

Finding Deadlocks in Large Concurrent Java Programs Using Genetic Algorithms

Enrique Alba
University of Málaga, Spain
eat@lcc.uma.es

Francisco Chicano
University of Málaga, Spain
chicano@lcc.uma.es

Marco Ferreira
Instituto Politécnico de Leiria,
Portugal
mpmf@estg.ipleiria.pt

Juan Gomez-Pulido
University of Extremadura,
Spain
jangomez@unex.es

ABSTRACT

Model checking is a fully automatic technique for checking concurrent software properties in which the states of a concurrent system are explored in an explicit or implicit way. However, the state explosion problem limits the size of the models that are possible to check. Genetic Algorithms (GAs) are metaheuristic techniques that have obtained good results in problems in which exhaustive techniques fail due to the size of the search space. Unlike exact techniques, metaheuristic techniques can not be used to verify that a program satisfies a given property, but they can find errors on the software using a lower amount of resources than exact techniques. In this paper, we compare a GA against classical exact techniques and we propose a new operator for this problem, called memory operator, that allows the GA to explore even larger search spaces. We implemented our ideas in the Java Pathfinder (JPF) model checker to validate them and present our results. To the best of our knowledge, this is the first implementation of a Genetic Algorithm in this model checker.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Model checking*; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Verification

Keywords

Model Checking, GA, Memory Operator, Graph Search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO '08 Atlanta, Georgia, USA

Copyright 2008 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

1. INTRODUCTION

Concurrent systems are the target of subtle errors that are very difficult to detect as they may depend on the order the operating system chooses to execute the different threads of the application. Some examples of this kind of errors are deadlocks, livelocks and starvation. One technique used to validate and verify programs against several properties like the ones mentioned is *model checking*. Basically, a model checker uses an abstract simplification of the program (called the model), creating and traversing a graph representing all the possible states of that model to find a path starting in the initial state that violates the given properties. Usually, the size of the graph is very large, which leads to the main problem in model checking, known as *state explosion*. This problem consists in the fact that realistic and complex models have too many possible states, sometimes even an infinite number, to allow efficient model checking. If, instead of using a model, the real application is used, the number of states is even higher.

We show in this work that a variation on Genetic Algorithms (GAs) combined with Java Pathfinder, a well known model checker, can be used to search for errors in complex applications. We will show that good results, while not optimal, can be obtained faster and with fewer resources than using the standard exact search methods. We will show that, using a new operator, even more complex applications can be searched with good results. While we will focus on searching for deadlock, any kind of safety property violation could be searched for with our algorithms, changing only the fitness function.

The rest of this paper is structured in the following way: Section 2 gives background on model checking. Section 3 describes our proposal of a new Genetic Algorithm that can be used to discover paths to objective nodes on large state graphs. Section 4 presents our experiments and results and Section 5 describes our proposal of a new operator to improve our Genetic Algorithm. Finally, Section 6 summarizes our conclusions and refers to further work in this area.

2. BACKGROUND

Model Checking is an automatic technique for checking if a reactive system fulfills a given property. Properties can be classified into two groups: *safety* and *liveness* properties. Informally, a safety property asserts that “something

bad does not happen” while a liveness property asserts that “something good eventually happens” [12]. In order to find a safety property violation, it is only needed to find a finite path to a violating state (where something bad happened, like for example a deadlock state). However, to find a liveness property violation, it is needed to find an infinite path in which a desired state never occurs (that is, the expected good never happened) [8].

To find safety property violations, model checking works by searching through all the states of a model (either explicitly or implicitly) and validating each of them against the specified properties. If all the states are valid, it is proved that those properties are valid in the model. If any of those properties fail, the model checker can give an execution path that leads to that error. This execution path is also known as the *counterexample* or the *error trail*. As it is needed to run through all possible states in the model to prove correctness, the number of states must be relatively small, otherwise the model checker reaches memory constraints.

There are techniques for reducing the search space, like Partial Order Reduction [5] and Symmetry Reduction [11], that allow model checkers to tackle larger models, however they are not yet sufficient to deal with large and complex programs. Symbolic model checking [4] is also a very popular alternative to the explicit state model checking that can reduce the amount of memory required for the verification by means of a compact representation for set of states. When the search space is very large, it may be impossible to prove correctness of the model. However, an error might be found without searching the entire search space, proving the incorrectness of the model.

Since typically not all state-space can be explored, the method used to search through the states is of critical importance to determine if and how quickly an error is found. Also, the size of the execution path that leads to the error is important to debug: it must be as short as possible. That leads to the question: what is, if any, the best search algorithm?

Nested Depth First Search has been utilized in model checkers for checking liveness and safety properties. In the case of safety properties, Depth First Search (DFS) and Breadth First Search (BFS) have also been used, since they can be applied to the search for one objective node in a graph. The use of heuristic-based search algorithms such as A* and Best-First Search has been studied also in the context of safety properties [9].

These search methods are not appropriate to deal with the state-explosion problem, as they may need to search the entire search space. To limit the memory used and time consumed, some heuristic searches are used not to prove the correctness of the model but to find an error in it. This kind of search may not search the whole search space. One example is the Beam Search Algorithm [9].

Another type of search algorithms, very popular in the optimization domain, are the *metaheuristic search algorithms*. Unlike simple heuristic algorithms, metaheuristic search algorithms try to find new solutions using information available from previous solutions [14]. There are several metaheuristic search algorithms, both single-solution based and population based, some of the most popular being Simulated Annealing, Tabu Search, Evolutionary Algorithms, Scatter Search, and Ant Colony Optimization (ACO) [3].

Although metaheuristic search algorithms have been widely

used in optimization problems of many different areas, there have been few incursions to model checking. It may be due to the fact that metaheuristic searches are not optimal and complete and as a consequence cannot be used to prove correctness of a model. Previous work using metaheuristics in model checking includes the usage of genetic algorithms in Protocol Validation [2] and to explore large state-space problems [7], and using ACO algorithms to find short counterexamples [1]. Both [2] and [7] show that it is possible, and advantageous, to use genetic algorithms in model checking. However, Alba [2] uses an encoding specific to protocol validation and Godefroid [7] implemented his GA in VeriSoft, a model checker used with C programs.

3. USING GAs IN MODEL CHECKING

A Genetic Algorithm is a population based metaheuristic search algorithm. Its basic operation is depicted in Algorithm 1.

Algorithm 1 Pseudocode of a Genetic Algorithm

```

1:  $P = \text{generateInitialPopulation}()$ ;
2:  $\text{evaluate}(P)$ ;
3: while not stoppingCondition() do
4:    $P' = \text{selectParents}(P)$ ;
5:    $P' = \text{applyVariationOperators}(P')$ ;
6:    $\text{evaluate}(P')$ ;
7:    $P = \text{selectNewPopulation}(P, P')$ ;
8: end while
9: return the best found solution

```

Basically, a GA starts by creating an initial population P and evaluating it. Then, it creates a new population (P') by selecting parents and applying variation operators on those parents. These variation operators are usually crossover and mutation operators. Having evaluated the new population, the next generation is selected using elements from both the old and the new population.

As with all metaheuristic algorithms, the GA is more like a framework and as such must be adapted to the problem it will solve. We will show which parameters and methods can be used to adapt that framework to find safety property violations with model checking. We will discuss how to encode solutions as individuals, how to perform the crossover and mutation of those individuals and the fitness function used to evaluate them. We call the resulting algorithm geGA, a graph-exploring Genetic Algorithm.

3.1 Individual Encoding

In a GA an individual represents a possible solution to the problem. In our case, the problem is to find a path to a goal state (a state that causes a safety property violation in the model). A path can be described as the sequence of transitions that occurred from an initial state to a final state. Figure 1 shows an example of a state graph evidencing the path described by transitions 0, 1, 1.

In the particular problem we are solving, the number of transitions needed to reach a goal state (the path size) is unknown beforehand, that is, we do not know what is the length of the shortest path from the initial state to a goal state. If the individuals were composed of a fixed length sequence of transitions, the algorithm might always fail in finding a goal state. We solve the path size problem using

variable-length chromosomes. This allows our GA to create solutions of variable number of transitions and determine which size is best.

As is shown in Figure 1, the number of enabled transitions in each state is not always the same and is unknown until that state is visited by the model checker. One possible solution to deal with unknown number of transitions would be to use an integer to identify the transition. This raises two questions: what should be the maximum value of that integer? and, what to do if in a particular state there are fewer transitions than the corresponding value in the chromosome? There is no easy answer to these questions. One might use the maximum number of transitions available in a model as the maximum integer, but that number is not always easy to figure out, and sometimes is simply impossible to determine. Then comes the second problem. If the next integer in the chromosome is s and the state had only n transitions available (with $s > n$) which transition t would we choose? We could choose to cut s at n with $t = \text{Min}(s, n)$ but that would create a bias toward n . Something similar, but harder to detect, would happen if we used a modulo operation like $t = (s \bmod n)$. Consider the following example: a state has 3 transitions and the GA uses the maximum transition number as 5. Thus, in that particular state, the choices available to the GA would be: $s = 0 \Rightarrow t = 0$, $s = 1 \Rightarrow t = 1$, $s = 2 \Rightarrow t = 2$, $s = 3 \Rightarrow t = 0$, $s = 4 \Rightarrow t = 1$. That would result in transition 2 to have half the chances of being selected than transition 0 or 1.

We chose to use a representation for the transition other than an integer. To represent the selected transition s in a state we use a floating-point number in the range $[0..1)$. To determine the transition t in a state with n transitions, we just use the formula $t = \lfloor s \times n \rfloor$. Since s is normalized (between 0 and 1, exclusive), the resulting value t is an integer number between 0 and the number of possible transitions in the state, uniquely identifying a transition. This allows an unknown number of transitions while maintaining a linear conversion from the selected transition to the available transition. However, this may create a problem with the mutation operator because the number of possible values for the gene (the alphabet) is (theoretically) infinite, but the meaning of that alphabet is finite. The mutation operator must be able to guarantee that, when changing the value of the gene, the new value means a new path choice after the linear conversion.

3.2 Crossover and Mutation

To allow the chromosome to grow and shrink, the crossover operator has been adapted. Our crossover chooses a differ-

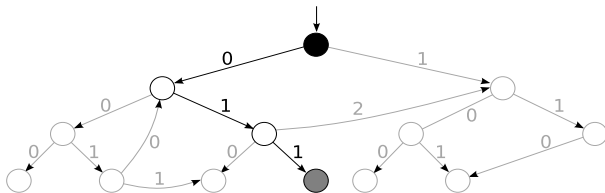


Figure 1: Example graph showing the states of a program and the variable number of transitions in each state.

ent position for each parent and then swaps the genes after that position to form the offspring. The offspring will probably have different sizes than their parents. Also, as we use floating-point numbers instead of a more traditional bit encoding, the cut positions are always between transition choices. There is little sense in using two positions per parent because, when inserting at the middle different values and probably different number of transitions, the remaining values lose their meaning. An example of the crossover operator described can be seen in Figure 2.

Our mutation operator traverses all the genes in the individual and, with a given probability, creates new random values for those genes. However, since we are using floating-point numbers to encode the individual, we must guarantee that when we change the gene value, that change really affects the path choice that gene represents. We make this guarantee by traversing the path identified by the chromosome up to the selected gene and checking how many transitions are available at that point. We then check which of those transitions corresponds to the current gene value and generate new random values until one is found corresponding to a different transition and use it to replace the gene value. Our mutation operator does not change the size of the chromosome, and that might be a change to consider in the future.

3.3 Fitness Function

Genetic Algorithms are function optimizers. They usually maximize or minimize the value of a given function. In our case, we want to detect paths that lead to deadlocks, and prefer shorter paths. As such, our fitness function $f(x)$ is defined as shown in (1), where the variable *numblocked* represents the number of blocked threads generated by the path while *pathlen* represents the number of transitions in the path and *deadlock* is 1 if a deadlock was found, 0 otherwise. The Genetic Algorithm will try to maximize $f(x)$.

$$f(x) = \text{deadlock} + \text{numblocked} + \frac{1}{1 + \text{pathlen}} \quad (1)$$

Equation (1) assumes that the number of threads in the model to be checked is constant or that a deadlock only occurs when all the threads are blocked. If that is not the case, a much bigger *deadlock* value must be used when a deadlock is found.

4. EXPERIMENTS

In this section we present the results of our experiments. The experiments have been performed with Java Pathfinder on an Intel Core Duo T2400 (1.83 Mhz) processor, with a Java virtual machine restricted to 512Mb of memory (Sun JVM 1.6.0_01-b06). To detect deadlocks, we have used two well-known problems in the Java Pathfinder and the model checking community: the Dining Philosophers [6] and the Stable Marriage problem [10].

The Dining Philosophers problem is an illustrative exam-

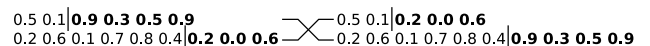


Figure 2: The crossover we implemented allows for chromosome length changes to occur

ple of common concurrency problems. It consists of a number of philosophers (originally 5) around a round table. Each philosopher shares a fork with the philosopher to his right and another fork with the philosopher to his left. Whenever a philosopher gets hungry, he picks the left fork, then the right fork, eats and drops the left fork and finally the right fork, returning to a thinking state. As the forks are shared between philosophers, whenever one philosopher is eating, the other two close to him can not eat. If all the philosophers decide to eat at the same time, they will all pick their left forks but will all be waiting for the right fork to be available (which will never happen), thus creating a deadlock.

The Stable Marriage problem has two sets on n elements, the man and the women. Each element of each set ranks each element of the other set in order of preference. The problem consists in pairing each man to a woman such that there exists no men and women who are not assigned to each other but who would both prefer each other to their present partners. When that happens, we have a stable marriage.

For the Dining Philosophers problem we used the two different implementations included with JPF. In the first one, called DiningPhilosophers, each philosopher cycles through the pick forks, eat, drop forks and think states. In the second implementation, called DiningPhil, each philosopher only pick the forks, eat and drop the forks once, thus limiting the number of possible deadlocks.

For the Stable Marriage Problem (SMP), we converted a distributed solution from a model in Promela. This solution has an intentional bug created by splitting a critical section in two, giving the opportunity to another thread to interleave and create a deadlock. Without that intentional error, the solution have been proven correct with JPF using complete search methods (such as BFS), but restricting the number of couples to 3.

While these problems seem simple, the amount of states generated by them is huge as they grow exponentially with the number of philosophers or couples. We have tried to explore the entire graph of these problems to have an idea of their size. DiningPhilosophers grows from 2094 to 120544 states when the number of philosophers is increased from 3 to 4. We could not explore the entire graph with 5 philosophers. The same happens to SMP, growing from 10509 to 773843 states when increasing the number of couples from 3 to 4.

Our purpose is to have short error trails. We do not want to stop as soon as the algorithm finds a deadlock. Instead, we want to give geGA a chance for that solution to improve. To allow some improvement we set the stopping criterion of geGA to end the search only after a number of iterations, even if a deadlock has been found before. As it can be seen in Table 1, we used a well-known and studied selection operator, the tournament selection with size of 2. As the mutation probability we have used the value of 0.01, which means that every gene as a 1% (in average) chance to have its value mutated. We also used an elitist replacement, which means that the best individual of the previous generation survives to the next generation. The parameters for each problem are the result of a previous study aimed at finding good configurations for tackling the respective problem. Notice that due to the increased difficulty of the DiningPhil problem we had to use more iterations and even a bigger population to find property violations.

Table 1: Configuration of the geGA

Problem	Size	Parameter	Value
Common to all		Selection	Tournament(2)
		Mut. Prob.	1%
		Replacement	Elitism(1)
DiningPhilosophers	4	Pop. Size	50
		Max Length	50
		Iterations	50
DiningPhilosophers	8	Pop. Size	50
		Max. Length	60
		Iterations	50
DiningPhilosophers	16	Pop. Size	50
		Max Length	120
		Iterations	50
DiningPhil	4	Pop. Size	50
		Max Length	25
		Iterations	100
DiningPhil	8	Pop. Size	50
		Max Length	50
		Iterations	100
DiningPhil	16	Pop. Size	200
		Max Length	100
		Iterations	150
SMP	3	Pop. Size	50
		Max Length	50
		Iterations	50
SMP	4	Pop. Size	50
		Max Length	60
		Iterations	50
SMP	5	Pop. Size	50
		Max Length	60
		Iterations	50

Since Genetic Algorithms are stochastic algorithms, the results of a single execution is not sufficient to conclude anything about its performance, so we have run the algorithm several times and report the average of the measures considered for each problem. Our results are shown in Tables 2, 3 and 4. Because of the time given to geGA to improve solutions, for each of the problems, we show the time taken to find an error (Time to 1st Error) and the total execution time of the search (Total Time), which includes the error trail improvement time. These are always the same for DFS and BFS as these search methods stop as soon as they find an error. We also present the depth of the first error found (Depth of 1st Error) and the error depth after improvement (Final Depth). Times are expressed in seconds and depths are expressed in number of states traversed to reach the error state. Finally we show the memory used by the search algorithm, expressed in Megabytes.

We also show in the tables the results of a statistical test (with level of significance $\alpha = 0.05$) in order to check if there exist statistically significant differences. An item in DFS or BFS is marked with a + sign in parenthesis if the difference with geGA is significant and it is marked with a - sign if it is not. We use the one sample Wilcoxon sign rank test because we compare one sample (the results of geGA) with one single value (the result of the corresponding exhaustive algorithm). The better results in each measure

Table 2: Results of the algorithms with DiningPhilosophers

Search Method	Number of Philosophers	Time to 1 st error (s)	Total Time (s)	Depth of 1 st error	Final Depth	Memory (MB)	errors / runs
DFS	4	(+)2.00	(+)2.00	(+)147.00	(+)147.00	(+)20.00	1/1
BFS	4	(+)36.00	(+)36.00	(+)20.00	(-)20.00	(+)168.00	1/1
geGA	4	0.00	5.82	32.68	20.00	81.12	50/50
DFS	8	(+)10.00	(+)10.00	(+)24248.00	(+)24248.00	(+)71.00	1/1
BFS	8	-	-	-	-	-	0/1
geGA	8	1.90	14.50	55.32	41.72	170.06	50/50
DFS	16	-	-	-	-	-	0/1
BFS	16	-	-	-	-	-	0/1
geGA	16	16.62	53.16	112.14	95.66	408.9	50/50

are highlighted in boldface.

The results presented in Table 2 show that geGA was the fastest algorithm to find a property violation in the Dining-Philosophers problem. However, the total execution time of our algorithm is greater than the one of DFS for 4 and 8 philosophers. The improvement provided by that extra number of iterations is clearly visible, reaching more than 30% of improvement in the error depth. In terms of finding the deadlock, geGA was the only algorithm capable of finding it with 16 philosophers. We aborted the DFS search for 16 philosophers after 2 hours without finding any deadlock. Comparing the quality of the error trail, we can verify that geGA gives short error trails, very close to the smallest possible. Finally, in terms of memory, BFS uses more than the other algorithms, running out of memory from 8 philosophers on. Overall, geGA provided the best solutions in the shortest amount of time, but requires a large amount of memory.

Table 3 shows the results of our experiments with the DiningPhil problem. Again, our algorithm was the fastest in finding a property violation in all instances of the problem. All the algorithms have the same error trail length because the lack of cycles in the graph. Comparing the memory requirements, we can verify that geGA requires more memory than DFS, but much less than BFS which could not solve the problem with 8 philosophers due to insufficient memory. Again, geGA was the only algorithm to find the deadlock with 16 philosophers (DFS did not find it in 2 hours of execution). In this problem, geGA was not able to find the deadlock in every run, but reached a hit rate of 90%.

In the Stable Marriage Problem, the geGA was not the fastest algorithm. DFS was faster but the resultant path quality was the worst for every instance of the SMP. BFS was the slowest algorithm and gave the best error trails up to 4 couples. BFS ran out of memory with 5 couples in the SMP. geGA delivered good results, both performance wise and quality wise. Again, its memory requirements are bigger than DFS and smaller than BFS. Notice that although achieving good results, geGA couldn't deliver a 100% hit rate, finding a deadlock in 47 of the 50 runs.

From these experiments one can clearly see that geGA poses a problem regarding the memory requirements. GAs are population based and as such represent many solutions at any given generation. All the states in those solutions must be expanded to be evaluated. Unfortunately, JPF does not allow to move forward through a known transition. In-

stead, the method that is used is to expand the first state, save the new state in memory, go back, expand the second state, and so on. After all states are expanded and stored in memory, one can instruct JPF to go directly to a saved state. Fortunately, geGA does not need to remember previously visited states. This allows us to free all the memory used to store those states. We use this as last resort because freeing the memory means we might need to expand some states again which consumes time. In these experiments we cleared the memory whenever the available memory became less than 100 MB. This explains why geGA never consumes more than approximately 400 MB of memory.

We compared our results to Godefroid's results in [7]. In order to be able to make this comparison fair, we had to change the DiningPhilosophers implementation. Godefroid's implementation chooses non-deterministically the fork to pick-up first, meaning that a philosopher can pick his left fork first and then the right fork, or can choose to pick the right fork first and only then the right fork. We implemented this non-determinism using the `randomBool()` method of the `Verify JPF` class. We used the same number of iterations Godefroid used as stopping criterion (50 iterations) and the same number of philosophers: 17. We used a smaller population of only 50 individuals, compared to 200 used by Godefroid. Table 5 summarizes our results. Due to differences in the environment used, we cannot compare the times nor the depth of the error directly. We can compare the hit rate, where geGA shows a clear improvement over Godefroid's GA, specially taking into consideration that geGA had to look deeper to find the deadlock).

Table 5: Comparison of geGA vs. Godefroid's GA

Algorithm	Errors / Runs	Time(s)	Depth
Godefroid GA	26/50	177	65
geGA	48/50	101	170

5. THE MEMORY OPERATOR

We now introduce a novel operator, called *memory operator* (MO), to allow the path to extend indefinitely while preserving the memory required to evaluate individuals.

The memory required to evaluate a large individual is a direct consequence of the number of states the model checker

Table 3: Results of the algorithms with DiningPhil

Search Method	Number of Philosophers	Time to 1 st error (s)	Total Time (s)	Depth of 1 st error	Final Depth	Memory (MB)	errors / runs
DFS	4	(+)3.00	(+)3.00	(-)16.00	(-)16.00	(+)24.00	1/1
BFS	4	(+)3.00	(+)3.00	(-)16.00	(-)16.00	(-)31.00	1/1
geGA	4	0.44	1.00	16.00	16.00	31.11	45/50
DFS	8	(+)132.00	(+)132.00	(-)33.00	(-)33.00	(+)39.00	1/1
BFS	8	-	-	-	-	-	0/1
geGA	8	1.42	6.25	33.00	33.00	76.00	45/50
DFS	16	-	-	-	-	-	0/1
BFS	16	-	-	-	-	-	0/1
geGA	16	44.40	118.87	65.00	65.00	408.07	48/50

Table 4: Results of the algorithms with SMP

Search Method	Num. Couples	Time to 1 st error (s)	Total Time (s)	Depth of 1 st error	Final Depth	Memory (MB)	errors / runs
DFS	3	(+)0.00	(+)0.00	(+)39.00	(+)39.00	(+)25.00	1/1
BFS	3	(+)2.00	(+)2.00	(+)22.00	(+)22.00	(+)35.00	1/1
geGA	3	0.02	3.78	28.16	23.04	54.06	50/50
DFS	4	(-)1.00	(+)1.00	(+)67.00	(+)67.00	(+)31.00	1/1
BFS	4	(+)84.00	(+)84.00	(+)30.00	(+)30.00	(+)397.00	1/1
geGA	4	2.34	10.12	45.88	34.36	104.12	50/50
DFS	5	(+)2.00	(+)2.00	(+)103	(+)103	(+)34	1/1
BFS	5	-	-	-	-	-	0/1
geGA	5	2.55	10.36	55.7	49.62	119.62	47/50

has to expand/generate. In the GA described before, the model checker started with the initial state (there is only one in every model) and expanded it to follow the selected transition and repeat the process until all transitions defined by the individual had been traversed. As the population evolves and grows, the first transitions in the individual tend to stabilize, but the model checker still has to evaluate them every generation. We propose saving the resulting states of those stable first transitions in memory slots and use them as the starting point for next generations. The advantages are obvious: less memory and time are required to evaluate an individual and the path can maintain a constant growth without requiring more memory. There are, of course, disadvantages: part of the search space is being discarded, and a good solution might be in that part.

To avoid discarding search space where a good solution might be found, we suggest that an individual may always use the initial state as a starting point for its path. We use the memory slot 0 to store that initial state. Also, we propose that a selection operator be used to select which states to put in the rest of memory slots. If that selection operator is too elitist, we may easily fall in a local maxima. If that selection is well balanced, however, the algorithm becomes less greedy and more capable of leaving that local maxima. Figure 3 shows how the search space is segmented and where the search concentrates as the memory operator is used. In this figure, each triangle represents the model state graph with the top of the triangle representing the initial state. Each triangle represents a moment in time, with the time increasing from left to right. Each segment

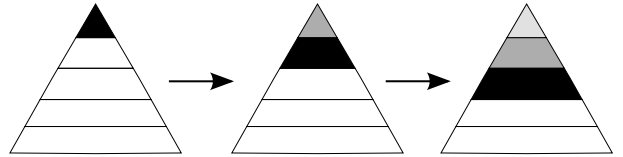


Figure 3: Segmentation of the search space and concentration of the individuals at 3 different moments. The darker the segment is, the higher the individual concentration is on that segment.

of a triangle represents a subgraph having a minimum and maximum depth. This operator is based on the, so called, missionary technique used in [1] for reaching deep graph regions.

5.1 Using the Memory Operator

Our use of the memory operator is as follows: the geGA executes as described before using a relatively small maximum chromosome length. After a predetermined number of iterations have been executed the memory operator selects some of the individuals and stores their final states in the memory slots and removes all other states from memory. This does not pose a problem for the counterexamples because JPF stores in each state the complete path of choices made to reach that state, which means that having a state is enough to describe the path leading to it. Then geGA resumes its normal operation creating an execution cycle.

For the memory operator to work, each individual must be able to choose its starting state. That could be done by storing in the chromosome which memory slot is to be used. Fortunately, our encoding of the individual described in Section 3.1 can still be applied. However, the first element in the chromosome is not used to describe a transition from the initial state, but to select the starting state. Since it is a floating-point number between [0..1) we just need to multiply it by the number of memory slots to determine which will be the starting state of the individual’s path.

5.2 Experimental Results with the MO

In this section we will try to find deadlocks on the Dining Philosopher and Stable Marriage problems using geGA with and without the memory operator. We will use larger problem instances than the previous to verify if the MO helps with very large graphs. We will then compare the results obtained using both algorithms.

Many more iterations are needed when using the memory operator (geGA^{MO}) than when not using it (gaGA) because the individual length is much smaller than required and the individuals must reach deep regions in the states graph. To make both algorithms comparable we decided to use the same computational effort on both algorithms. This means that, contrary to our previous experiments, our stopping criterion now is the computational time (measured in seconds). Table 6 shows the configuration used in each of the problems. Notice that we must estimate a maximum length of the chromosome when we are not using the MO, but we always use the same size (50 state transitions) when using MO. The MO will make the individuals move in the search-space to reach the goal at a deeper depth. We also use the same number of memory slots as individuals in the population. This allows to store a different starting position to each individual. We also chose to use 20 iterations as the MO frequency, which will allow the individuals to move further rather quickly, while having a little time to optimize the solutions in each search-space segment. With SMP we have used only 10 iterations as the frequency because a previous study showed it to yield better results. The selection method, crossover, and mutation parameters are the same as described in the previous section.

In Table 7 we present the results obtained using geGA with and without the memory operator. In this case we use the Kruskal-Wallis test [13] for checking the statistical significance of the differences, since we are comparing two samples.

The DiningPhilosophers problem was very well tackled with geGA^{MO}. As seen in Table 7 geGA^{MO} is faster to find a deadlock, requires less memory than geGA and has a better hit rate (geGA^{MO} found an error in every run on both instances of the problem). With 16 philosophers, however, geGA provided the error trail with better quality. With 32 philosophers geGA only found one deadlock in one of the 50 runs of the algorithm.

With the DiningPhil problem the geGA^{MO} is again faster, consumes less memory and is more effective (better hit rate) than geGA.

Finally, with SMP we have the opposite situation. Table 7 shows that not only geGA was the fastest algorithm of the two to find the path to the deadlock, but it found shorter paths and had a better hit rate. geGA^{MO} still used less memory but had more difficulty to find the deadlock.

Table 6: Configuration of the geGA with and without the memory operator

Algorithm	Problem	Size	Parameters
Common to all			Pop. Size 50
geGA	DiningPhilosophers	16	Max Length 120
			Stop At 60
geGA	DiningPhilosophers	32	Max Length 260
			Stop At 60
geGA ^{MO}	DiningPhilosophers	16	Max Length 50
			MO Slots 50
			MO Freq. 20
			Stop At 60
geGA ^{MO}	DiningPhilosophers	32	Max Length 50
			MO Slots 50
			MO Freq. 20
			Stop At 60
geGA	DiningPhil	16	Max Length 70
			Stop At 60
geGA ^{MO}	DiningPhil	16	Max Length 50
			MO Slots 50
			MO Freq. 20
			Stop At 60
geGA	SMP	8	Max Length 140
			Stop At 20
geGA ^{MO}	SMP	8	Max Length 50
			MO Slots 50
			MO Freq. 10
			Stop At 20

geGA^{MO} fulfills its promise of being able to tackle with larger search-spaces. The segmentation of the search-space allows the algorithm to be faster and consume less memory. However, there is a compromise between these advantages and the quality of the solution.

The use of the MO is not a good solution to every problem as we have seen with SMP. In this problem, the last steps depend greatly on the choices made in the beginning. The MO segments the search space and geGA^{MO} tries to find the best solution in each segment, starting from the previous segment solutions. However, it may be necessary to have a bad (partial) solution in the first segments to find a good (global) solution on the last segments. Further study is required to reach a definitive conclusion, but a better fitness function might help for these problems, instead of the generic one we used.

6. CONCLUSION AND FUTURE WORKS

We have presented here a genetic algorithm for the problem of finding safety errors in concurrent programs. This algorithm, called geGA, has been implemented inside the Java Pathfinder model checker, that is able to check Java programs. We compared this algorithm to the standard exhaustive search algorithms, DFS and BFS, using them to detect deadlocks (one example of a safety property) in large concurrent programs. We have shown that while it generally requires more time than DFS to reach an error, the error trails generated by geGA are smaller and therefore simpler to use in the debugging of the program. geGA is also faster

Table 7: Results of geGA and geGA^{MO} with the problems

Problem	Search Method	Time to 1 st error (s)	Total Time (s)	Depth of 1 st error	Final Depth	Memory (MB)	errors / runs
DiningPhilosophers 16	geGA	23.56	60.54	114.10	99.00	408.48	48/50
	geGA ^{MO}	(+)12.82	60.06	(+)125.84	(+)114.9	(+)106.34	(-)50/50
DiningPhilosophers 32	geGA	51.00	63.00	256.00	256.00	417.00	1/50
	geGA ^{MO}	(+)36.26	60.00	(-)260.02	(+)248.06	(+)154.88	(+)50/50
DiningPhil 16	geGA	41.67	60.00	65.00	65.00	408.00	6/50
	geGA ^{MO}	(+)19.03	60.00	(-)65.00	(-)65.00	(+)56.83	(+)35/50
SMP 8	geGA	2.58	20.42	126.88	104.12	267.54	50/50
	geGA ^{MO}	(+)13.40	20.00	(-)134.45	(+)122.34	(+)62.60	(+)38/50

than BFS and the lengths of the error trails that geGA obtains are very close to the BFS' ones. Besides, it can be used in larger problems, where BFS can not be used due to the required amount of memory. There are, however, problems remaining with geGA. One problem lies in the fact that one must limit the growth of the chromosomes in the individuals. This requires having an a priori estimate of the length of the error trail, which is frequently impossible to know. The other problem is the memory requirements of this algorithm. We proposed a novel operator, the memory operator, and have shown that using it with our geGA, a combination we called geGA^{MO}, could solve those problems while being able to tackle with even larger problems.

Some research can still be done to improve our algorithms. It is possible to modify geGA to be usable in the detection of liveness properties. That would imply adding a second phase to the Algorithm that, instead of looking for an error state, would look for a cycle including the state found in the first phase. Early studies we made suggest that the frequency of the memory operator execution and the maximum chromosome length affects the compromise between speed, memory and quality of the solutions: low frequencies and bigger chromosomes leads to better quality of the solutions at the expense of slower execution. We plan to study this issue in the future.

7. ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish Ministry of Education and Science and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project), and also by European CELTIC through the Spanish Ministry of Industry funds from FIT-330225-2007-1 (the CARLINK project).

8. REFERENCES

- [1] E. Alba and F. Chicano. Finding safety errors with ACO. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1066–1073, New York, NY, USA, 2007. ACM Press.
- [2] E. Alba and J. Troya. Genetic algorithms for protocol validation. *Proceedings of the 4th conference on Parallel Problem Solving (PPSN IV), Berlin, Germany, September, 1996*.
- [3] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.
- [4] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. Technical report, Pittsburgh, PA, USA, 1993.
- [5] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, 1999.
- [6] E. Dijkstra. Hierarchical ordering of sequential processes. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, 2002.
- [7] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):117–127, 2004.
- [8] G. Gopalakrishnan. *Computation Engineering: Applied Automata Theory and Logic*. Springer US, 2006.
- [9] A. Groce and W. Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.
- [10] D. Gusfield and R. Irving. *The stable marriage problem*. MIT Press, 1989.
- [11] A. L. Lafuente. Symmetry reduction and heuristic search for error detection in model checking. Workshop on Model Checking and Artificial Intelligence, August 2003.
- [12] L. Lamport. Verification and specification of concurrent programs. *A Decade of Concurrency: Reflections and Perspectives: REX School/symposium, Noordwijkerhout, the Netherlands, June 1-4, 1993: Proceedings*, 1994.
- [13] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.
- [14] M. Yagiura and T. Ibaraki. On metaheuristic algorithms for combinatorial optimization problems. *Systems and Computers in Japan*, 32(3):33–55, 2001.