Cross-Entropy Based Testing

Hana Chockler*, Eitan Farchi*, Benny Godlin* and Sergey Novikov*
*IBM Haifa Research Laboratory
Mount Carmel, Haifa 31905, Israel
Email: {hanac,farchi,godlin,novikov}@il.ibm.com

***Abstract*—In simulation-based verification, we check the correctness of a given program by executing it on some input vectors. Even for medium-size programs, exhaustive testing is impossible. Thus, many errors are left undetected. The problem of increasing the exhaustiveness of testing and decreasing the number of undetected errors is the main problem of software testing. In this paper, we present a novel approach to software testing, which allows us to dramatically raise the probability of catching rare errors in large programs. Our approach is based on the *cross-entropy method*. We define a performance function, which is higher in the neighborhood of an error or a pattern we are looking for. Then, the program is executed many times, choosing input vectors from some random distribution. The starting distribution is usually uniform, and it is changed at each iteration based on the vectors with highest value of the performance function in the previous iteration. The cross-entropy method was shown to be very efficient in estimating the probabilities of rare events and in searching for solutions for hard optimization problems. Our experiments show that the cross-entropy method is also very efficient in locating rare bugs and patterns in large programs. We show the experimental results of our cross-entropy based testing tool and compare them to the performance of ConTest and of Java scheduler.**

I. Introduction

Software testing (also called *simulation-based verification*) is a family of analyses that involve an automatic or semi-automatic exploration of the state space of a program. Simulation-based verification is traditionally used in order to check the program with respect to some input vectors [2]. It is widely used today as a primary means for checking the correctness of programs. Thus,in simulation-based verification, the challenge of making the verification process as exhaustive as possible is crucial. Each input vector induces a different execution of the program, and a program is correct if it behaves as required for all possible input vectors. In the ideal world, a program would be tested on all input vectors. This approach, however, is infeasible even for medium-size programs, whereas today it is common to find programs with a few million lines of code. To make matters worse, reuse of components (that may have never been tested with the new use in mind) makes it possible to assemble products orders of magnitude larger, in the same time.

Since simulation-based verification is a technique that replaces the infeasible task of checking all input vectors, it is very important to increase the capability of simulation-based verification to find errors and to make the testing process as thorough as possible. The research in this area focuses on several different directions, which we describe in more detail in Section II. While the existing techniques have been successful in specific applications, none of them has proven successful in efficiently finding a large variety of rare bugs in large programs. The problem is especially acute in concurrent programs, which many have exponentially many possible behaviors resulting from different possible schedulings of threads.

In this paper we propose a new approach to testing of large programs, which is based on the *cross-entropy method*. The cross-entropy (CE) method is a generic approach to rare event simulation [19]. It derives its name from the cross-entropy or the Kullback-Leibler distance, which is a fundamental concept of modern information theory [13]. It is an iterative approach, and is based on minimizing the cross-entropy or the Kullback-Leibler distance between probability distributions. The CE method was motivated by an adaptive algorithm for estimating probabilities of rare events in complex stochastic networks [17]. Then, it was realized that a simple modification allows to use this method also for solving hard combinatorial optimization problems, in which there is a performance function associated with the inputs. The CE method involves an iterative procedure, where each iteration consists of two phases:

1) Generate a random data sample according to a specified mechanism.
2) Update the parameters of the random mechanism based on the data to produce a "better" sample in the next iteration, where "better" is chosen according to the predefined performance function.

A sample is evaluated according to a predefined performance function. The procedure terminates when the "best" sample, that is, a sample with the maximal value of the performance function (or, if the global maximum is unknown in advance, with a sufficiently small relative deviation), is generated. The CE method is used in many areas, including buffer allocation [1], neural computation [8], DNA sequence alignment [12], scheduling [14], and graph problems [18].

In testing, in order to make the CE method applicable we shift the focus from searching for bugs to focusing our attention on the most error-prone areas of a program. The following example helps to clarify the difference between the approaches. Assume that we are given a program, which we want to test for buffer overflow errors. A bug-oriented testing tool searches for executions in which the buffer is overflowed. These executions might be very rare and might escape our testing efforts. In cross-entropy-based testing, we

direct the execution to the areas in which the buffer reaches its maximal capacity. The advantage of this approach is that this area, while can be small, is not of negligible size, and can be discovered during random testing. Also, if there exists an erroneous execution in which the buffer is overflowed, it surely occurs in this area.

We argue that many common bugs and patterns which may contain bugs have a natural performance function. For example, in case of the buffer overflow, the natural performance function for an execution gives the value which is equal to the maximal size of the buffer in this execution. We describe many more examples of common bugs and bug patterns and their associated performance functions.

Our way of using CE method in testing is suitable for programs with many points of non-deterministic decisions. Informally, such programs create a large control-flow graph with many branching points, which allows us to view the testing setting as a variation of a graph optimization problem. While a serialized program can, in theory, have many points with non-deterministic decisions (for example, statements conditional on the result of coin-tossing), the most common example of such programs is concurrent programs. In concurrent programs, at each synchronization point the decision of which process (or thread) makes the next step is made by the scheduler and can be viewed as non-deterministic when analysing the program. In this work, we focus on concurrent programs and apply the CE method to errors related to concurrency (such as a way of scheduling the threads that leads to buffer overflow). Our work can, in theory, be extended to arbitrary non-deterministic programs. However, we believe that testing concurrent programs is the area in which CE method has the maximal advantage.

We implemented our algorithm in a tool named *ConCEnter*, which stands for "Concurrent Cross-Entropy based Error Revealer". We provide the details of implementation and the experimental results of running ConCEnter on several programs with different bug patterns[1]. We compare the performance of ConCEnter with the performance of ConTest, a randomized testing tool for concurrent programs developed in IBM [9]. We show that ConCEnter performs better than ConTest by an order of magnitude.

II. Related Work

In this section, we briefly survey the existing work in the area of improving the capability of simulation-based verification to find bugs.

One direction is to focus on bugs that manifest themselves under a heavy load on the system during runtime, such as buffer overflows, timeouts, etc. The traditional method of testing for these bugs is called stress-testing. The method takes single-thread tests and creates stress tests by cloning them many times and executing them simultaneously (see, for example, [10]).

[1]The executable of ConCEnter with some test programs is available from the authors on request.

Another approach to testing is to create random interference in the scheduler which leads to randomized interleavings of the thread executions (these tools are also called "noise makers") [9], [22]. The interference is created by injecting noise to the scheduler forcing the scheduler to create uncommon interleavings, which are rare during usual execution. However, a chance of finding a very rare bug is small, since the random distribution (created by injecting noise) is not adjusted to a specific pattern.

The most commonly used method to maximize the exhaustiveness of testing is *coverage estimation*. There is an extensive research simulation-based verification community on coverage metrics, which provide a measure of exhaustiveness of a test [23]. Coverage metrics are used in order to monitor progress of the verification process, estimate whether more input sequences are needed, and direct simulation towards unexplored areas of the program. Essentially, the metrics measure the part of the design that has been activated by the input sequences. For example, code-based coverage metrics measure the number of code lines executed during the simulation. There is a variety of metrics used in coverage estimation (see, for example, [7], [15], [16], [23], [26]), and they play an important role in the design validation effort [24]. Still, since there is no special effort directed to a possibly erroneous pattern in the code, rare bugs can remain in the code even after testing reached a high coverage of it.

Existing testing methods that look for predefined patterns in programs rely on heuristics specifically targeted to a pattern, for example, delaying read/write instructions [3], or denying the application certain services [25]. This approach is quite efficient for finding predefined patterns, however, it requires a different heuristic to be invented for each new bug.

An approach based on exploiting genetic algorithms [11] in testing led to several prototypes of testing tools implementing this idea (see, for instance, [20], [21]). These algorithms, however, normally require a very long execution time, and they have also been shown to be ineffective in finding rare and hard-to-reproduce bugs.

III. Preliminaries

A. The Cross-entropy Method in Optimization Problems

The cross-entropy (CE) method was developed in order to efficiently estimate probabilities of rare events, where an event e is considered a rare event if its probability is very small, say, smaller than 10^{-5}. In this method, we are given a very large probability space and a function S from this space to $\mathbb{R}^+$, and we say that e occurs on an input X (from the probability space) if $S(X) > \gamma$ for some predefined value $\gamma \in \mathbb{R}^+$. Since the space is very large, it is infeasible to search it exhaustively, therefore the estimation of the probability l of e is made by sampling. A straightforward way to estimate l is to draw a random sample according to the given probability distribution f on inputs, and then estimate l by examining the sample. The problem is, that when e is a rare event, the sample might have to be very large in order to estimate l accurately. A better way is to draw the sample according to some other

probability distribution g that raises the probability of e. The ideal probability distribution here would be g_l, which gives the probability 0 to inputs that do not contain e. The CE method attempts to approximate g_l. The distance between two distributions that is used in this approximation is the Kullback-Leibler "distance" (also called cross-entropy). The Kullback-Leibler "distance" between g and h defined as:

$$\mathcal{D}(g,h) = E_g \ln \frac{g(X)}{h(X)} = \int g(x) \ln g(x) dx - \int g(x) \ln h(x) dx.$$

Note that this is not a distance in the formal sense, since in general it is not symmetric. Since g_l is unknown, the approximation is done iteratively, where in iteration i we draw a random sample according to the current probability distribution f_i and compute the (approximate) cross-entropy between the f_i and g_l based on this sample. Then we construct f_{i+1} by updating f_i based on the cross-entropy result. The reader is referred to Appendix for more formal explanation and to the book on cross-entropy for the complete description of the method [19].

Our method of testing large program is based on the application of CE method to graph optimization problems. In these problems, we are given a (possibly weighted) graph $G = \langle V, E \rangle$, and the probability space is the set of paths in G represented by the sets of traversed vertices. This setting matches, for example, the definitions of the traveling salesman problem and the Hamiltonian path problem in the context of CE method. This is also the setting which we use in our approach. Our goal is to find executions of the program under test in which the system enters a predefined state. For example, in order to test stack overflows, we are interested in executions in which the stack occupancy is maximal. As we explain in Section III-B, we represent the program under test as a graph. Then, paths in this graph correspond to executions of the program. We define S (which we call *a performance function*) so that $S(X)$ reaches its global maximum on paths in which the system is brought to the state we are testing. As we argue in Section IV-C, many common bugs and bug patterns admit a natural performance function. The probabilities can be assigned to edges or to vertices of the graph (depending on how the sample is drawn). W.l.o.g, we assume that the probabilities are assigned to edges. We start with uniform probability distribution. In each iteration i, we sort the sample $\mathcal{X}_i = \{X_1, \ldots, X_N\}$ generated in this iteration in ascending order of their performance function values. That is, $S(X_1) \leq S(X_2) \leq \ldots \leq S(X_N)$. For some $q \ll 1$, let

$$Q(\mathcal{X}_i) = \{X_{\lfloor (1-q)N \rfloor}, X_{\lfloor (1-q)N+1 \rfloor}, \ldots, X_N\}$$

be the best q-part of the sample. The probability update formula in our setting is

$$f'(e) = \frac{|Q(e)|}{|Q(v)|}, \tag{1}$$

where $e \in E$ is an edge of G that originates in the vertex v, $Q(v)$ are the paths in $Q(\mathcal{X}_i)$ which go through v and $Q(e)$ are the paths in $Q(\mathcal{X}_i)$ which go through e. Intuitively, the edge e "competes" with other edges that originate in v and participate in paths in $Q(v)$. We continue in the next iteration with the updated probability distribution f'. The procedure terminates when a sample has a relative standard deviation below a predefined threshold parameter (usually between 1% and 5%).

Remark 3.1 (Smoothed updating): In optimization problems involving discrete random variables, such as graph optimization problems, the following equation is used in updating the probability function instead of Equation 1:

$$f''(e) = \alpha f'(e) + (1-\alpha) f(e), \tag{2}$$

where $0 < \alpha \leq 1$ is the *smoothing parameter*. Clearly, for $\alpha = 1$ we have the original updating equation. Usually, a value of α between 0.4 and 0.9 is used. The main reason why the smoothed updating performs better is that it prevents losing a good sample forever (if one of its edges is assigned 0 in one of the iterations).

B. Definitions

In this work we focus on finite multi-threaded programs. Let t stand for the number of threads in the program. As all executions of the program are finite, we can talk about the unwound code of the program. That is, the code of each loop is duplicated the maximum number of times it may run and all function called are embedded in the code of the main function.

Definition 3.2 (PL): Program location (PL) is a line number in unwound code of the program.

Definition 3.3 (CFG_i): A *control flow graph* (CFG_i) of thread i ($i \in [t]$) is a directed graph $G_i = \langle L_i, E_i, \mu_i \rangle$ where L_i is the set of all program locations in the unwound code of the thread, E_i is the set of edges such that $\langle v, u \rangle \in E_i$ if a statement at location u can be executed immediately after the statement at location v, and $\mu_i \in L$ is the initial program location of the thread.

Definition 3.4 (PLV): Program location vector (PLV) $\vec{v}$ is a t dimensional vector such that for each $i \in [t]$ $v_i \in L_i$.
At each moment m during the execution of the program, we say that the execution is at PLV $\vec{v}$ if for each $i \in [t]$ v_i is the next program location to be executed in thread i.

Clearly, the set of all PLVs is equal to the cross product of the L_is.

Definition 3.5 (JCG): The joint control graph (JCG) of the tested program is a graph $\langle V, E \rangle$ whose vertices are the PLVs. There is an edge in the JCG between vertices u and w if there exists an execution path in which w is the immediate successor of u.
Note that at each moment only one thread can make a move. Therefore, the branching degree of each vertex is at most t. Since the code is unwound, every statement in it is executed at most once and the statements are executed in the increasing order of their program locations. Therefore, JCG is a finite directed acyclic graph (DAG). The source vertex of the JCG is a PLV which is composed of the initial PL μ_i for each thread i.

Definition 3.6 (PF): Probability function $PF : V(JCG) \times [t] \mapsto [0, 1]$ such that the probability sum over the outgoing edges of each vertex is 1.

This function defines for each vertex v and each of its outgoing edges e_i the probability of the thread i to advance when the execution reaches v. If not all threads are enabled at v, we take the relative probabilities of the enabled ones. If $T_{en} \subseteq [t]$ is the set of the enabled threads at this moment then the relative probabilities are:

$$RP(v, i) \doteq \begin{cases} \frac{PF(v,i)}{\sum_{j \in T_{en}} PF(v,j)} & \text{if } i \in T_{en} \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.7 (Visible thread state): A *visible thread state* (VTS) s of a thread i is the pair $\langle v, \sigma, \tau \rangle$ where $v \in L_i$ denotes the current PL, σ denotes the valuation of all local variables, and τ denotes the valuation of the global variables accessed by the statement at program location v.

Intuitively, the set of visible thread states are all possible states of the system as visible by the thread when it accesses various parts of this system.

We assume that correct locking policy was implemented in the program (this can be checked statically). Therefore, any VTS that can be reached by any program execution, can be also reached by some program execution when context switches are allowed only at locking or unlocking statements. Thus, under such context-switch policy the only vertices of the JCG which may have more than one successor are those PLVs with program locations of lock or unlock statements. Then, all vertices with a single successor can be collapsed without losing any synchronization information, and therefore we can assume that the JCG graph contains only program locations of locking/unlocking statements [2]. This restriction on the context-switch policy was introduced in [4], [5].

For a single execution of the program, we call the sequence of vertices of JCG that it visits an *execution path* in JCG.

IV. Cross-entropy for testing

A. Algorithm

The algorithm starts with choosing the performance function S according to the state of the system to which we want to converge (we list some of the more natural performance functions in Section IV-C). A concurrent program is viewed as a JCG, which is a product of DAGs of its threads. Each DAG corresponds to the unwound program of a thread. The JCG is not constructed in advance - edges between PLVs are discovered during the execution of the algorithm.

Essentially, the algorithm simulates a random walk on the control graph of the program by deciding, at each synchronization point, which thread makes the next step[3]. The structure of the algorithm is presented in Figure 1, and we also describe it in more detail below.

1) The probability distribution table contains probability distribution on edges and is updated at each iteration. Initially, the edges are unknown, so the table is empty, assuming uniform distribution.
2) The program is instrumented by adding callbacks at synchronization points. This enables the algorithm to stop the execution at these points and decide which edge is going to be traversed next (that is, which thread makes a move). At this point, the algorithm gathers the knowledge of the edges of JCG and stores it.
3) At each iteration, the algorithm performs the following tasks:
   a) The istrumented program is executed a number of times that is sufficient to collect a meaningful sample (the number of executions depends on the size of JCG). Executions are forced to perform scheduling decisions according to the current probability distribution on edges. Stopping the execution and deciding which thread is allowed to run next is performed by separate components in our implementation.
   b) The executions are used in order to compute the new probability distribution (see Equation 1 and Equation 2). We discuss the choice of the parameters q and α in Section V.
4) The algorithm terminates when the collected sample of the current iteration has a sufficiently small relative standard deviation (between 1% and 5%, as discussed in Section III-A).

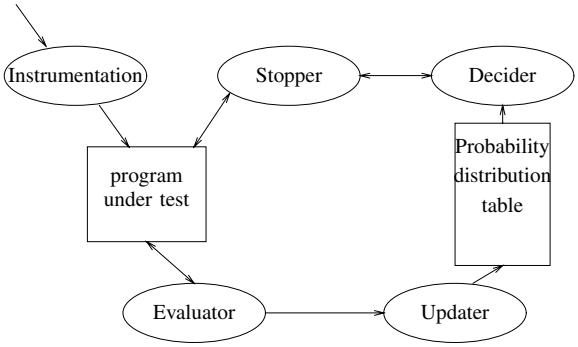

Fig. 1. The structure of our algorithm

B. Heuristics for improving performance

Dealing with unwinding: To generate the full JCG of the unwound program (without actually generating the unwound code), a program location is composed from the line number in the original (not unwound) code and an additional parameter, which reflects the unwinding. This parameter is a vector of values of loop counters for all loops the program is currently in. The example in Figure 4.1 illustrates why a single counter is not enough to determine the program location uniquely.

Example 4.1: The program block in Figure 2 contains nested loops. The values of $maxI$ and $maxJ$ are received from other thread and define the number of iterations of each loop. Consider two cases: (1) $maxI = 1$ and $maxJ = 2$,

[2]We assume a single statement per program location.

[3]Recall that we assume that all nodes are synchronization points.

```
assert(maxI ≤ 10);
assert(maxJ ≤ 10);
for(i = 0; i ≤ maxI; i++)
  for(j = 0; j ≤ maxJ; j++)
    foo(i, j);
```

Fig. 2. An example of nested loops

and (2) $maxI = 2$ and $maxJ = 1$. Consider the second call to $foo(i, j)$. In each of these cases, this call will appear in a different place in the unwound program, but the value of a single counter in both cases would be the same.

For the general case, we present a better parameter, using which, a 1-1 mapping can be created for any inner iterative structure of the program. This parameter will take the form of a stack. Each time the thread execution reaches a loop it pushes a new counter with value 0 on top of the stack, before entering the loop. Each time the execution starts a new iteration of a loop it increases the value of the top counter on the stack. Each time the execution exits the loop scope we pop the top counter of the stack. The same idea can be applied to function calls. Currently we do not test programs with recursion, though the method can be applied to recursive programs after minor changes.

Modulo reduction: In practice, the JCG of the unwound program can be huge even for medium-size programs. The problem with this is twofold. First, the graph itself may be two big to fit in the memory and to search efficiently. Second and more important, the set of execution samples generated at each iteration of the algorithm may be too sparse when projected on the JCG. In such case, the probability is updated only for a small fraction of the nodes. The consequence is that the algorithm converges to an arbitrary local maximum, instead of the target global maximum. We solve both problems by introducing the *modulo reduction*, in which the counters in program locations are computed modulo some small integer. The modulo reduction creates a *modulo joint control graph* (MJCG) form the original JCG. Each vertex of JCG is mapped to a vertex of MJCG by taking the modulo value of all its counters. The modulo reduction decreases the size of the graph significantly, however, it does not preserve the desired 1-1 mapping to unwound program locations. Our experiments show that with some fine tuning, it is possible to find the optimal modulo parameter, in which the many-to-one mapping does not confuse the CE computation, and yet, the MJCG is dense enough to allow the CE method to converge to a global (and not local) maximum.

C. Applications

In this section we show that CE method is useful for testing. The necessary prerequisite of using the CE method to find a rare pattern or bug is the ability to define a performance function for this pattern. We show that many (if not the majority) of potential problems in programs admit a natural performance function, and thus can be discovered using the CE method. As an easy example, consider searching for buffer overflows. Using the CE method, we can direct the executions of the program to areas where the buffer occupancy is maximized. The natural metrics for this problem is the number of elements in the buffer. The following is a partial list of other common patterns in programs, often associated with bugs, for which there exist natural metrics, and which, therefore, can be found using the CE method.

- Discovering data races: the function is the number of shared resources accessed during the execution.
- Testing error paths: the function is the number of error paths taken during the execution.
- Incorrect use of synchronization primitives: the function is the number of calls to mutually exclusive functions in an execution.
- Bugs caused or related to wait-notify situations: the function is the number of lost notifies.
- Testing recovery from multiple failures of threads/tasks: the function is the number of canceled threads/tasks.
- Simulation of the environment that triggers many exceptions: the function is the number of generated exceptions.
- Not releasing resources properly or exhaustion of resource pools: the function is the number of allocated resources minus the number of released ones.

V. Implementation and Experimental Results

The algorithm we describe in Section IV-A is implemented in Java and tested on several examples that we constructed. In this section, we briefly describe the implementation details and present our experimental results.

A. Implementation

The cross-entropy-based testing tool ConCEnter is written in Java. Its structure is reflected in Figure 1, and we briefly describe each part of it below.

- **Instrumenter** is an instrumentation tool that adds callbacks at synchronization points, i.e., immediately before and after each synchronized block.
- **Decider** receives a node v of the JCG of the program under test and chooses which thread is allowed to run next according to current relative probabilities $RP(v)$ on the control graph edges.
- **Stopper**: on callbacks from the instrumented code it stops the currently running thread using a mutex designated for this thread. Then, it calls notify() on the mutex of the thread that can make the next step (based on Decider's decision).
- **Evaluator** collects the edges of the JCG traversed by the execution path. At the end of each execution it computes the S value of the execution.
- **Updater** updates the probability distribution table for the next iteration based on the computations of Evaluator.

During the execution, Decider and Evaluator collect big amounts of data. To minimize the sizes of the memory buffers for this data, it is periodically written to files.

B. Experimental Results

We performed several experiments on different metrics and different types of bugs. The tests were written in Java version 1.5.0 and executed on a 4 CPU machine 64bit "Dual Core AMD Opteron(tm) Processor 280" with clock rate of 2.4GHz and 1MB cache size each. The total memory of the machine is 8GB. The operation system it runs is GNU/Linux 2.4.21-37.ELsmp.

We constructed several examples of programs with bugs that admit a natural performance function. The programs and bugs introduced in them are as follows.

1) Buffer overflow and underflow in a standard producer-consumer program. The test program consists of four threads (2 producers and 2 consumers) running concurrently. The buffer size is 5, and each thread can perform 10 push or pop operations.
2) Buffer overflow in a scenario similar to the previous case but with the buffer size 45 and 25 operations for each thread.
3) Buffer overflow and underflow similar to test 1 but with modulo parameter.
4) Stack overflow in a program with two types of threads, A and B, and two threads of each type. The pseudo-code of a thread is presented In Figure 3. Each thread performs 10 operations, and the size of the stack is 36. We note that in this test, buffer overflows are very rare (the probability of experiencing a bufer overflow in a random execution under uniform distribution on edges of the JCG is $O(1/2^n)$, where n is the size of the buffer), since the buffer is filled up only in executions in which thread types continuously alternate.

```
myName = A; // or B
for(i = 0; i < 10; i++)
  synchronized(buffer) {
    if((top of stack)== myName)
      pop();
    else
      push(myName);
  } // end loop
```

Fig. 3. The pseudo-code of the stack overflow example with two types of processes

5) Deadlocks. The test program has 10 mutexes and three locking threads. Most of the time, a correct locking policy is enforced (mutexes are numbered and the locking is attempted only in the order of their numbering). However, in a small percentage of the executions, a thread locks mutex 8 before locking mutex 7 and thus can cause deadlock.

In each test, we fine-tuned the parameters q (the best part of the sample), α (the smoothing parameter), and the modulo parameter to achieve the fastest convergence to the global maximum. The results of ConCEnter are compared with the results of running ConTest and Java scheduler on the same examples. We summarize the results in Table 4. In all examples ConCEnter converged in less than 20 iterations.

Buffer Overflow/Underflow			
	ConCEnter	ConTest	Java scheduler
% successful executions	100%	unstable	99%
runtime	15 sec	10 sec for 2000 tests	3 sec
Buffer Overflow with a large buffer			
	ConCEnter	ConTest	Java scheduler
% successful executions	100%	from 0.02% to 30%	99%
runtime	20 sec	30 sec for 5000 tests	3 sec
Buffer Overflow with MJCG			
	ConCEnter	ConTest	Java scheduler
% successful executions	100%	unstable	99%
runtime	15 sec	10 sec for 2000 tests	3 sec
Stack Overflow with A and B threads			
	ConCEnter	ConTest	Java scheduler
% successful executions	90%	$\ll 2.5\%$	0%
runtime	75 sec	< 40 sec	2 sec
Deadlock			
	ConCEnter	ConTest	Java scheduler
% successful executions	90%	at most 1 out of 5000 tests	$\approx 0\%$
runtime	20 min	300 sec	2 sec

Fig. 4. Experimental results

We also studied the influence of various CE method parameters on the performance of ConCEnter on the example of test 3 (stack overflow with A and B threads). In our experiments, we did not see a significant influence of q and α parameters on the performance of ConCEnter. The conclusion is that the modulo parameter is the parameter that seems to have a crucial role in the performance of ConCEnter. It seems that there is a lower and an upper bound for it, between which ConCEnter performs well. If the modulo parameter is smaller than its lower bound, then MJCG loses too much information and thus the CE method does not converge. If, on the other hand, the modulo parameter is larger than its upper bound, then MJCG is too sparse and CE method converges to a local maximum. We summarize these results in Table 5. The runs that did not converge neither to the global nor to the local maximum, did not converge at all after 40 iterations.

modulo	# samples per iteration	% convergence to global maximum	% convergence to local maximum
none	100, 200, 400	0	100%
2	200	30%	70%
2	400	15%	0%
3	100	20%	55%
3	200	55%	25%
3	400	75%	10%
4	100	5%	80%
4	200	50%	45%
4	400	85%	5%
8	100, 200, 400, 1000	0	100%

Fig. 5. Influence of modulo parameter

VI. Conclusions and Future Work

We showed a way to find rare bugs in large programs using the CE method by describing an algorithm that adapts the setting of testing of a program to the setting of the combinatorial optimization problem and performs an iterative procedure of CE method for a given performance function. We listed many interesting (potentially buggy) patterns in programs which have natural performance functions. We implemented a testing tool based on the CE method, called ConCEnter. We described the structure of ConCEnter and presented experimental results of running it on several patterns. We compared the performance of ConCEnter to this of ConTest, an IBM testing tool. In future work, we will apply ConCEnter to other patterns. We will also investigate other, more sophisticated ways to use CE method in testing. Mainly, we are interested in the following directions:

- **Fine-tuning ConCEnter parameters**. Our experiments show that in many cases choosing the best parameters for a given performance function is hard. It is interesting to devise an automatic method to find optimal parameters.
- **Finding the second best**. Our method converges towards some parts of interleaving space where the performance function is maximized locally or globally. There may be other areas where the performance function is maximized. We would like to develop a method to find these other arias which where not covered in previous executions of the algorithm.
- **Increasing the modulo in a part of the graph**. The parameter that defines the modulo reduction seems to be critical to the convergence of the method and it is hard to find the value which will be optimal to all nodes of the JCG. The best strategy here is probably to define a different modulo value for each node. We guess that this can be done by analyzing the graph of previous executions.
- **Coverage**. As mentioned above, increasing test coverage is one of most widely used methods of program verification. In testing, a huge effort is made to create tests which cover more of the program. We intend to use CE method with "find the second best" technique to iteratively create execution paths which increase coverage.
- **Partial Replay**. It is often important to replay a previous execution of the program (for example, in order to reproduce a bug). Unfortunately, full replay is very hard or even impossible to achieve in practice. It seems that using the CE method with the performance function that reaches its maximal value on the execution we are trying to replay can produce an execution that resembles it to a high extend.

Acknowledgment

We thank Reuven Y. Rubinstein and Andrey Dolgin for helpful discussions on the cross-entropy method.

References

[1] G. Alon, D.P. Kroese, T. Raviv, and R.Y. Rubinshtein. Application of the cross-entropy method to buffer allocation problem in simulation-based environment. *Annals of Operations Research*, 2004.

[2] L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Kluwer Academic Publishers, 2000.

[3] M. Biberstein, E. Farchi, and S. Ur. Fidgeting to the point of no return. In *PDPS*, volume 26-30, April 2004.

[4] Edmund Clarke, Himanshu Jain, and Daniel Kroening. Verification of SpecC and Verilog using predicate abstraction. In *MEMOCODE*, pages 7–16. IEEE, 2004.

[5] Edmund Clarke, Himanshu Jain, and Daniel Kroening. Verification of SpecC using predicate abstraction. *FMSD*, 30(1):5–28, February 2007.

[6] T. Homem de Mello and R.Y. Rubinstein. Rare event simulation and combinatorial optimization using cross entropy: estimation of rare event probabilities using cross-entropy. In *Winter Simulation Conference*, pages 310–319, 2002.

[7] D.L. Dill. What's between simulation and formal verification? In *35st DAC*, pages 328–329. IEEE Computer Society, 1998.

[8] U. Dubin. The cross-entropy method for combinatorial optimization with applications. Master Thesis, The Technion, 2002.

[9] O. Edelstein, E. Farchi, Y. Nir, G. Ratzaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(3):111–125, 2002.

[10] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *SEA*, Acta Press, 2002.

[11] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[12] J.M. Keith and D.P. Kroese. Rare event simulation and combinatorial optimization using cross entropy: sequence alignment by rare event simulation. In *Winter Simulation Conference*, pages 320–327. ACM, 2002.

[13] S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86, 1951.

[14] L. Margolin. Cross-entropy method for combinatorial optimization. Master Thesis, The Technion, 2002.

[15] D. Peled. *Software reliability methods*. Springer-Verlag, 2001.

[16] G. Ratsaby, B. Sterin, and S. Ur. Improvements in coverability analysis. In *FME*, pages 41–56, 2002.

[17] R.Y. Rubinshtein. Optimization of computer simulation models with rare events. *European Journal on Operations Research*, 99:89–112, 1997.

[18] R.Y. Rubinshtein. The cross-entropy method and rare-events for maximal cut and bipartition problems. *ACM Transactions on Modelling and Computer Simulation*, 12(1):27–53, 2002.

[19] R.Y. Rubinstein and D.P. Kroese. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Information Science and Statistics. Springer-Verlag, 2004.

[20] E.M. Rudnick, J.H. Patel, G.S. Greenstein, and T.M. Niermann. Sequential circuit test generation in a genetic algorithm framework. In *DAC*, pages 698–704. ACM/IEEE, 1994.

[21] D.G. Saab, Y.G. Saab, and J. Abraham. A test cultivation program for sequential VLSI circuits. In *CAV*, LNCS, pages 216–219, 1992.
[22] S.D. Stoller. Testing concurrent java programs using randomized scheduling. In *RV*, volume 70(4) of *ENTCS*. Elsevier, 2002.
[23] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.
[24] Verisity. Surecove's code coverage technology. http://www.verisity.com/products/surecov.html, 2003.
[25] J.A. Whittaker. *How to Break Software*. Addison-Wesley, 2003.
[26] H. Zhu, P.V. Hall, and J.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

APPENDIX

Here we present the more formal explanation of CE method. The main ideas behind the CE algorithm are as follows. Let $\mathcal{X}$ be a space of vectors. Let $\{f(\cdot;v)\}$ be a family of probability density functions (pdfs) on $\mathcal{X}$,where v is a parameter vector. Let $S:\mathcal{X}\rightarrow\mathbb{R}^+$ be a function that gives each vector in $\mathcal{X}$ a non-negative value. In the general version of the CE method, we estimate the probability

$$l=P_u(S(X)\geq\gamma)=E_uI_{\{S(X)\geq\gamma\}} \tag{3}$$

for some $\gamma\in\mathbb{R}$ - under input $f(\cdot;u)$ pdf. If this probability is very small, say, smaller than 10^{-5}, we say that $\{S(X)\geq\gamma\}$ is a rare event. A straightforward way to estimate l is to draw a random sample $X_1,\ldots,X_N$ from $\mathcal{X}$ according to $f(\cdot;u)$, and then estimate l by examining the sample. The problem is, that when l is a rare event, the sample might have to be very large in order to estimate l accurately. A better way is to draw the sample according to some other probability density function g that raises the probability of $S(X)\geq\gamma$. Using the density g we can rewrite equation 3 as

$$l=\int I_{\{S(X)\geq\gamma\}}\frac{f(x;u)}{g(x)}g(x)dx=E_gI_{\{S(X)\geq\gamma\}}\frac{f(x;u)}{g(x)}.$$

The g called *importance sampling* density. An unbiased estimator of l is

$$\hat{l}=\frac{1}{N}\sum_{i=1}^{N}I_{\{S(X_i)\geq\gamma\}}\frac{f(X_i;u)}{g(X_i)}, \tag{4}$$

where $\hat{l}$ called *importance sampling* (IS) estimator.

The best g to estimate l is

$$g^*(x)=\frac{I_{\{S(x)\geq\gamma\}}f(x;u)}{l} \tag{5}$$

by using this change of measure in equation 4 we get

$$l=I_{\{S(x)\geq\gamma\}}\frac{f(X_i;u)}{g(X_i)}$$

for all i. The g^* function depends on l which is unknown. It is convenient to choose g from the $\{f(\cdot;v)\}$ family. The idea is to choose *tilting parameter* v such that distance between g^* and $f(\cdot:v)$ is minimal. For this purpose the CE method uses Kullback-Leibler "distance" (also called cross-entropy). The Kullback-Leibler "distance" between g and h defined as:

$$\mathcal{D}(g,h)=E_g\ln\frac{g(X)}{h(X)}=\int g(x)\ln g(x)dx-\int g(x)\ln h(x)dx.$$

Minimizing the Kullback-Leibler "distance" between g^* in equation 5 and $f(\cdot;v)$ is equivalent to solving the maximization problem

$$max_v\int g^*(x)\ln f(x;v)dx,$$

since the the value of $\int g^*(x)\ln g^*(x)dx$ is constant. Substituting g^* from equation 5 we obtain equivalent maximization problem

$$max_v\int\frac{I_{\{S(x)\geq\gamma\}}f(x;u)}{l}\ln f(x;v)dx,$$

which is equivalent to

$$max_vD(v)=max_vE_uI_{\{S(x)\geq\gamma\}}\ln f(x;v),$$

and using importance sampling again with a change measure $f(\cdot;w)$ we can write

$$max_vD(v)=max_vE_wI_{\{S(x)\geq\gamma\}}\frac{f(x;u)}{f(x;w)}\ln f(x;v).$$

The optimal solution is

$$v*=argmax_vE_wI_{\{S(x)\geq\gamma\}}\frac{f(x;u)}{f(x;w)}\ln f(x;v)$$

and can be estimated by following stochastic program

$$max_v\hat{D}(v)=max_v\frac{1}{N}\sum_{i=1}^{N}I_{\{S(X_i)\geq\gamma\}}\frac{f(X_i;u)}{f(X_i;w)}\ln f(X_i;v), \tag{6}$$

where $X_1,\ldots,X_N$ is a random sample from $f(\cdot;w)$.

The CE method in each iteration solves Equation 6 based on the solution of previous iteration. We demonstrate the solution in one iteration by example, which is very similar to the use of the cross-entropy method in the paper.

Let $\vec{X}$ be a random vector $(X_1,\ldots,X_N)\sim Ber(p)$, and parameter $v=p$. The probability density function is

$$f(\vec{X};p)=\prod_{i=1}^{N}p_i^{X_i}(1-p_i)^{1-X_i},$$

and since each X_j can only be 0 or 1, we have

$$\frac{\partial}{\partial p_j}\ln f(X;p)=\frac{X_j}{p_j}-\frac{1-X_j}{1-p_j}=\frac{X_j-p_j}{(1-p_j)p_j}.$$

We can compute the maximum in Equation 6 by setting the first derivatives with respect to p_j equal to zero, for $j=1,\ldots,N$:

$$\frac{\partial}{\partial p_j}\sum_{i=N}^{N}I_{\{S(X_i)\geq\gamma\}}\ln f(X_i;p)=$$

$$=\frac{1}{(1-p_j)p_j}\sum_{i=N}^{N}I_{\{S(X_i)\geq\gamma\}}(X_{ij}-p_j)=0,$$

where X_{ij} is X_j in sample i.

Thus, we have

$$p_j=\frac{\sum_{i=N}^{N}I_{\{S(X_i)\geq\gamma\}}X_{ij}}{\sum_{i=N}^{N}I_{\{S(X_i)\geq\gamma\}}}.$$