Concurrent Java Test Generation as a Search Problem

Yaniv Eytani [1]

Computer Science Dept., University of Haifa

Abstract

A Random test generator generates executable tests together with their expected results. In the form of a noise-maker, it seeds the program with conditional scheduling primitives (such as yield()) that may cause context switches. As a result different interleavings are potentially produced in different executions of the program. Determining a-priori the set of seeded locations required for a bug to manifest itself is rarely possible.

This work proposes to reformulate random test generation of concurrent Java programs as a search problem. Hence, it allows applying a set of well known search techniques from the domain of AI to the input space of the test generator. By iteratively refining the input parameters fed to the test generator, the search process creates testing scenarios (i.e. interleavings) that maximizes predefined objective functions. We develop geneticFinder, a noise-maker that uses a genetic algorithm as a search method. We demonstrate our approach by maximizing two objective functions: the high manifestation rate of concurrent bugs and the exporting of a high degree of debugging information to the user. Experimental results show our approach is effective.

1 Introduction

The increasing popularity of concurrent programming on the Internet as well as on the server side has brought the issue of concurrent defect analysis to the forefront. Java's scheduling mechanism introduces non determinism to the program's executions (each execution is called an interleaving). The set of possible interleavings is huge, and it is not practical to try them all. Only a few of these interleavings actually produce concurrent faults. Thus, the probability of producing a concurrent fault is very low. As a result, defects

[1] The author gratefully acknowledges the support of the Caesarea Edmond Benjamin the Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science, at the University of Haifa. Email: ieytani@cs.haifa.ac.il Web: http://cs.haifa.ac.il/~ieytani/

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

such as unintentional race conditions and deadlocks are difficult and expensive to uncover and analyze and often escape to the field.

A related problem is that the JVM scheduler is usually deterministic; executing the same tests many times will not help, because the same set of interleavings will be created. This is true for simple and average complexity tests re-executed in a similar environment, regardless of the environment. The problem of testing multi-threaded programs is compounded by the fact that tests that reveal a concurrent fault in the field or in a stress test are usually long and run under different environmental conditions. As a result, such tests are not necessarily repeatable, and when a fault is detected, much effort must be invested to recreate the conditions under which it occurred.

In previous work, we developed the raceFinder tool, a heuristic noise-maker for Java that was found effective in uncovering concurrent bugs [2]. RaceFinder seeds the program with conditional synchronization primitives that potentially change the runtime interleavings the JVM scheduler produces and increases the probability that concurrent bugs manifest. This capability, however, strongly depends on raceFinder's set of inputs (called a "configuration"). A configuration is a subset of the program locations and associated parameters that determine the amount of scheduling noise (seeded context switches and delays) to be applied. The number of possible configurations can be very large and only a small subset may uncover bugs. Some configurations are even not feasible for testing. Different programs usually have different sets of configurations that are effective for the purpose of finding bugs and determining them a-priori is rarely possible.

This work proposes to reformulate random test generation of concurrent Java programs (in the form of noise-making [5]) as a search problem. A random test generator generates executable tests together with their expected results. It receives as input test directives or specifications that tell it what kinds of tests are required. Treating test generation as a search problem allows applying a set of well known search techniques from the AI domain to the input space of the generator. Changing the inputs of the generator changes its output, namely the conditional scheduling noise inserted to the program's execution and potentially changes the interleaving generated. Thus, the search process iteratively tunes the inputs to create testing scenarios (i.e. interleavings) that maximize predefined objectives functions over a single execution (or possibly a group of executions).

To demonstrate our approach we developed a new variant of the raceFinder tool (called geneticFinder), based on the use of a genetic algorithm as the search method. Genetic algorithms [13] (GAs hereafter) search for optimal or near optimal solutions by sampling the search space at random and creating a set of possible solutions (chromosomes) called "population". Each chromosome has an assigned fitness. These chromosomes undergo either recombination, mutation or survive and are chosen with a probability that depends on their fitness level to evolve into a new generation of solutions.

GeneticFinder currently aims at maximizing two predefined objective functions: (1) achieving a high probability that the bug manifests in every execution of the program, and (2) exporting a high amount of debugging information to the user. The latter is achieved by applying high degree of scheduling noise on a minimal set of variables and program locations. Both objective functions correlate well (this experience is drawn from [2][6]) and we refer to configurations that maximize both functions as effective configurations.

Previous works were aimed specifically at either manifesting bugs [2] or achieving coverage objectives for the purpose of creating effective regression suites [10]. This work generalizes this notion by providing the flexibility to define various new objective functions (or a weighted combination of these functions). In addition, our framework could easily interact with other testing tools from different paradigms such as static analysis [4], race detection [1][16] and model checking [18]. One can use information from either static analysis or dynamic analysis (such as race detection) as starting points to the search process. Other possibility is to use the noise-maker to provoke potential incorrect behavior of race conditions that are bugs (as opposed to benign races). By defining the appropriate objective functions, the generator's output could be used to guide run-time analysis of tools such as a model checker (as suggested in [14]). The results of the model checker analysis can then be fed back to the noise-maker and initiate another iteration of the search process. We intend to explore such possibilities in future work.

We experimented with a set of programs taken from a publicly available benchmark [7]. Each program's bugs are documented, and so we could evaluate our approach to show it is effective.

The paper is organized as follows: Section 2 discusses concurrent test generation with the raceFinder tool. Section 3 formulates noise-making as a search problem. Section 4 provides the implementation details of the genetic variant of raceFinder. Section 5 presents experimental results and analysis. Section 6 lists our conclusions and future work.

2 Concurrent test generation with raceFinder

When a concurrent bug manifests either in a small number of the interleavings or at ones that are not produced under the scheduling policy of the JVM, it may occur with a very low probability or not at all. To increase the probability such bugs manifest we had previously developed the raceFinder tool [2], a random test generator for concurrent Java programs.

In a typical raceFinder user scenario, a given functional test t is repeatedly executed against the program P, when the latter is executed. In each execution the program is seeded with conditional synchronization primitives (such as yield()) at various program locations. The seeded primitives may cause context switches, and as a result potentially different interleavings are produced. This process creates irregular testing scenarios and increases the

probability that concurrent bugs are uncovered (elaborated in [2][5][17]).

RaceFinder performs a heuristic search based on a two-level scheme. The first level determines, for each concurrent event (program locations that access shared variables, referred to as events [2]), whether or not to force context switches using the seeded primitives. At the second level, different seeded primitives are applied to the chosen concurrent events. Probabilistic parameters control the number of context switches and delays (referred to as noise or scheduling noise) introduced at run-time.

2.1 Probabilistic models for noise-making

Previous work [2] shows that the policy of applying noise to a single variable related to the concurrent bug significantly increases the probability that the bug manifests itself in comparison to randomly executing context switches over all of the program's events (referred to as white-noise). To justify seeding a subset of the program's events, recent work [6] classifies the program's events as good, bad and neutral, based on the effect of applying a context switch at each event (before or after it is executed). Other types of conditional seeding are possible and will be discussed in the following section. We briefly motivate the proposed classification in the next paragraphs.

Taxonomy of concurrent bugs [8] [15] and research of data races [16] and atomicity [12][19] indicate that a concurrent bug manifests when a sequence of events occur in a specific order. This sequence usually involves events from different threads. Hence, a context switch is required after (or before) some of events in the sequence execute, for the bug to manifest. We refer to such events as good events. Experimental studies show that having too many context switches tends to mask the concurrent bug [2]. Hence, for the sequence to expose the concurrent bug, it is best that no context switches occur before or after most other events in the sequence. We refer to latter as bad events. Thus, for a given concurrent bug, we distinguish between three types of events:

- Good events – events in which a context switch increases the probability that a concurrent bug manifests.
- Bad events – events in which a context switch decreases the probability that a concurrent bug manifests.
- Neutral events – events in which a context switch does not impact the probability that a concurrent bug manifests.

In the following example

Thread 1	*Thread* 2
$(1)If(x! = 0)\{$	$(3)x = 1;$
$(2)y = 1/x;$	$(4)x = 0;$
$\}$	$(5)z = 8$

Events (1) and (3) are good since a context switch executed after these events increases the probability a division by zero occurs. However, if the program is changed by replacing (1) with if (x == 0), then event (1) becomes bad.

Events can be naturally grouped as the set of events that access a shared variable, or the set of events that occur at a given program location. We will use the terms events, program locations and variables interchangeably to denote such sets according to the context.

For each shared variable or program location, we loosely assign a probability of whether it is as good, bad, or neutral. This assignment is done basing on the probability that good, bad, or neutral concurrent events associated with it are executed, assuming it is executed. Probability issues naturally arise since the specific interleaving the scheduler chooses depends on external and unpredicted events such as the current load of other programs running on the underlying machine. In addition, interaction between conditional seeding at different program locations can affect the scheduling in unpredicted ways, justifying the above assignments of probabilities.

2.2 Scheduling heuristics

As described earlier, raceFinder uses a two-level scheme to apply noise to the program. The first level seeds only a subset of the programs events in order to increase the probability a bug manifest. RaceFinder further increases the probability that the bug manifest at the second level by using scheduling heuristics (i.e. multiple context switches and delays) over performing a single context switch at every event of the chosen set. We motivate this using the following example:

Thread 1	*Thread* 2	*Thread* 3
(1)$If(x! = 0)\{$	(3)$z = 1;$	(5)$z = 1;$
"$noise$"	(4)$x = 0;$	(6)$x = 1;$
(2)$y = 1/x;$		
$\}$		

In the example, a scheduling heuristic is applied after event (1) is executed. Assume that the scheduler always first transfers control to thread 3 and once it finishes, returns the control to thread 1 (due to a predefined JVM scheduling policy). A single context switch after event (1) will not cause the bug to manifest. However, seeding (1) with a long delay (e.g., using wait()) could force the scheduler to execute thread 2 and increase the probability that the bug appears. This example intuitively illustrates that different scheduling heuristic can potentially increase or decrease the probability a bug manifest. We briefly describe three possible heuristics:

- Yield() - At a chosen event, the yield() heuristic randomly determines whether to execute the yield() primitive. This is repeated a random number

of times. The yield() scheduling heuristic has the advantage that no time out is used. If a time out is used, the CPU might be idle at times, causing an unacceptable performance impact (see the sleep() heuristics discussed in [5]).

- Wait() - The wait() heuristic is implemented by adding wait() statements at chosen events during the execution of the program. To prevent a deadlock, the thread's waiting is timed. Locks are acquired and released before and after the wait() primitive in order to flush values from the thread local area and the heap. As a result, other threads see the changes that occurred to shared variables.
- Halt one thread - An unlikely timing scenario (elaborated in [8]) occurs when a thread halts for a long time while other threads progress and change a shared variable value. This heuristic prevents a thread that reached a chosen program location from advancing. Only when the rest of the program cannot proceed and waits for this thread will the thread continue advance.

Clearly, other scheduling heuristics are possible (see [5][17] for details). However, we believe these heuristics capture the general notion of the irregularity that leads to concurrent bugs in real programs.

3 Concurrent test generation as a search problem

A search problem could be described as finding a set of parameters for which an objective function has a maximum or a minimum. When searching optimal or near optimal solutions to a problem within a large multi-modal search space, it may be infeasible to solve analytically. Meta-heuristics algorithms [20] are a set of techniques that are typically applied to such search spaces. Noise-making nicely fits with this definition, as a set of competing constraints (choosing good locations while not choosing bad locations) have to be balanced. Moreover, a solution is sometimes reached using a multi-objective function (such as a combining the configuration's bug manifestation rate and the configuration's size). Previous work on noise-making shows that heuristic measures can sometime be effective by narrowing down the search space, applying noise only to shared variables that are accessed frequently [2]. Formulating it as a search problem allows the use search techniques (for example, a genetic algorithm) that disregard such limiting assumptions.

Solving a search problem is associated with the three following steps: representing the problem (a model), defining a fitness function (the objective functions) and defining a set of manipulation operators (search operators). In the next sub-sections we describe the notations and objective functions required for applying meta-heuristic search to noise-making.

3.1 Notations

To formulate noise-making as a search problem we define a search space over a set of all possible multi-thread programs in Java. We denote $\mathcal{P}$ to be the set of all Java programs. For each program $p \in \mathcal{P}$ we denote the following sets:

- $\mathcal{V}$ be the set of the program's variables,
- $\mathcal{S}_V$ be the set of the program variables, where each one of the program's variable is represented by two elements, before and after it is accessed.
- $\mathcal{L}$ be the set of the program's program locations, where each one of the program's location is represented by two elements, before and after it is executed.

Intuitively, this partitions the program's events into sets that are based on the memory locations they access at various levels of granularity (the number of events each element represents). We differentiate the accesses to "before" and "after" the access itself due to potentially different results obtained by applying scheduling noise before or after these locations.

We denote the following sets:

(i) $\mathcal{H}$ to be a finite set of available scheduling heuristics, e.g. $\mathcal{H} = \{yield(), wait(), halt\ one\ thread\}$.

(ii) $\mathcal{N}$ to be a finite set of noise strength (probability of execution of seeded primitives and seeded delays length): $\mathcal{N} = \{0, \ldots, 100\}$.

(iii) $\mathcal{S}_L$, $\mathcal{S}_V$ and S_S to be the Cartesian products of three predefined sets:

- $\mathcal{S}_L = \mathcal{L} \times \mathcal{H} \times \mathcal{N}$
- $\mathcal{S}_V = \mathcal{V} \times \mathcal{H} \times \mathcal{N}$
- $S_S = \mathcal{S}_V \times \mathcal{H} \times \mathcal{N}$

Each seeding element S_i of the set $\mathcal{S}_L$ (or $\mathcal{S}_V$ and $\mathcal{S}_S$) intuitively represents a scheduling heuristic of type H_i applied at location $L_i \in \mathcal{L}$ (or $V_i \in \mathcal{V}$ and $S_{V_i} \in \mathcal{S}_V$). We use the terms variables to denote both variables and split access variables (elements of $\mathcal{S}_V$). For sake of simplicity, we assume each heuristic has only one associated noise strength parameter $N_i \in \mathcal{N}$.

Definition 3.1 Configuration set $\mathcal{C}$ is a lattice constructed as the as the powerset of the disjoint union of sets: $\mathcal{S}_L$, $\mathcal{S}_V$, S_S

Each configuration C_i is a subset of seeding objects of types: $\mathcal{S}_L$, $\mathcal{S}_V$ or $\mathcal{S}_V$. We abuse notation and write $C_i(t)$ to denote the number of times the configuration C_i had been executed after t program executions, and $f_b(C_i)$ to denote the number of times the bug manifested during the program's executions (seeded with the configuration C_i).

3.2 Objective functions

A search problem could be described as finding a set of parameters that maximizes a predefined objective function (or functions). An objective function characterizes what is considered to be a good solution by imposing an ordinal scale upon the individual solution it is applied on. At each step of the search process the objective function allows to choose one solution from a set of two or more solutions (configurations, for the problem at hand).

In this work, we will define two such functions: a function that maps each configuration to a probability that the concurrent bug manifests and a function that maps each configuration to a size representing the amount of debugging information it provides.

Definition 3.2 For every program P_i there exists a function $f_p : \mathcal{C} \rightarrow [0, 1]$, The function f_p maps each configuration to the probability that a concurrent bug will manifest, where seeding objects of the configuration C_i are applied during the execution of the program.

Definition 3.3 For every program P_i there exists a function $f_s : \mathcal{C} \rightarrow \overline{\mathcal{N}}$ (where $\overline{\mathcal{N}}$ extends over a finite prefix of the natural numbers), The function f_s maps each configuration to a size basing the amount of scheduling noise inserted by the configuration C_i and the way it is partitioned between the locations. The function f_s aim at concentrating a high degree of noise among a minimal number of locations.

Clearly, other objective functions are possible. For example, a function that maximizes the number of data races occurring over a single run or a function that maximizes a coverage criteria (as used in [5]). A natural extension of the objective functions proposed in this work could be functions that receive a set of configurations as their input. Such functions aim to maximize objectives for a group of configurations (over possibly a multiply number of executions). These functions could measure overall concurrent coverage criteria [10] or collect overall statistics about the program [11]. Consider, for example, a function that maximizes the total number of different data races occurring for a given number of tests to be such an objective.

The computation of the function f_s is given explicitly by a formula. The function f_p, however, is unknown at every single point. Thus, f_p must be derived by applying the Bayes rule to yield posterior probabilities using observations about the number of times a bug has manifested with each configuration (for elaborated details see [9]). We describe the process of evaluating both functions and details about the search operators in the next section

4 GeneticFinder

We implemented a variant of raceFinder that uses a genetic algorithm as a search method (call geneticFinder). Details of the representation, search

operators and fitness evaluation for this implementation follows.

4.1 Representation and operators

A genetic algorithm operates on a population of individuals representing possible solutions (chromosomes). These candidates, initially created randomly, are combined and mutated to evolve into a new generation of solutions, which may be fitter. Recombination provides a mechanism for mixing genetic material within the population. Mutations introduce new genetic material thereby preventing the search from stagnating. The next population of solutions is chosen from the parent and the offspring generations in accordance with a survival strategy that favors fit individuals.

In geneticFinder, each chromosome models one configuration using a three-level hash table. The first level in the hash table contains the heuristic names, used as a key to access the second level that contains, for each heuristic, the variables and program locations noised by this heuristic. Each heuristic's noise parameters (for each location or variable) are found in the third hash table level and are accessed in a similar manner.

Recombination is done by randomly choosing a pair of "parent" configurations. Each parent contributes a random subset of its heuristics (and then a subset of variables and program locations) to the new "offspring" configuration. When both parents add the same heuristic and location (or variable) to the offspring, its noise parameters are randomly chosen and bounded by the parents' values. Mutation is preceded by choosing a subset of a heuristic's variables or locations and removing them from the "offspring' configuration. This is done to create smaller configuration that still manifest the concurrent bugs. The "traditional" role of the mutation operator, i.e., to introduce new genetic material, is achieved by randomly creating a number of new configurations each generation.

4.2 Fitness evaluation

GAs success in finding an optimum solution strongly depends on the choice of a fitness function that directs the search along promising pathways. The fitness function is a weighted combination of objective functions and characterizes what is considered to be a good solution. Given two possible solutions, it determines which of them supplies a better set of desired properties. Here, the fitness function is a mixture of the functions defined in section 3.2. In the next paragraphs we describe how these functions are evaluated.

First, we discuss evaluation of the f_s function. A desired property for a good solution should be its ability to provide information about the causes of the bug in source code. Intuitively, the search process can be depicted as filtering out locations (and variables) not related to the bug until only a small group of related locations remains. Having a high degree of noise over the locations or variables in the remaining group usually correlates highly with

source code locations related to the bug. We evaluate f_s by combining two different functions: *size* and *entropy*.

The function *size* aims to prefer solutions that have a small number of "noised"locations or variables. Program locations are of smaller granularity that variables, since each variable groups together a set program's locations that perform access to a memory location associated with it (for example, all program locations accessing variable "x"). Thus we prefer solutions containing program location over solutions containing similar number of variables. Noise is applied to each variable or program location with different probabilistic strength. Hence, the function *size* calculates a weighted sum of the noise strength applied to the program locations and variables. It is evaluated as follows: Let $N_{i,j}$ be the value of N in the j'th seeding object of the configuration C_i where the elements of C_i are enumerated in an arbitrary manner. Let the function *elementSize* map each type seeding object to a predefined size (e.g. 1, 2 or 8 depending on the type of item j: location, split access variable or variable respectively). The function $size(C_i)$ is evaluated as follows:

$$size = \sum_{j=1}^{|C_i|} N_{i,j} \cdot elementSize\,(C_{i,j}) \tag{1}$$

This function maps configurations with a total high degree of noise to higher values. Of course, such preference does accurately represent the amount of debugging information a configuration contains (consider the cases where no noise is applied or noise it applied to many variables). Thus, there is a need to evaluate the way the noise is partitioned between the noised locations and variables. Preference is given to configurations having a high degree of noise distributed among a small number of locations (loosely similar to the entropy measure [3]). Thus, we denote this function as entropy and evaluate as follows:

$$entropy = -\sum_{j=1}^{|C_i|} \frac{N_{i,j}}{size\,(C_i)} \log\left(\frac{N_{i,j}}{size\,(C_i)}\right) \tag{2}$$

The function f_s is constructed as a combination of both size and entropy:

$$f_s = size\,(C_i) + \frac{1}{entropy\,(C_i)} \tag{3}$$

Integrating both functions into the fitness function leads the search process towards selecting configurations having fewer seeded program locations with high degree of noise applied on them.

In addition, we aim to prefer a configuration (or configurations) that manifest bugs with high probability. As the scheduler is assumed deterministic, we can assume that a given configuration (executed a sufficient number of times) manifest the bug at a certain unknown probability. We cannot estimate this probability before the bug actually manifests. We can, however, estimate an upper bound on that probability during testing and evaluate the fitness accordingly (this is derived by using Bayes rule). For example, if no bug was

found in the 100 first runs we assume that the probability is lower than 0.01. Accuracy depends on the number of times a configuration has been executed and we consider results basing on more executions to be more significant. We evaluate f_p as follows:

$$f_p = \frac{f_b(Ci)}{C_i(t)} + \log\left(C_i(t)\right). \tag{4}$$

4.3 Genetic search process

At the initialization phase of the search a random pool of configurations is created. The search progress is measured by generations. Each generation every configuration found in the pool is executed a predefined number of times. After a configuration complete executing, a fitness function is calculated for that configuration. When all configurations for a given generation have finished their execution they are sorted by their fitness level. A set of the highest ranked configurations remain unchanged in the pool to be executed again in the next generation. Another set, the lowest ranked configurations, is replaced with a set of new configurations created randomly. The remaining configurations form a third set. Configurations are randomly chosen from the latter set and the set of high ranking configurations to be mated and mutated as explained in the previous section. This process continues for a predefined number of generations determined before the search starts.

5 Experimental Results

We experimented with programs taken from the publicly available multi-threaded benchmark[7], containing programs with documented concurrent bugs. Each program has different variables and program locations associated with the bug and it is necessary to apply noise only at these locations for the bug to manifest itself. Prior knowledge of these locations is used only to evaluate the success of the search process. The programs of the benchmark report after each execution whether or not a bug had manifested and this information serves as the test results. Information about the program's variables, split access variables and program locations is collected by executing the program a number of predefined times before the search process starts.

To demonstrate our approach we focus on one example program. We analyze this program and discuss related experimental results in details. Results obtained from other programs we experimented with were similar.

5.1 Detailed example

We briefly describe an example program called distributive maximum. This program outputs the maximum value of an array of integers it receives as an input (called *integersArray*). The maximum is calculated in a distributive manner using a set of threads. Each thread is associated with one cell in the

array and compares the value of its associated cell to a globally shared variable, containing an updated global maximum value (called *globalMaximum*) found preceding to that point in program's execution. This results in the value of *globalMaximum* increasing monotonically until it reaches the maximal value of the array. When a thread begins to run, a local thread variable is initialized with the offset in the array of its associated cell (called *index*). During execution, each thread reaches the following code block:

```
(1) If (integersArray[index] > globalMaximum)

(2) globalMaximum = integersArray[index]
```

The above code contains two data races related to the variable *globalMaximum*. Thus, there are two possible irregular scheduling scenarios that lead to a bug manifesting (i.e. wrong calculation of the maximum): (a) a context switch occurs before *globalMaximum* is read in (1). In this scenario the value of *globalMaximum* is changed by another thread. However this change is not seen by the first thread and once it regains control the condition of its *if* statement could be incorrect. The bug occurs since the condition is not evaluated again and consequently a wrong lower value could be written to *globalMaximum*(b) a context switch occurs before *globalMaximum* is written in (2). In a similar manner, once the first thread regains control, it is possible that a higher value has already been written to *globalMaximum*. In this scenario *globalMaximum* might be written again with an incorrect lower value. Note that it is sufficient that either (a) or (b) occur for the bug to manifest itself.

5.2 Results and analysis

To test our approach we applied the genetic search process using different programs from the benchmark. We present detailed results for the distributive maximum program.

We experimented with a set of small populations (10-20 chromosomes). In our experiments the best configurations usually alternated between applying noise on either program location (1) or (2) (from the above code example) and other locations and variables not related to the bug. Very high ranked configurations (by overall fitness) usually contained significantly lower number of seeding objects. Given enough runtime (usually 20-30 minutes), the search process converges to configurations containing only "correct"program locations and also manifested the bug at high rate.

In the experiments we measured changes both to the value of the configuration size (using the *size* function) and the total quality of solutions obtained (using the *fitness* function) as search progresses. Progress was measured by the number of generations passed since the start of the search. We measured the following values for both functions:

- The value of the best configuration in each generation.
- The average value of all configurations in the configurations' pool.
- The average value of all configurations that remained unchanged in the configurations' pool in the past generation.

The configurations in the pool were sorted by their fitness level every generation.

We found that the results for all three groups behaved similarly. We noticed that genetic search process generally progressed towards narrowing down the number of unnecessary seeded variables and program locations. Smaller configurations became high ranked configuration every small number of generations. The overall fitness of the solutions improved constantly until convergence. Further improvement was only achieved by the GA running the same configurations over and over again and thus increasing the level of confidence in the probability of bug manifestation they predicted. During the run the fitness level would sometimes decrease by a small amount due to the fact that the number of bugs found is only an approximation of the configuration probability for finding bugs and thus it is sometimes incorrect (underestimates the correct probability).

In early experiments the search process sometimes converged to the single split access variable "*before globalMaximum*". Configurations containing only the split access variable had high manifestation rate of the bug that dominated the fitness calculation. To assure the search process converges into either program locations (1) or (2) and provides more debugging information we increased the weights of the variable and split access variable in the *elementSize* function (from 1,2,4 to 1,4,8 for program locations, split access variables and variables respectively).

To verify the fact that new configurations are created during the progress of the genetic process we measured the age of each configuration (i.e. number of generations since it was created). We noticed that different configurations participated in the search process (we measured the age of the highest ranked configuration and an average age of all configurations in the pool). After the search process started to converge to a solution the age of the configurations rises steadily. This behavior correlates with the fitness and size measured results.

6 Conclusions and future work

This work proposes to reformulate random test generation of concurrent Java programs (in the form of noise-making) as a search problem. Treating test generations as a search problem allows applying a set of well known AI search techniques to the input space of the generator. The search process iteratively tunes the scheduling noise to create testing scenarios that maximize predefined objectives functions over a single execution or possibly a group of executions.

To demonstrate our approach we developed geneticFinder, a new variant of the raceFinder tool, based on the use of genetic algorithm as the search method. GeneticFinder aims at maximizing two predefined objective functions, first achieving high probability that the bug manifests in every execution of the program and second to export high degree of debugging information to the user. Our experimental results show this approach is effective. Moreover, our developed framework generalizes over previous works aimed achieving specific goals by providing flexibility to define various objective functions or a weighted combination of these functions. In Addition, it could allow in the future easy interactions with other testing tools from different testing paradigms.

RaceFinder is an on-going research project. Further research is needed to formulate a stop criterion for the search and improve the fitness function to include more features. It is known that parallel execution of GA improves its results. We plan to experiment with such implementation in the future. We also plan to compare and combine GA with other nature inspired methods such as particle swarm optimization and ant colony optimization.

Acknowledgments.
I thank Sadek Jbara for implementing many features in geneticFinder, and Shlomo Berkovsky, Edward Furman, Larry Manevitz and Shmuel Ur for their help during the preparation of this paper.

References

[1] C. Artho, K. Havelund, A. Biere. "High-level data races". Software Testing, Verification & Reliability 13(4): 207-227 (2003)

[2] Y. Ben-Asher, Y. Eytani, and E. Farchi, "Heuristics for Finding Concurrent Bugs," Workshop on Parallel and Distributed Testing and Debugging, Nice, 2003.

[3] C. E. Shannon. "A Mathematical Theory of Communication". Bell System Tech. J. 27(1948), 379-423, 623-659.

[4] J.D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. "Escape Analysis for Java". Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999.

[5] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur. "Framework for Testing Multi-threaded Java Programs". Concurrency and Computation: Practice and Experience 15(3-5): 485-499 (2003).

[6] Y. Eytani."Efficient Framework for Finding Concurrent Bugs in Java". Master's thesis. Computer Science department, University of Haifa ,Submitted.

[7] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. "Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs". Concurrency and Computation: Practice & Experience, to appear.

[8] E. Farchi, Y. Nir, and S. Ur. "Concurrent Bug Patterns and How to Test Them." In Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.

[9] S. Fine, A. Ziv. "Coverage Directed Test Generation for Functional Verification Using Bayesian Networks."In proceeding of DAC, 2003

[10] S. Fine, S. Ur, A. Ziv. "Probabilistic Regression Suites for Functional Verification."In proceeding of DAC, 2004

[11] B. Finkbeiner, S. Sankaranarayanan and H. Sipma. "Collecting Statistics over Runtime Executions."In the Second Workshop on Runtime Verification (RV), Volume 70(4), Electronic Notes in Theoretical Computer Science. Elsevier 2002.

[12] C. Flanagan and S. N. Freund. "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs."31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), Jan 2004.

[13] D.E.Goldberg, R.Burch. "Genetic Algorithms in Search, Optimization, and Machine Learning."AW Publ., 1989.

[14] K. Havelund. "Using Runtime Analysis to Guide Model Checking of Java Programs". In proceeding of SPIN, 2000.

[15] B. Long and P. A. Strooper. "A Classification of Concurrency Failures in Java Components." In Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs."ACM Transactions on Computer Systems, 1997.

[17] S. D. Stoller. "Testing Concurrent Java Programs Using Randomized Scheduling."In the Second Workshop on Runtime Verification (RV), Volume 70(4), Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[18] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. "Model Checking Programs."International Journal on Robust Software Engineering 10(2), April 2003.

[19] L. Wang and S. D. Stoller. "Run-time Analysis for Atomicity."In the Third Workshop on Runtime Verification (RV), Volume 89(2), Electronic Notes in Theoretical Computer Science. Elsevier, 2003.

[20] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper and M. Shepperd. "Reformulating Software Engineering as a Search Problem."IEE Proceedings - Software 150(3): 161-175, 2003.