ACME: Automata with Counters, Monoids and Equivalence

Nathanaël Fijalkow, Denis Kuperberg

ACME: Automata with Counters, Monoids and Equivalence*

Nathanaël Fijalkow[1,2] and Denis Kuperberg[2]

[1] LIAFA, Paris 7
[2] University of Warsaw

Abstract. We present ACME, a tool implementing algebraic techniques to solve decision problems from automata theory. The core generic algorithm takes as input an automaton and computes its stabilization monoid, which is a generalization of its transition monoid.
Using the stabilization monoid, one can solve many problems: determine whether a B-automaton (which is a special kind of automata with counters) is limited, whether two B-automata are equivalent, and whether a probabilistic leaktight automaton has value 1.

The dedicated webpage where the tool ACME can be downloaded is

http://www.liafa.univ-paris-diderot.fr/~nath/acme.htm .

1 Stabilization Monoids for B- and Probabilistic Automata

The notion of stabilization monoids appears in two distinct contexts. It has first been developed in the theory of regular cost functions, introduced by Colcombet [Col09,Col13]. The underlying ideas have then been transferred to the setting of probabilistic automata [FGO12].

1.1 Stabilization Monoids in the Theory of Regular Cost Functions

At the heart of the theory of regular cost functions lies the equivalence between different formalisms: a logical formalism, cost MSO, two automata model, B- and S-automata, and an algebraic counterpart, stabilization monoids.

Here we briefly describe the model of B-automata, and their transformations to stabilization monoid. This automaton model generalizes the non-deterministic automata by adding a finite set of counters. Instead of accepting or rejecting a word as a non-deterministic automaton does, a B-automaton associates an

* The research leading to these results has received funding from the French ANR project 2010 BLAN 0202 02 FREC, the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement 259454 (GALE) and 239850 (SOSNA).

integer value to each input word. Formally, a B-automaton is a tuple $\mathcal{A} = \langle A, Q, \Gamma, I, F, \Delta \rangle$, where A is a finite alphabet, Q is a finite set of states, Γ is a finite set of counters, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times A \times \{\mathbf{ic}, \varepsilon, \mathbf{r}\}^\Gamma \times Q$ is the set of transitions.

A transition (p, a, τ, q) allows the automaton to go from state p to state q while reading letter a and performing action $\tau(\gamma)$ on counter γ. Action $\mathbf{ic}$ increments the current counter value by 1, ε leaves the counter unchanged, and $\mathbf{r}$ resets the counter to 0.

The value of a run is the maximal value assumed by any of the counters during the run. The semantics of a B-automaton $\mathcal{A}$ is defined on a word w by $[\![\mathcal{A}]\!](w) = \inf\{\mathrm{val}(\rho) \mid \rho \text{ is a run of } \mathcal{A} \text{ on } w\}$. In other words, the automaton uses the non determinism to minimize the value among all runs. In particular, if $\mathcal{A}$ has no run on w, then $[\![\mathcal{A}]\!](w) = \infty$.

The main decision problem in the theory of regular cost functions is the limitedness problem. We say that a B-automaton $\mathcal{A}$ is *limited* if there exists N such that for all words w, if $[\![\mathcal{A}]\!](w) < \infty$, then $[\![\mathcal{A}]\!](w) < N$.

One way to solve the limitedness problem is by computing the stabilization monoid. It is a monoid of matrices over the semiring of counter actions $\{\mathbf{ic}, \varepsilon, \mathbf{r}, \omega\}^\Gamma$. There are two operations on matrices: a binary composition called product, giving the monoid structure, and a unary operation called stabilization. The stabilization monoid of a B-automaton is the set of matrices containing the matrices corresponding to each letter, and closed under the two operations, product and stabilization. As shown in [Col09,Col13], the stabilization monoid of a B-automaton $\mathcal{A}$ contains an unlimited witness if and only if it is not limited, implying a conceptually simple solution to the limitedness problem: compute the stabilization monoid and check for the existence of unlimited witnesses.

1.2 Stabilization Monoids for Probabilistic Automata

The notion of stabilization monoids also appeared for probabilistic automata, for the Markov Monoid Algorithm. This algorithm was introduced in [FGO12] to partially solves the value 1 problem: given a probabilistic automaton $\mathcal{A}$, does there exist $(u_n)_{n\in\mathbb{N}}$ a sequence of words such that $\lim_n \mathbb{P}_\mathcal{A}(u_n) = 1$?

Although the value 1 problem is undecidable, it has been shown that the Markov Monoid Algorithm correctly determines whether a probabilistic automaton has value 1 under the *leaktight* restriction. It has been recently shown that all classes of probabilistic automata for which the value 1 problem has been shown decidable are included in the class of leaktight automata [FGKO14], hence the Markov Monoid Algorithm is the *most correct* algorithm known to (partially) solve the value 1 problem.

As for the case of B-automata, the stabilization monoid of a probabilistic automaton is the set of matrices containing the matrices corresponding to each letter, and closed under the two operations, product and stabilization.

Note that the main point is that both the products and the stabilizations depend on which type of automata is considered, B-automata or probabilistic automata.

2 Computing the Stabilization Monoid of an Automaton

We report here on some implementation issues regarding the following algorithmic task:

> We are given as input:
>
> – A finite set of matrices S,
> – A binary associative operation on matrices, the product, denoted $\cdot$,
> – A unary operation on matrices, the stabilization, denoted $\sharp$.
>
> The aim is to compute the closure of S under product and stabilization, called the stabilization monoid generated by S.

Our choice of OCaml allowed for a generic implementation of the algorithm.

Note that if we ignore the stabilization operation, this reduces to computing the monoid generated by a finite set of matrices, *i.e.* the transition monoid of a non-deterministic automaton. It is well-known that this monoid can be exponential in the size of the automaton, so the crucial aspect here is space optimization.

In our application, the initial set of matrices is given by matrices M_a for $a \in A$, where A is the finite alphabet of the automaton. Hence we naturally associate to every element of the stabilization monoid a $\sharp$-expression: $(M_a \cdot M_b^\sharp \cdot M_a)^\sharp$ is associated to $(ab^\sharp a)^\sharp$. Many $\sharp$-expressions actually correspond to the same matrix: for instance, it may be that $(M_a \cdot M_a \cdot M_b)^\sharp = M_a^\sharp$; in such case, we would like to associate this matrix to the $\sharp$-expression $a^\sharp$, which is "simpler" than $(aab)^\sharp$.

There are two data structures: a table and a queue. The table is of fixed size (a large prime number), and is used to keep track of all the matrices found so far, through their hash value. The queue stores the elements to be treated. The pseudo-code of the algorithm is presented in Algorithm 1.

Data: $S = \{M_a \mid a \in A\}$
Result: The stabilization monoid generated by S
Initialization: an empty table T and an empty queue Q;
for $a \in A$ **do**
    push (a, M_a) onto Q for every $a \in A$;
    add M_a to T;
end
while Q *is not empty* **do**
    let (s, M) the first element in Q;
    search M in T (through its hash value);
    if M *is not in* T *(new)* **then**
        add M to T;
        for $N \in T$ **do**
            push $M \cdot N$ onto Q;
            push $N \cdot M$ onto Q;
        end
        push $M^\sharp$ onto Q;
    end
end

Algorithm 1: Computing the stabilization monoid

3 Minimizing the Stabilization Monoid

To test whether two B-automata are equivalent, we follow [CKL10]: for both automata we construct its stabilization monoid, then we minimize them and check whether the minimal stabilization monoids are isomorphic.

We do not explain in details how to check whether two stabilization monoids are isomorphic. This is in general a very hard problem, theoretically not well understood; for instance there is no polynomial-time algorithm to check whether two groups are isomorphic. Our setting here makes the task much easier, as we look for an isomorphism extending two given morphisms (associating to each letter an element), leading to a simple linear-time algorithm.

Let M be a stabilization monoid, whose elements are denoted $m_1, \cdots, m_n$, and an ideal $I \subseteq M$. The algorithm constructs an increasing sequence of partitions, starting from the partition that separates I from $M \setminus I$.

Consider a partition P of the elements. For an element $m \in M$, we denote by $[m]_P$ its equivalence class with respect to P. The type of m with respect to

P is the following vector of equivalence classes:

$$([m]_P,\ [m^{\omega\sharp}]_P, [m \cdot m_1]_P, \cdots, [m \cdot m_n]_P, [m_1 \cdot m]_P, \cdots, [m_n \cdot m]_P)\ .$$

Note that the second component uses the $\omega\sharp$ operator, defined using the $\sharp$ operator. Relying on the $\sharp$ operator would not be correct, as it is partial (only defined for idempotent elements).

Using the types, we construct a larger partition P', such that two elements are equivalent for P' if they have the same type with respect to P.

There are three data structures: two *union-find* tables to handle partitions of the elements and a table of types. We present in Algorithm 2 a pseudo-code of the minimization algorithm.

Data: A stabilization monoid $(M, \cdot, \sharp)$, an ideal $I \subseteq M$
Result: The minimal stabilization monoid with respect to (M, I)
Initialization: a partition P separating I and $M \setminus I$ and an empty table T of types;
while *unstable* **do**
    Compute the types (with respect to P) in T;
    Create a new partition P' such that two elements are equivalent for P' if they have the same types;
    $P \longleftarrow P'$;
end

Algorithm 2: Minimizing the stabilization monoid

References

[CKL10] Thomas Colcombet, Denis Kuperberg, and Sylvain Lombardy. Regular temporal cost functions. In *ICALP (2)*, pages 563–574, 2010.

[Col09] Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In *ICALP (2)*, pages 139–150, 2009.

[Col13] Thomas Colcombet. Regular cost-functions, part I: Logic and algebra over words. *Logical Methods in Computer Science*, 9(3), 2013.

[FGKO14] Nathanaël Fijalkow, Hugo Gimbert, Edon Kelmendi, and Youssouf Oualhadj. Deciding the value 1 problem for probabilistic leaktight automata. 2014.

[FGO12] Nathanaël Fijalkow, Hugo Gimbert, and Youssouf Oualhadj. Deciding the value 1 problem for probabilistic leaktight automata. In *LICS*, pages 295–304, 2012.