Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2018-IC-005

2018-IC-005

Chasing Errors Using Biasing Automata

Lei Bu, Doron Peled, Dashuan Shen, Yael Tzirulnikov

Leveraging Applications of Formal Methods, Verification and Validation 2018

Chasing Errors Using Biasing Automata

Lei Bu[2], Doron Peled[1(✉)], Dashuan Shen[2], and Yael Tzirulnikov[1]

[1] Department of Computer Science, Bar Ilan University, Ramat Gan, Israel
doron.peled@gmail.com
[2] State Key Laboratory for Novel Software Technology,
Nanjing University, Nanjing, China

Abstract. Randomized testing is a lightweight approach for searching for bugs. It presents a tradeoff between the number of testing experiments performed and the probability to find errors. An important challenge in random testing is when the errors that we try to detect are scattered with very low probability among the different executions, forming a "rare event". We suggest here the use of a "biasing automaton", which observes the tested sequence and controls the distribution of the different choices of extending it. By the careful selection of a biasing automaton, we can increase the chance of errors to be found and consequently reduce the number of tests we need to perform. The biasing automaton is constructed through repeated testing of variants of the system under test. We show how to construct biasing automata based on genetic programming.

1 Introduction

Testing and model checking are complementary methods for achieving system reliability. While model checking is comprehensive, covering *all* the executions in a model of the system, testing is sometimes more affordable, allowing to *sample* the executions. The border between these techniques is becoming blurred, as new hybrid methods are developed. The choice between methods and tools is indeed not easy, with a clear tradeoff between coverage and complexity. *Statistical model checking* [13,14,17], applies repeated verification of executions against a specification (as in run-time verification), based on random sampling. The number of samples that need to be checked grows when the focus is on detecting event that contain executions that occur with low probability. As the expectation to hit an execution that belongs to such an event during random (Monte Carlo) testing is low, the potential advantage of using it over model checking diminishes.

Our goal is to provide a light-weight approach for random search for errors. It is similar to statistical model checking in that the executions are sampled

The research in this paper was partially funded by an ISF-NSFC grant "Runtime Measuring and Checking of Cyber Physical Systems" (ISF award 2239/15, NSFC No. 61561146394). The authors from Nanjing University were also partially funded by a National Natural Science Foundation of China grant No. 61572249.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11245, pp. 271–286, 2018.
https://doi.org/10.1007/978-3-030-03421-4_18

and analyzed against a temporal specification. However, we do not attempt to calculate the probability of occurrence of erroneous executions, but rather to discover them. The method is based on learning a *biasing automaton*, which inspects the current execution and provides the distribution on the different choices for extending the tested execution. In this paper, we study the problem of learning effective biasing automata using genetic programming.

We are interested in constructing biasing automata that will improve the choice of tested executions for finding an error in a family of programs. Thus, we expect an effective biasing automata to be useful in the process of regression testing. The use of a collection of candidate programs, within genetic programming, helps to discover inherent structure that is important for identifying errors. We embed the learning and use of biasing automata in the process of genetic synthesis of correct-by-design code. We apply a *coevolution* process [12], where both solutions and biasing automata are constructed simultaneously, helping to improve each other.

A related technique for handling rare events, in the context of statistical model checking, is *importance splitting* [8], one uses a splitting of the test sequences into cases. Then one can zoom into checking cases where the rare events are believed to appear more frequently. The statistical result for each such case is the *conditional probability* of the rare event to occur under that case. The distribution of the different cases are assumed to be known, and the statistical results of the different cases are multiplied by the distribution of selecting the different cases in order to normalize the overall statistical result. Biasing automata can also be seen as splitting the test cases (according to their prefixes) for concentrating more on some cases. However, it is meaningless to normalize the experiment results in order to obtain a statistical measurement on the relative occurrence rate of the rare event. In a sense, a biasing automaton can be considered to be an approximation to a fault model [15], which helps detecting erroneous executions on future versions of the tested system.

2 Preliminaries

2.1 Labeled Markov Chains

A *labeled Markov Chain* is a stochastic model, describing sequences of events, in which the probability of choosing the next event depends only on the state reached so far after the previous event. We describe a labeled Markov Chain as a tuple $\mathcal{M} = (Q, q_0, A, B, T, D, L)$ where

- Q is a finite set of *states*, with $q_0 \in Q$ the initial state.
- A is a finite set of *actions*.
- B is a finite set of *propositions*.
- $T : Q \times A \mapsto Q$ is the *transition function*.
- $D : Q \times A \mapsto [0, 1]$ is the *distribution* on selecting the next action from the current state, where $\Sigma_{a \in A} P(q, a) = 1$.
- $L : Q \mapsto 2^B$ is a labeling function.

An *execution* of a Markov Chain is a finite alternating sequence of states and actions $\sigma = q^0c^1q^1c^2 \dots c^nq^n$, where $q^0 = q_0$, and $T(q^{i-1}, c^i) = q^i$, the length $|\sigma|$ of the execution is n. From each state q^{i-1} in the execution, an action c^i is selected with probability $D(q^{i-1}, c^i) > 0$ (we say that c^i is *enabled* from q^i); then the state is changed to $q^i = T(q^{i-1}, c_i)$. The probability of selecting the particular execution $\sigma = q^0c^1q^1c^2 \dots c^nq^n$ is $p(\sigma) = D(q^0, c^1) \times D(q^1, c^2) \times \dots \times D(q^{n-1}, c^n)$.

2.2 Testing Experiments

Our testing process consists of generating random walks on a Markov Chain and checking them against a given specification. A *property* φ is a set of finite sequences over 2^B. An execution $\sigma = q^0c^1q^1c^2 \dots c^nq^n$ *satisfies* the property φ if $L(\sigma) = L(q^0)L(q^1) \dots L(q^n) \in \varphi$. That is, the sequence of labels of the execution σ is in the set φ. We also write in this case that $\sigma \models \varphi$. The formalism used for specifying the property φ needs to be effective for checking whether a given sequence satisfies it or not. We use biasing automata only for checking violation of "safety properties" [1], where violations can be detected on finite executions that cannot be completed anymore into executions that satisfy the specification.

A *testing experiment* is an execution (random walk) σ of $\mathcal{M}$, limited to some predefined number n of actions, and terminating as soon as it first violates the checked specification φ.

2.3 Genetic Programming

During the 1970s, Holland [5] established the field known as *Genetic Algorithms* (GA). Individual candidate solutions are represented as fixed length strings of bits, corresponding to chromosomes in biological systems. Candidates are evaluated using a *fitness* function; It approximates the distance of the candidate from a desired solution. Genetic algorithms evolves a *set of candidates* into a successor set. Each such set forms a *generation*, and there is no backtracking. Candidates are usually represented as fixed length strings. They progress from one generation to the next one according to one of the following cases:

Reproduction. Part of the candidates are selected to propagate from one generation to the subsequent one. The reproduction is done at random, with probability relative to the relation between the fitness of the individual candidate and the average of fitness values in the current generation.

Crossover. Some pairs of the candidates, selected at random for reproduction, are combined using the crossover operation. This operation takes parts of bit strings from two parent solutions and combines them into new solutions, which potentially inherit useful attributes from their parents.

Mutation. This operation randomly alters the content of small number of bits from candidates selected for reproduction (this can also be done after performing crossover). One can decide on mutating each bit separately with some probability.

The different candidates in a single generation have a combined effect on the search; progress tends to promote, improve and combine candidates that are better than others in the same generation. The process of selecting candidates from the previous generation and deciding whether to apply crossover or mutation continues until we complete a new generation. All generations are of some predefined fixed size N. This can be, typically, a number between 50 and 500. Genetic algorithms perform the following steps:

1. Randomly generate N initial candidates.
2. Evaluate the fitness of the candidates.
3. If a satisfactory solution is found, or the number of generations created exceeds a predefined limit (say hundreds or a few thousands), terminate.
4. Otherwise, select candidates for reproduction using randomization, proportional to the fitness values and apply crossover or mutation on some of them, again using randomization, until N candidates are obtained.
5. Go to step 2.

If the algorithm does not terminate with a satisfying solution after a predefined limit on the number of generations, we can restart it with a new random seed, or change the way that we calculate the fitness function.

Genetic programming, suggested by Koza [12], is a direct successor of genetic algorithms. Each individual organism represents a computer program. Programs are represented by variable length structures, such as syntax trees or a sequences of instructions. Each node is classified as *code*, *Boolean*, *condition* or *expression*. Leaf nodes are variables or constants, and other nodes have successors according to their type. For example, a while node (of type *code*) has one successor of type *Boolean* and one successor of type *code* (for the loop body); the *Boolean* node "and" has two successors that can be of type *Boolean* or *condition*, and a *condition* node "<" has two successors of type *expression*. The genetic operations need to respect typing restrictions, e.g., *expressions* cannot be exchanged with *Booleans*.

Crossover is performed on a pair of trees by selecting a subtree rooted with the same node type in each tree, and then swapping between them. This results in two new programs, each having parts from both of its parents. There are several kinds of mutation transformations on syntax trees. First, a node, which roots a subtree, is selected at random. Then, one of the following mutations is performed:

Replacement. Throw away the selected subtree and replace it with a randomly generated subtree of the same type.

Insertion. Generate a new node of the same type as the selected subtree and insert it as its parent node. Then complete the other descendants of the newly inserted node, if necessary.

Reduction. The selected node is replaced with one of its descendants, and the rest of the descendant are deleted.

Deletion. Delete the selected subtree and update its ancestor nodes recursively.

The fitness is often calculated in GP by running the candidate programs on a large set of test cases and evaluating the results, but also using model checking [9–11].

3 Biasing Automata

A biasing automaton is a tool for controlling the probabilities of random walks. It observes the sequence of states of the execution selected so far, and provides the probability distribution for selecting the next action that extends the execution. Based on its current state and the (labeling on the) last observed state of the random walk, the biasing automaton can change its state and provide a different distribution.

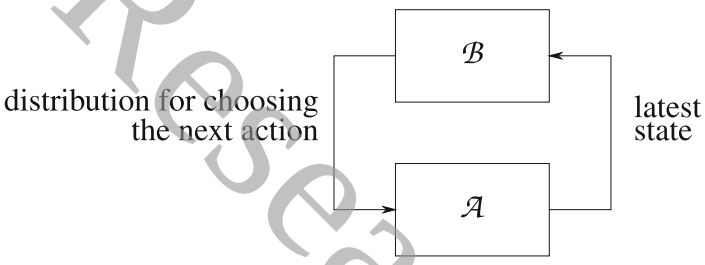

Fig. 1. Controlling random walks using a biasing automaton

A model automaton represents the state space and transition relation of the system under test (SUT). The goal of a biasing automaton is to provide probabilities for selecting the tested execution sequences, in a way that increases the chance of finding violations of the specification that appear among the executions of the model automaton with low probability. A biasing automaton provides, in our context, the probabilities for making the individual selections between possible actions during the random walk. Figure 1 describes the combination of an automaton $\mathcal{A}$, representing the state space for the random walks, and a biasing automaton $\mathcal{B}$ that *controls* the probabilities of selecting the next action of $\mathcal{A}$. The combination of $\mathcal{A}$ and $\mathcal{B}$ forms a Markov Chain.

A *model automaton* $\mathcal{A} = \{S, s_0, A, B, \Delta, M\}$ represents the system under test, where transitions are marked by actions, and states by propositions from a finite set B. It is defined as follows:

- S is a finite set of *states*, with $s_0 \in S$ the *initial state*.
- A is a finite set of actions.
- B is a a set of Booelan *propositions*.
- $\Delta : S \times A \mapsto S$ is the *transition relation*.
- $M : S \mapsto 2^B$ is the *labeling* on the states.

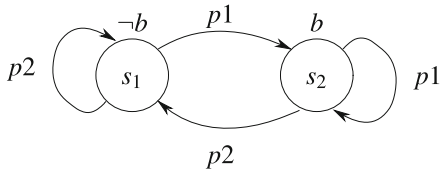

Fig. 2. A model automaton with initial state s_1

The actions of the model automaton in Fig. 2 are $p1$ and $p2$ (corresponding in our context to process names). There is a single Boolean variable b, hence $B = \{b\}$ and the nodes of the model automaton are marked either by b (corresponding to the propositions $\{b\}$ holding in the state), or $\neg b$ (corresponding to $\emptyset$).

A biasing automaton $\mathcal{B} = \{G, g_0, A, B, \Gamma, \mathcal{D}, o\}$ for a family of model automata with a set of actions A and a set of propositions B is a finite automaton with states labeled by a probability distribution between the actions of $\mathcal{A}$ as the output of the automaton (hence, it is a Moore machine). The input to the automaton is a set of predicates from B. Based on the input, the biasing automaton will move to a new state and consequently change the distribution output.

- G is a finite set of *states* with $g_0 \in G$ the initial state.
- A is a set of *actions*.
- B is a finite set of *propositions*.
- $\Gamma : G \times 2^B \mapsto G$ is the *transition function*.
- $\mathcal{D}$ is the space of a distributions over A (i.e., each $D \in \mathcal{D}$ is a *distribution function* $D : A \mapsto [0, 1]$).
- $o : G \mapsto \mathcal{D}$ is the *output function* on states. It returns a distribution function on the set of actions A.

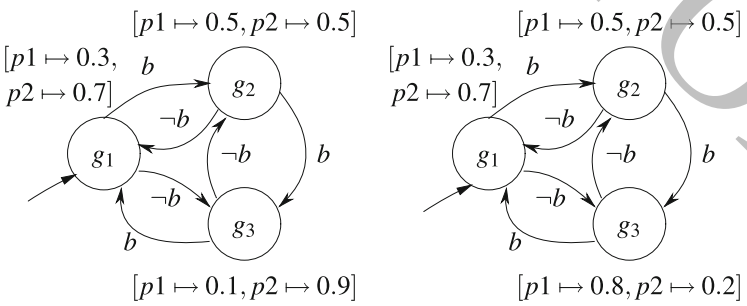

Fig. 3. Two variants of biasing automata

In Fig. 3 we have an example of two variants of biasing automata for the model in Fig. 2. The biasing automata in Fig. 3 differ from one another in the probabilities in state g_3.

The *convolution* of a model automaton $\mathcal{A}$ and a biasing automaton $\mathcal{B}$, denoted $\mathcal{A}[\mathcal{B}]$ over a common set of actions A and a set of Boolean propositions B, is a synchronization of both automata. The probability distribution on actions from the same state is given by the output on the state of the biasing automaton. The convolution produces a Markov chain, obtained as follows:

- The set of *states* is $S \times G$. The *initial state* is (s_0, g_0).
- A is the set of *actions*.
- B is the set of *propositions*.
- The *transition* function $Tr : (S \times G \times A) \mapsto (S \times G)$, where $Tr : ((s, g), a) = (s', g')$ if $\Delta(s, a) = s'$, $\Gamma(g, M(s')) = g'$.
- $D((s, g), a) = o(g)(a)$.
- $L(s, g) = M(s)$.

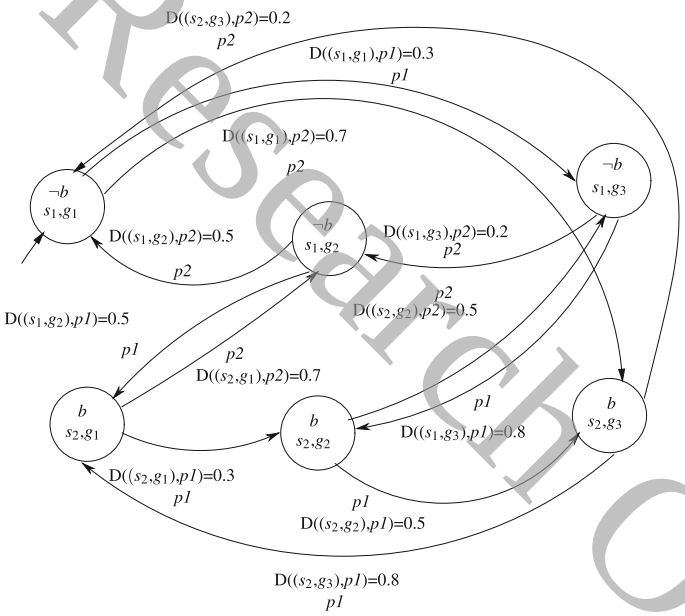

Fig. 4. The convolution automaton

The convolution automaton $\mathcal{A}[\mathcal{B}]$ is a labeled Markov Chain. The automaton in Fig. 4 is the convolution of $\mathcal{A}$, which appears in Fig. 2, and $\mathcal{B}$,which appears on the righthand side of Fig. 3.

4 Obtaining Biasing Automata Through Genetic Programming

Constructing a biasing automaton that improves the chance of finding executions that manifest an event with low probability (e.g., the event that consists

of the executions that violate the specification) involves a tradeoff between optimality and time complexity. One can perform a complete analysis (based on model checking) of the executions of the system under test (SUT), and check weather there are executions that demonstrate the problem we want to trace. If such an execution exists, we can construct a biasing automaton that allows this execution exclusively. However, the complexity of the complete analysis is high, and this would defy using random testing as an alternative method for the complete analysis. Moreover, changes to the SUT, e.g., implementing a correction that would eliminate this erroneous execution, and the success rate in finding erroneous executions using this biasing automaton will be reduced to 0%. In fact, we are interested in constructing biasing automata that are robust to such changes, and applicable to a family of related SUTs. The process we propose here for obtaining a useful biasing automaton is thus based on experiments with different versions of the SUT. We use genetic programming both to mutate the SUTs tested, and also to construct biasing automata.

In our experiments, we use the genetic process to synthesize solutions for concurrency control problems. This is done during the construction of the biasing automata, where both kinds of objects: solutions for concurrency control and biasing automata are used to improve the population of each other in a process called *coevolution*. In genetic programming, coevolution is a method to solve a large problem by refining it into numerous smaller problems, letting them interact for the sake of cooperative evaluation. This is inspired by biology, where two or more species affect each others evolution reciprocally. The populations challenge each other and compete with each other, thus making each other improve as the generations evolve. Together, both the programs and the automata evolve and as generations progress, improving both candidate programs and biasing automata.

4.1 Fitness Functions

In the coevolution process, the fitness of the candidate programs in each generation is calculated using random testing, guided by the biasing automata. We select a fixed number K for biasing automata that will help us to perform the random testing. In each generation, the K biasing automata with highest fitness are used to guide the testing. For each random walk we first randomly select which one of these automata will guide it. The biasing automata observe the predicates on the states of the current random walk, and provide the distribution for selecting the next process to extend it. To express the fitness for the biasing automata we define the following:

- G_1 is the set of all programs in the current generation.
- G_2 is the set of all biasing automata in the current generation.
- $s(\mathcal{A}[\mathcal{B}], \varphi, n, N)$ is the number of times we found a violation of φ when running N executions of the program represented by automaton $\mathcal{A}$ according to bias automaton B with at most n actions per execution. This is an experiment that could yield different results each time.
  We have that when $n, N \rightsquigarrow \infty$, then $s(\mathcal{A}[\mathcal{B}], \varphi, n, N) \rightsquigarrow p(\mathcal{A}[\mathcal{B}], \varphi)$.

- $minScore = \min_{\mathcal{B} \in G_2} \Sigma_{\mathcal{A} \in G_1} s(\mathcal{A}[\mathcal{B}], \varphi, n, N)$ as the minimum fitness score on all biasing automata in the generation.
- $maxScore = \max_{\mathcal{B} \in G_2} \Sigma_{\mathcal{A} \in G_1} s(\mathcal{A}[\mathcal{B}], \varphi, n, N)$ as the maximum fitness score on all biasing automata in the generation.

We use N=300 and n=150. The fitness function is defined as:

$$fitness(\mathcal{B}) = \frac{\Sigma_{\mathcal{A} \in G_1} s(\mathcal{A}[\mathcal{B}], \varphi, n, N) - minScore}{maxScore - minScore}$$

The fitness of the candidate programs are based on the randomized testing (using the co-generated biasing automata) of their correctness properties. Because the testing is based on inspecting limited length sequences, and due to the sampling nature of testing, we also make light use of model checking for providing fitness for the candidate programs: if a candidate is detected with a very high fitness value, which is above a certain threshold that we define, we run the SPIN model checker [6] on this candidate to check whether indeed it satisfies the desired properties. For more details on the fitness function used on candidate programs see [3].

4.2 Mutations

For the biasing automata, we define the following types of mutations:

Change a transition. Choose randomly a state to mutate and redirect one of its outgoing edges.

Change probabilities. Choose a state and assign new distribution on choosing the actions from it.

Add a state. Generate a new state and connect it to the other states in the automaton graph.

Delete a state. Choose randomly a state, delete it from the automaton and assign a random target state for each edge that previously led to it.

Sub-automaton. Create new sub-automaton. Choose one of the states and delete all states with index larger than it. Grow a new automaton with some number of states and merge it to the remaining states of the original automaton.

As an example, consider the automaton in Fig. 5. The labeling we use is over the set of propositions $\{p1\ in\ CS_1, p2\ in\ CS_2, \}$. Each edge is labeled with a subset of these propositions, where the lack of a label, e.g., $p2\ in\ CS_2$ is denoted by $\neg p2\ in\ CS_2$. We compact the presentation, where several edges have the same source and target, by depicting a single edge, marked by a Boolean formula that is equivalent to the disjunction of the formulas on the edges.

After a *change transitions* mutation was performed, and state q_2 is chosen as the mutation point, edges coming out of state q_2 were randomly changed. Specifically, the depicted self edge from q_2 to itself represents two edges: one labeled with $\neg(p1\ in\ CS_1) \wedge p2\ in\ CS_2$ and the other with $\neg(p1 in\ CS_1) \wedge \neg(p2\ in\ CS_2)$. The former edged was replaced with an edge from q_2 to q_0. We thus obtained from 5 the automaton in Fig. 6.

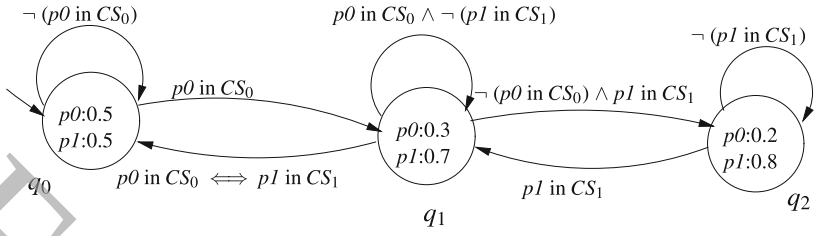

Fig. 5. Biasing automaton before mutation

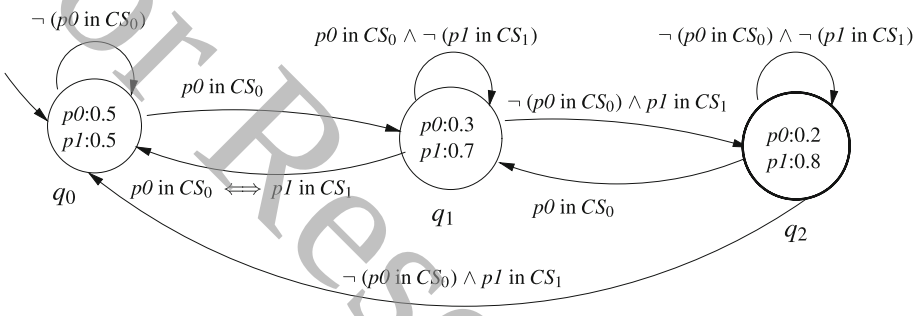

Fig. 6. Biasing automaton after a change transition mutation

Extrapolating Mutating Probabilities. The mutation and crossover operations of genetic programming allow us to jump from one candidate solution to another, while preserving part of the structure of the former candidate. Two other search heuristics allow us to make finer changes around the candidates, in order to search for local or global optima. In *simulated annealing* [16], one controls the amount of change allowed between candidates; the probability of making big changes decreases when the fitness increases. Accordingly, a small change tends to search around the candidates that look more promising, while a big change tends to avoid hill climbing. In *gradient descent*, one uses exterpolation of a multivariable function, based on the direction of its gradient (multivariable derivative), to assist in progressing to a better solution.

Inspired by these techniques, we refine the genetic programming search. In particular, we want to make the "change-probabilities" mutation more sensitive to the results of the testing experiments. $s_1, \ldots, s_n$. Define a function

$$f(\mathcal{B}, \mathcal{A}, D(s_1, p1), \ldots, D(s_n, p1)) \quad (1)$$

that returns the probability of random walks over a model automaton $\mathcal{A}$, performed according to the distributions on the nodes of $\mathcal{B}$, to satisfy the checked property φ. We assume a fixed structure for the biasing automata that are allowed for the parameter $\mathcal{B}$, in particular, they all have n states. These automata

can differ only in the probability distributions attached to the states[1], which appear explicitly as parameters to this function. Since our goal is to increase the chance of observing violations of φ, we would like to select these distributions in a way that will minimize the value of that function over the family of models $\mathcal{A}$ that we consider.

It is of course pointless to try to calculate the actual value of the function f. But we can try to approximate its behavior using experiments. In addition to the above mutations, we allow selecting to change the distribution parameters using extrapolation approximation that is based on such experiments. Consider the case where there are two automata $\mathcal{B}$, and $\mathcal{B}'$, where $\mathcal{B}'$ is obtained from $\mathcal{B}$ using change probability mutation. Suppose, without loss of generality, that $\mathcal{B}'$ returns better results, i.e., more violations, when applied to various model automata that we consider. Then we can change the distributions proportional to the change between the distributions of $\mathcal{B}$ and $\mathcal{B}'$.

5 Implementation and Experiments

In [3] we presented the use of genetic programming, based on statistical model checking (SMC) and model checking [4], for synthesizing concurrent code from temporal specification. One of the goals was to replace most of the use of model checking in [9–11] by lighter reliability methods. However, we quickly realized that using the random-based sampling approaches is very sensitive to the occurrence of events with low probability, which may be missed. This required several adaptations, e.g., bringing back the use of model checking, even if in a very limited way, at the last stages of the genetic synthesis. The observations made in that work motivated the definition and use of biasing automata in the current paper, in order to better control the statistical evaluation during the genetic process. We provide here experimental results based on integrating the software reported in [3] with an implementation of a genetic coevolution process for constructing and using biasing automata.

5.1 Running Example: Mutual Exclusion

We take the synthesis of solutions for the mutual exclusion problem as a running example. Following is the template that needs to be concretized.

$p0$: While W0 do	$p1$: While W1 do
NonCrit0	NonCrit1
preCS0	preCS1
CS0	CS1
postCS0	postCS1
end while	end while

[1] For the case of two processes, $D(s_i, p2) = 1 - D(s_i, p1)$, hence this is a unique function of the distributions. In case of k choices, we need to use $k - 1$ parameters for each state.

NonCSi represents the actions of process *pi* outside the critical section. It can actually be fixed as empty code. CSi represents the critical section, which both processes want to enter a finite or unbounded number of times. It is not part of the synthesis task, and can be represented by trivial code (which serves only to allow checking that it is eventually entered upon request). The goal of the mutual exclusion problem is to allow eventual access to the critical section each time a process wants to enter it, but to disallow both processes to enter the critical section at the same time. Entering and exiting the critical section is controlled by the code in preCSi and postCSi. These are the program segments that consist of the mutual exclusion protocol and are the focus of the synthesis.

The properties we want to satisfy here are:

- *Safety* :$\Box\neg(p0\ in\ CS_0 \wedge p1\ in\ CS_1)$
- *Liveness* :$\Box(pi\ in\ preCS_i \rightarrow \Diamond pi\ in\ CS_i)$

5.2 Experiment Setting

Our program is written in C and runs in Linux. We run the GP procedure for at most 2000 generations, with 100 candidates in each generation. For the fitness evaluation, we randomly simulate each candidate 300 times per generation. Based on coevolution, we also generate candidate biasing automata. Each generation includes 5–20 biasing automata and each biasing automaton is allowed to have at most 50 states.

Mutation is performed on 40% randomly chosen candidates among the one that are selected to propagate to the next generation. The distribution on the different mutations is according to the Table 1, where the *create sub-automaton* mutation is selected with a very low probability, due to its rather extensive effect.

Table 1. Distribution of types of mutations

	Mutation rate
Change transitions	24%
Change probabilities	24%
Add state	24%
Remove state	24%
Create sub-automaton	4%

5.3 Biasing Automata Learned

The automaton in Fig. 7 is one of the biasing automaton that the system learned. State q_1 corresponds to process $p0$ being in the critical section, while $p1$ is not

in its critical section. Then, higher probability is given to process $p1$ to progress. This biasing automaton helps capturing both processes at their critical section, violating the safety property; it reduces the probability to generate an execution sequence where $p0$ enters its critical section, immediately and independently of $p1$ leaving the critical section, and only then $p1$ enters its critical section. Intuitively, it encourages $p0$, when in its critical section, to wait and give chance for $p1$ to also enter its critical section, rather than leaving immediately.

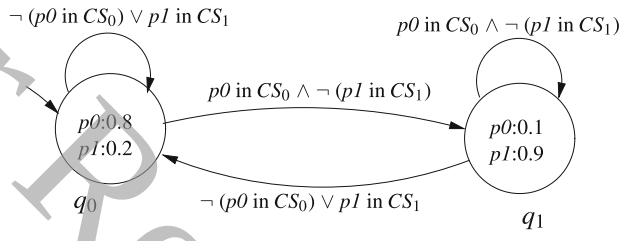

Fig. 7. A biasing automaton generated during the GP synthesis of mutual exclusion

5.4 Performance Evaluation

We conducted experiments to check whether the integration of biasing automata can help with detecting errors. We picked up a biasing automaton that was generated through a coevolution process. Then we randomly selected 20 candidates for solution for the mutual exclusion, also generated as part of the genetic process, and run each of them 300 times with both random simulation and biasing automata guided simulation respectively. With the help of biasing automata we locate 348 executions that triggered error states, while we only found 120 erroneous executions by using random simulation.

Table 2. Applying a single biasing automaton in the GP synthesis procedure

No. of biasing aut. per generation	Success rate	Average iterations until convergence	Average time per execution (minutes)
Without biasing aut.	20%	1930	20
5	28%	1890	31
10	32%	1700	40
15	31%	1720	61
20	30%	1731	83

We next evaluated whether using biasing automata can help, when integrated into the GP procedure, in generating correct solutions with greater success rate and less iterations. In this experiment, we select the best biasing automaton in each generations We change the number of biasing automata candidates generated in each generation to guide the testing. The data is reported in Table 2. From the table we can observe that the best performance, in terms of the success rate of finding correct solutions, was with 10 biasing automata per generation. Not surprisingly, the overhead of learning more biasing automata per generation affects the execution time. Hence, we selected for further experiments using 10 biasing automata.

As different biasing automata may address different aspects of errors, we also experimented with using multiple biasing automata instead of just the best one to guide the simulation. In this case, for each random execution we first selected (randomly) the biasing automata that will guide it among the k best biasing automata in the generation. The results of this experiment are shown in Table 3. We can observe from the table that the best success rate was with using 3 automata. The declined success above 3 automata can possibly be explained by adding lower fitness biasing automata to guide the random selection.

Table 3. Applying multiple biasing automata in the simulation of mutual exclusion

k: No. of best biasing automata used per generation	Fitness value of the kth automata in the last generation	Success rate	Average iterations until convergence
1	1	32%	1700
2	0.95	34%	1693
3	0.93	40%	1640
4	0.81	36%	1712
5	0.73	33%	1730
6	0.69	32.5%	1746
7	0.67	31.9%	1751

We performed further experiments, e.g., synthesizing solutions for the dinning philosophers problem, where we found also an improvement in the number of errors that could be found, when we introduced a biasing automaton to guide the random testing. However, the improvement of the success rate of the genetic search on that example was more modest than for the mutual exclusion problem, only up to 5%.

6 Conclusions

Random simulations are widely used in software testing and verification. However, the low probability ("rare event") errors which appear with very low probability may be elusive to discover. This may cause incorrect assessment of the

system behavior and quality. In this paper, we presented biasing automata, which helps the randomized selection of test cases in a way that is leaned towards the sought errors.

We implemented this idea as part of a genetic programming synthesis system for concurrent code from temporal specification. Our experiments show that with the help of biasing automata, we can locate more errors during random testing. In addition, the success rate of the synthesis process was significantly improved. The experiments lead us to believe that the approach presented here is effective for enhancing the results of random testing. Of course, further experiments, and additional work on the implementation are called for.

The embedding of the generation of a biasing automaton in a genetic coevolution, as reported in this paper, was a matter of convenience, as we already experimented with the genetic synthesis separately. In addition, it turned out to improve the success of generating correct-by-design solutions. However, biasing automata can also be generated by mutating the already available system under test. In that sense, the process is related to "mutation testing" [2]. In mutation testing, one checks if a collected set of test suite is strong enough to cover the testing of a program. Here, mutation can used to find a biasing automaton that would improve the guidance for random testing, in order to increase the chance of finding errors.

Acknowledgements. The second author would like to thank Sergiy Bogomolov and Ken McMillan for interesting discussions on this subject.

References

1. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2008)
3. Bu, L., Peled, D., Shen, D., Zhuang, Y.: Genetic synthesis of concurrent code using mode checking and statistical model checking. In: SPIN 2018, pp. 275–291 (2018)
4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
5. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge (1992)
6. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, New Jersey (2004)
7. Grosu, R., Smolka, S.A.: Monte Carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_18
8. Jegourel, C., Legay, A., Sedwards, S.: An effective heuristic for adaptive importance splitting in statistical model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8803, pp. 143–159. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45231-8_11

9. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71605-1_11
10. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 141–156. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_11
11. Gal Katz, D.: Peled: Synthesizing, correcting and improving code, using model checking-based genetic programming. STTT **19**(4), 449–464 (2017)
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
13. Larsen, K.G., Legay, A.: On the power of statistical model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 843–862. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_62
14. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
15. Pretschner, A., Holling, D., Eschbach, R., Gemmar, M.: A generic fault model for quality assurance. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 87–103. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41533-3_6
16. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
17. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_17