Exploring Very Large State Spaces Using Genetic Algorithms

Patrice Godefroid[1] and Sarfraz Khurshid[2]

[1] Bell Laboratories, Lucent Technologies, god@bell-labs.com
[2] Laboratory for Computer Science, Massachusetts Institute of Technology, khurshid@lcs.mit.edu

Abstract. We present a novel framework for exploring very large state spaces of concurrent reactive systems. Our framework exploits application-independent heuristics using genetic algorithms to guide a state-space search towards error states. We have implemented this framework in conjunction with VeriSoft, a tool for exploring the state spaces of software applications composed of several concurrent processes executing arbitrary code. We present experimental results obtained with several examples of programs, including a C implementation of a public-key authentication protocol. We discuss heuristics and properties of state spaces that help a genetic search detect deadlocks and assertion violations. For finding errors in very large state spaces, our experiments show that a genetic search using simple heuristics can significantly outperform random and systematic searches.

1 Introduction

Model checking [4] is an automatic technique for verifying finite-state concurrent systems. The *state space* of a concurrent system is a directed graph that represents the combined behavior of all the concurrent components in the system. Model checking typically involves exhaustively searching the state space of a system to determine whether some property of the system is satisfied or not. State-space exploration techniques have been used successfully to detect subtle yet important errors in the design and implementation of several complex hardware and software concurrent reactive systems (e.g., see [1, 3, 9, 19]). It is worth noting that the main practical interest of systematic state-space exploration (and of "verification" in general) is to find errors that would be hard to detect and reproduce otherwise.

The main practical limitation when model checking real systems is dealing with the so-called *state-explosion* problem: the number of states contained in the state space of large complex systems can be huge, even infinite, thereby making exhaustive state-space exploration intractable. Several approaches have been proposed to address the state-explosion problem, including symbolic verification, partial-order methods and symmetry methods. Although these approaches have increased the scope of model checking to state spaces that are several orders of magnitude larger, many realistic state spaces are still too large to be handled, and state explosion remains a fundamental problem in model checking.

When a problem is computationally too hard to solve using an exact and complete algorithm, it is common in computer science to explore the use of heuristics in order to find approximate solutions to the problem, or to converge faster towards some solutions. Maybe surprisingly, the idea of exploiting heuristics for model checking has received very little attention so far. This may be due to two reasons. First, model checking is not an optimization problem: the primary goal is not to find a best solution (e.g., the shortest path leading to some state), it is to find any solution (e.g., any reachable error state). Second, the historic emphasis in model checking has been on completeness: the primary goal is to exhaustively check every reachable state of the system.

In this paper, we explore the use of *genetic algorithms* [11] for exploring very large state spaces in search for error states. Genetic algorithms are search algorithms inspired by the mechanics of genetics and natural selection. These search algorithms combine survival of the fittest among chromosome-like string structures with a structured yet randomized information exchange. Genetic algorithms are often viewed as function optimizers, although the range of problems they have been applied to is quite broad [18].

We present a framework that uses genetic algorithms to exploit heuristics for guiding a search in the state space of a concurrent reactive system towards errors like deadlocks and assertion violations. At each visited state during a state-space exploration, the genetic algorithm decides which transition to explore next when there are more than one enabled outgoing transitions. We have implemented this framework in conjunction with VeriSoft [8], an existing tool for exploring the state spaces of systems composed of several concurrent software processes executing arbitrary code. We present experimental results obtained with several examples of programs, including a C implementation of a public-key authentication protocol. From these experiments, we discuss general properties of state spaces that seem to help a genetic search find errors quickly. When the state space to be explored is very large, our experiments show that a genetic search using simple application-independent heuristics can significantly outperform random and systematic searches.

The rest of the paper is organized as follows. In Section 2, we recall the basic principles of genetic algorithms. Section 3 describes our framework and the genetic algorithms we use. We discuss how to modify a model checker to let its search be guided by a genetic algorithm. In Section 4, we describe several programs and properties we have analyzed using our implementation. We then discuss results of experiments, and study the influence of various parameters on the effectiveness of a genetic search. In Section 5 we present concluding remarks and discuss related work.

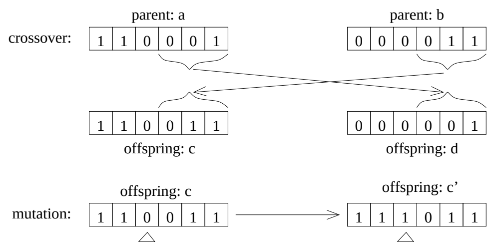

Fig. 1. Illustration of crossover and mutation operators. Candidate solutions are encoded as strings of bits. Parents a and b are recombined to produce offsprings c and d: a crossover is performed at the 4th bit, i.e., the tails of both parents are swapped starting from the 4th bit. Offspring c is then mutated to produce c': a mutation is performed at the 3rd bit, i.e., the value of the 3rd bit is flipped.

2 Genetic Algorithms

A *genetic algorithm* provides an algorithmic framework for exploiting heuristics that simulates natural-evolution processes like selection and mutation. It evolves candidate solutions to problems that have large solution spaces and are not amenable to exhaustive search or traditional optimization techniques. Genetic algorithms have been applied to a broad range of learning and optimization problems [18] since their inception by Holland [11].

Typically, a genetic algorithm starts with a random population of encoded candidate solutions, called *chromosomes*. Through a *recombination* process and *mutation* operators, it evolves the population towards an optimal solution. Generating an optimal solution is not guaranteed and the challenge is thus to design a "genetic" process that maximizes the likelihood of generating such a solution. The first step is typically to evaluate the *fitness* of each candidate solution in the current population, and to select the fittest candidate solutions to act as parents of the next generation of candidate solutions. After being selected for reproduction, parents are recombined (using a *crossover* operator) and mutated (using a *mutation* operator) to generate offsprings (see Figure 1 for a description of these operators). The fittest parents and the new offsprings form a new population, from which the process is repeated to create new populations. Figure 2 gives a standard genetic algorithm in pseudocode.

To illustrate an iteration of a genetic algorithm, consider the boolean satisfiability problem. Assume that we want to find a satisfying assignment to the following boolean formula: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee x_5 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_5 \vee \neg x_6) \wedge (x_3 \vee \neg x_5 \vee \neg x_6) \wedge (x_3 \vee x_4 \vee x_5)$. Let's say we have two (randomly generated) candidate solutions, $a:\ \{x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 1\}$ and $b:\ \{x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 1, x_6 = 1\}$. If we evaluate the formula on a, we see that clauses 3 and 6 are false, whereas evaluating the formula on b makes clauses 2 and 5 false. Both a and b are not satisfying assignments. We now recombine a and b to produce an offspring $c:\ \{x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1, x_6 = 1\}$, which takes the first three variable assignments from a and the last three from b. Offspring c does not define a satisfying assignment

```
gen := 0;
P[gen] := random population;
fitness[gen] := evaluate(P[gen]);
while (fitness[gen] < T) {        // fitness has not reached desired level
  gen++;
  S[gen] := select(P[gen-1]);     // select fittest chromosomes
  CM[gen] := crossover(S[gen]);   // perform crossover on pairs
  CM[gen] := mutate(CM[gen]);     // mutate resulting chromosomes
  P[gen] := S[gen] + CM[gen];     // produce next generation
  fitness[gen] := evaluate(P[gen]); }
```

Fig. 2. Pseudocode for a standard genetic algorithm

either since it makes clause 5 false. However, if we mutate the value c assigns to x_3 to produce $d: \{x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 1\}$, we see that d does provide a satisfying assignment to our boolean formula.

The operations of evaluation, selection, recombination and mutation are usually performed many times in a genetic algorithm. Selection, recombination, and mutation are generic operations in any genetic algorithm and have been thoroughly investigated in the literature. On the other hand, evaluation is problem specific and relates directly to the structure of the solutions (i.e., how candidate solutions are encoded as chromosomes and relate to each other). Therefore, in a genetic algorithm, a major issue is the choice of the structure of solutions and of the method of evaluation (fitness function). Other parameters include the size of the population, the portion of the population taking part in recombination, and the mutation rate. The mutation rate defines the probability with which a bit is flipped in a chromosome that is produced by a crossover.

3 Genetic Algorithms for State-Space Exploration

In this section, we discuss how genetic algorithms can be used to guide a search in the state space of a concurrent reactive system.

3.1 Combining Genetic Algorithms and Model Checking

In our context, the search space to be explored is the (possibly inifinite) state space of the system. For simplicity and without loss of generality, we assume that the state space has a unique initial state. Candidate solutions are finite sequences of transitions in the state space starting from the initial state. Each candidate solution is encoded by a chromosome, i.e., a finite string of bits. Figure 3 shows a simple example of encoding. How to encode finite paths in a graph using chromosomes is discussed in details below.

To evaluate the fitness of a chromosome, the genetic algorithm *executes* the path encoded by the chromosome. This is done by combining the genetic algorithm with a model checker. Given a representation of a system, the model checker determines the state space to explore. The execution of a path starts in

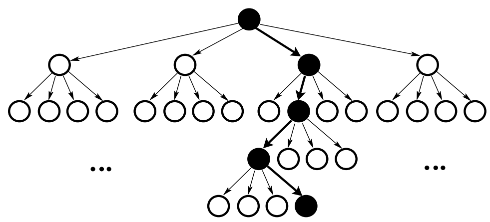

Fig. 3. Example encoding. Assume a state space with fixed branching (of 4) and fixed depth (of 4); 8 bits are used to represent a chromosome. The chromosome '10 01 00 11' encodes the path that visits the filled states (following the bold edges) in state space.

the initial state. If there are more than one possible transitions from the current state, the model checker informs the genetic algorithm about the number of possibilities. The genetic algorithm decodes a part of the chromosome it is currently processing and informs the model checker of which transition to take. The model checker then checks whether the state following that transition is an error state. If so, the current path is saved and the user is notified. Otherwise, the model checker repeats this process from the new state.

Since a chromosome can only encode a finite number of transitions, the state space is explored up to a fixed depth. Whenever the model checker has explored a path up to this maximum depth, it prompts the genetic algorithm to evaluate the fitness of the current chromosome. This operation is discussed further below. Once the fitness of the current chromosome has been computed, another chromosome of the current population is evaluated using the same process.

3.2 Genetic Encoding

We now discuss a novel chromosome-encoding scheme that can be applied to arbitrary state spaces. Indeed, the simple encoding technique described in Figure 3 is not satisfactory for several reasons.

First, the number of enabled transitions in a state is typically not constant. Moreover, an upper bound on the number of enabled transitions in a state may not be known a priori[1]. Therefore, a practical encoding cannot use a fixed number of bits to encode a single transition. We resolve this issue by *dynamically* interpreting a chromosome: if there are n enabled transitions from the current state being processed during the state-space search, we read the next $log(n)$ bits from the current chromosome to decide which next transition to explore.

Second, the number of enabled transitions in a state is not necessarily a power of 2. This means that we may have to deal with *spurious* encodings: encodings that fall outside the desired interval of values. The traditional approach [10] to deal with this issue is to map the decoded integer linearly into the desired interval. This approach, however, typically introduces bias toward some values in the desired interval. Therefore, we deal with spurious encodings by *updating*

[1] We assume that the number of enabled transitions in any given state is finite, and hence that such a bound exists.

such chromosomes instead: if there are n enabled transitions from the current state and the next $log(n)$ bits of the current chromosome decode to a value greater than or equal to n, we randomly generate a number between 0 and $n-1$ and replace the last $log(n)$ bits read of the chromosome by the binary encoding of this number. Note that our procedure for updating chromosome bits in this case is necessary to avoid multiple fitness evaluations of the same chromosome to evaluate to different values.

Third, a suitable length (i.e., number of bits) for chromosomes cannot be determined in advance. Since a chromosome can only encode a finite number of transitions, the model checker only explores paths up to a fixed depth. For a maximum depth d, we use sufficiently long chromosomes so that they can encode any path of length up to d, and we track the *effective* length of chromosomes. The effective length at any point during a genetic evolution is the maximum number of bits that have been read from any single chromosome up to that point in the search. Mutations and crossovers are performed only on initial segments of chromosomes up to the (current) effective length.

3.3 Fitness Function

An important parameter of a genetic algorithm is the fitness function that defines the fitness of each chromosome. We consider in this work two classes of errors that we wish to detect in state spaces: deadlocks and assertion violations. Deadlocks are states with no outgoing transitions (all the processes in the system are blocked). Assertions are boolean expressions involving program variables that are said to be violated when the corresponding boolean expression evaluates to false. We now discuss heuristics for guiding a genetic search towards both classes of error states.

For deadlock detection, a simple heuristic to measure the fitness of a chromosome is to sum the number of enabled transitions at each state along the execution path represented by the chromosome. The intuition behind this heuristic is that chromosomes with a smaller sum seem more likely to lead to deadlocks, and should therefore be considered fitter.

For detecting assertion violations, a possible heuristic is to attempt maximizing assertion evaluations. To achieve this, one can award *bonus* scores for chromosomes that lead to as many as possible assertion evaluations. One can also award bonuses to chromosomes that make choices leading towards assertion statements at control points in the control flow graph of the program; this can be done by instrumenting the execution of tests (such as "if-then-else" statements) in the program using a static analysis of the program text.

When analyzing protocols with message exchanges, a sensible heuristic is to attempt maximizing the number of messages being exchanged. We use this simple heuristic in the analysis of Needham-Schroeder public key authentication protocol [15] and identify a (previously known [14]) attack on the protocol (see Section 4 for details).

Note that our framework can be used to discover multiple (independent) errors of a same type in a system without requiring to fix previously detected errors. This can be done by awarding *penalty* scores to chromosomes that lead to states where a previously discovered error is detected. Application-specific heuristics can also be used in our framework to fine tune the performance of the genetic algorithm if needed.

3.4 Dynamically Adapting Parameters

The genetic algorithm we use in this work is a slight variation of the pseudocode in Figure 2 where the value of some parameters are adapted as the genetic evolution progresses. In particular, we keep track of the best and worst chromosome fitness in each generation, and, if both fitness values become equal, we increase the mutation rate, in order to help the genetic evolution get out of local maximas. Once there is an improvement in the overall fitness, we restore the original mutation rate to continue evolution as normal.

As mentioned in Section 3.2, we also update the effective length of chromosomes during evolution.

If evolution stabilizes (i.e., the fitness does not seem to improve for several generations) and the search does not find any error, we re-start the genetic algorithm with the initial default parameter values and a new randomly generated seed to generate a new random initial population. This reduces any bias that may have been introduced in a previous run that used a "bad" seed.

4 Experimental Evaluation

We have implemented the framework presented in the previous section in conjunction with VeriSoft [8], a tool that implements model-checking algorithms for exploring the state spaces of systems composed of several concurrent software processes executing arbitrary code written in full-fledged programming languages such as C or C++. We report in this section results of experiments comparing the performances of four state-space search algorithms:

· GA is the genetic algorithm described in the previous section;
· GA_M is GA with no crossovers (only mutations);
· $RAND$ is a "random search" that explores random paths in a state space; and
· EXH is a search algorithm that systematically explores the state space up to some fixed depth[2], and attempts to explore it exhaustively.

The purpose of these experiments is also to identify heuristics and properties of state spaces that help a genetic search detect deadlocks and assertion violations.

4.1 Examples of Programs and Properties

We report experiments performed with two sample C programs.

[2] Note that, in general, the depth of the state space of a software system composed of processes executing arbitrary C or C++ code may not be bounded, making the state space infinite and a fully exhaustive search impossible.

Dining philosophers Consider the following variant of the well-known dining-philosophers problem:

```
while (true) {
  think;
  nondeterministically
    pick left-fork; pick right-fork;
    OR
    pick right-fork; pick left-fork;
  eat;
  drop left-fork; drop right-fork; }
```

The above pseudocode describes a philosopher process. A philosopher starts by thinking, which then makes him hungry at which point he nondeterministically decides to either pick his left fork followed by his right fork, or to pick his right fork followed by his left fork. Once a philosopher has both forks adjacent to him in his hands, he eats. Finally, he drops first the left fork and then the right fork back onto the table, and repeats this process indefinitely. Since several philosophers are sitting around the same table and hence sharing one fork with each of their two adjacent neighbors, they compete for forks with each other. For instance, if all philosophers around the table have picked their left fork, the entire system is then in a deadlock.

We denote by $PHIL$ a C implementation of the above system with 17 philosophers. We arbitrarily choose this large number of processes so that it is not possible to explore the state space of the system exhaustively within a reasonable amount of time. Nondeterminism is simulated using the system call `VS_toss` supported by VeriSoft (see [8]). In what follows, we compare the effectiveness of various search algorithms to find deadlocks in this system.

Needham-Schroeder protocol The Needham-Schroeder public-key authentication protocol [15] aims at providing mutual authentication, so that two parties can verify each other's identity before engaging in a transaction. The protocol involves a sequence of message exchanges between an *initiator*, a *responder*, and a mutually-trusted key server. The exact details of the protocol are not necessary for the discussion that follows and we omit these here. An attack against the original protocol involving six message exchanges was reported in [14]: an intruder *Carol* is able to impersonate an initiator *Alice* to set up a false session with responder *Bob*, while *Bob* thinks he is talking to *Alice*.

We denote by $AUTH$ a C implementation[3] of the Needham-Schroeder protocol. This implementation is described by about 500 lines of C code and is much more detailed than the protocol description analyzed in [14]. The C code also contains an assertion that is violated whenever an attack to the protocol occurs. We compare below the effectiveness of various search algorithms to find assertion violations representing attacks to this implementation of the protocol.

[3] John Havlicek provided us this implementation.

		error found?	$\frac{\#errors}{\#runs}$	*runtime (hrs)*	*average time to find error*	*depth searched*
$PHIL$	GA	yes	26/50	1:16:21	2 min 57 sec	65
	$RAND$	no	0/1	8:00:00	-	65
	EXH	no	0/1	8:00:00	-	34
$AUTH$	GA	yes	3/100	2:33:24	51 min 8 sec	110
	$RAND$	no	0/1	8:00:00	-	110
	EXH	no	0/1	8:00:00	-	45

Table 1. Genetic search versus random and exhaustive search

4.2 Experimental Results

In the experiments that follow, whenever a genetic search is applied to $PHIL$ to detect deadlocks, the heuristic (fitness function) used is to minimize the sum of enabled transitions along a single execution path. In contrast, whenever a genetic search is applied to $AUTH$ to detect protocol attacks, the heuristic used in the experiments below is to maximize the number of messages exchanged among parties involved in the protocol along a single execution path. All experiments were performed on a Pentium III 700 MHz processor with 256 MB of RAM.

The genetic parameters we use are as follows. The population size is set to 200 chromosomes. The best 100 chromosomes in a generation reproduce. The default mutation rate is 200, i.e., each bit of a chromosome that is produced by a crossover is flipped with probability 1/200. The effective length of chromosomes varies between 70 and 320 bits.

Genetic Search versus Random and Exhaustive Searches We compare the performance of the search algorithms GA, $RAND$ and EXH for analyzing $PHIL$ and $AUTH$. For GA and $RAND$, we limit the depth of the search to paths of length about twice the length of the shortest path that leads to an error. (We discuss this choice later in this section.) For EXH, we limit the search depth to about the length of the shortest path that leads to an error (with the hope of helping EXH as much as possible).

Table 1 summarizes our results. For $PHIL$, we run GA 50 times (each run starts with a randomly-generated seed), and let it evolve for 50 generations in each run. More than 50% of the runs identify a deadlock. In contrast, both $RAND$ and EXH are unable to find a deadlock in 8 hours of search. For $AUTH$, we run GA 100 times (each run uses a randomly-generated seed), and let it evolve for 100 generations in each run. Only 3 runs identify an attack on the C implementation of the Needham-Schroeder protocol. Again, both $RAND$ and EXH are unable to find an attack in 8 hours.

Despite that GA is able to find an attack in $AUTH$, its performance is worse than when analyzing $PHIL$. This may be due to our choices of fitness functions: the heuristic for finding deadlocks may be a better measure of fitness, than the simple heuristic of maximizing message exchanges used when exploring the state space of $AUTH$. We chose to use and evaluate these particular heuristics in our

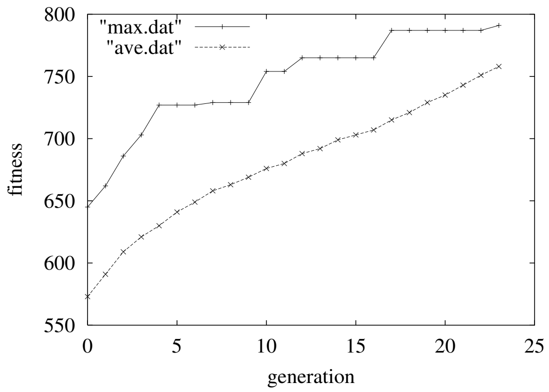

Fig. 4. GA deadlock-detection performance. The maximum and average fitness among the parent chromosomes in a generation is plotted against the generation number.

experiments because they are application-independent and hence can be used to analyze other applications.

Figure 4 illustrates a run of GA on $PHIL$. Typically, a genetic algorithm makes quick progress in the beginning stages of evolution. Then, there are phases when it hits local maximas before mutations further improve its performance. Notice how the average fitness of the parents steadily increases. This indicates that the genetic operators are effective in maximizing fitness while exploring the state space. It should not come as a surprise that the maximum (average) fitness among parents never decreases since we are using the so-called *elitist* model, in which the best chromosomes always survive to the next generation.

Search Deeper We now investigate how the effectiveness of a genetic search varies as we increase the maximum depth of the search. In these experiments, we consider a simplified version of $AUTH$ where the first two message exchanges from a known attack (involving a path of 42 steps in the state space) are hard-wired into the search algorithm and the algorithm needs only to find the last 4 message exchanges necessary to complete the attack. We call this simpler problem $AUTH_2$, and use it in the experiments below in order to amplify differences between results we observe.

Table 2 tabulates our results. We run GA on $PHIL$ for 50 generations. We compare the results of 20 runs using each of the depths 34, 51 and 68, where 34 is the minimum depth required to find a deadlock in $PHIL$. When using depths 34 and 51, GA is unable to detect a deadlock, whereas when we increase the depth to 68, 14 out of 20 runs detect a deadlock. When exploring the state space of $AUTH_2$ using a depth of 42, GA is unable to find an attack in 20 tries, whereas when we increase the depth to 60, GA finds an attack 6 times.

The reason why a deeper maximum search depth can actually help a genetic search may be the following. From most reachable states in the state spaces of $PHIL$ and $AUTH$, there exists a path that leads to an error state. Chromosomes

	error found?	$\frac{\#errors}{\#runs}$	runtime (hrs)	average time to find error	depth searched
$PHIL$	no	0/20	0:25:33	-	34
	no	0/20	0:30:01	-	51
	yes	14/20	0:33:44	2 min 24 sec	68
$AUTH_2$	no	0/20	0:33:47	-	42
	yes	6/20	0:28:31	4 min 45 sec	60

Table 2. GA performance as maximum search depth changes

		error found?	$\frac{\#errors}{\#runs}$	runtime (hrs)	average time to find error	depth searched
$PHIL$	GA	yes	26/50	1:16:21	2 min 57 sec	65
	GA_M	yes	26/50	0:59:16	2 min 16 sec	65
$AUTH_2$	GA	yes	16/50	1:11:18	4 min 27 sec	60
	GA_M	yes	3/50	1:27:07	29 min 2 sec	60

Table 3. Genetic search with (GA) and without (GA_M) crossover operator

that encode "bad" initial segments are therefore not necessarily penalized since their tails may contain a path that leads to an error state and are sufficient to detect the error. If the exploration was limited to the minimum depth necessary to find an error, chromosomes that encoded the "wrong" first moves would have a very low probability of producing an offspring that corrects these first moves.

On the other hand, increasing the depth of the search should be done with caution since it obviously increases the search space and hence the length of chromosomes, which in turn leads to slower genetic operations and convergence of the algorithm.

Mutation Alone Here, we investigate the effectiveness of the crossover operator by comparing the performance of GA and GA_M, i.e., GA without crossover operations, when exploring the state spaces of $PHIL$ and $AUTH_2$. The same parameter values are used for both GA and GA_M.

Table 3 summarizes our results. The performances of GA and GA_M are comparable on $PHIL$: both algorithms find the deadlock 26 times out of 50 runs. This may be explained as follows. A deadlock in $PHIL$ results from a *set* of choices made by the philosophers, namely that they all choose to pick their left forks or they all choose to pick their right forks. In particular, it does not matter in which order the philosophers pick their forks; what matters is which fork they pick. Mutations alone seem effective in finding a deadlock since each mutation alters some philosopher's choice and once the right set of choices is attained, a deadlock is reached.

In contrast, GA is more effective than GA_M in finding an attack on $AUTH_2$. An attack on the protocol is formed by a specific *sequence* of message exchanges that allows intrusion: the messages have to be exchanged in a precise order,

	error found?	$\frac{\#errors}{\#runs}$	*runtime (hrs)*	*average time to find error*	*depth searched*
$PHIL$	yes	26/50	1:16:21	2 min 57 sec	65
$PHIL^{PO}$	yes	5/50	1:07:32	13 min 30 sec	65
$AUTH_2$	yes	16/50	1:11:18	4 min 27 sec	60
$AUTH_2^{PO}$	yes	1/50	1:24:34	1 hr 24 min 34 sec	60

Table 4. GA performance with and without partial-order reduction

simply finding the exact set of messages involved in the attack is not sufficient. Since crossovers combine and preserve sub-sequences (of messages in this case), their effect in converging quickly toward a solution becomes more important.

Therefore, it seems preferable to use GA over GA_M when exploring arbitrary state spaces, since GA is effective irrespective of the search being for a set or a sequence of transitions.

Partial-Order Reduction Finally, we investigate how the use of partial-order reduction techniques (e.g., see [7]) affects the performance of a genetic search. Roughly speaking, partial-order reduction algorithms can dynamically prune the state space of a concurrent system in a completely reliable way (i.e., without missing any errors) by taking advantage of independent (i.e., commutative) actions executed by concurrent processes, hence avoiding to consider all their interleavings during a state-space exploration. The pruned state space defined with partial-order algorithms is thus a subset of the full state space. In the following experiments, we consider a partial-order reduction algorithm using a combination of the persistent-set and sleep-set techniques as implemented in VeriSoft (see [8]). Let $PHIL^{PO}$ and $AUTH_2^{PO}$ denote the reduced state spaces of $PHIL$ and $AUTH_2$, respectively, that are explored when partial-order reduction is used.

Results of experiments are tabulated in Table 4. When exploring $PHIL^{PO}$, GA detects a deadlock only 5 times out of 50 runs. Recall that GA detected a deadlock 26 times during a same number of runs when exploring $PHIL$. A similar decrease in performance is observed when GA explores $AUTH_2^{PO}$.

A possible explanation for this phenomenon is the following. In the reduced state space resulting from partial-order reduction, most reachable states have few outgoing transitions that can be selected to be explored next (thanks to the pruning). Hence, the set of actions corresponding to a set of possible next transitions can vary a lot from state to state. This means that selecting transition number i in a state s may result in executing a program action totally different from the action executed when selecting transition i in another state s'. In other words, same transition choices made in different context may yield totally different program actions, especially when using partial-order reduction. After a crossover or mutation operation, the tail of each resulting chromosome may be interpreted in an entirely different context, which harms the beneficial effect of these operators.

5 Conclusion and Related work

We have shown in this paper that, when exploring very large state spaces of concurrent reactive systems, genetic algorithms using simple application-independent heuristics can significantly outperform traditional random and systematic state-space searches used in current model checkers. We have discussed in detail the engineering challenges faced when extending a model checker with a genetic search algorithm. We believe the use of heuristics in model checking could contribute to broadening its applicability by several additional orders of magnitude. Further experiments and studies are needed to validate this claim.

As mentioned in the introduction, genetic algorithms have already been used for a broad range of applications. In particular, genetic algorithms have been used to perform structural and functional testing of sequential programs. For instance, Pargas et al. [16] present a goal-oriented technique for automatic test-data generation using a genetic algorithm guided by program control dependencies; their implementation aims at achieving statement and branch coverage. Jones et al. [12] use genetic algorithms to generate test sets that satisfy the requirements for test-data-set adequacy of structural testing. More recently, Bueno et al. [2] build upon [12] and present a tool for the automation of both test-data generation and infeasible-path identification. In [13], a framework using genetic algorithms is developed for testing methods manipulating complicated data structures; this framework was successfully applied to identify several flaws in a naming architecture for dynamic networks of devices and computers.

In contrast with all this previous work, the problem addressed in this paper is the exploration of (very large) state spaces of concurrent reactive systems as defined with a model checker. This requires the use of original chromosome encodings and fitness functions suitable for the application domain considered here. We are not aware of any other work where genetic algorithms have been used for state-space exploration.

Heuristics for choosing a search order that favor visiting first successor states that are most likely to lead to an error ("best-first search") are discussed in [21] in the context of symbolic model checking and in [5] in the context of explicit model checking. It is worth noting that a best-first search (BFS) can be viewed as a particular case of genetic search (GS). Indeed, the latter can simulate the former as follows: GS uses the same fitness function as that of BFS; crossover and mutation rates are set to 0; the effective length of chromosomes is set to n where n is the current generation; only a single best chromosome in a generation produces the next generation; the number of offsprings produced by this unique parent is the number of outgoing transitions at the last state visited by the parent and each offspring contains the entire parent path plus one more (unique) transition. Backtracking strategies (breadth-first, depth-first, etc.) that can be used in conjunction with BFS can also be simulated by dynamically adapting parameters of GS and appropriately defining the creation of next generation. In contrast, a best-first search cannot simulate a genetic search in general since its "fitness function" is restricted to local heuristics based on the current state and next possible transitions, and hence lacks the ability to simulate the global evaluation of an entire chromosome. Intuitively, a best-first search is also more "deterministic" than a genetic search since it is less general and does not include randomized operations like crossovers and mutations, which improve robustness with respect to sub-optimal fitness functions by helping the search avoid being trapped in local maxima. Further studies are needed to determine which parameter values of a genetic search (including BFS) are best suited for analyzing specific classes of programs and properties.

Heuristics for over and under approximating BDD representations when these become too large or for finding pseudo-optimal BDD-variable orderings are also commonly used in symbolic verification. Such heuristics tackle different problems related to model checking and are of different nature than the ones used here.

The issue of changing parameter values during the run of a genetic algorithm is an active area of research in genetic algorithms. A recent survey is given in [6]. The "1/5 rule" of Rechenberg [17] constitutes a classical adaptive method for setting the mutation rate. This rule states that the ratio of mutations in which the offspring is fitter than the parent, to all mutations should be 1/5, hence if the ratio is greater than 1/5, the mutation rate is increased, and if the ratio is less than 1/5, the mutation rate is decreased.

The "Dynamic Parameter Encoding" [20] (DPE) algorithm provides the ability to encode real-valued parameters of arbitrary precision. DPE first searches for optimal values of more significant digits of the parameters. Next it fixes the values discovered and progressively searches for lesser significant digits. This way the same fixed length chromosome encodes different digits of parameters at different points during the algorithm execution. Notice that DPE requires a priori knowledge of an upper bound on parameter values.

Our dynamic encoding of paths in a state space is novel to the best of our knowledge; it does not require a priori knowledge of the maximum number of enabled transitions in any given state of a state space.

Acknowledgments

We thank John Havlicek for sharing with us his implementation of the Needham-Schroeder protocol. We also thank Darko Marinov and Audris Mockus for helpful comments on this paper, and Enoch Peserico for inspiring discussions on genetic algorithms. The work of the second author was done partly while visiting Bell Laboratories and was also funded in part by ITR grant #0086154 from the National Science Foundation.

References

1. B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the ACCESS.bus protocol using SPIN. In *Proceedings of Formal Methods Europe'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478, Oxford, March 1996. Springer-Verlag.

2. Paul Marcos Siqueira Bueno and Mario Jino. Identification of potentially infeasible program paths by monitoring the search for test data. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000.
3. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Apllications*. North-Holland, 1993.
4. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
5. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In *Proceedings of the 2001 SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.
6. A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.
7. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.
8. Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, January 1997.
9. Patrice Godefroid, Robert Hanmer, and Lalita Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of ACM SIGSOFT ISSTA'98 (International Symposium on Software Testing and Analysis)*, pages 124–133, Clearwater Beach, March 1998.
10. David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.
11. John Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.
12. B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, Sep 1996.
13. Sarfraz Khurshid. Testing an intentional naming system using genetic algorithms. In *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Genova, Italy, April 2001.
14. Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 1995.
15. Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
16. Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification, and Reliability*, 9(4):263–282, 1999.
17. Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzbog, Stuttgart, 1973.
18. Peter Ross and Dave Corne. Applications of genetic algorithms. *AISB Quaterly on Evolutionary Computation*, pages 23–30, Autumn 1994.
19. H. Rudin. Protocol development success stories: Part I. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.
20. Nicol N. Schraudolph and Richard K. Belew. Dynamic parameter encoding for genetic algorithms. *Machine Learning*, 9(1):9–21, 1992.
21. C. H. Yang. *Prioritized Model Checking*. PhD thesis, Stanford University, 1998.