Language Equivalence for Probabilistic Automata*

Stefan Kiefer[1], Andrzej S. Murawski[2], Joël Ouaknine[1],
Björn Wachter[1], and James Worrell[1]

[1] Department of Computer Science, University of Oxford, UK
[2] Department of Computer Science, University of Leicester, UK

Abstract. In this paper, we propose a new randomised algorithm for deciding language equivalence for probabilistic automata. This algorithm is based on polynomial identity testing and thus returns an answer with an error probability that can be made arbitrarily small. We implemented our algorithm, as well as deterministic algorithms of Tzeng and Doyen et al., optimised for running time whilst adequately handling issues of numerical stability. We conducted extensive benchmarking experiments, including the verification of randomised anonymity protocols, the outcome of which establishes that the randomised algorithm significantly outperforms the deterministic ones in a majority of our test cases. Finally, we also provide fine-grained analytical bounds on the complexity of these algorithms, accounting for the differences in performance.

1 Introduction

Probabilistic automata were introduced by Michael Rabin [16] as a variation on non-deterministic finite automata. In a probabilistic automaton the transition behaviour is determined by a stochastic matrix for each input letter. Probabilistic automata are one of a variety of probabilistic models, including Markov chains and Markov decision processes, that are of interest to the verification community. However a distinguishing feature of work on probabilistic automata is the emphasis on language theory and related notions of equivalence as opposed to temporal-logic model checking [1, 4, 8, 12–14].

Probabilistic automata accept weighted languages: with each word one associates the probability with which it is accepted. In this context two automata $\mathcal{A}$ and $\mathcal{B}$ are said to be equivalent if each word is accepted with the same probability by both $\mathcal{A}$ and $\mathcal{B}$. It was shown by Tzeng [21] that equivalence for probabilistic automata is decidable in polynomial time, although this fact can also be extracted from results of Schützenberger [17] on minimising weighted automata. By contrast the natural analog of language inclusion—that $\mathcal{B}$ accepts each word with probability at least as great as $\mathcal{A}$—is undecidable [6] even for automata of fixed dimension [2].

* Research supported by EPSRC grant EP/G069158. The first author is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

The motivation of Tzeng to study equivalence of probabilistic automata came from learning theory. Here our motivation is software verification as we continue to develop APEX [12], an automated equivalence checker for probabilistic imperative programs. APEX is able to verify contextual equivalence of probabilistic programs[3], which in turn can be used to express a broad range of interesting specifications.

APEX works by translating probabilistic programs into automata-theoretic representations of their game semantics [8]. The translation applies to open programs (i.e., programs with undefined components) and the resultant automata are highly abstracted forms of the program operational semantics. Intuitively, only externally observable computational steps are visible in the corresponding automata and internal actions such as local-variable manipulation remain hidden. Crucially, the translation relates (probabilistic) contextual equivalence with (probabilistic) language equivalence [14]. Accordingly, after two programs have been translated into corresponding probabilistic automata, we can apply any language equivalence testing routine to decide their contextual equivalence. In some of our case studies, we shall use automata generated by APEX for benchmarking the various algorithms discussed in the paper.

In conjunction with the ability of the APEX tool to control which parts of a given program are externally visible, the technique of verification by equivalence checking has proven surprisingly flexible. The technique is particularly suited to properties such as obliviousness and anonymity, which are difficult to formalise in temporal logic.

In this paper we develop a new randomised algorithm for checking equivalence of probabilistic automata (more generally of real-valued weighted automata). The complexity of the algorithm is $O(nm)$, where n is the number of states of the two automata being compared and m is the total number of labelled transitions. Tzeng [21] states the complexity of his algorithm as $O(|\Sigma| \cdot n^4)$; the same complexity bound is likewise claimed by [5, 10] for variants of Tzeng's procedure. Here we observe that the procedure can be implemented with complexity $O(|\Sigma| \cdot n^3)$, which is still slower than the randomised algorithm.

This theoretical complexity improvement of the randomised algorithm over the deterministic algorithm is borne out in practice. We have tested the algorithms on automata generated by APEX based on an anonymity protocol and Herman's self-stabilisation protocol. We have also generated test cases for the equivalence algorithms by randomly performing certain structural transformations on automata. Our results suggest that the randomised algorithm is around ten times faster than the deterministic algorithm. Another feature of our experiments is that the performance of both the randomised and deterministic algorithms is affected by whether they are implemented using forward reachability from the initial states of the automata or using backward reachability from the accepting states of the automata.

[3] Two programs are contextually equivalent if and only if they can be used interchangeably in any context.

2 Algorithms for Equivalence Checking

Rabin's probabilistic automata [16] are *reactive* according to the taxonomy of Segala [19]: the transition function has type $Q \times \Sigma \to \mathrm{Dist}(Q)$, where Q is the set of states, Σ the alphabet and $\mathrm{Dist}(Q)$ the set of probability distributions on Q. Another type of probabilistic automata are the so-called *generative* automata in which the transition function has type $Q \to \mathrm{Dist}(\Sigma \times Q)$. The automata produced by APEX combine both reactive and generative states. In this paper we work with a notion of $\mathbb{R}$-*weighted automaton* which encompasses both reactive and generative transition modes as well as negative transition weights, which can be useful in finding minimal representations of the languages of probabilistic automata [17]. In the context of equivalence checking the generalisation from probabilistic automata to $\mathbb{R}$-weighted automata is completely innocuous.

Definition 1. *An* $\mathbb{R}$-weighted automaton $\mathcal{A} = (n, \Sigma, M, \alpha, \eta)$ *consists of a positive integer* $n \in \mathbb{N}$ *representing the number of states, a finite alphabet* Σ*, a map* $M : \Sigma \to \mathbb{R}^{n \times n}$ *assigning a transition matrix to each alphabet symbol, an initial (row) vector* $\alpha \in \mathbb{R}^n$*, and a final (column) vector* $\eta \in \mathbb{R}^n$*. The automaton* $\mathcal{A}$ *assigns each word* $w = \sigma_1 \cdots \sigma_k \in \Sigma^*$ *a* weight $\mathcal{A}(w) \in \mathbb{R}$*, where* $\mathcal{A}(w) := \alpha M(\sigma_1) \cdots M(\sigma_k)\eta$*. Two automata* $\mathcal{A}, \mathcal{B}$ *over the same alphabet* Σ *are said to be* equivalent *if* $\mathcal{A}(w) = \mathcal{B}(w)$ *for all* $w \in \Sigma^*$*.*

From now on, we fix automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$, and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$, and define $n := n^{(\mathcal{A})} + n^{(\mathcal{B})}$. We want to check $\mathcal{A}$ and $\mathcal{B}$ for equivalence. It is known that if $\mathcal{A}$ and $\mathcal{B}$ are not equivalent then there is a short word witnessing their inequivalence [15]:

Proposition 2. *If there exists a word* $w \in \Sigma^*$ *such that* $\mathcal{A}(w) \neq \mathcal{B}(w)$ *then there exists a word* w *of length at most* n *such that* $\mathcal{A}(w) \neq \mathcal{B}(w)$*.*

It follows immediately from Proposition 2 that the equivalence problem for weighted automata is in co-NP. In fact this problem can be decided in polynomial time. For instance, Tzeng's algorithm [21] explores the tree of possible input words and computes for each word w in the tree and for both automata the distribution of states after having read w. The tree is suitably pruned by eliminating nodes whose associated state distributions are linearly dependent on previously discovered state distributions. Tzeng's procedure has a runtime of $O(|\Sigma| \cdot n^4)$. Another algorithm was recently given in [10]. Although presented in different terms, their algorithm is fundamentally a backward version of Tzeng's algorithm and has the same complexity. (An advantage of a "backward" algorithm as in [10] is that repeated equivalence checks for different initial state distributions are cheaper, as results from the first equivalence check can be reused.) Variants of Tzeng's algorithm were also presented in [5] for checking linear-time equivalences on probabilistic labelled transition systems, including probabilistic trace equivalence and probabilistic failures equivalence. These variants all have run time $O(|\Sigma| \cdot n^4)$ as for Tzeng's algorithm. It has been pointed out in [7] that more efficient algorithms can actually be obtained using techniques from the 60s: One can – in linear time – combine $\mathcal{A}$ and $\mathcal{B}$ to form a weighted automaton that assigns each word w the weight $\mathcal{A}(w) - \mathcal{B}(w)$. An algorithm of Schützenberger [17] allows to compute a minimal equivalent automaton, which is the empty automaton if and only if $\mathcal{A}$ and $\mathcal{B}$ are equivalent. As sketched in [7], Schützenberger's algorithm runs in $O(|\Sigma| \cdot n^3)$, yielding an overall runtime of $O(|\Sigma| \cdot n^3)$ for checking equivalence.

2.1 Deterministic Algorithms

Since the focus of this paper is language equivalence and not minimisation, we have not implemented Schützenberger's algorithm. Instead we have managed to speed up the algorithms of [21] and [10] from $O(|\Sigma| \cdot n^4)$ to $O(|\Sigma| \cdot n^3)$ by efficient bookkeeping of the involved vector spaces. Whereas [7] suggests to maintain an LU decomposition to represent the involved vector spaces, we prefer a QR decomposition for stability reasons. We have implemented two versions of a deterministic algorithm for equivalence checking: a forward version (related to [21]) and a backward version (related to [10]). In the following we describe the backward algorithm; the forward version is obtained essentially by transposing the transition matrices and exchanging initial and final states.

Figure 1 shows the backward algorithm. The algorithm computes a basis Q of the set of state distributions

$$\left\{ \begin{pmatrix} M^{(\mathcal{A})}(\sigma_1) \ldots M^{(\mathcal{A})}(\sigma_k)\eta^{(\mathcal{A})} \\ M^{(\mathcal{B})}(\sigma_1) \ldots M^{(\mathcal{B})}(\sigma_k)\eta^{(\mathcal{B})} \end{pmatrix} \mid \sigma_1 \ldots \sigma_k \in \Sigma^* \right\}$$

represented by n-dimensional column vectors. It is clear that $\mathcal{A}$ and $\mathcal{B}$ are equivalent if and only if $\alpha^{(\mathcal{A})} u^{(\mathcal{A})} = \alpha^{(\mathcal{B})} u^{(\mathcal{B})}$ for all vectors $\begin{pmatrix} u^{(\mathcal{A})} \\ u^{(\mathcal{B})} \end{pmatrix}$ in Q.

Computing a basis Q of the set of state distributions requires a membership test to decide if a new vector is already included in the current vector space. While a naive membership test can be carried out in $O(n^3)$ time, maintaining the basis in a canonical form (here as an orthogonal set) admits a test in $O(n^2)$.

Given a set $Q = \{q_1, \ldots, q_k\}$ of $k \leq n$ orthogonal vectors and another vector $u \in \mathbb{R}^n$, we denote by $u \downarrow Q$ the orthogonal projection of u along Q onto the space orthogonal to Q. Vector $u \in \mathbb{R}^n$ is included in the vector space given by Q, if u is equal to its projection into Q, i.e., the vector $u - u \downarrow Q$. Vector $u \downarrow Q$ can be computed by Gram-Schmidt iteration:

```
for $i$ from 1 to $k$
    $u := u - \frac{q_i^T u}{q_i^T q_i} q_i$
return $u$
```

Thereby q_i^T denotes the transpose of vector q_i. Note that this iteration takes time $O(nk)$.

The improvement from the $O(|\Sigma| \cdot n^4)$ bound for equivalence checking reported in $[5, 10, 21]$ to $O(|\Sigma| \cdot n^3)$ arises from the fact that we keep the basis in canonical form which reduces the $O(n^3)$ complexity $[5, 10, 21]$ for the membership test to $O(n^2)$. Since each iteration of the equivalence checking algorithm adds a new vector to Q or decreases the size of *worklist* by one, $|\Sigma| \cdot n$ iterations suffice for termination. Thus the overall complexity is $O(|\Sigma| \cdot n^3)$.

For efficiency, we consider an implementation based on finite-precision floating point numbers. Therefore we need to take care of numerical issues. To this end, we use the modified Gram-Schmidt iteration rather than the classical Gram-Schmidt iteration, as the former is numerically more stable, see [20].

To be more robust in presence of rounding errors, vectors are compared up to a relative error $\epsilon > 0$. Two vectors $v, v' \in \mathbb{R}^n$ are equal up to relative error $\epsilon \in \mathbb{R}$, denoted by $v \approx_\epsilon v'$, iff $\frac{\|v-v'\|}{\|v\|} < \epsilon$. The relative criterion is used to compare the weight of a word in automaton $\mathcal{A}$ and $\mathcal{B}$, and to check if a vector $u \in \mathbb{R}^n$ is included in a vector space given by a set of base vectors Q, i.e., $u \approx_\epsilon u - u \downarrow Q$.

Algorithm EQUIV$_{\leftarrow,\mathrm{det}}$

Input: Automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$

```
if $\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$
    return "$\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} = \mathcal{A}(\varepsilon) \neq \mathcal{B}(\varepsilon) = \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$"
$worklist$:= $\{(\eta^{(\mathcal{A})}, \eta^{(\mathcal{B})}, \varepsilon)\}$
$\eta := \begin{pmatrix} \eta^{(\mathcal{A})} \\ \eta^{(\mathcal{B})} \end{pmatrix}$
if $\|\eta\| = 0$
    return "$\mathcal{A}$ and $\mathcal{B}$ are both empty and hence equivalent"
$Q := \{\eta\}$
while $worklist \neq \emptyset$
    choose and remove $(v^{(\mathcal{A})}, v^{(\mathcal{B})}, w)$ from $worklist$
    forall $\sigma \in \Sigma$
        $u^{(\mathcal{A})} := M^{(\mathcal{A})}(\sigma)v^{(\mathcal{A})}$; $u^{(\mathcal{B})} := M^{(\mathcal{B})}(\sigma)v^{(\mathcal{B})}$; $w' := \sigma w$
        if not $\alpha^{(\mathcal{A})}u^{(\mathcal{A})} \approx_\epsilon \alpha^{(\mathcal{B})}u^{(\mathcal{B})}$
            return "$\alpha^{(\mathcal{A})}u^{(\mathcal{A})} = \mathcal{A}(w') \neq \mathcal{B}(w') = \alpha^{(\mathcal{B})}u^{(\mathcal{B})}$"
        $u := \begin{pmatrix} u^{(\mathcal{A})} \\ u^{(\mathcal{B})} \end{pmatrix}$
        $q := u \downarrow Q$
        if not $u - q \approx_\epsilon u$
            add $(u^{(\mathcal{A})}, u^{(\mathcal{B})}, w')$ to $worklist$
            $Q := Q \cup \{q\}$
return "$\mathcal{A}$ and $\mathcal{B}$ are equivalent"
```

Fig. 1. Deterministic algorithm (backward version ←).

2.2 Randomised Algorithms

In this section we present a new randomised algorithm for deciding equivalence of $\mathbb{R}$-weighted automata. Recall from Proposition 2 that two $\mathbb{R}$-weighted automata with combined number of states n are equivalent if and only if they have the same *n-bounded language*, that is, they assign the same weight to all words of length at most n. Inspired by the work of Blum, Carter and Wegman on free Boolean graphs [3] we represent the *n-bounded language* of an automaton by a polynomial in which each monomial represents a word and the coefficient of the monomial represents the weight of the word. We thereby reduce the equivalence problem to polynomial identity testing, for which there is a number of efficient randomised procedures.

Let $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$ be two $\mathbb{R}$-weighted automata over the same alphabet Σ. Let $n := n^{(\mathcal{A})} + n^{(\mathcal{B})}$ be the total number of states of the two automata. We introduce a family of variables $\mathbf{x} = \{x_{\sigma,i} : \sigma \in \Sigma, 1 \leq i \leq n\}$ and associate the monomial $x_{\sigma_1,1} x_{\sigma_2,2} \dots x_{\sigma_k,k}$ with a word $w = \sigma_1 \sigma_2 \dots \sigma_k$ of length $k \leq n$. Then we define the polynomial $P^{(\mathcal{A})}(\mathbf{x})$ by

$$P^{(\mathcal{A})}(\mathbf{x}) := \sum_{k=0}^{n} \sum_{w \in \Sigma^k} \mathcal{A}(w) \cdot x_{\sigma_1,1} x_{\sigma_2,2} \dots x_{\sigma_k,k} \,. \tag{1}$$

The polynomial $P^{(\mathcal{B})}(\mathbf{x})$ is defined likewise, and it is immediate from Proposition 2 that $P^{(\mathcal{A})} \equiv P^{(\mathcal{B})}$ if and only if $\mathcal{A}$ and $\mathcal{B}$ have the same weighted language.

To test equivalence of $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ we select a value for each variable $x_{i,\sigma}$ independently and uniformly at random from a set of rationals of size Kn, for some constant K. Clearly if $P^{(\mathcal{A})} \equiv P^{(\mathcal{B})}$ then both polynomials will yield the same value. On the other hand, if $P^{(\mathcal{A})} \not\equiv P^{(\mathcal{B})}$ then the polynomials will yield different values with probability at least $(K-1)/K$ by the following result of De Millo and Lipton [9], Schwartz [18] and Zippel [22] and the fact that $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ both have degree n.

Theorem 3. *Let $\mathbb{F}$ be a field and $Q(x_1, \dots, x_n) \in \mathbb{F}[x_1, \dots, x_n]$ a multivariate polynomial of total degree d. Fix a finite set $\mathbb{S} \subseteq \mathbb{F}$, and let $r_1, \dots, r_n$ be chosen independently and uniformly at random from $\mathbb{S}$. Then*

$$\mathbf{Pr}[Q(r_1, \dots, r_n) = 0 \mid Q(x_1, \dots, x_n) \not\equiv 0] \leq \frac{d}{|\mathbb{S}|} \,.$$

While the number of monomials in $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ is proportional to $|\Sigma|^n$, i.e., exponential in n, writing

$$P^{(\mathcal{A})}(\mathbf{x}) = \alpha^{(\mathcal{A})} \left(\sum_{i=0}^{n} \prod_{j=1}^{i} \sum_{\sigma \in \Sigma} x_{\sigma,j} \cdot M^{(\mathcal{A})}(\sigma) \right) \eta^{(\mathcal{A})} \tag{2}$$

it is clear that $P^{(\mathcal{A})}$ and $P^{(\mathcal{B})}$ can be evaluated on a particular set of numerical arguments in time polynomial in n. The formula (2) can be evaluated in a forward direction, starting with the initial state vector $\alpha^{(\mathcal{A})}$ and post-multiplying by the transition matrices, or in a backward direction, starting with the final state vector $\eta^{(\mathcal{A})}$ and pre-multiplying by the transition matrices. In either case we get a polynomial-time Monte-Carlo algorithm for testing equivalence of $\mathbb{R}$-automata. The backward variant is shown in Figure 2.

Algorithm EQUIV$_{\leftarrow,\text{rand}}$

Input: Automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$

```
if $\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$
    return "$\alpha^{(\mathcal{A})}\eta^{(\mathcal{A})} = \mathcal{A}(\varepsilon) \neq \mathcal{B}(\varepsilon) = \alpha^{(\mathcal{B})}\eta^{(\mathcal{B})}$"
$v^{(\mathcal{A})} := \eta^{(\mathcal{A})}$; $v^{(\mathcal{B})} := \eta^{(\mathcal{B})}$
for $i$ from 1 to $n$ do
    choose a random vector $r \in \{1, 2, \ldots, Kn\}^{\Sigma}$
    $v^{(\mathcal{A})} := \sum_{\sigma\in\Sigma} r(\sigma)M^{(\mathcal{A})}(\sigma)v^{(\mathcal{A})}$
    $v^{(\mathcal{B})} := \sum_{\sigma\in\Sigma} r(\sigma)M^{(\mathcal{B})}(\sigma)v^{(\mathcal{B})}$
    if $\alpha^{(\mathcal{A})}v^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})}v^{(\mathcal{B})}$
        return "$\exists w$ with $|w| = i$ such that $\mathcal{A}(w) \neq \mathcal{B}(w)$"
return "$\mathcal{A}$ and $\mathcal{B}$ are equivalent with probability at least $(K-1)/K$"
```

Fig. 2. Randomised algorithm, backward version

Algorithm EQUIV$_{\leftarrow,\text{rand}}$

Input: Automata $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, M^{(\mathcal{A})}, \alpha^{(\mathcal{A})}, \eta^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \alpha^{(\mathcal{B})}, \eta^{(\mathcal{B})})$

```
$v_0^{(\mathcal{A})} := \eta^{(\mathcal{A})}$; $v_0^{(\mathcal{B})} := \eta^{(\mathcal{B})}$;
for $i$ from 1 to $n$ do
    choose a random vector $r \in \{1, 2, \ldots, Kn\}^{\Sigma}$
    $v_i^{(\mathcal{A})} := \left(\sum_{\sigma\in\Sigma} r(\sigma)M^{(\mathcal{A})}(\sigma)\right) v_{i-1}^{(\mathcal{A})}$;
    $v_i^{(\mathcal{B})} := \left(\sum_{\sigma\in\Sigma} r(\sigma)M^{(\mathcal{B})}(\sigma)\right) v_{i-1}^{(\mathcal{B})}$;
    if $\alpha^{(\mathcal{A})}v_i^{(\mathcal{A})} \neq \alpha^{(\mathcal{B})}v_i^{(\mathcal{B})}$
        $w := \varepsilon$;
        $u^{(\mathcal{A})} := \alpha^{(\mathcal{A})}$; $u^{(\mathcal{B})} := \alpha^{(\mathcal{B})}$;
        for $j$ from $i$ downto 1 do
            choose $\sigma \in \Sigma$ with $u^{(\mathcal{A})}M^{(\mathcal{A})}(\sigma)v_{j-1}^{(\mathcal{A})} \neq u^{(\mathcal{B})}M^{(\mathcal{B})}(\sigma)v_{j-1}^{(\mathcal{B})}$;
            $w := w\sigma$;
            $u^{(\mathcal{A})} := u^{(\mathcal{A})}M^{(\mathcal{A})}(\sigma)$;
            $u^{(\mathcal{B})} := u^{(\mathcal{B})}M^{(\mathcal{B})}(\sigma)$;
        return "$\alpha^{(\mathcal{A})}u^{(\mathcal{A})} = \mathcal{A}(w') \neq \mathcal{B}(w') = \alpha^{(\mathcal{B})}u^{(\mathcal{B})}$"
return "$\mathcal{A}$ and $\mathcal{B}$ are equivalent with probability at least $(K-1)/K$"
```

Fig. 3. Randomised algorithm with counterexamples, backward version

2.3 Runtime

The randomised algorithm is simpler to implement than the deterministic algorithm since there is no need to solve large systems of linear equations. While the deterministic algorithm, carefully implemented, runs in time $O(|\Sigma|\cdot n^3)$, the randomised algorithm runs in time $O(n \cdot |M|)$, where $|M|$ is the number of nonzero entries in all $M(\sigma)$, provided that sparse-matrix representations are used. In almost all of our case studies, below, the randomised algorithm outperforms the deterministic algorithm.

2.4 Extracting Counter-examples

We can obtain counter-examples from the randomised algorithm by exploiting a self-reducible structure of the equivalence problem. We generate counter-examples incrementally, starting with the empty string and using the randomised algorithm as an oracle to know at each stage what to choose as the next letter in our counter-example. Here the important thing is to avoid repeatedly running the randomised algorithm, which would negate the complexity advantages of this procedure over the deterministic algorithm. In fact this can all be made to work with some post-processing following a single run of the randomised procedure as we now explain.

Once again assume that $\mathcal{A}$ and $\mathcal{B}$ are weighted automata over the same alphabet Σ and with combined number of states n. To evaluate the polynomial $P^{(\mathcal{A})}$ we substitute a set of randomly chosen rational values $\mathbf{r} = \{r_{\sigma,i} : \sigma \in \Sigma, 1 \le i \le n\}$ into the expression (2). Here we will generalize this to a notion of *partial evaluation* $P^{(\mathcal{A})}_w(\mathbf{r})$ of polynomial $P^{(\mathcal{A})}$ with respect to values $\mathbf{r}$ and a word $w \in \Sigma^m$, $m \le n$:

$$P^{(\mathcal{A})}_w(\mathbf{r}) = \alpha^{(\mathcal{A})} M^{(\mathcal{A})}(w) \left(\sum_{i=m}^{n} \prod_{j=m+1}^{n} \sum_{\sigma\in\Sigma} r_{\sigma,j} \cdot M^{(\mathcal{A})}(\sigma) \right) \eta^{(\mathcal{A})} \qquad (3)$$

Notice that $P^{(\mathcal{A})}_\varepsilon(\mathbf{r}) = P^{(\mathcal{A})}(\mathbf{r})$ where ε is the empty word and, at the other extreme, $P^{(\mathcal{A})}_w(\mathbf{r}) = \mathcal{A}(w)$ for any word w of length n. We define $P^{(\mathcal{B})}_w$ similarly.

Proposition 4. *If $P^{(\mathcal{A})}_w(\mathbf{r}) \neq P^{(\mathcal{B})}_w(\mathbf{r})$ then either $\mathcal{A}(w) \neq \mathcal{B}(w)$ or $m < n$ and there exists $\sigma \in \Sigma$ with $P^{(\mathcal{A})}_{w\sigma}(\mathbf{r}) \neq P^{(\mathcal{B})}_{w\sigma}(\mathbf{r})$*

Proof. Considering the equation (3) and the corresponding statement for $\mathcal{B}$ the contrapositive of the proposition is obvious: if $\mathcal{A}(w) = \mathcal{B}(w)$ and $P^{(\mathcal{A})}_{w\sigma}(\mathbf{r}) \neq P^{(\mathcal{B})}_{w\sigma}(\mathbf{r})$ for each $\sigma \in \Sigma$ then $P^{(\mathcal{A})}_w(\mathbf{r}) = P^{(\mathcal{B})}_w(\mathbf{r})$.

From Proposition 4 it is clear that the algorithm in Figure 3 generates a counter-example trace given $\mathbf{r}$ such that $P^{(\mathcal{A})}(\mathbf{r}) \neq P^{(\mathcal{B})}(\mathbf{r})$.

3 Experiments

Implementation. To evaluate the deterministic and randomised algorithms, we have implemented an equivalence checker for probabilistic automata in C++ and tested their performance on a 3.07 GHz workstation with 48GB of memory. The forward and the backward algorithm have about 100 lines of code each, and the randomised one around 80, not counting supporting code. We use the Boost uBLAS library to carry out vector and matrix operations. Numbers are represented as double-precision floating point numbers. In order to benefit from the sparsity of vectors and transition matrices, we store them in a sparse representation which only stores non-zero entries.

Sparsity also plays a role in the Gram-Schmidt procedure because, in our context, the argument vector u often shares non-zeros with only a few base vectors. We have implemented an optimised version of Gram-Schmidt that only projects out base vectors that share non-zero positions with the argument. This can lead to dramatic speed-ups in the deterministic algorithms (the randomised algorithm does not use Gram-Schmidt).

3.1 Herman's protocol

We consider Herman's self-stabilization protocol [11], in which N processes are arranged in a ring. Initially each process holds a token. The objective of Herman's protocol is to evolve the processes into a *stable* state, where only one process holds a token. The algorithm proceeds in rounds: in every round, each process holding a token tosses a coin to decide whether to keep the token or to pass it to its left neighbour; if a process holding a token receives an additional token, both of these tokens expire. We will be interested in the number of rounds that it takes to reach the stable state. The corresponding APEX model "announces" each new round by making calls to an undefined (external) procedure *round* inside the control loop of the protocol. Making the announcements at different program points within the loop (e.g. at the very beginning or the very end) results in different automata shown in Figure 4 (for $N = 3$).

A few remarks are due regarding our graphical representation of automata. Initial states are marked by a grey background. Accepting states are surrounded by a double border and labelled by a distribution over return values, e.g., the automaton on the left returns value 0 with a probability of $\frac{3}{4}$.

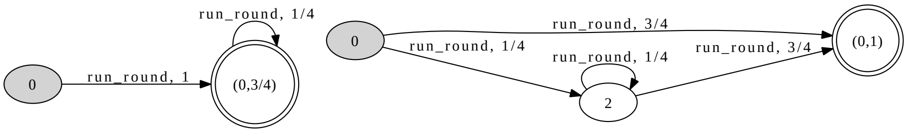

Fig. 4. Herman's protocol: automata for early (left) and late (right) announcement.

These automata are structurally different and not bisimilar. However, intuitively, it should not matter at which position within the loop the announcement is made and therefore we check if the resulting automata are equivalent. Here is how the four algorithms perform in this case study.

	states		equivalence checking			
N	s_{spec}	s_{prot}	$t_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$	$t_{\rightarrow,\text{rand}}$	$t_{\rightarrow,\text{det}}$
9	23	24	0.00	0.00	0.00	0.00
11	63	64	0.00	0.09	0.05	0.13
13	190	191	0.15	2.66	1.19	3.60
15	612	613	3.98	93.00	30.32	119.21

3.2 Grade Protocol

Assume that a group of students has been graded and would like to find out the sum of all their grades, e.g. to compute the average. However, none wants to reveal their individual grade in the process.

The task can be accomplished with the following randomised algorithm. Let $S \in \mathbb{N}$ be the number of students, and let $\{0, \cdots, G-1\}$ ($G \in \mathbb{N}$) be the set of grades. Define $N = (G-1) \cdot S + 1$. Further we assume that the students are arranged in a ring and that each pair of adjacent students shares a random integer between 0 and $N-1$. Thus a student shares a number l with the student on the left and a number r with the student on the right, respectively. Denoting the student's grade by g, the student announces the number $(g + l - r)\ mod\ N$. Because of the ring structure, each number will be reported twice, once as l and once as r, so the sum of all announcements (modulo N) will be the same as the sum of all grades. What we can verify is that only this sum can be gleaned from the announcements by a casual observer, i.e. the observer cannot gain access to any extra information about individual grades. This correctness condition can be formalised by a specification, in which the students make random announcements subject to the condition that their sum equals the sum of their grades.

Correct protocol. To check correctness of the Grade protocol, we check if the automaton resulting from the implementation of the protocol is equivalent to the specification automaton. Both automata are presented in Figure 5 for $G = 2$ and $S = 3$. Both automata accept words which are sequences of grades and announcements. The words are accepted with the same probability as that of producing the announcements for the given grades. Adherence to specification can then be verified via language equivalence.

In Figure 6 we report various data related to the performance of APEX and the equivalence checking algorithms. The reported times t_{spec} and t_{prot} are those needed to construct the automata corresponding to specification and the protocol respectively (we also give their sizes $s_{\text{spec}}, s_{\text{prot}}$). The columns labelled by $t_{\leftarrow,\text{rand}}, t_{\leftarrow,\text{det}}, t_{\rightarrow,\text{rand}}, t_{\rightarrow,\text{det}}$ give the running time of the language equivalence check of the respective automata using the four algorithms. The symbol $-$ means that the computation timed out after 10 minutes.

The running time of the deterministic backwards algorithm is higher than the one of the forward algorithm because the backwards algorithm constructs a different vector space with a higher number of dimensions.

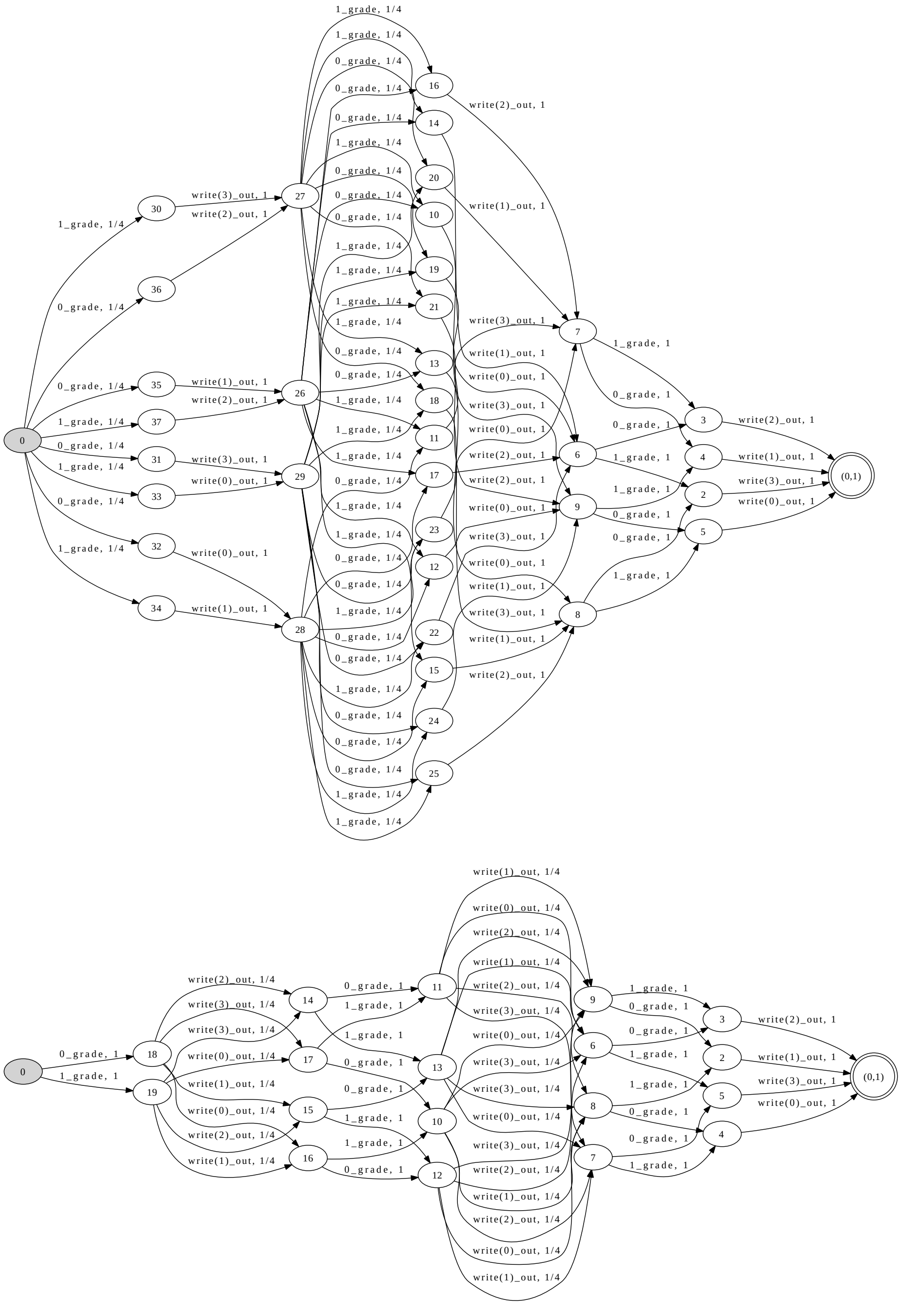

Fig. 5. Automata generated by APEX: protocol (top) and specification (bottom)

		APEX		states		equivalence checking			
G	S	t_{spec}	t_{prot}	s_{spec}	s_{prot}	$t_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$	$t_{\rightarrow,\text{rand}}$	$t_{\rightarrow,\text{det}}$
2	10	0.00	0.02	202	1102	0.00	0.41	0.00	0.02
2	20	0.05	0.24	802	8402	0.10	26.09	0.03	0.26
2	50	1.75	6.46	5,002	127,502	8.08	-	1.21	10.07
2	100	25.36	82.26	20,002	1,010,002	240.97	-	10.56	159.39
2	200	396.55	-	80,002	-				
3	10	0.02	0.16	383	3803	0.02	5.79	0.01	0.09
3	20	0.22	1.64	1,563	31,203	0.69	598.78	0.23	1.33
3	50	6.99	39.27	9,903	495,003	61.76	-	9.29	63.16
3	100	102.42	-	39,803					
3	200	-	-						
4	10	0.04	0.56	564	8124	0.07	37.32	0.04	0.36
4	20	0.53	5.47	2,324	68,444	2.25	-	0.77	4.95
4	50	15.94	142.59	14,804	1,102,604	210.00	-	30.27	222.50
4	100	232.07	-	59,604					
4	200	-	-						
5	10	0.09	1.46	745	14,065	0.17	164.87	0.10	0.68
5	20	0.96	16.39	3,085	120,125	5.36	-	1.92	14.49
5	50	28.81	417.58	19,705	1,950,305	508.86	-	62.64	335.01
5	100	417.59	-	79,405					
5	200	-	-						

Fig. 6. Equivalence checks for the Grade Protocol

		APEX		states		equivalence checking					
G	S	t_{spec}	t_{prot}	s_{spec}	s_{prot}	$t_{\leftarrow,\text{rand}}$	$t^{*}_{\leftarrow,\text{rand}}$	$t_{\leftarrow,\text{det}}$	$t_{\rightarrow,\text{rand}}$	$t^{*}_{\rightarrow,\text{rand}}$	$t_{\rightarrow,\text{det}}$
2	10	0.00	0.10	202	8,845	0.02	0.02	6.71	0.01	0.01	0.26
2	20	0.05	2.24	802	151,285	1.85	1.86	-	0.12	0.14	11.28
2	50	1.75	170.40	5,002	6,120,205	-	-	-	17.38	18.60	-
3	10	0.00	0.10	383	63685	0.51	0.51	-	0.06	0.07	6.81
3	20	0.05	2.24	1,563	1,150,565	86.57	86.61	-	2.51	2.84	441
3	50	1.75	170.40	9,903	-						
4	10	0.00	0.10	564	207,125	8.04	8.04	-	0.34	0.35	60.57
4	20	0.05	2.24	2,324	3,816,145	-	-	-	17.11	17.31	-
4	50	1.75	170.40	14,804	-						
5	10	0.00	0.10	745	481,765	27.22	27.25	-	1.33	1.37	269.71
5	20	0.05	2.24	3,085	8,966,725	-	-	-	72.29	72.53	-
5	50	1.75	170.40	19,705	-						

Fig. 7. Inequivalence checks for the Grade Protocol

Faulty protocol. Moving to test cases for inequivalence checking, we compare the specification for the Grade Protocol with a faulty version of the protocol in which the students draw numbers between 0 and $N-2$ (rather than $N-1$, as in the original protocol). The running times and automata sizes are recorded in Figure 7.

Both the deterministic and the randomised algorithms are decision algorithms for equivalence *and* can also generate counterexamples, which could serve, e.g., to repair a faulty protocol. We test counterexample generation on the faulty version of the protocol.

The deterministic algorithm stores words in the work list which immediately yield a counterexample in case of inequivalence. In the context of the randomised algorithm (as described in Section 2.4), counterexample generation requires extra work: words need to be reconstructed using matrix vector multiplications. To quantify the runtime overhead, we record running times with counterexample generation switched on and off.

The running time of the backward randomised algorithm with counterexample generation is denoted by $t^*_{\leftarrow,\mathrm{rand}}$ and analogously for the forward algorithm. The runtime overhead of counterexample generation remains below 12%.

3.3 Randomised Inflation

Next we discuss two procedures that can be used to produce random instances of equivalent automata. We introduce two directional variants (by analogy to the forward and backward algorithms) that, given an input automaton, produce an equivalent automaton with one more state. Intuitively, equivalence is preserved because the new state is a linear combination of existing states, which are then suitably adjusted.

Backward Inflate From a given automaton $\mathcal{A} = (n, \Sigma, M, \alpha, \eta)$, we construct an equivalent automaton $\mathcal{A}'$ with an additional state such that the automaton $\mathcal{A}'$ is not bisimilar to the original automaton $\mathcal{A}$. The automaton $\mathcal{A}' = (n+1, \Sigma, M', \alpha', \eta')$ has an additional state that corresponds to a linear combination of states in $\mathcal{A}$. To this end, the inflation procedure randomly picks several vectors:

- a single row vector $\delta \in \mathbb{R}^n$ which controls the transitions out of the new state; to obtain a substochastic transition matrix, we ensure $\delta \geq (0, \ldots, 0)$ (where $\geq$ is meant componentwise) and $\delta(1, \ldots, 1)^\top = 1$;
- column vectors $s(\sigma) \in \mathbb{R}^n$ for all $\sigma \in \Sigma$ that define the probability that flows into the new state; for probabilistic automata one should ensure $(0, \ldots, 0)^\top \leq s(\sigma) \leq (1, \ldots, 1)^\top$ and $s(\sigma)\delta \leq M(\sigma)$.

We define the transition matrices such that the new state $n+1$ is a linear combination of the old states, weighted according to δ. The transition matrix for $\sigma \in \Sigma$ is defined as folllows:

$$M'(\sigma) := \begin{pmatrix} M(\sigma) - s(\sigma)\delta & s(\sigma) \\ \delta M(\sigma) & 0 \end{pmatrix} \quad \text{and} \quad \alpha' := (\alpha \;\; 0) \quad \text{and} \quad \eta' := \begin{pmatrix} \eta \\ \delta\eta \end{pmatrix} .$$

Proposition 5. *$\mathcal{A}$ and $\mathcal{A}'$ are equivalent.*

Proof. For any word $w = \sigma_1 \cdots \sigma_k \in \Sigma^*$, define $c(w) := M(\sigma_1) \cdots M(\sigma_k)\eta$ and $c'(w) := M'(\sigma_1) \cdots M'(\sigma_k)\eta'$. We will show:

$$c'(w) = \begin{pmatrix} c(w) \\ \delta c(w) \end{pmatrix} \quad \text{for all } w \in \Sigma^*. \tag{4}$$

Equation (4) implies the proposition, because $\mathcal{A}'(w) = \alpha' c'(w) \overset{(4)}{=} \alpha c(w) = \mathcal{A}(w)$. We prove (4) by induction on the length of w. For the induction base, we have $c'(\varepsilon) = \eta' = \begin{pmatrix} \eta \\ \delta\eta \end{pmatrix} = \begin{pmatrix} c(\varepsilon) \\ \delta c(\varepsilon) \end{pmatrix}$. For the induction step, assume (4) holds for w. Let $\sigma \in \Sigma$. Then we have

$$\begin{aligned} c'(\sigma w) = M'(\sigma)c'(w) &= \begin{pmatrix} M(\sigma) - s(\sigma)\delta & s(\sigma) \\ \delta M(\sigma) & 0 \end{pmatrix} \begin{pmatrix} c(w) \\ \delta c(w) \end{pmatrix} \\ &= \begin{pmatrix} M(\sigma)c(w) \\ \delta M(\sigma)c(w) \end{pmatrix} = \begin{pmatrix} c(\sigma w) \\ \delta c(\sigma w) \end{pmatrix} , \end{aligned}$$

completing the induction proof. □

Forward Inflate Forward Inflate is a variant obtained by "transposing" Backward Inflate. In this variant it is hard to keep the matrices stochastic. However, nonnegativity can be preserved. Construct an automaton $\mathcal{A}' = (n + 1, \Sigma, M', \alpha', \eta')$ as follows. Pick randomly:

- a column vector $\delta \in \mathbb{R}^n$;
- row vectors $s(\sigma) \in \mathbb{R}^n$ for all $\sigma \in \Sigma$; for nonnegative automata one should ensure $\delta s(\sigma) \leq M(\sigma)$.

Define

$$M'(\sigma) := \begin{pmatrix} M(\sigma) - \delta s(\sigma) & M(\sigma)\delta \\ s(\sigma) & 0 \end{pmatrix} \quad \text{and} \quad \alpha' := (\alpha \;\; \alpha\delta) \quad \text{and} \quad \eta' := \begin{pmatrix} \eta \\ 0 \end{pmatrix} .$$

Proposition 6. *$\mathcal{A}$ and $\mathcal{A}'$ are equivalent.*

The proof is analogous to that of Proposition 5.

We have conducted experiments in which we compare automata with their multiply inflated versions. Our initial automaton is the doubly-linked ring of size 10, i.e. the states are $\{0, \cdots, 10\}$, there are two letters l and r such that the probability of moving from i to $(i+1)$ *mod* 10 on r is 0.5 and the probability of moving from i to $(i-1)$ *mod* 10 on l is 0.5 (otherwise the transition probabilities are equal to 0). The table below lists times needed to verify the equivalence of the ring with its backward inflations.

#inflations	$t_{\leftarrow,\mathrm{rand}}$	$t_{\leftarrow,\mathrm{det}}$
1	0.001282	0.00209
5	0.001863	0.004165
10	0.002986	0.008560
20	0.003256	0.007932
50	0.009647	0.058917
100	0.045588	0.306103
200	0.276887	2.014591
500	3.767168	29.578838

4 Conclusion

This paper has presented equivalence checking algorithms for probabilistic automata. We have modified Tzeng's original deterministic algorithm [21]: by keeping an orthogonal basis of the underlying vector spaces, we have managed to improve the original quartic runtime complexity to a cubic bound. Further, we have presented a novel randomised algorithm with an even lower theoretical complexity for sparse automata. Our randomised algorithm is based on polynomial identity testing. The randomised algorithm may report two inequalent automata to be equivalent. However, its error probability, established by a theorem of Schwarz and Zippel, can be made arbitrarily small by increasing the size of the sample set and by repeating the algorithm with different random seeds. The randomised algorithm outperforms the deterministic algorithm on a wide range of benchmarks. Confirming the theoretical error probabilities the algorithm has detected all inequivalent automata and was able to generate suitable counterexample words.

References

1. C. Baier, N. Bertrand, and M. Größer. Probabilistic automata over infinite words: Expressiveness, efficiency, and decidability. In *DCFS*, volume 3 of *EPTCS*, pages 3–16, 2009.
2. V. D. Blondel and V. Canterini. Undecidable problems for probabilistic automata of fixed dimension. *Theoretical Computer Science*, 36 (3):231–245, 2003.
3. M. Blum, A. Chandra, and M. Wegman. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Inf. Process. Lett.*, 10(2):80–82, 1980.
4. K. Chatterjee, L. Doyen, and T. A. Henzinger. Probabilistic weighted automata. In *CONCUR*, volume 5710 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 2009.
5. Linda Christoff and Ivan Christoff. Efficient algorithms for verification of equivalences for probabilistic processes. In *CAV*, volume 575 of *Lecture Notes in Computer Science*, pages 310–321. Springer, 1991.
6. A. Condon and R. Lipton. On the complexity of space bounded interactive proofs (extended abstract). In *Proceedings of FOCS*, pages 462–467, 1989.
7. C. Cortes, M. Mohri, and A. Rastogi. On the computation of some standard distances between probabilistic automata. In *Proc. of CIAA*, pages 137–149, 2006.
8. V. Danos and R. Harmer. Probabilistic game semantics. *ACM Transactions on Computational Logic*, 3(3):359–382, 2002.

9. R. DeMillo and R. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978.
10. L. Doyen, T. Henzinger, and J.-F. Raskin. Equivalence of labeled Markov chains. *International Journal of Foundations of Computer Science*, 19(3):549–563, 2008.
11. T. Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.
12. A. Legay, A. S. Murawski, J. Ouaknine, and J. Worrell. On automated verification of probabilistic programs. In *Proceedings of TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2008.
13. Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Compositionality for probabilistic automata. In *CONCUR*, volume 2761 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2003.
14. A. S. Murawski and J. Ouaknine. On probabilistic program equivalence and refinement. In *Proceedings of CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2005.
15. A. Paz. *Introduction to Probabilistic Automata.* Academic Press, 1971.
16. M. O. Rabin. Probabilistic automata. *Information and Control*, 6 (3):230–245, 1963.
17. M.-P. Schützenberger. On the definition of a family of automata. *Inf. and Control*, 4:245–270, 1961.
18. J. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.
19. R. Segala. Modeling and verification of randomized distributed real -time systems. Technical report, MIT, Cambridge MA, USA, 1996.
20. L.N. Trefethen and D. Bau. *Numerical Linear Algebra.* SIAM, 1997.
21. W. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21(2):216–227, 1992.
22. R. Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 1979.