Learning Markov Decision Processes Based on Genetic Programming*

Rong Wu
School of Software Engineering
Tongji University
Shanghai, China
wu_rong@tongji.edu.cn

Jin Xu
School of Software Engineering
Tongji University
Shanghai, China
slhjin@tongji.edu.cn

***Abstract*—Model checking is used to verify the security of communication protocols in which the behavior is stochastic influenced by the environment. Automata learning settles the problem of obtaining formal models from observable data of black-box systems. It is available for different variations of finite automata to in model checking. Genetic Programming is a machine learning technique that automatically generates programs and outputs a fittest program. In this paper, we present an approach to learn markov decision progresses based on the framework of genetic programming. The approach outputs the fittest model with a set of system traces by refining iteratively models. We evaluate our method on one probabilistic system from the literature and 30 randomly generated examples.**

Keywords—automata learning, genetic programming, markov decision processes, model checking

I. Introduction (*Heading 1*)

Model checking is a important technique of verifying system security. Probabilistic model checking can offer the probability of satisfying some property, e.g. model checking tools Prism [10] accepts LTL or PCTL formulae describing the property and outputs the optimal probability. In practical ap- plications, the internal structure of a system may be unknown since the third-party modules. Automata learning settles the problem of obtaining formal models from observable test data of black-box systems. In fact, it is available for many variations of finite automata in LearnLib [12] which efficiently implements the learning algorithms of different automata, e.g. deterministic finite automata (DFA) and mealy machines.

Genetic programming [2] is a machine learning technique that automatically generates programs and modifies iteratively these programs. Using the technique of model-based testing [13], some learning algorithms have learnt successfully models via genetic programming, such as DFA [7], [8], timed automata [4],[6]. It can learn a precise models consistent with a set of traces of black-box systems. However it is unavailable for Markov Decision Process (MDPs). MDPs are used to model stochastic systems, e.g. communication protocol. For protocol verification it is crucial to obtain the formal models from the protocols using automata learning.

In this paper, we indicate that MDPs can be learned based the framework of genetic programming and we evaluate our method on one academic probabilistic system and 30 randomly generated examples. The contribution of this paper is we propose a heuristic method to learn MDPs that shows the learned models are closed to true models. The outline of this article shows as follows: Sect. II reviews the concepts of markov decision processes and genetic programming. Sect. III presents the method of learning MDPs based on genetic programming. The experimental results of our method are listed in Sect. IV and we provide the discussion in Sect. V.

II. Preliminaries

A. *Markov Decision Processes*

Markov decision processes (MDPs) can express the systems with stochastic behaviors, in which a transition happens with some probability depending on the chosen input. We learn deterministic labelled MDPs as in [1]. A labelled markov decision process is a tuple $\mathcal{M} = (Q, \Sigma^I, \Sigma^O, q_0, \delta, L)$ where Q is a finite non-empty set of states; Σ^I and Σ^O are finite sets of input and output symbols respectively, $q_0 \in Q$ is the initial state; $\delta \subseteq Q \times \Sigma^I \times \mathrm{Dist}(Q)$ is the set of probabilistic transitions. $\mathrm{Dist}(Q)$ denotes a set of probabilistic distributions. A distribution $\mu\colon q \to [0,1]$ over set Q is a function mapping q to $p \in [0,1]$ such that $\sum_{q\in Q} \mu(q) = 1$. Let μ_q denotes the *point distribution* at q which assigns 1 to q; and $L\colon Q \to \Sigma^O$ is the labelling function. An MDP is deterministic if $\forall (q, i, \mu) \in \delta\colon \mu(q') > 0 \wedge \mu(q'') > 0 \to q' = q'' \vee L(q') \neq L(q'')$.

We use the terms traces and test sequences similarly to [1]. A path ρ through an MDP is an alternating sequence of states and input symbols starting in the initial state q_0, i.e. $\rho = q_0 \cdot i_1 \cdot q_1 \cdot i_2 \cdot q_2 \cdots i_n \cdot q_n$. In each state q_k, The next state q_{k+1} is selected probabilistically depending on the distribution μ_{k+1} such that $(q_k, i_{k+1}, \mu_{k+1}) \in \delta$. We observe a trace $L(\rho) = t = o_0 \cdot i_1 \cdot o_1 \cdot i_2 \cdot o_2 \cdots i_n \cdot o_n$ during the execution of a path ρ with $L(q_k) = o_k, 0 \leq k \leq n$. Notably, a trace leads to a unique state of an MDP $\mathcal{M}$. $\delta^*(t)$ denotes the state reached by the trace t or $\perp$ if t is unreachable i.e. the path of MDP $\mathcal{M}$ corresponding trace t is nonexistent. A test sequence s consists of a trace t and a input i, thus after executing test sequence $s = t \cdot i$, $\mathcal{M}$ reaches the state $\delta^*(t)$ and enables input i to tests the response.

The semantics of a MDP $\mathcal{M} = (Q, \Sigma^I, \Sigma^O, q_0, \delta, L)$ is defined as a function M which maps a test sequence s to a distribution μ if s is reachable on $\mathcal{M}$ or $\perp$ otherwise. Concretely, we have $M(s)$ as follows: a) $M(s) = \mu_{L(q_0)}$ if $s =$

ϵ; b) for $s = t \cdot i, t \in \mathcal{TR}, i \in \Sigma^I$, $M(s) = \perp$ if $\delta^*(t) = $, or $M(s)(o) = p$ otherwise if $(\delta^*(t), i, \mu) \in \delta \wedge \mu(q) = p > 0 \wedge L(q) = o$. $\mathcal{TR} = \Sigma^O \times (\Sigma^I \times \Sigma^O)^*$ denotes the set of traces. The equivalence is approximated as conformance checking between learned model (hypothesis) and the system under test (SUT). After executing test sequences, we check conformance between the SUT's outputs and the distributions predicted by the hypothesis.

B. Genetic Programming

Genetic programming [2] is similar to genetic algorithm [3] that evolves computer programs. Populations in genetic programming consists of individuals (programs), and each individual represents some potential solution to a specific problem. Generally speaking, genetic programming modify iteratively individuals in the population until an acceptable solution is found or the maximum of iterations has been achieved. Initial population is randomly created and then the population is updated by genetic operators. A fitness function is used to evaluate individuals in the population, and the next generation of individuals are generated by applying genetic operators to individuals selected. The selection method bases on the fitness function to select the fittest members of the population. Three types of genetic operators are commonly used: reproduction, mutation, and crossover.

- Reproduction: copy an individual as a new individual in the population.
- Mutation: change an individual introducing individual different from the last generation.
- Crossover: make an exchange of two individuals randomly chosen to create offspring.

III. Genetic Programming for Markov Decision Processes

The genetic-programming framework for MDP is similar with [4]. Fig. 1 depicts the procedure of genetic programming for MDPs in which each individual is a MDP. An initial population is generated using random strategy based on the provided inputs and outputs of the SUT. Then each MDP (all n_{pop}) of this population are evaluated by n_{test} test sequences generated before-hand and a fitness function. The fitness function considers the amount of passing test sequences and features of the learned MDP, e.g. size and non-deterministic behavior. The termination criteria depends on the amount of generations gen having been performed and the best individual $best$ of the population that passes all test cases. In other words, the procedure terminates if either $best$ exists and the fitness is unchanged during g_{change} generations or gen has reached g_{max}, where g_{change} and g_{max} are parameters. The fittest MDP of final generation is the output of the procedure as learned model.

New Population is created if termination criteria is not met. New individuals are based on the individuals existing. Afterward, for all individuals selected we mutate them with probability p_m, copy it to new population with probability $\frac{1-p_m}{2}$ and randomly select other individual to create together their offsetting with probability $\frac{1-p_m}{2}$. We have a new population after performing n_{pop} genetic operators to individuals chosen. Then the new population is evaluated to find the fittest MDP.

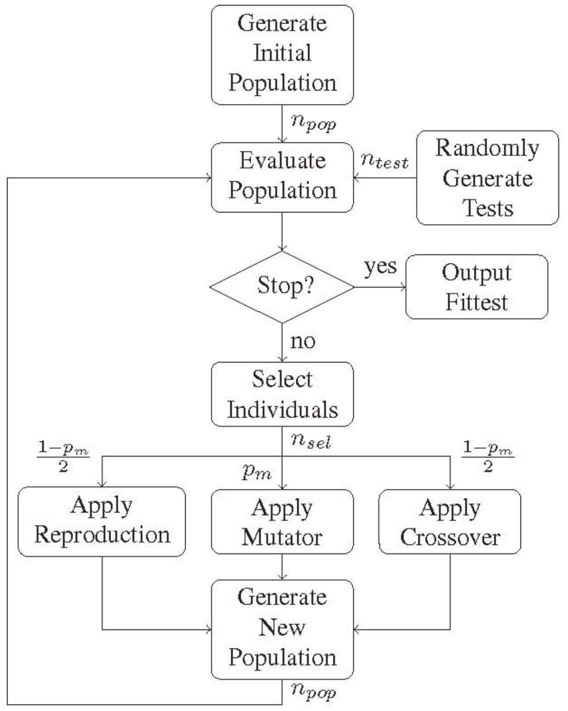

Fig. 1 Procedure of Genetic Programming for MDP

The initial population is created by randomly generating n_{pop} MDPs. A MDP is generated though randomly adding states (the number of states at lease 2), and randomly selecting a source state, a input and a distribution for transitions. The number of transitions is controlled by whether the initial state of the MDP is reachable with all the states and probability p_{trans}. Distributions are generated after randomly selecting target states and assigning a probability bigger than 0 to them such that the sum of the probabilities equal 1. Notably, the labels of all the target states included by the same distribution must be different for each pair. Subsequently, we will illustrate the creation of test sequences, fitness function and the procedure of creating a new population in detail.

A. Creation of Test sequences

As illustrated, a test sequence is a alternating sequence of outputs and inputs, thus the creation of test sequences is not feasible only using random strategy since MDPs are not output-enabledness for any output. on the other hand, the equivalence between MDPs depends on the output distributions obtained by executing test sequences on them. We have MDP $\mathcal{M}_1$ with semantics M_1 and MDP $\mathcal{M}_2$ with semantics M_2 are equivalent if both output distributions i.e. $M_1(s)$ and $M_2(s)$ are equivalent for every test sequence s. However, the output distribution of SUT is not available directly since it is black-box. The interfaces of SUT capturing MDP $\mathcal{M}$ we use as follow:

- $reset$. reset SUT to the initial state, i.e. current state is initial state.

- $step$. execute an input on SUT and select probabilistically a state according to the distribution of the probabilistic transition in $\mathcal{M}$ which starts from current state before $step$ and its input is the same with the input of executing. The current state update to selected state after $step$.

After executing, $reset$ and $step$ both return the label of current state.

Based on the both operations $reset$ and $step$, test sequences are generated by executing inputs step by step on SUT. Namely, a test sequence $s = t \cdot i$ implies MDP $\mathcal{M}$ underlying SUT with inputs Σ^I exists a path ρ such that $L(\rho) = t$ where t is a trace and $i \in \Sigma^I$. We have that for all prefixes $s' = t' \cdot i', i' \in \Sigma^I$ of any test sequence s, t' is reachable on $\mathcal{M}$. It is worth noting that the length of test sequences, the amount of inputs in test sequences, is geometrically distributed with p_{stop}.

As for the output distributions of SUT, we approximate them using statistical method. In other words, we execute test sequence s on SUT to observe the outputs, and record the frequency of every observable output by executing repeatedly s. Thus the probability of any observable output is the ratio of itself frequency and the sum of frequencies recording all the observable outputs after executing s. For example, we have observed outputs o_1 and o_2 after n executing with s on SUT, and the frequencies of o_1 and o_2 are f_1 and f_2 respectively, so the output distribution μ for s is $\mu(s)(o_1) = \frac{f_1}{n}, \mu(s)(o_2) = \frac{f_2}{n}$. It is reliable for the output distributions obtained by statistical method if the number of executing the same test sequence exceeds a big enough number n_o on SUT according to the law of large numbers. The value of n_o is decided by specific situation.

We use $test\ case$ to denote the structure consisting of a test sequence and corresponding output distribution of SUT. Algorithm 1 shows the method obtaining test sequences and corresponding output distributions of SUT as discussed. The fitness function illustrated later evaluates individuals using $test\ case$.

B. Fitness Evaluation

It is important for genetic programming to calculate rationally the fitness of individuals. Intuitively, the individual with the highest score of fitness has the same reactions with SUT if executing any test sequence. As illustrated, a deterministic MDP $\mathcal{M}$ returns either a distribution μ or the undefined behavior ⊥ when executing test sequence s. The undefined behavior ⊥ means that trace t is unreachable on $\mathcal{M}$ if $s = t \cdot i, i \in \Sigma^I$. Here we define that for MDP $\mathcal{M}$ with semantics M a test sequence s is $Fail$ if $M(s) = \perp$.

It is not certain for a test sequence that is not $Fail$. In general MDPs of population are deterministic, and we just have a result that either is a distribution or ⊥ if we execute a test sequence on learned model. However, mutation and crossover may introduce non-deterministic behaviour to new MDPs. Non-deterministic MDPs may lead to multiple paths existing if executing the same test sequence. Our goal is finding a deterministic and equivalent model in the population, thus for a learned model $\mathcal{M}$ a test sequence s is $Pass$ if neither shows non-deterministic behavior nor is unreachable during executing s on $\mathcal{M}$. It is easy to know a test sequence s is $Nondet$ for MDP $\mathcal{M}$ if $\mathcal{M}$ exhibits non-deterministic behavior during executing s.

Algorithm 1 The Generation of Test Sequences

```
Require:
    SUT; the threshold of reliable outputs for test sequences $n_o$;
    the probability of stopping test $p_{stop}$;
    the sets of inputs $\Sigma^I$ and outputs $\Sigma^O$
Ensure:
    a test sequence $s$ and corresponding output distribution $\mu$
 1: $reset$;
 2: $i = \epsilon, s = \epsilon$;
 3: repeat
 3:     $o = step(i)$;
 3:     randomly select a input $i \in \Sigma^I$;
 3:     $s = s \cdot o \cdot i$;
 4: until $p_{stop}$ is satisfied
 5: for all $o \in \Sigma^O$, $f(o) = 0$;
 6: for $k : 1 \to n_o$ do
 7:     execute $s$ on SUT, and observe the output $o'$ of SUT;
 8:     $f(o') = f(o') + 1$;
 9: end for
10: for all $o \in \Sigma^O$, $\mu(o) = f(o)/n_o$;
11: return $s$ and $\mu$;
```

Another factor to influence the fitness of individuals is the distance between distributions. The verdict of test sequence just consider the traces between learned model and SUT, and we add the distance of output distribution as a part of fitness function. Here we use the Kullback-Leibler Divergence (KLD) 5 to measure the distance of distribution. According to KLD, the distance of both the output distributions μ_h, μ_s obtained by executing the same test sequence on learned model $\mathcal{M}$ and SUT respective, is defined as

$$D_{KL}(\mu_h || \mu_s) = \sum_{o \in support(\mu_s)} \mu_h(o) log_2 \left(\frac{\mu_h(o)}{\mu_s(o)}\right) \qquad (1)$$

where $support(\mu_s) = \{o \in \Sigma^O | \mu_s(o) > 0\}$. We set a constant c to negate the distance since smaller $D_{KL}(\mu_h || \mu_s)$ implies the learned model is more fitted. Thus we have $Dis(\mu_h, \mu_s) = c - D_{KL}(\mu_h, \mu_s)$ denotes the similarity of between both the output distributions μ_h, μ_s from learned model $\mathcal{M}$ and SUT respective.

We use weights to control the fitness of individuals. As discussed, $Fit(s|\mathcal{M})$ denotes the fitness of the test sequence s for MDP $\mathcal{M}$, controlled by weights ω_{pass}, ω_{fail}, ω_{nonde} and ω_{pro}. we have $Fit(s|\mathcal{M}) =$

$$\begin{cases} \omega_{pass} + \omega_{pro} Dis(\mu_h, \mu_s), & Ver(s|\mathcal{M}) = Pass \\ \omega_{fail}, & Ver(s|\mathcal{M}) = Fail \\ \omega_{nonde}, & Ver(s|\mathcal{M}) = Nondet \end{cases} \qquad (2)$$

where $Ver(s|\mathcal{M})$ is the verdict of test sequence s for MDP $\mathcal{M}$. The fitness function is defined as

$$Fit(\mathcal{M}) = \sum_{s\in\mathcal{TS}} Fit(s|\mathcal{M}) - \omega_{size}(\mathcal{M}) \qquad (3)$$

where $\mathcal{TS}$ is the set of test sequences obtained before-hand, and $\omega_{size}(\mathcal{M})$ is related to the size of $\mathcal{M}$, e.g. the numbers of states and transitions. If two individuals have the same fitness of test sequence for every test sequence but the size of them is different, the individual of bigger size is redundant.

C. *Creation of New Population*

As discussed, a new population generated by applying genetic operators to selected individuals. Here, we use tournament selection to select n_{sel} individuals according to their fitness. Tournament selection chooses the fittest individual of a subset of population, and the subset of population also randomly selected.

Based on Mutation, we design some operations to change a part of origin MDP respectively. TABLE I. lists operators related to distribution. The others is the same with [4] (except for operations related to clocks), such as changing the source state of a transition, removing or adding a transition, splitting a state or a transition and adding or merging states.

TABLE I. MUTATION OPERATIONS

Operation	**Short description**
add target	selects a distribution of a transition, adds a new state to the distribution with label different with labels existing, and assigns probabilities to target states of the distribution
change target	selects a distribution of a transition and changes one of target states into a selected state with label different with labels existing
remove target	selects a distribution of a transition and removes one of target state
change probabilities	selects a distribution of a transition and assigns new probabilities to origin targets

We implement crossover as a parallel composition of two individuals defined in [11]. Namely, for the parents $\mathcal{M}^k = (Q^k, \Sigma^I, \Sigma^O, q_0^k, \delta^k, L^k), k = \{1,2\}$ we have the offspring of them is $\mathcal{M}^1 \parallel \mathcal{M}^2 = (Q^1 \times Q^2, \Sigma^I, \Sigma^O, (q_0^1, q_0^2), \delta, L)$ such that, for each state pair $(q^1, q^2) \in Q^1 \times Q^2$ and input $i \in \Sigma^I$:

- $\forall\big((q^1, q^2), i, \mu\big) \in$, we have

$$\mu = \begin{cases} \mu^1 \otimes \mu^2, \ (q^1, i, \mu^1) \in \delta^1 \wedge (q^2, i, \mu^2) \in \delta^2 \\ \mu^1 \otimes \mu_{q^2}, \ (q^1, i, \mu^1) \in \delta^1 \wedge \nexists \mu^2 : (q^2, i, \mu^2) \in \delta^2 \\ \mu_{q^1} \otimes \mu^2, \ (q^2, i, \mu^2) \in \delta^2 \wedge \nexists \mu^1 : (q^1, i, \mu^1) \in \delta^1 \end{cases} \qquad (4)$$

where $\mu \otimes \mu' \in Dist(Q^1 \times Q^2)$ such that $\mu \otimes \mu'(q^1, q^2) = \mu(q^1) \cdot \mu'(q^2)$ for $\mu \in Dist(Q^1)$ and $\mu' \in Dist(Q^2)$.

- the labelling function is given by $L(q^1, q^2) = Rand\big(L^1(q^1), L^2(q^2)\big)$, i.e. randomly selected between $L^1(q^1)$ and $L^2(q^2)$.

IV. CASE STUDY

It is difficult to achieve complete equivalence between a learned model and the target model represented by the SUT if the models exhibit probabilistic behavior. So we evaluate the accuracy between learned models and target models using two measurements as in [1]: the discounted bisimilarity distance [9] between the learned model and the target model and compare probabilities of temporal properties predefined with all models by PRISM [10].

For deciding the distribution whether adequately close true output distribution, we set $n_o = 2000$. As for the value of constraint c, we set $c = 55$ according to the experiments. The parameters related to genetic programming is similar for all experiments. we set $n_{pop} = 1000$, $n_{sel} = n_{pop}/10$, $g_{max} = 1000$, $g_{change} = 20$. We set $p_{tran} = 0.2$, $p_{stop} = 0.2$ to control the generation of MDPs and test sequences. As for the weights of fitness function, we set $\omega_{pass} = 5$, $\omega_{fail} = \omega_{nonde} = 0$, $\omega_{size} = 1$, $\omega_{pro} = 1$. We all evaluate 3 academic probabilistic systems. All Experiments run on a notebook with 16 GB RAM, an Intel Core i7-9750H CPU operating at 2.6 GHz and running Windows 10. The measurement results of 32 target models show in TABLE II. The outputs represents the number of executed $step$.

In addition to the academic probabilistic system (coffee machine, grid world robot), we have three groups of random MDPs, each containing ten MDPs: 10_4_4, 20_4_4, 30_4_4, where the first number gives the number of states, the second and the third the number of inputs and outputs. For random Examples, we have not verify the temporal properties, and we just evaluate the discounted bisimilarity distance with $\lambda = 0.9$. We set $n_{test} = 300$. The average discounted bisimilarity distance are at most 0.1977 for the group 10_4_4, 20_4_4 and 30_4_4.

TABLE II. MEASUREMENT RESULTS

target models	**states**	**$\delta_{0.9}$**	**outputs**	**time(min)**
coffee machine	3	0.0329	65868	0.05
grid world robot	36	0.1997	76525285	376
10_4_4	10	0.1873	21463472	20
20_4_6	20	0.1937	48413822	81
30_4_10	30	0.1963	70138697	137

Coffee Machine is a small and easy model to express the behavior of obtaining a coffee with the probability 0.8 and occurring a faulty with the probability 0.2 if putting a coin in the machine. The true model of coffee machine has 3 different states and 2 input symbols. We set $n_{test} = 200$, $\omega_{pass} = 10$, $\omega_{fail} = \omega_{nonde} = 0$, $\omega_{size} = 40$, $\omega_{pro} = 0.01$. Specially, it is adequate to set $n_{pop} = 500$ and $g_{change} = 5$. The evaluation of bisimilarity distance with $\lambda = 0.9$ to the true model is 0.0329.

Grid world robot is a control strategy similar with [1], in which a robot move from an initial location to a goal location. But the movements of the robot may arise faulty with a certain probability. The true model of grid world has 35 different states and 4 input symbols. We set $n_{test} = 500$. The evaluation of bisimilarity distance with $\lambda = 0.9$ to the true model is 0.1997. The probabilities of temporal properties between true model and learned model are listed in TABLE III. These temporal properties ask for the maximum probability of reaching the goal within a vary amount of steps. p_{true} and $p_{learned}$ represent the probabilities of true model and learned model respectively. The absolute different between and $p_{learned}$ is at most 0.0391.

TABLE III. EVALUATION OF GRID WORLD ROBOT FOR TEMPORAL PROPERTIES

temporal properties	p_{true}	$p_{learned}$
$\mathbb{P}_{max}(F^{\leq 11}(goal))$	0.9622	0.9441
$\mathbb{P}_{max}(F^{\leq 14}(goal))$	0.6499	0.6139
$\mathbb{P}_{max}(F^{\leq 16}(goal))$	0.6912	0.6521

V. CONCLUSION

In this paper, we present a heuristic method to learn MDPs that is based the framework of genetic programming. According to the general framework of genetic programming, we make an adaption in the generation of test sequences, the fitness function and the implementation of creating new population. The method outputs a learned model that is consistent with traces from black-box system. We evaluate one academic probabilistic system and 30 random examples that could be closed to true models.

But this work has some aspects that are worthy of improvement. The accuracy of the learned models depends on whether the test data include completely the behavior of the target model. Although the test sequences we use are randomly generated, it outputs precise learned models based on massive data. Thus the future works we are considering include to reduce the amount of data by interacting with the systems.

REFERENCES

[1] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G. Larsen. 2019. L*-Based Learning of Markov Decision Processes (Extended Version). CoRR abs/1906.12239 (2019). arXiv:1906.12239 http://arxiv.org/abs/190

[2] J. R. Koza. Genetic Programming: On the programming of Computers by Means of Natural Selection. MIT Press, London, England, 1992.

[3] Melanie Mitchell. An introduction to genetic algorithms. MIT Press, 1998.

[4] Tappler, M., Aichernig, B.K., Larsen, K.G., and Lorber, F. (2018). Learning Timed Automata via Genetic Programming. ArXiv, abs/1808.07744.

[5] T. M. Cover and J. A. Thomas. Elements of Information Theory. Wiley Series in Telecommunications. John Wiley and Sons, 1991.

[6] Aichernig B.K., Pferscher A., Tappler M. (2020) From Passive to Active: Learning Timed Automata Efficiently. In: Lee R., Jha S., Mavridou A., Giannakopoulou D. (eds) NASA Formal Methods. NFM 2020. Lecture Notes in Computer Science, vol 12229. Springer, Cham.

[7] B. D. Dunay, F. E. Petry and B. P. Buckles, "Regular language induction with genetic programming," Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, 1994, pp. 396-400 vol.1.

[8] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. 2012. Automatic generation of regular expressions from examples with genetic programming. In Proceedings of the 14th annual conference companion on Genetic and evolutionary computation (GECCO '12). Association for Computing Machinery, New York, NY, USA, 1477–1478.

[9] Bacci, G., Bacci, G., Larsen, K.G., Mardare, R.: The BisimDist library: Efficient computation of bisimilarity distances for Markovian models. In: Joshi, K.R., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8054, pp. 278–281.Springer (2013).

[10] Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585– 591. Springer (2011).

[11] Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2), 250–273 (1995).

[12] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib: A Framework for Active Automata Learning. https://doi.org/10.1007/978-3-319-216

[13] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. 2017. Model-Based Testing IoT Communication via Active Automata Learning. In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). 276–287. https://doi.org/10.1109/ICS

[14] James Aspnes and Maurice Herlihy. 1990. Fast randomized consensus using shared memory. Journal of Algorithms 11, 3 (Sept. 1990), 44.