither $\sigma(t) = 0$ nor $\sigma(t) > 0$, and can never be applied to split on the same term twice on the same branch.

The rule Expand expands an $\textsc{Im}_{\mathcal{A},h}(\sigma, m)$ predicate into the more basic predicate $\textsc{Conn}(\mathcal{A}, \sigma)$, as well as linear equations relating the transitions mentioned by σ with the monoid element m, and linear flow equations described by Flow (below). Since Conn and Im are partially redundant and the difference is covered by Flow, we can remove the instance of Im when applying Expand. In this sense, we split the semantics of the Im predicate into its counting aspect (covered by Flow) and its connectedness aspect (covered by Conn).

In Expand, we use the shorthand notation $h(q \xrightarrow{l} q') = h(l)$, i.e., we allow the homomorphism h to be applied also to transitions t. The predicate $\textsc{Flow}(\mathcal{A}, \sigma)$ represents the flow equations to be generated when expanding $\textsc{Im}_{\mathcal{A},h}(\sigma, m)$. We assume that each application of the predicate introduces fresh integer variables f_q for every accepting state $q \in F$, and define:

$$\begin{aligned}
\textsc{Flow}(\mathcal{A}, \sigma) &= \bigwedge_{q\in F} f_q \geq 0 \wedge \sum_{q\in F} f_q = 1 \wedge \bigwedge_{q\in Q} \textsc{In}(q,\sigma) - \textsc{Out}(q,\sigma) = \textsc{Sink}(q)\\
\textsc{Sink}(q) &= 0 \text{ if } q \notin F, f_q \text{ otherwise.}\\
\textsc{In}(q,\sigma) &= \textsc{StartFlow}(q) + \sum_{t\in(\longrightarrow q)} \sigma(t)\\
\textsc{StartFlow}(q) &= 1 \text{ if } q = q_I, \text{ otherwise } 0.\\
\textsc{Out}(q,\sigma) &= \sum_{t\in(q\longrightarrow)} \sigma(t)
\end{aligned}$$

The rule Split allows us to branch the proof tree by trying to exclude a transition from a potential solution before concluding that it must be included. Intuitively, this is what guarantees our ability to make forward progress by eliminating paths through $\mathcal{A}$.

The Prop rule allows us to propagate (dis-)connectedness across $\mathcal{A}$. It states that we are only allowed to use transitions attached to a reachable state, and is necessary to ensure connectedness in the presence of cycles in $\mathcal{A}$. The rule makes use of the notion of *dominating* sets of transitions, defined as follows:

Definition 5.4. A set C of transitions of an automaton $\mathcal{A}$ *dominates* a transition t, written $\textsc{Dom}(C, \mathcal{A}, t)$, if every accepting path p of $\mathcal{A}$ with $t \in p$ contains at least one transition from C. Notably, $\textsc{Dom}(\emptyset, \mathcal{A}, t)$ for every unreachable transition t, and $\textsc{Dom}(\{t\}, \mathcal{A}, t)$ for every transition t.

The Dom relation can be efficiently implemented in a solver by using standard Ford-Fulkerson/ Edmonds-Karp min-cut between a state and the initial state after removing transitions where $\sigma(t) = 0$. By only performing this computation after such filtering, a solver additionally avoids breaking the rule of adding clauses that already appear in the formula.

Finally, the rule Subsume can be applied when the connectedness of an automaton has been ensured by exhaustive application of the other rules. This suggests a proof strategy where you Prop when you can, Split when you must, and Subsume when neither is possible anymore.

In addition to Table 1, we assume the existence of a rule Presburger-Close, corresponding to a sound and complete solver for Presburger arithmetic formulas, and for constraints over M.

A decision procedure would start from one or multiple predicates $\textsc{Im}_{\mathcal{A},h}(\sigma, m)$ to be satisfied, possibly in combination with other constraints about m. It would then first expand the predicates using the Expand rule, and subsequently apply the other rules to search for a solution.

As illustrated in Section 2, a decision procedure can also perform arithmetic rewriting of the occurring terms and equations. Such reasoning is not necessary for correctness or completeness, but it shortens the examples considerably; we will therefore assume the existence of a rule Algebra that allows us to perform standard algebraic reasoning on linear arithmetic constraints.

5.1 An Example

Here we will return to $\mathcal{A}'$ from Section 2 and perform the steps of Sections 2.2.2 and 2.2.3 with the formal calculus we just established, but exclude Constraint (ii) and therefore the entire automaton $\mathcal{B}$, since we introduce support for products of automata in Section 6. The example then becomes satisfiable, as $s_1 = \mathsf{dd}, i = 1, s_2 = \mathsf{adda}, l_A = 4$ is a satisfying assignment to Constraint (i) and Constraint (iii) to Constraint (v).

Starting with $\mathcal{A}'$, h extracts the increments of a transition, $h(q_1 \xrightarrow{c/i} q_2) = \vec{v}$ for the increment vector $\vec{v}$, concretely $h(I_A \xrightarrow{\Sigma/[l_{A+}, i_+]} I_A) = [l_{A+}, i_+]$. We use the same compact notation here as in Section 2 to represent what is essentially a sparse vector of 1 and 0 coefficients, i.e. $[l_{A+}, i_+] = [1, 1, 0]$. The reader is advised to review Fig. 1a from Section 2 while going through this example.

Initially, we let σ map to fresh variables to obtain, after some simplifications:

$$\begin{array}{llll} \sigma(D \xrightarrow{d/[l_{A+}, n_+]} F_A) = x_1 & \sigma(F_A \xrightarrow{\Sigma/[l_{A+}]} F_A) = x_2 & \sigma(I_A \xrightarrow{c/[l_{A+}, n_+]} F_A) = x_3 & \\ \sigma(I_A \xrightarrow{d/[l_{A+}, n_+]} D) = x_4 & \sigma(I_A \xrightarrow{\Sigma/[l_{A+}, i_+]} I_A) = x_5 & & \end{array}$$

$$\textsc{Flow}(\mathcal{A}', \sigma) = \begin{array}{c} 1 + \cancel{x_5} = \cancel{x_5} + x_4 + x_3 \wedge \\ x_4 = x_1 \wedge \\ x_3 + x_1 + \cancel{x_2} = \cancel{x_2} + 1 \end{array} \tag{5}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x_1 = 1 \land x_2 = l_A - 3 \land x_3 = 0 \land x_4 = 1 \land x_5 = 1 \land n = 2 \land i = 1 \land l_A > 3}\ \textsc{Subsume}}{\textsc{Conn}(A', \sigma) \land x_1 = 1 \land x_2 = l_A - 3 \land x_3 = 0 \land x_4 = 1 \land x_5 = 1 \land n = 2 \land i = 1 \land l_A > 3}\ \textsc{Algebra: =-simp.}}{\textsc{Conn}(A', \sigma) \land x_1 = n - 1 \land x_2 = l_A - n - i \land x_3 = 2 - n \land x_4 = n - 1 \land x_5 = i \land n = 2 \land i = 1 \land l_A > 3}\ \textsc{Algebra: >-reasoning}}{\textsc{Conn}(A', \sigma) \land x_1 = n - 1 \land x_2 = l_A - n - i \land x_3 = 2 - n \land x_4 = n - 1 \land x_5 = i \land n > i \land l_A - i - n > 0 \land i > 0}\ \textsc{Algebra: =-simp.}}{\textsc{Conn}(A', \sigma) \land 1 = x_4 + x_3 \land x_4 = x_1 \land x_3 + x_1 = 1 \land l_A = x_1 + x_2 + x_3 + x_4 + x_5 \land i = x_5 \land n = x_1 + x_3 + x_4 \land n > i \land l_A - i - n > 0 \land i > 0}\ \text{(Expanding Flow)}}{\textsc{Conn}(A', \sigma) \land \textsc{Flow}(A', \sigma) \land l_A = x_1 + x_2 + x_3 + x_4 + x_5 \land i = x_5 \land n = x_1 + x_3 + x_4 \land n > i \land l_A - i - n > 0 \land i > 0}\ \text{(Expanding sum)}}{\textsc{Conn}(A', \sigma) \land \textsc{Flow}(A', \sigma) \land [l_A, i, n] = \sum_{t \in \delta_{A'}} h(t) \cdot \sigma(t) \land n > i \land l_A - i - n > 0 \land i > 0}\ \textsc{Expand}}{\textsc{Im}_{A', h}(\sigma, [l_A, i, n]) \land n > i \land l_A - i - n > 0 \land i > 0}$$

Fig. 4. A proof tree for Constraint (i) and Constraint (iii) to Constraint (v) from Example 2.1, corresponding to handling the Parikh automaton $\mathcal{A}'$ of Fig. 1a.

The proof tree is shown in Fig. 4. Like in Sections 2.2.1 and 2.2.2, by arithmetic reasoning the calculus ends up with fixed values for several of the transition variables x_k, and eventually concludes that the transition directly from state I_A to F_A (variable x_3) is incompatible with the constraint. It is then possible to remove the Conn predicate using Subsume, since Prop is not able to infer further constraints, and no non-trivial applications of Split remain to be done. After this, by further arithmetic reasoning we can derive a solution $l_A = 4, i = 1, n = 2$, and conclude that the root constraint is satisfiable. To obtain values for the string variables s_1, s_2 corresponding to the solution, one can construct an accepting path of the automaton with each transition t taken $\sigma(t)$ times.

5.2 Correctness of PC*

Our correctness proof of PC* consists of two main parts: first, we show that the construction of a proof always terminates, and then that each of the proof rules in Table 1 is an equivalence transformation, i.e., does not change the set of satisfying assignments of a formula. In combination, those two results immediately imply that PC* gives rise to a decision procedure.

5.2.1 *PC* Terminates.*

Lemma 5.1. *Suppose Γ is a set of formulas in which the predicates Im and Conn only occur positively. There is no infinite sequence of proofs $P_0, P_1, P_2, \ldots$ in which P_0 has Γ as root, and each P_{i+1} is derived from P_i by applying one of the rules in Table 1.*

Proof. The rule Expand can only be applied finitely often since each application removes one Im predicate, and none of the rules introduce new instances of the predicate. The rule Subsume can only be applied finitely often since it strictly decreases the combined number of Im and Conn predicates in sets of formulas, and none of the rules increases that number.

To show termination of Split and Prop, observe that the σ in a predicate $\textsc{Conn}(\mathcal{A}, \sigma)$ is never updated on a proof branch, which means that the set of terms $\sigma(t)$ for $t \in \mathcal{A}$ on every branch is finite. Each application of Split and Prop adds a new formula $\sigma(t) = 0$ or $\sigma(t) > 0$ to a proof goal, which can only happen finitely often. □

5.2.2 *The Rules in Table 1 are Solution-Preserving.*

Lemma 5.2. *Consider an application of one of the rules in Table 1, with premises $\Gamma_1, \ldots, \Gamma_k$ and conclusion Γ. An assignment β satisfies the conclusion Γ if and only if it satisfies one of the premises Γ_i.*

Proof. This property has to be shown by analysing the possible applications of each proof rule.

Expand unfolds the definition of the Im predicate. To show that the rule is solution-preserving, we prove the equivalence of the upper and lower sets of formulas:

- Assume that β satisfies the conclusion, which means that there is some accepting path $p = \langle q_0 l_1 q_1 \ldots q_n \rangle \in \text{Paths}(\mathcal{A})$ with $\text{val}_\beta(\sigma(t)) = \#(t, p)$ and $\text{val}_\beta(m) = h(w(p))$. This immediately implies that β satisfies $\textsc{Conn}(\mathcal{A}, \sigma)$, since a path is connected, and $\textsc{Flow}(\mathcal{A}, \sigma)$ since an accepting path satisfies the flow equations. The equation $m = \sum_{t \in \delta_\mathcal{A}} \sigma(t) \cdot h(t)$ holds because of $\text{val}_\beta(\sigma(t)) = \#(t, p)$.
- Assume that β satisfies the premise, which implies that $\text{val}_\beta(\sigma)$ describes a consistent, connected flow of the automaton. By the same argument as in [Verma et al. 2005], this flow can be mapped to an accepting path p of $\mathcal{A}$ such that each transition t occurs on p exactly $\text{val}_\beta(\sigma(t))$ times. Together with the equation $\text{val}_\beta(\sigma(t)) = \#(t, p)$, this implies that β satisfies $\textsc{Im}_{\mathcal{A},h}(\sigma, m)$.

In Split, we make use of the fact that $\sigma(t)$ is $\mathbb{N}$-valued by definition. For any β, clearly exactly one of $\sigma(t) = 0$ or $\sigma(t) > 0$ will be satisfied, implying the property.

For Prop, suppose that $\textsc{Dom}(C, \mathcal{A}, t)$, which means that every accepting path containing t contains at least one of the transitions in C. For a β satisfying $\textsc{Conn}(\mathcal{A}, \sigma)$. This means that every accepting path p where $t \in p$ has at least one transition $t' \in C$ such that $\sigma(t') = 0$. This means that also $\text{val}_\beta(\sigma(t)) = 0$ has to hold since no unbroken path containing t exists.

Finally, for Subsume, observe that if Split cannot be applied, then a goal must contain $\sigma(t) = 0$ or $\sigma(t) > 0$ for every t. In case the formulas in E are inconsistent, an application of Subsume is trivially solution-preserving; therefore assume that E is consistent, which means that it contains exactly one of $\sigma(t) = 0$ or $\sigma(t) > 0$ for each t. Since Prop is not applicable, the transitions t with $\sigma(t) > 0$ must form a connect sub-graph of the automaton; this means that $\textsc{Conn}(\mathcal{A}, \sigma)$ is redundant as it is implied by E. □

6 PARIKH IMAGES FROM PRODUCTS OF AUTOMATA

We now generalise our calculus to natively work with intersections of regular languages, or equivalently products of automata. For this extension, we change the main predicate Im to be indexed by a vector of automata $\langle \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle$. For simplicity, we assume that the sets of states of the k automata (and therefore also the transition sets) are pairwise disjoint.

Definition 6.1. Suppose $\mathcal{A}_1, \ldots, \mathcal{A}_k$ are automata, $h : \Sigma^* \to M$ is a homomorphism to a commutative monoid M, $\sigma : \bigcup_{i=1}^{k} \delta_{\mathcal{A}_i} \to \mathbb{N}$ is a transition selection function, and $m \in M$. The predicate $\textsc{Im}_{\langle \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle, h}(\sigma, m)$ is true exactly when there are accepting paths $p_1, \ldots, p_k$ of the respective automata, such that for each $i \in \{1, \ldots, k\}$ and $t \in \delta_{\mathcal{A}_i}$ it holds that

Table 2. Additional derivation rules for products of arbitrarily many automata.

Name	Rule	Side conditions
ExpandM	$\dfrac{\left\{\begin{array}{c}\textsc{Flow}(\mathcal{A}_i,\sigma),\textsc{Conn}(\mathcal{A}_i,\sigma),\\ m=\sum_{t\in\delta_{\mathcal{A}_i}}\sigma(t)\cdot h(t)\end{array}\right\}_{i=1}^{k},\ \textsc{Im}_{\langle\mathcal{A}_1,\ldots,\mathcal{A}_k\rangle,h}(\sigma,m),\ E,\Gamma}{\textsc{Im}_{\langle\mathcal{A}_1,\ldots,\mathcal{A}_k\rangle,h}(\sigma,m),E,\Gamma}$	None
Materialise	$\dfrac{\begin{array}{c}\textsc{Flow}(\mathcal{A}',\sigma'),\\ \textsc{Conn}(\mathcal{A}',\sigma'),\textsc{Bind}(\mathcal{A}_i,\mathcal{A}_j,\mathcal{A}',\sigma'),\\ \textsc{Im}_{\langle\mathcal{A}_1,\mathcal{A}_{i-1},\mathcal{A}_{i+1},\ldots,\mathcal{A}_{j-1},\mathcal{A}_{j+1},\ldots,\mathcal{A}_k,\mathcal{A}'\rangle,h}(\sigma',m),E,\Gamma\end{array}}{\textsc{Im}_{\langle\mathcal{A}_1,\ldots,\mathcal{A}_i,\ldots,\mathcal{A}_j,\ldots,\mathcal{A}_k\rangle,h}(\sigma,m)E,\Gamma}$	$\sigma'=\textsc{Extend}(\mathcal{A}',\sigma)$, $1\le i<j\le k$, $\mathcal{A}'=\sigma_E(\mathcal{A}_i)\times\sigma_E(\mathcal{A}_j)$

- the multiplicity of t on p_i is consistent with σ, that is, $\sigma(t)=\#(t,p_i)$,
- the automata all accept the same word $w(p_i)=w(p_1)$, and
- the accepted word is mapped to $m=h(w(p_i))=h(w(p_1))$.

For the calculus (Table 2), we first extend Expand to generate flow equations and instances of Conn for each automaton, resulting in a new rule ExpandM. Unlike Expand, ExpandM does not remove the Im predicate, since it is needed to keep track of the currently considered partial products.

The rule Materialise introduces the product of two individual automata $\mathcal{A}_i,\mathcal{A}_j$; this step eliminates $\mathcal{A}_i,\mathcal{A}_j$ as index of the Im predicate, and instead adds the product of the automata restricted to the transitions that can still be taken. The rule also introduces the flow equations and the Conn predicate.

In Materialise, we use the following notation for pruning away parts of an automaton based on the transition selection function σ, only keeping those transitions for which σ is positive:

$$\sigma_E(\mathcal{A})=\langle Q,q_I,F,\{t\in\delta\mid(\sigma(t)=0)\notin E\}\rangle$$

This filtering operation can be optimised to also eliminate states from Q and F that become unreachable; this is kept implicit for sake of presentation at this point.

The rule Materialise has to connect the newly introduced product to the previous automata $\mathcal{A}_i,\mathcal{A}_j$. This is done by extending the selection function σ to σ', mapping the transitions of the product to fresh variables x_t:

$$\textsc{Extend}(\mathcal{A}',\sigma)(t)\ =\ \begin{cases}x_t & t\in\delta_{\mathcal{A}'}\\ \sigma(t) & \text{otherwise}\end{cases}$$

The multiplicity of transitions in the product then has to be related to the multiplicities in the individual automata, modelled using the Bind predicate. The predicate expresses that the multiplicity of a transition $t\in\delta_{\mathcal{A}_i}$ in $\mathcal{A}_i$ has to coincide with the sum of the multiplicities of

transitions in $\sigma_E(\mathcal{A}_i) \times \sigma_E(\mathcal{A}_j)$ derived from t, and similarly for $\mathcal{A}_j$:

$$\textsc{Bind}(\mathcal{A}_i, \mathcal{A}_j, \mathcal{A}', \sigma') = \left\{ \sigma'(t) = \sum_{t' = \langle q, q_R \rangle \xrightarrow{l} \langle q', q'_R \rangle \in \delta_{\mathcal{A}'}} \sigma'(t') \mid t = q \xrightarrow{l} q' \in \delta_{\mathcal{A}_i} \right\} \cup \left\{ \sigma'(t) = \sum_{t' = \langle q_L, q \rangle \xrightarrow{l} \langle q'_L, q' \rangle \in \delta_{\mathcal{A}'}} \sigma'(t') \mid t = q \xrightarrow{l} q' \in \delta_{\mathcal{A}_j} \right\}$$

For precisely one automaton, neither rule applies, and we perform the calculus as before.

6.1 An Example

We return to our example in Section 2. We also reuse the definition of h as the simple projection to extract the counter increments, so in this case, $h(t) = [l_{B+}]$ for every transition t since $\mathcal{B}$ only counts length. The reader is advised to review Figs. 3a and 3b from Section 2 while going through this example. The transition variables of the figures match the ones used here.

We extend Eq. (5) from Section 5.1 with definitions for B shown in Eq. (6):

$$\begin{array}{lll} \sigma(A \xrightarrow{a/l_{B+}} F_B) = x_6 & \sigma(A \xrightarrow{c/l_{B+}} A) = x_7 & \sigma(B \xrightarrow{b/l_{B+}} I_B) = x_8 \\ \sigma(B \xrightarrow{c/l_{B+}} B) = x_9 & \sigma(B \xrightarrow{c/l_{B+}} F_B) = x_{10} & \sigma(I_B \xrightarrow{a/l_{B+}} A) = x_{11} \\ \sigma(I_B \xrightarrow{b/l_{B+}} B) = x_{12} & & \end{array} \tag{6}$$

$$\textsc{Flow}(\mathcal{B}, \sigma) = \begin{array}{r} 1 + x_8 = x_{11} + x_{12} \wedge \\ x_{11} = x_6 \wedge \\ x_{12} = x_8 + x_{10} \\ x_6 + x_{10} = 1 \end{array}$$

The only possible rule at the start is ExpandM, which we use to add the corresponding constraints on each automaton of the product as we would have had in the single-automaton version. After that, we expand the various sums and apply light equality propagation.

We then continue by repeating the steps of Section 5.1, shown in Fig. 4 since the steps to propagate Example 2.1 across the automaton $\mathcal{A}'$ still apply for the product. Note that this fixes the values of all transitions but one of $\mathcal{A}'$, including the free counter variables i, n, whose new constant values we propagate everywhere.

Once we have performed reasoning by equality to eliminate transition variables, we obtain the version of $\mathcal{B}$ shown in Fig. 3b and represented by the clauses in the final node in the tree. This corresponds to the end of Section 2.2.2. Our only options now are to either split on which path we take from I_B by case-splitting on x_{12} or to directly invoke Materialise. The latter would produce a shorter tree in this instance but might lead to a larger product being computed, so we pick Split. This leads to an opportunity for propagation since $x_{12} = 0$ cuts off x_9 from I_B on the left branch. Note that the same opportunity does not present itself on the right branch; the use of the upper path to state A does not preclude visiting state B.

In both cases the automata are now simpler, so we apply Materialise to obtain their product, which for both branches is empty. An empty product will have no outgoing transitions from its initial state, and so will lead to a flow equation $1 = 0$. We can then close the proof and know that the problem is unsatisfiable. A full derivation tree for the example can be found in Fig. 5.

$$\dfrac{\dfrac{\dfrac{\dots \boxed{\wedge 1 = 0} \\ \wedge \textsc{Im}_{\emptyset,h}(\sigma, [l_A, 1, 2, l_A])}{x_9 = 0 \wedge x_7 = l_A - 2 \\ \wedge x_2 = l_A - 3 \wedge x_8 = 0 \\ \wedge x_6 = x_{11} = 1 \wedge x_{10} = 0 \\ \wedge \textsc{Conn}(B, \sigma) \\ \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}\ \textsc{Materialise} \qquad \dfrac{\dots \boxed{\wedge 1 = 0} \\ \wedge \textsc{Im}_{\emptyset,h}(\sigma, [l_A, 1, 2, l_A])}{x_{12} > 0 \wedge \\ x_7 = l_A - x_9 - 2x_{11} - 2x_{12} \wedge \\ x_2 = l_A - 3 \wedge \\ x_8 = x_{11} + x_{12} - 1 \wedge \\ x_6 = x_{11} \wedge \\ x_{10} = 1 - x_{11} \wedge \\ \textsc{Conn}(\mathcal{B}, \sigma) \wedge \\ \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}\ \textsc{Materialise}}{x_{12} = 0 \wedge x_7 = l_A - x_9 - 2 \\ \wedge x_2 = l_A - 3 \wedge x_8 = 0 \\ \wedge x_6 = x_{11} = 1 \\ \wedge x_{10} = 0 \wedge \textsc{Conn}(\mathcal{B}, \sigma) \\ \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}\ \textsc{Prop}}{x_7 = l_A - x_9 - 2x_{11} - 2x_{12} \\ \wedge x_2 = l_A - 3 \\ \wedge x_8 = x_{11} + x_{12} - 1 \\ \wedge x_6 = x_{11} \\ \wedge x_{10} = 1 - x_{11} \\ \wedge \textsc{Conn}(\mathcal{B}, \sigma) \\ \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}\ \textsc{Split}\ x_{12} = 0$$

$$\dfrac{l_A = \sum_{k=6}^{k=12} x_k \wedge x_2 = l_A - 3 \wedge 1 + x_8 = x_{11} + x_{12} \wedge x_{11} = x_6 \\ \wedge x_{12} = x_8 + x_{10} \wedge \textsc{Conn}(\mathcal{B}, \sigma) \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}{x_7 = l_A - x_9 - 2x_{11} - 2x_{12} \\ \wedge x_2 = l_A - 3 \\ \wedge x_8 = x_{11} + x_{12} - 1 \\ \wedge x_6 = x_{11} \\ \wedge x_{10} = 1 - x_{11} \\ \wedge \textsc{Conn}(\mathcal{B}, \sigma) \\ \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}\ \text{Expand } \textstyle\sum, \textsc{Algebra}$$

$$\dfrac{l_A = \sum_{k=6}^{k=12} x_k \wedge x_2 = l_A - 3 \wedge \textsc{Flow}(\mathcal{B}, \sigma) \\ \wedge \textsc{Conn}(\mathcal{B}, \sigma) \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}{l_A = \sum_{k=6}^{k=12} x_k \wedge x_2 = l_A - 3 \wedge 1 + x_8 = x_{11} + x_{12} \wedge x_{11} = x_6 \\ \wedge x_{12} = x_8 + x_{10} \wedge \textsc{Conn}(\mathcal{B}, \sigma) \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}\ \textsc{Algebra}\text{: expand } \textsc{Flow}$$

$$\dfrac{l_B = \sum_{k=6}^{k=12} x_k \wedge i = x_5 \wedge l_A = \sum_{k=1}^{k=5} x_k \\ \wedge n = x_4 + x_1 \wedge \textsc{Flow}(\mathcal{B}, \sigma) \wedge \textsc{Conn}(\mathcal{A}', \sigma) \\ \wedge \textsc{Conn}(\mathcal{B}, \sigma) \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, i, n, l_B]) \\ \wedge l_A = l_B \wedge l_A - i - n > 0 \wedge 0 < i < n}{\textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, i, n, l_B]) \wedge l_A = l_B \wedge l_A - i - n > 0 \wedge 0 < i < n}\ \textsc{ExpandM}$$

$$\dfrac{l_A = \sum_{k=6}^{k=12} x_k \wedge x_2 = l_A - 3 \wedge \textsc{Flow}(\mathcal{B}, \sigma) \\ \wedge \textsc{Conn}(\mathcal{B}, \sigma) \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, 1, 2, l_A])}{l_B = \sum_{k=6}^{k=12} x_k \wedge i = x_5 \wedge l_A = \sum_{k=1}^{k=5} x_k \\ \wedge n = x_4 + x_1 \wedge \textsc{Flow}(\mathcal{B}, \sigma) \wedge \textsc{Conn}(\mathcal{A}', \sigma) \\ \wedge \textsc{Conn}(\mathcal{B}, \sigma) \wedge \textsc{Im}_{\langle \mathcal{A}', \mathcal{B} \rangle, h}(\sigma, [l_A, i, n, l_B]) \\ \wedge l_A = l_B \wedge l_A - i - n > 0 \wedge 0 < i < n}\ \text{Repeat Fig. 4, } l_A = l_B.$$

Fig. 5. A derivation for PC* on the Parikh image strings for the constraints of Example 2.1. Note the constant propagation for n!

6.2 Correctness of PC* for Products of Automata

Since Table 2 only extends the existing rules of Table 1, we focus on the differences compared to the calculus for a single automaton.

6.2.1 *PC* for Products of Automata Terminates.*

Lemma 6.1. *Suppose Γ is a set of formulas in which the product version of Im only occurs positively. There is no infinite sequence of proofs $P_0, P_1, P_2, \dots$ in which P_0 has Γ as root, and each P_{i+1} is derived from P_i by applying one of the rules in Table 2.*

Proof. The rule Materialise can similarly only be used finitely many times, as each application reduces the number of automata in the product of Im by one automaton, until only one remains and Lemma 5.1 for single-automaton instances apply.

ExpandM can only be applied precisely once per Im term since each application introduces an identical set of formulas and we have a generic side condition that no rule may add only redundant formulas. □

6.2.2 The Rules in Table 2 are Solution-preserving. Since our calculus now includes a rule introducing new variables, the Materialise rule, we have to slightly generalise the notion of solution-preservation:

Lemma 6.2. *Consider an application of one of the rules in Table 2, with premises $\Gamma_1, \ldots, \Gamma_k$ and conclusion Γ. An assignment β (over the symbols in Γ) satisfies the conclusion Γ if and only if there is an extension β' of β satisfying one of the premises Γ_i.*

Proof. We have to consider the two new rules in Table 2. The result is immediate for ExpandM, since this rule does not remove the Im predicate from a proof goal, and the newly introduced formulas are all implied by the Im predicate.

For Materialise, observe that the existence of an accepting path in $\mathcal{A}_i \times \mathcal{A}_j$ is equivalent to the existence of individual paths in $\mathcal{A}_i, \mathcal{A}_j$ accepting the same word. The path in the product will satisfy the flow equations and connectedness, and it will be related to the individual paths as stipulated by the Bind predicate. □

7 IMPLEMENTATION

We implement PC* for Parikh automata as described in Section 4.1. The artefact submitted along with this paper is a program that reads an instance file with one or more products of one or more Parikh automaton with transition labels defined as ranges of Unicode characters, along with a set of constraints on the final values of their registers expressed as Presburger arithmetic in a C-like syntax. We call this program Catra.

Catra is written in Scala, with the calculus described in this paper implemented as a theory plug-in for the Princess automated theorem prover [Rümmer 2008], which also performs the Presburger reasoning. For comparison, we also provide an implementation of the baseline method from [Verma et al. 2005], a direct translation that uses the nuXmv symbolic model checker [Cavada et al. 2014] to solve our constraints, and the approximation described in [Janků and Turoňová 2020] on top of the standard baseline back-end. An example of an input file corresponding to our running example introduced in Section 2 can be found in the root directory of the artefact [Stjerna and Rümmer 2024].

Catra uses symbolic labels for automata. A symbolic label is defined as a finite range of Unicode code points. This allows representing regular expression patterns like `(a-z)` as `a -> b [a, z]` when it would have otherwise required 27 non-symbolic transitions.

In satisfaction mode, supported by all backends, Catra tries to satisfy the constraints expressed by the input file, reporting Sat with register assignments or Unsat. Additionally, baseline and PC* also support generating the Presburger formula describing the constraints of the input file, i.e., computing a closed-form representation of the complete Parikh image.

7.1 Implementing the Baseline

As a baseline, we use the same Presburger solver (Princess), input file parser, and automaton implementation as Catra. We do this to better analyse the impact of the calculus rules themselves. Adapting [Verma et al. 2005], we produce quantified Presburger formulae for each successive

term and add them to Princess. We compute the product incrementally term by term, checking satisfiability at each step. We use a priority queue to select automata for each step and order them by their number of transitions. We use this heuristic to put off computing large (and therefore slow) products until we have to, hoping to find an empty intermittent product. This is roughly similar to the approach taken in [Janků and Turoňová 2020].

Algorithm 1: How we implement the baseline approach

Data: $\mathcal{A}_1, \ldots, \mathcal{A}_n$ automata, other constraints Γ
Result: Sat or Unsat
$p \leftarrow$ `newTheoremProver()`
`assert`(p, Γ)
foreach $\mathcal{A}_i$ **do**
  `assert`$(p, \psi(\mathcal{A}_i))$
  if p *is* Unsat **then**
    break
  end
end
$q \leftarrow$ `newPriorityQueue`$(\mathcal{A}_1, \ldots, \mathcal{A}_n)$
while p *not* Unsat *and* $|q| > 1$ **do**
  $\mathcal{A}, \mathcal{A}' \leftarrow$ `dequeue`(q)
  `assert`$(p, \psi(\mathcal{A} \times \mathcal{A}'))$
  `enqueue`$(q, \mathcal{A} \times \mathcal{A}')$
end
return *p's SAT status*

As an optimisation, our automata (including intermittent products) have dead states eliminated during construction. Any automaton we produce contains only states that are both reachable from the initial state and have a path to an accepting state. We never perform any other minimisation on the automata for either backend. More complex minimisation was left out since performing minimisation on automata with counters is non-trivial.

7.2 Heuristics and Search Strategies

PC* as described in Sections 5 and 6 leaves some choices unspecified, including the priority of rules and the order of their arguments. In this section, we address these and describe additional implementation details and techniques used to enhance Catra.

7.2.1 Splitting, Materialisation, and Propagation. We order our rule applications as follows: first, propagate connectedness if possible, then perform materialisation if tractable as defined below, then finally resort to splitting as a last resort.

In addition to applying Split as described in Table 1 to randomly selected transitions, we prefer splitting to sever a strongly connected component from the initial state. We randomly select an automaton where we can compute a cut between an SCC and the initial state, that is, where the SCC does not contain the initial state and where the sum of the transition variables of the transitions in the cut is not known to be positive. If there are multiple such strongly connected components, we choose one randomly. We then proceed to split on the sum of the transition variables of the cut as if it were a regular transition, e.g., its sum being zero or nonzero. In this way, we drive PC* towards applying Prop.

The implementation of the connectedness constraint is opportunistic and straightforward. We compute a set of dead states by performing forward and backward reachability computations on

an automaton, where we disregard any transition whose associated variable is known to be zero. After that we add clauses ensuring any transition variable associated with a transition starting in a dead state is zero.

Product materialisation is the final piece of the puzzle. In the current implementation we put off computing intermediate products until at most six transition variables of one of the automata are not known to always be used (> 0) or always unused (≤ 0). The number was chosen experimentally. The other automaton for the product is selected randomly.

7.2.2 Clause Learning. Catra enables clause learning by default when using our backend, as it has been experimentally shown to increase the performance in aggregate (though not strictly). We currently only implement minimal clause learning based on forward-reachability cuts. No sophisticated clause learning for products has been implemented.

7.2.3 Random Restarts. Finally, we perform restarts scaled by the Luby series [Luby et al. 1993]. Experimental results have shown this to have a large improvement in performance, which is unsurprising given how many random choices we make during solving and how tail-heavy our problem is.

8 EVALUATION

We evaluate the performance of Catra on 37 497 instances of Parikh automata intersection problems generated by the Ostrich+ string constraint solver [Chen et al. 2020] when solving the PyEx benchmarks, which are string constraints from symbolic execution that are known to be hard for many solvers [Reynolds et al. 2017]. After generating an initial 38 227, we remove 314 instances solved in under five seconds by baseline as well as 416 instances that were duplicates of other instances in the set.

We also attempted to benchmark instances generated by Ostrich+ solving the Kaluza benchmarks [Saxena et al. 2010] (38 227 instances), but discarded them since they all turned out to be trivial for our tool Catra: every instance could be solved in under five seconds, with a mean runtime of about 0.1s. The benchmarks are run on commit `d54e33b` of Catra.

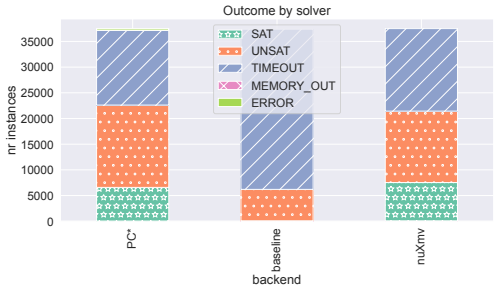
The benchmarks for PC* and nuXmv are executed in parallel on a server running Ubuntu 22.04.3 LTS with an AMD Ryzen 9 5900X processor at 2.27 GHz and 12 cores, with 4 threads sharing the same JVM. Baseline was executed one JVM instance per input as 6 jobs in with a 4 GB heap each on an Intel i5-10600, 3.3 GHz CPU with 6 cores and 12 threads. Simultaneously, StarExec ran the same experiment and produced almost identical results.

We compiled the code using Scala 2.13.12 and executed the experiments on OpenJDK Java 1.8 with a maximum heap of 100 GB. We used nuXmv version 2.0.0 invoked as a subprocess for each instance. Instances were executed in batches of 10, each given a fresh JVM. Each JVM was warmed up for 10 s on a random benchmark from the set before starting to execute. We believe this represents a realistic use case where PC* is used to support, e.g., a string solver. Experiments were executed in random order for all backends. Each instance got a time budget of 30 s.

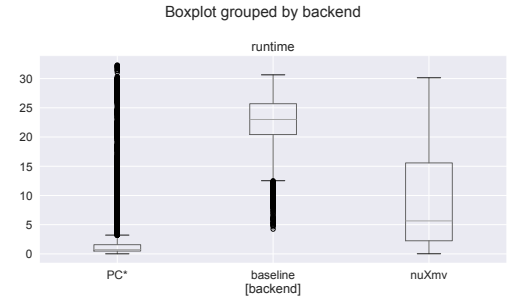
All runtimes are measured in wall-clock time as observed by the JVM when executing the instance, and exclude time spent parsing (usually far below 0.1 s).

8.1 Execution Time and Ability to Solve Instances

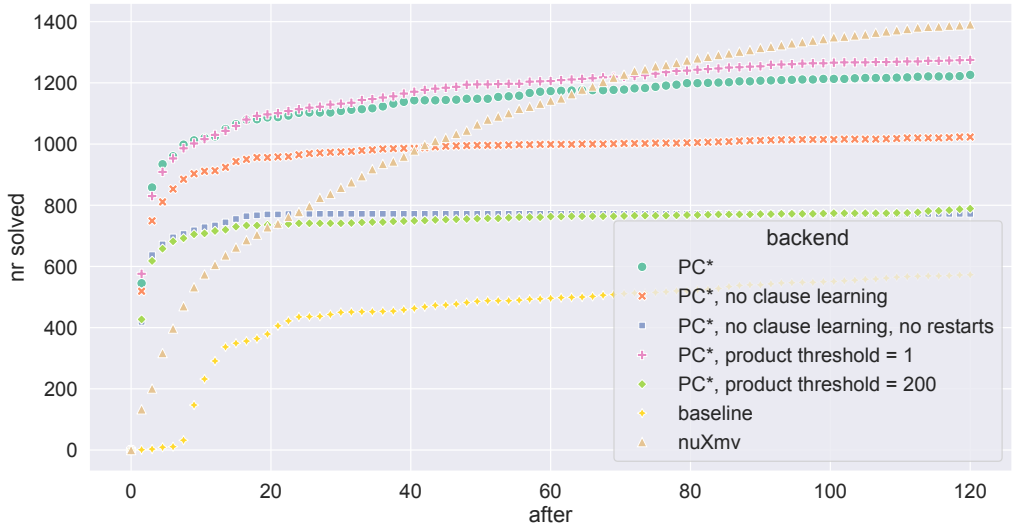
In Fig. 6a, we show how many of the 37 497 instances the respective back-end could solve. A summary of their outcomes by instance type is also available in Table 3. Note that many instances lack a ground truth as they are solved by only one backend. We see that PC* generally outperforms nuXmv on determining unsatisfiability, as does baseline, while being similar at satisfiable instances.



(a) The division of statuses per backend.



(b) The distribution of runtimes for solved instances per backend, assuming they were solved within the 30 s timeout. The whiskers use the default placement in Pandas, 1,5 times the IQR (interquartile range).



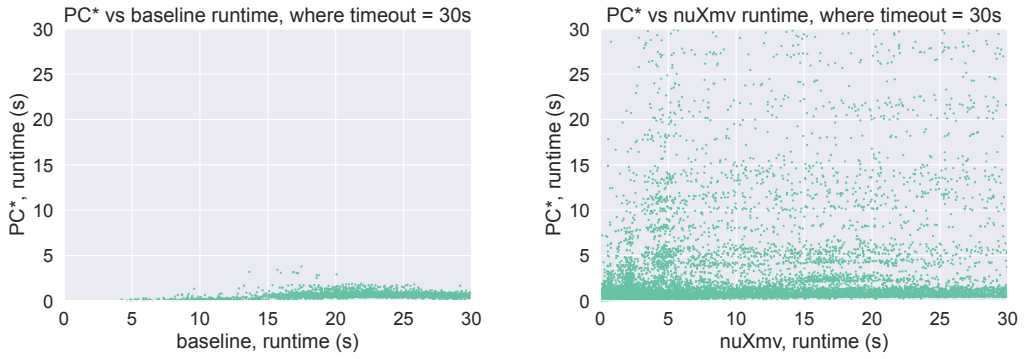
(c) The number of instances solved as the time budget increases, simulated from one 120 s-timeout run.

Fig. 6. Evaluation of PC* solving Parikh automata problems generated by Ostrich.

Both PC* and nuXmv outperforms baseline on satisfiable instances. On satisfiable instances, nuXmv and PC* have similar performance.

Baseline performs worse on satisfiable instances because it executes a heuristic meant to detect unsatisfiability early, similar to [Janků and Turoňová 2020]. The heuristic is enabled since the improvement is significant compared to the extra cost.

Finally, in Figs. 6b, 7a and 7b and we compare runtimes for solved instances between the backends. We see that nuXmv has a more even spread of runtimes up to 20 s, while both PC* and baseline tend towards solving their instances quickly or not at all, though PC* does have a long tail of outlier instances that finish as the timeout increaseses. Notably, Fig. 7a shows that any instance solved by baseline in 30 s is also solved by PC* in under 5 s.



(a) Per-instance runtime between baseline and PC* (b) Per-instance runtime between nuXmv and PC*

Fig. 7. Per-instance duels between backends.

Table 3. Number of successful results within a timeout of 30 s. Instances solved by no backend within the timeout (about half of the set) are omitted from the table.

backend kind	PC*	baseline	nuXmv
SAT	6587	8	7532
UNSAT	16022	6193	13903

8.1.1 Scalability. To investigate scaling we additionally execute 2 000 randomly sampled (without replacement) benchmarks with a 120-second timeout. We add runs of PC* with clause learning and restarts disabled, and with only clause learning disabled to show the impact of Section 7.2.3, and Section 7.2.2 respectively. These experiments are executed on a smaller machine, a six-core AMD Ryzen 5 2600 but the same Java version. Their max heap size is set to 20 GB since this system has less RAM. The baseline experiments are executed separately as only two threads.

A cactus plot showing the number of instances solved within a given timeout can be seen in Fig. 6c. We see that PC* outperforms nuXmv in general but that nuXmv might scale better on very long runtimes. This is likely due to two factors. First, nuXmv is more mature than PC*, and its more general model checking methods might pay off on more difficult instances compared to problem-specific methods. Second, for longer-running calculations clause learning as described in Section 7.2.2 might matter more. As clause learning in CATRA is rudimentary and generalises poorly across product computations, there is a performance hit.

8.2 Evaluation in OSTRICH

To evaluate the effectiveness of PC* in a string solver, CATRA was experimentally integrated as the Parikh automata product solver into the string solver OSTRICH version 1.3. OSTRICH is an independently developed solver that participated in the recent SMT-COMP 2023[1], winning the single-query track for quantifier-free strings (QF_S), as well as dominating other solvers on unsatisfiable string benchmarks [SMT-COMP 2023].

[1] https://smt-comp.github.io/2023/participants/ostrich

Table 4. Number of solved benchmarks in the set of quantifier-free strings with linear integer arithmetic constraints (QF_SLIA) at SMT-COMP 2023. The numbers are from the competition results, except for CA-Str, which is executed by us on the same cluster as the competition with the same resources, and the two virtual portfolio solvers Ostrich+CA and Competition which aggregate the best results from Ostrich/CA-Str and all the competition results for non-Ostrich solvers respectively.

solver kind	CA-Str	Competitors	Ostrich 1.3	Ostrich+CA	Z3-Noodler	cvc5	z3alpha
Sat	6411	14207	9586	9913	7344	14069	13599
Unsat	5566	7574	7356	7389	5367	7377	7298

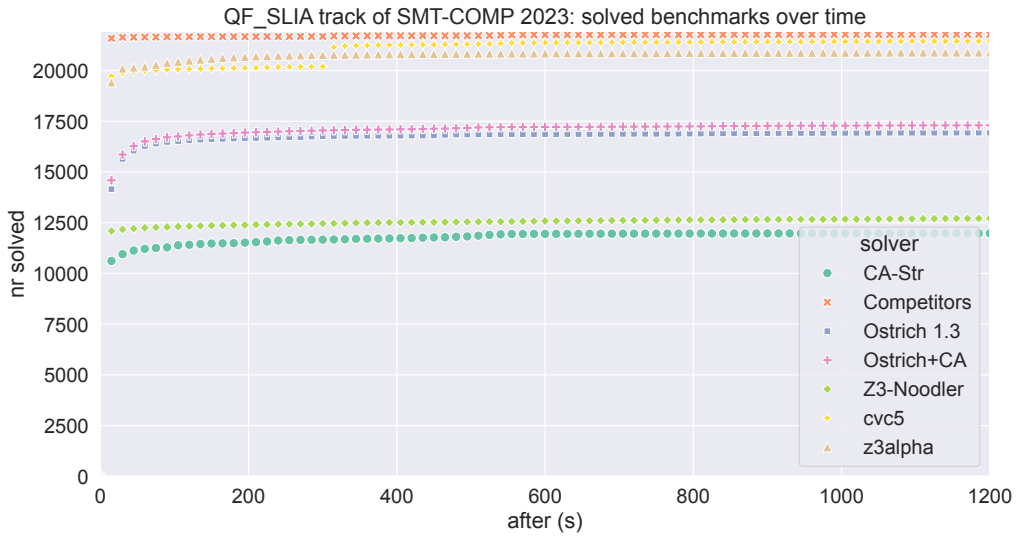


Fig. 8. The number of instances solved in the QF_SLIA track of SMT-COMP 2023 as the time budget increases.

At SMT-COMP, Ostrich competed using a portfolio of three different back-ends:

- **BW-Str:** a backward-propagation-based solver, not utilising Parikh automata [Chen et al. 2019].
- **ADT-Str:** a solver based on algebraic data-types.
- **CE-Str:** a re-implementation of the Ostrich+ algorithm [Chen et al. 2020], using Parikh automata and the encoding from [Verma et al. 2005] (referred to in this paper as baseline).

For our experiments, we modify the CE-Str solver to apply Catra (with PC* as its chosen backend) instead of the previous baseline method, resulting in a new back-end **CA-Str**. Combining the results from SMT-COMP for Ostrich 1.3 with our new results running CA-Str we construct a virtual portfolio Ostrich+CA that simulates running CA-Str as a fourth back-end to Ostrich. We similarly combine the results of all non-Ostrich solvers to obtain the virtual portfolio solver **Competition**.

We extend the results of SMT-COMP 2023 with our modified Ostrich on the single-query string solving with linear integer arithmetic constraints track (QF_SLIA). We picked QF_SLIA since Ostrich already performed well on the other two string solving tracks. In fact, every solver,

including Ostrich, handled every benchmark in QF_SNIA within the timeout. Additionally, Parikh intersection problems would mainly be generated by Ostrich when solving constraints involving integers, meaning that CA-Str would be of little or no help on the QF_S track.

We obtain the results by executing Ostrich 1.3 with CA-Str using the same benchmarking infrastructure (the StarExec cluster) and configuration that ran SMT-COMP 2023, combining our new results for CA-Str with the published results of SMT-COMP 2023. [2] Where available, we use the revised, out-of competition version of the results for solvers with bugs discovered during the competition, including Ostrich and Z3-Noodler. [3] Note that Z3-Noodler abstained on 70 instances, and thus has a lower total number of results. We used the parallel results for all solvers.

As can be seen from Table 4 and Fig. 8, integrating Catra as a back-end leads to gains both on satisfiable and unsatisfiable problems. On satisfiable problems, the combination Ostrich+CA is still outperformed by cvc5 and z3alpha. On unsatisfiable benchmarks, Ostrich+CA now narrowly beats the other solvers squeezing past cvc5, which is promising given that Ostrich 1.3 (without CA-Str), cvc5, and z3alpha all show very strong performance on this class of benchmarks.

These results are not unexpected. It is known that automata-based string solvers (Ostrich, Z3-Noodler) tend to perform better on unsatisfiable than on satisfiable benchmarks, compared to solvers that do not utilize automata and directly work on regular expressions (cvc5, z3alpha). The computation of automata representations of regular constraints can be expensive, and might be unnecessary for satisfiable formulas. In addition, the algorithm in Ostrich has stronger theoretical completeness guarantees than cvc5 and Z3-Str, and ensuring completeness often has an adverse effect on performance in practice [Barbosa et al. 2022; Chen et al. 2019; Zheng et al. 2013].

The results show that PC* can be used to enhance the performance of an automaton-based string solver. Moreover, the cactus plot in Table 4 illustrates that CA-Str is immediately useful, boosting the results for the Ostrich portfolio even at the first datapoint (though marginally). These results should be considered preliminary, however, as we believe that a deeper integration of Catra into string solvers can lead to significant performance gains. In particular, the integration layer is currently too shallow to allow Ostrich to learn clauses generated by Catra and additionally incurs overhead from serialising and deserialising the current Parikh automata problem into Catra's input format.

8.3 Threats to Validity

The most obvious threat to validity would be an unsound implementation. To address this we have validated all reported solutions made by PC* with nuXmv. A previous version contained a race with random restarts during product materialisation causing non-deterministic unsoundness in 0,7 of instances. Additionally, both we and the artefact evaluation committee independently discovered a soundness issue in CA-Str on one instance in QF_S where the `20230329-automatark-lu/instance08425.smt2` instance was incorrectly reported as Unsat. The underlying issue was that Catra generated a too broad blocking clause for a product without transitions. We implemented a fix and re-evaluated all benchmarks. Performance was not measurably affected.

Since addressing those bugs we have observed no further soundness issues in neither Catra nor in CA-Str. Benchmarking results for both runs are included in the artefact of this paper, and show virtually identical performance characteristics. We have additionally executed all of the benchmarks on machines with widely different performance characteristics and have observed the trends to be robust.

[2] StarExec Job ID 59410, "SMT-COMP 2023 single query final", available at https://www.starexec.org/starexec/secure/details/job.jsp?id=59410.
[3] StarExec Job ID 59668, "SMT-COMP 2023 single query final fixed" available at https://www.starexec.org/starexec/secure/details/job.jsp?id=59668.

The second threat to validity is our implementation of automata operations. As PC* by design offloads some of the product computation work onto Princess, the baseline could be unfairly disadvantaged by a slow automata product implementation. We believe this is not an issue since similar performance issues with the baseline approach have been reported for other string solvers, as well as for a previous implementation in Ostrich. Additionally, profiling shows that baseline spends most of its time in Princess, suggesting that the automaton implementation is not the bottleneck. Finally, the difference in performance between PC* and baseline was unaffected by significant optimisation of the automata library, additionally strengthening this thesis.

9 CONCLUSION

In this paper, we have introduced a calculus to compute commutating operations on intersections of regular languages that we call PC*. We have evaluated it on 37 497 Parikh automata intersection problems generated by the Ostrich+ string solver [Chen et al. 2020] solving the PyEx benchmark suite [Reynolds et al. 2017] using our Parikh automata solver Catra.

Within Catra, PC* shows astonishing performance in terms of solve-time compared to the baseline approach laid out in [Verma et al. 2005] when implemented on the same underlying automated theorem prover (Princess, [Rümmer 2008]). It is also competitive with the nuXmv model checker [Cavada et al. 2014], outperforming it on unsatisfiable instances and generally outperforming it for timeouts under 30 seconds with its advantage increasing drastically for even shorter timeouts. 30 seconds would generally be considered a long timeout for our intended use as supporting infrastructure to a string constraint solver.

Future investigations involve two tracks. The first one is integration into existing string solvers (wich Ostrich being a particularly promising candidate due to its shared use of Princess), and further adaptation to that use case. Closer inspection of the instances where we currently time out should be useful to further improve our heuristics.

The second track for future improvements is the extension into other problem domains, including other logics, model checking problems, as well as to more powerful automata such as transducers. In principle, we can already express stronger constraints than Parikh automata due to our use of an automated theorem prover which allows rich constraints on counter variables.

10 DATA-AVAILABILITY STATEMENT

A reproduction package featuring all relevant logs and all benchmarked instances is available [Stjerna and Rümmer 2024]. Catra and Ostrich are both available as living software under a 3-clause BSD license on GitHub at https://github.com/amandasystems/catra and https://github.com/uuverifiers/ostrich respectively. A version of Ostrich using CA-Str is available in the `cea-catra` branch. Code for the paper itself is available at https://github.com/amandasystems/oopsla-artefact.

ACKNOWLEDGMENTS

We thank the reviewers for insightful comments. This work was supported by the Swedish Research Council (VR) under grants 2018-04727 and 2021-06327, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and the Wallenberg project UPDATE.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and conquer: a framework for efficient analysis of string constraints. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 602–617. https://doi.org/10.1145/3062341.3062384

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT Solver for String Constraints. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 462–469. https://doi.org/10.1007/978-3-319-21690-4_29

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A String Solver with Theory-aware Heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, Vienna, Austria, 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. 2021. An SMT Solver for Regular Expressions and Linear Arithmetic over String Length. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 289–312. https://doi.org/10.1007/978-3-030-81688-9_14

Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2017. *String Analysis for Software Verification and Security*. Springer, Cham. https://doi.org/10.1007/978-3-319-68670-7

Michaël Cadilhac, Alain Finkel, and Pierre McKenzie. 2011. On the expressiveness of Parikh automata and related models. arXiv:1101.1547 [cs.FL]

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 334–342. https://doi.org/10.1007/978-3-319-08867-9_22

Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 325–342. https://doi.org/10.1007/978-3-030-59152-6_18

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* 3, POPL, Article 49 (jan 2019), 30 pages. https://doi.org/10.1145/3290362

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. 2011. Parikh's theorem: A simple and direct automaton construction. *Inform. Process. Lett.* 111, 12 (2011), 614–619. https://doi.org/10.1016/j.ipl.2011.03.019

Diego Figueira and Leonid Libkin. 2015. Path Logics for Querying Graphs: Combining Expressiveness and Efficiency. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, Kyoto, Japan, 329–340. https://doi.org/10.1109/LICS.2015.39

Melvin C. Fitting. 1996. *First-Order Logic and Automated Theorem Proving* (2nd ed.). Springer-Verlag, New York, NY, USA.

John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, Shaftesbury Road, Cambridge, UK.

Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. 2017. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.* 2, POPL, Article 4 (dec 2017), 32 pages. https://doi.org/10.1145/3158092

Petr Janků and Lenka Turoňová. 2020. Solving String Constraints with Approximate Parikh Image. In *Computer Aided Systems Theory – EUROCAST 2019*, Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia (Eds.). Springer International Publishing, Cham, 491–498. https://doi.org/10.1007/978-3-030-45093-9_59

Juhani Karhumäki. 1980. Generalized Parikh mappings and homomorphisms. *Information and Control* 47, 3 (1980), 155–165. https://doi.org/10.1016/S0019-9958(80)90493-3

Felix Klaedtke and Harald Rueß. 2002. *Parikh Automata and Monadic Second-Order Logics with Linear Cardinality Constraints*. Technical Report 177. Albert-Ludwigs-Universität Freiburg. https://tr.informatik.uni-freiburg.de/reports/report177/report00177.ps.gz

Dexter C. Kozen. 1997. *Automata and computability*. Springer, New York.

Giovanna J. Lavado, Giovanni Pighizzini, and Shinnosuke Seki. 2013. Converting nondeterministic automata and context-free grammars into Parikh equivalent one-way and two-way deterministic automata. *Information and Computation* 228-229 (2013), 1–15. https://doi.org/10.1016/j.ic.2013.06.003

Michael Luby, Alistair Sinclair, and David Zuckerman. 1993. Optimal speedup of Las Vegas algorithms. *Inform. Process. Lett.* 47, 4 (1993), 173–180. https://doi.org/10.1016/0020-0190(93)90029-9

Kimbal Marriott and Peter Stuckey. 1998. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/5625.001.0001

Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. Z3str4: A Multi-armed String Solver. In *Formal Methods*, Marieke Huisman, Corina Păsăreanu, and Naijun Zhan (Eds.). Springer International Publishing, Cham, 389–406. https://doi.org/10.1007/978-3-030-90870-6_21

Rohit J. Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (oct 1966), 570–581. https://doi.org/10.1145/321356.321364

Rodrigo Raya. 2023. *Decision Procedures for Power Structures*. Ph. D. Dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-10546

Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling Up DPLL(T) String Solvers Using Context-Dependent Simplification. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 453–474. https://doi.org/10.1007/978-3-319-63390-9_24

Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Iliano Cervesato, Helmut Veith, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–289. https://doi.org/10.1007/978-3-540-89439-1_20

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*. IEEE, Oakland, CA, USA, 513–528. https://doi.org/10.1109/SP.2010.38

Helmut Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. 2004. Counting in Trees for Free. In *Automata, Languages and Programming*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1136–1149. https://doi.org/10.1007/978-3-540-27836-8_94

Rani Siromoney and V. Rajkumar Dare. 1985. A generalization of the Parikh vector for finite and infinite words. In *Foundations of Software Technology and Theoretical Computer Science*, S. N. Maheshwari (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–302.

SMT-COMP. 2023. *SMT-COMP 2023 Results*. The SMT Steering Committee. https://smt-comp.github.io/2023/results.html

Daniel Stan and Anthony W. Lin. 2021. Regular Model Checking Approach to Knowledge Reasoning over Parameterized Systems. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems* (Virtual Event, United Kingdom) *(AAMAS '21)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1254–1262. https://doi.org/10.5555/3463952.3464097

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *PLDI 21*. Association for Computing Machinery, New York, NY, USA, 620–635. https://doi.org/10.1145/3453483.3454066

Amanda Stjerna and Philipp Rümmer. 2024. Reproduction Package for 'A Constraint Solving Approach to Parikh Images of Regular Languages'. https://doi.org/10.5281/zenodo.10796555

Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the Complexity of Equational Horn Clauses. In *Automated Deduction – CADE-20*, Robert Nieuwenhuis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–352. https://doi.org/10.1007/11532231_25

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 114–124. https://doi.org/10.1145/2491411.2491456

Received 21-OCT-2023; accepted 2024-02-24