An Efficient Calculus for Generalised Parikh Images of Regular Languages

some cool authors

Uppsala University, Sweden

Abstract

The image of the Parikh map is important in automata theory, offering a compact characterisation of an automaton. We contribute a novel understanding of how Parikh maps can be combined with arbitrary commutative monoid morphisms to efficiently represent a wide range of logics on automata and automata-like structures. Furthermore, we show how this formulation can be efficiently implemented as a calculus in a theorem prover supporting Presburger logic. In particular, our calculus allows us to solve the common problem in string solvers of computing the Parikh image of products of arbitrarily many automata. We demonstrate this by implementing a tool called Catra, which we use to solve a set of constraints produced by *the PyEx benchmarks* when solved by the Ostrich string constraint solver. We show that this implementation in addition to having a lower memory footprint than the standard eager approach executed on the same underlying solver also outperforms the nuxmv model checker on the same problem*, as well as the over-approximation recently described in [1].*

1 Introduction

Parikh maps and their image appears naturally as part of many operations in model checking and the solving of string constraints in automata-based string solvers such as Ostrich, notably in representing constraints on string lengths. While it is possible to compute the Parikh image of an automaton using the method described in [3], this method produces large existentially quantified clauses which are costly to eliminate and in practice make many real-world problems intractable. Furthermore, applications to string analysis in theorem provers require symbolic labels to handle Unicode alphabets *citation needed*. Conjunctions of string constraints finally, and most crucially, lead to the computation of Parikh images of automata products. Applying the approach in [3] would require the computation of the product before the Parikh image can be computed. In several instances we have observed, the materialisation of the product of the automata exhausts the memory of any reasonable machine due to the exponential blow-up of the product.

Addressing these concerns, our specialised calculus for Parikh images allows us to handle symbolic transition labels naturally, while also allowing us to interleave the computation of arbitrarily deep products of automata with the Parikh image of their product. This allows us to let both calculations inform each other, eliminating unnecessary work, and pruning the size of the partial products used in the computation. Moreover, the method can be used iteratively to, as it were, chew off bite-sized chunks of the product, thereby avoiding the problem of memory-outs.

1.1 Motivating Example

2 Overview

We define an non-deterministic automaton $\mathcal{A}$ with alphabet Σ as $\mathcal{A} = \langle Q, q_I, F, \delta \rangle$ where $\delta = Q \times \Sigma \times Q$, Q is its states, q_I is w.l.o.g. assumed to be the single initial state, and F is its set of accepting states. Note that we will later generalise the labels of transitions further.

2.1 The Parikh Map of a Regular language

Formally, the *Parikh map* over a context-free language $\Sigma^* = \{a_1, \ldots, a_k\}$ is defined as in [2]:

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

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 subset of the language $\mathcal{L} \subseteq \Sigma^*$ as:

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

Thus we would have $\psi(\{ab, abb\}) = \{[1, 1], [1, 2]\}$.

2.2 The Parikh image of a regular language expressed in Presburger arithmetic

Following [3], we define the Parikh Image of a regular language recognised by a DFA $\mathcal{A} = \langle Q, q_I, F, \delta \rangle$ as:

$$\psi(\mathcal{A}) := \bigwedge_{\alpha \in \Sigma} c_\alpha = \sum_{\delta \in \delta} t_\delta \text{ where } \alpha \in \delta$$
$$\bigwedge_{q \in Q} \left(1 \text{ if } q \in I + \sum_{\delta = q' \to q} t_\delta - \sum_{\delta = q \to q'} t_\delta \right) \begin{cases} \geq 0 & \text{if } q \in I \\ = 0 & \text{otherwise} \end{cases}$$
$$\bigwedge_{\delta = q \to q'} t_\delta > 0 \implies z_{\prime q} > 0$$
$$\bigwedge_{q, q' \in Q} z_{q'} > 0 \implies \begin{cases} \bigvee_{\delta = q \to q'} z_{q'} = z_q + 1 \wedge t_\delta > 0 \wedge z_q > 0 & \text{if } q \notin I \\ z_{q'} = 1 \wedge t_\delta > 0 & \text{if } q \in I \end{cases}$$

where all variables z_i, t_i are existentially quantified and the variables c_α make up the actual image. Feeding this definition into a Presburger solver and performing quantifier elimination on z_i, t_i would produce the Parikh image in its Presburger form. In this paper we use this example (sometimes appropriately modified) as the baseline approach.

2.2.1 An Example

AN EXAMPLE!

2.3 Generalised Parikh Images

To generalise the logic over Parikh images, we intrdouce a morphism $\mathrm{h} : \Sigma \to M$ where M is a commutative monoid where the following holds:

- $\mathrm{h}(\epsilon) = 0$, where 0 is the neutral element of M
- $\mathrm{h}(a \bullet b) = \mathrm{h}(a) + \mathrm{h}(b)$

2.3.1 Example 1: String length

A useful example of such a morphism can express string length. We let $M = (\mathbb{Z}; +; 0)$ and let $\mathrm{h}(x) = 1$. Then the length of a string $s = s_1 \bullet \ldots \bullet s_n$ is given by $\mathrm{h}(s) = \sum \mathrm{h}(s_i) = 1 + \ldots + 1$. Similarly, vectors and vector addition can be used to obtain the regular Parikh map.

We also expect more efficient computation

2.3.2 Example 2: Parikh Automata

We apply it to Parikh Automata and Magic Happens

2.4 Lazy Computation of Parikh Images for Regular Languages

We assume an NFA $\mathcal{A} = \langle Q, q_I, F, \delta \rangle$ with t transitions $\delta = \{\delta_1, \ldots, \delta_t\}$ where we describe each such transition δ_i from state q to state q' with label $l \in \Sigma$ as $\delta_i = q \xrightarrow{l} q'$. When one or more of the states and labels of a transition are uninteresting we will omit it.

For convenience, we introduce the following supporting concepts and notations:

Definition 2.1. A *path* $p = \langle q_I l_1 q_1, \ldots, q_n \rangle$ of an automaton $\mathcal{A}$ represents a valid run of the automaton as a path starting in its initial state and taking zero or more transitions to arrive at a final accepting state $q_n \in F$, while passing a number of labels $l_1, \ldots, l_n$. If $q_I \in F$, $\langle q_I \rangle$ is a valid path.

More formally, we require that:

1. $\forall_k \delta(q_k, l_{k+1}, q_{k+1})$
2. $q_n \in F$

Definition 2.2. The *word* of a path $\mathrm{w}(p) = l_1 \bullet \ldots \bullet l_n$ is the word read out on its labels.

Definition 2.3. The *transition count*, $\#(t, p)$ is the number of times a transition $t = q_1 \xrightarrow{l} q_2 \in \delta$ appears in a path p.

We then introduce the two predicates into our calculus with the following definitions:

Definition 2.4. $\mathrm{Im}_($ $\mathcal{A}, \mathrm{h}, \sigma, m)$ where:

- $m \in M$, for some commutative monoid M, as described in Section 2.3.
- $\sigma : \delta \to \mathbb{N}$

holds when $\exists p = \langle q_I l_1 q_1, \ldots, q_n \rangle \in \mathcal{A}$ such that:

- $\forall_{t_i \in p}\, \sigma(t_i) = \#(t_i, p)$

- $\mathrm{w}(p) \in \mathcal{L}(\mathcal{A})$
- $m = \mathrm{h}(\mathrm{w}(p))$

Definition 2.5. $\textsc{Conn}(\mathcal{A}, \sigma)$ holds when $\exists p = \langle q_I l_1 q_1, \ldots, q_n \rangle \in \mathcal{A}$ such that:

- $\forall_{t_i \in \delta}\, \sigma(t_i) = \#(t_i, p)$
- $\mathrm{w}(p) \in \mathcal{L}(\mathcal{A})$

Intuitively, it represents the condition that $\mathcal{A}$ is connected with respect to the selection function σ. It is redundant to Im by design.

Finally, we present the rules of our calculus for one automaton. Note that we operate on sets of symbols (terms, clauses). Additionally, we also use the convention of splitting the terms into linear inequalities (E) and any other clauses (Γ).

$$\textsc{Expand}\ \frac{\textsc{Conn}(\mathcal{A}, \sigma) \wedge \textsc{Flow}(\mathcal{A}, \sigma) \wedge m = \sum_{t \in \delta} \sigma(t) \cdot \mathrm{h}(t), E, \Gamma}{\mathrm{Im}_(\, \mathcal{A}, \mathrm{h}, \sigma, m), E, \Gamma}$$

$$\textsc{Split}\ \frac{\textsc{Conn}(\mathcal{A}, \sigma), E, \Gamma, \sigma(t) = 0 \mid \textsc{Conn}(\mathcal{A}, \sigma), E, \Gamma, \sigma(t) > 0}{\textsc{Conn}(\mathcal{A}, \sigma), E, \Gamma}\ \text{if } \sigma(t) = 0 \notin E, \sigma(t) > 0 \notin E$$

$$\textsc{Propagate}\ \frac{\textsc{Conn}(\mathcal{A}, \sigma), E, \Gamma, \sigma(t) = 0}{\textsc{Conn}(\mathcal{A}, \sigma), E, \Gamma}\ \text{if } \sigma(t) = 0 \notin E, \forall p \in \mathcal{A} \,:\, \exists c \in p \,:\, \sigma(c) = 0 \in E$$

$$\textsc{Subsume}\ \frac{E, \Gamma}{\textsc{Conn}(\mathcal{A}, \sigma), E, \Gamma}\ \text{if } \forall t = s_0 \to s_1 \in \delta, \sigma(t) = 0 \notin E \,:\, \exists p \in \mathcal{A}, s_0 \in p \,:\, \forall c \in \mathrm{T}(p) \,:\, \sigma(c) > 0 \in E$$

We use the pseudo-function $\textsc{Flow}(\mathcal{A}, \sigma)$ that generates a set of existentially quantified linear inequalities with the following definition:

$$\textsc{Flow}(\mathcal{A}, \sigma) = \bigwedge_{q \in Q} \textsc{In}(q, \sigma) - \textsc{Out}(q, \sigma) \geq 0 \text{ if } q \in F, = 0 \text{ otherwise}$$

$$\textsc{In}(q, \sigma) = (1 \text{ if } q = q_I, \text{ otherwise } 0) + \sum_{t = q_0 \to q \in \delta} \sigma(t)$$

$$\textsc{Out}(q, \sigma) = \sum_{t = q \to q' \in \delta} \sigma(t)$$

Where we transparently assign fresh variables to each state $q \in Q$ and existentially quantify them in the whole clause.

Additionally, we assume the existence of a rule Presburger-Close, corresponding to a sound and complete solver for Presburger formulae, modulo the monoid M.

The Propagate 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 $\mathcal{A}$.

Expand expands the predicate into its most basic rules; one set of linear equations synchronising the transitions mentioned by σ to the corresponding Monoid element m, and the linear

flow equations of the standard Parikh image formulation, as described by Flow. 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).

Finally, Split allows us to branch the proof tree by trying to exclude a contested 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 $\mathcal{A}$.

A decision procedure for our predicate in a tableau-based automated theorem prover would start by expanding the predicate using the Expand rule. If $\mathcal{A}$ contains no loops that could be disconnected from a minimum spanning tree (MST) with its root in q_I, the initial state, this would be sufficient to reach subsumption immediately.

2.5 An Example

3 Parikh Images from Products of Automata

- additional rules needed for products
- backjumping and conflict-driven learning

3.1 An Example

3.2 The calculus is correct

To be as clear as possible about what it means for our calculus to be correct, we use more exact definitions than the traditional soundness and completeness.

3.2.1 The calculus terminates

3.2.2 The calculus allows no values outside the Parikh image

3.2.3 The calculus allows all values in the Parikh image

4 Extending and generalising the calculus

5 Implementation and Experiments

- length constraints
- Parikh automata, automata with registers
- Model-checking examples

6 Conclusions

References

[1] Petr Janků and Lenka Turoňová. "Solving String Constraints with Approximate Parikh Image". In: *EUROCAST*. 2019, pp. 491–498.

[2] Dexter C. Kozen. *Automata and computability*. New York: Springer, 1997. ISBN: 0387949070.

[3] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. "On the Complexity of Equational Horn Clauses". In: *CADE*. 2005, pp. 337–352.