Cross-Entropy-Based Replay of Concurrent Programs*

Hana Chockler[1], Eitan Farchi[1], Benny Godlin[1], and Sergey Novikov[2,**]

[1] IBM Haifa Research Laboratories
Haifa University, Mount Carmel
Haifa 31905, Israel
hanac,farchi,godlin@il.ibm.com
[2] Department of Computer Science
Weizmann Institute, Israel
sergey.novikov@weizmann.ac.il

Abstract. *Replay* is an important technique in program analysis, allowing to reproduce bugs, to track changes, and to repeat executions for better understanding of the results. Unfortunately, since re-executing a concurrent program does not necessarily produce the same ordering of events, replay of such programs becomes a difficult task. The most common approach to replay of concurrent programs is based on analyzing the logical dependencies among concurrent events and requires a complete recording of the execution we are trying to replay as well as a complete control over the program's scheduler. In realistic settings, we usually have only a partial recording of the execution and only partial control over the scheduling decisions, thus such an analysis is often impossible. In this paper, we present an approach for replay in the presence of partial information and partial control. Our approach is based on a novel application of the cross-entropy method, and it does not require any logical analysis of dependencies among concurrent events. Roughly speaking, given a partial recording R of an execution, we define a performance function on executions, which reaches its maximum on R (or any other execution that coincides with R on the recorded events). Then, the program is executed many times in iterations, on each iteration adjusting the probabilistic scheduling decisions so that the performance function is maximized. Our method is also applicable to debugging of concurrent programs, in which the program is changed before it replayed in order to increase the information from its execution. We implemented our replay method on concurrent Java programs and we show that it consistently achieves a close replay in presence of incomplete information and incomplete control, as well as when the program is changed before it is replayed.

1 Introduction

Software testing and debugging are nowadays the primary means of checking the correctness of programs. Repeated re-execution, or *replay*, is a widely accepted technique

* This work is partially supported by the European Community under the Information Society Technologies (IST) program of the 6th FP for RTD - project SHADOWS contract IST-035157. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

** Most of this work was performed when author was at IBM Haifa Research Laboratories.

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 201–215, 2009.

for debugging sequential deterministic programs. In these programs, a specific input vector always generates the same execution, so the task of replay is relatively easy, provided that the input vector is recorded. This approach, of recording the input vector and re-executing the program, does not work on concurrent programs, where the same input vector can generate many executions of the program, as a result of the different scheduling of concurrent events. *Replay of a concurrent execution from its recorded trace* is a widely used and extensively researched approach for analyzing and debugging concurrent behaviors. To overcome the problem of different scheduling, replay of a concurrent program is achieved by first recording an execution, and then enforcing the same order of events during the replay.

Existing work on replay of concurrent programs is based on analyzing the logical order between concurrent events and assigning time-stamps to events that need to be executed in a particular order. The idea of applying time-stamps to express logical order between events was first introduced in the seminal work of Lamport [15]. Lamport introduced the *happened-before* relation between concurrent events, which imposes a partial order on these events. A system of logical clocks is then used in order to capture this relation by assigning time-stamps to all concurrent events in the computation, consistent with the logical dependencies between these events. The computation of time-stamps for ordering of concurrent events is studied and improved upon in several works (see, for example, [3,14,21,18,19,1]), where the order of events determined by time-stamps is used for replay of programs. Replay of multi-threaded Java programs, which are inherently concurrent, is studied by Choi et al., who suggest methods for efficient recording of concurrent events and control of the Java scheduler in order to replay the recorded execution (see, for example, [5]). Here also, the logical dependencies between events are analyzed in order to construct a correct logical schedule, which is then used for replay.

The major drawback of the existing approaches to replay of concurrent programs is that they all study *deterministic replay*, that is, replay that has full control over the scheduler and follows a complete recording of a previous execution. In order to implement deterministic replay, it is necessary to record the events that are crucial for defining the logical order between all concurrent events. There are several works that attempt to reduce the size of the trace as much as possible [16,20]. However, there exists a minimal subset of events that is crucial for the correct logical order, and this subset has to be recorded in order to allow replay. Moreover, in replay, there has to be a full control over the scheduler, since even a slight change in the order of concurrent events can lead to deviation from the recorded execution and inability to resume replay. In realistic settings, both requirements, of full recording and full control over the scheduling, are often impractical. Software today incorporates third party code and library code of which we have no control, code is loaded at runtime, and due to standardization and open environments it is impossible to have control over all concurrent events. Moreover, due to space and performance considerations and the fact that often recordings of executions are created by the customer, getting a recording of an execution that is complete enough to determine logical order is unlikely. Finally, these approaches do not address a common debugging scenario, in which the user adds informative statements to the program before replaying it (usually print statements). These statements

affect the existing happened-before relation and the control over the program, and thus the program cannot be replayed based on the previously computed order of concurrent events.

In this paper, we study the problem of *approximate replay*, that is, replay in presence of incomplete recording and only partial control over the concurrent events, and where a program under test is modified before it is replayed. We present a novel approach to replay that is based on the cross-entropy method and the adaptation of this method to testing of concurrent programs. The *cross-entropy* (CE) method is a generic approach to rare event simulation [24]. 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 based on minimizing the cross-entropy or the Kullback-Leibler distance between two probability distributions. The CE method was motivated by an adaptive algorithm for estimating probabilities of rare events in complex stochastic networks [22]. 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 in optimization problems is used to find a set of inputs on which the performance function reaches its global maximum, where the input space is assumed to be too large to allow exhaustive exploration. The method is based on an iterative sampling of the input space according to a given probability distribution over this space. Each iteration consists of the following phases:

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

The initial probability distribution is assumed to be a part of the input. 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 [2], neural computation [7], DNA sequence alignment [10], scheduling [17], and graph problems [23].

In [4], we adapted the cross-entropy method to testing of concurrent programs. Informally, such programs induce 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, with the input space being the space of all possible paths on the graph. 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, decisions about the order of execution of concurrent threads are made by the scheduler, and thus can be viewed as non-deterministic when analyzing the program. Our tool, ConCEnter, is based on the cross-entropy method, and was shown to be effective in finding rare bugs in multi-threaded Java programs. The most natural initial probability distribution over the space of all possible paths (and the one we use in [4]) is the uniform distribution

over edges on each node, meaning that at each decision point each enabled thread can make a step with equal probability.

In this paper, we adapt the cross-entropy method to replay of concurrent programs as follows. We define the *distance* $D(e_1, e_2)$ between two executions e_1 and e_2 in a way that reflects the distance between e_1 and e_2 on the control graph of the program. Then, we define the performance function $S(e)$ over executions as $-D(e, e_{rec})$, where e_{rec} is the recorded execution. Since the distance is always non-negative, the performance function $S(e)$ reaches its global maximum on executions that are as close to the recorded execution as possible. Then, we apply the cross-entropy-based testing method (implemented in ConCEnter) in order to get an approximate replay of the recorded execution. In our approach, having only a partial recording of the execution or not having full control over the executions is handled naturally without any adjustments. The distance is computed based on the recorded events, and the probabilities of concurrent events over which we have control are adjusted according to the cross-entropy minimization computation. Moreover, our experimental results show that our method works even if the program is changed before replaying it, for example, by adding print statements (a common scenario for debugging).

As we discuss in Section 4.2, in replay we can use the recorded execution in order to compute an initial probability distribution over the space of all executions in a way that significantly improves the running time of ConCEnter compared to executing it on the same examples with the initial uniform distribution. We give edges that appear in the recorded execution the initial probability that is higher than their probability under the uniform distribution, and we adjust the probabilities of other edges accordingly.

We discuss the problem of losing diversity of the sample and the reasons why this problem is more pronounced in replay than in applications of cross-entropy based testing to bug searching. Essentially, the performance function based on the distance from a recorded execution has a very narrow global maximum (in fact, when the complete recording is available, the global maximum is reached on exactly one order of recorded events). This particular shape of the performance function increases the probability of missing the global maximum by converging to a local maximum or even to some random point. This problem, of losing diversity of the sample (and as a result, converging to a wrong point), is inherent for the cross-entropy method as well as for other methods that use iterative adjustments of the probability distribution, and it becomes more pronounced in replay, compared to testing for rare bugs. To overcome this problem, we use *random injection* at some of the iterations of the cross-entropy execution. Our method is inspired by *simulated annealing*, introduced in [11]. Simulated annealing is a general approach to finding good approximations for global optimization problems of functions in large search spaces. Essentially, in each iteration, simulated annealing replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the number of previous iterations. The randomization decreases during the process, thus allowing it to converge. Roughly speaking, simulated annealing slows the convergence process by introducing an additional dimension of randomness. The effect is two-fold: first, by slowing the convergence process, it increases the probability of it converging to the correct maximum[1]; and second, by introducing more

[1] The reason for this phenomenon in simulated annealing is the same as in cross-entropy.

randomness, it increases coverage of the search space, thus increasing the probability that a good solution will be eventually drawn. We achieve a similar effect by introducing random injection at some iterations, where the decision of whether to introduce a random injection depends on the relative standard deviation of the current sample.

We implemented replay with random injection in ConCEnter and tested it on multi-threaded Java programs[2]. Our experimental results show that we are able to efficiently produce a close replay even when we have only a partial recording of the execution. We compare the performance of ConCEnter with the performance of ConTest, a randomized tool for testing of concurrent programs developed at IBM [8].

2 Preliminaries

2.1 The Cross-Entropy Method in Optimization Problems

In this section we present a brief overview of the cross-entropy method for optimization problems. The reader is referred to [4] for more formal explanation and to the book on cross-entropy for the complete description of the method[24]. In our setting, we use the application of the cross-entropy method to graph optimization problems.

The cross-entropy (CE) method for optimization problems searches for a global maximum of a function S (called a *performance function*) defined on a very large probability space. Since the exhaustive search is impossible due to the size of the space, the method works in iterations, each time drawing a sample from the space and adjusting the probability distribution for the next iteration according to the values of S on the sample.

In graph optimization problems, we are given a (possibly weighted) graph $G = \langle V, E \rangle$, and the probability space is defined on the set of paths in G represented by the sets of traversed vertices. The probability distribution is defined by assigning probabilities to edges or vertices of the graph (depending on how the sample is drawn). 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 this paper.

2.2 Programs as Graphs

We view concurrent programs under test as *control flow graphs*, with nodes being synchronization points. We start with the definition of a control flow graph of a single thread. We define a *program location (PL)* as a line number in the code of the program, and we assume that it uniquely defines the state of a thread. In particular, this means that all loops are unwound to the maximal number of iterations and all function calls are embedded in the code of the main function.[3] We use t for the number of threads in the program.

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

[3] The unwound code of even a small program can be very large. In the previous paper, we reduced the size of the unwound code by using multi-dimensional modulo counters [4]. Unfortunately, when using modulo counters, we lose the one-to-one correspondence between the trace and its representation on the graph. Thus, we do not use this method in replay.

Definition 1 (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 2 (PLV). Program location vector *(PLV) $\boldsymbol{v}$ is a t-dimensional vector such that for each $i \in [t]$, $v_i \in L_i$.*

We say that at a given time m during the execution of the program, the execution is at PLV $\boldsymbol{v}$ iff 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 (JCG). *The joint control graph (JCG) of the program under test 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 step only one thread executes. 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 initial node of the JCG is a PLV which is composed of the initial PL μ_i for each thread i.

Definition 4 (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 threads. If $T_{en} \subseteq [t]$ is the set of the enabled threads at this moment then the relative probabilities are:

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

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

2.3 The Cross-Entropy Method for Replay

We describe the problem of replay in concurrent programs as a graph optimization problem, where the program is represented as a graph, executions are paths on the graph, and the performance function reaches its maximum on the recorded execution. In replay, an input to the procedure is a (partially) recorded execution. The performance function is based on the *distance* between two executions. Our distance metric is somewhat similar to L_1 *distance*, also known as *taxicab distance* or *rectilinear distance* (see [12]), and it expresses the distance between the paths on the control graph that correspond to the execution.

Definition 5. *For two executions e_1 and e_2 of the same program P represented by paths π_1 and π_2 on the JCG of P, respectively, the* distance $D(e_1, e_2)$ *between e_1 and e_2 is defined as the number of nodes that are present in one execution and absent from another. Formally, we have the following definition:*

1. *Let L_P be the vector of all nodes of the JCG of P (note that each node can appear only once in an execution because of unwinding). Let n be the length of L_P.*
2. *Let $L(\pi_k)$ be the binary vector of length n such that $L(\pi_k)[i] = 1$ iff $L_P[i] \in \pi_k$, for $k = 1, 2$.*
3. *The distance $D(e_1, e_2)$ is defined as $H(L(\pi_1), L(\pi_2))$, where $H()$ is the Hamming distance (Hamming distance is defined in [9]).*

The *replay performance function* $S^r(e)$ from the recorded execution r to an execution e is defined as $-D(r, e)$. Clearly, it reaches its maximum value 0 at $e = r$. When we have a partial recording of the execution, the distance is measured only with respect to the recorded nodes, even if they do not form a connected path on the graph. Thus, in case of a partial recording, there is a set of traces that have the distance 0 from the recorded execution.

We define the probability distribution on the set of executions by assigning probabilities to edges of the control flow graph. The initial probability distribution is either uniform or biased towards the recorded execution (see Section 3.1 for the discussion on the initial 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 $0 < q \ll 1$, 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 the control flow graph 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%), or an exact replay is achieved.

Smoothed updating and its importance for replay 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). According to [6], when the shape of the performance function makes convergence to the global maximum hard, it is advisable to use very low

smoothing parameters, around 0.01, thus decreasing the rate of update of the probability distribution and therefore also the convergence rate. Slowing the convergence rate increases the probability of a sample to find the global maximum, and hence is better for "difficult" functions. In our experiments, setting the smoothing parameter to a very low value increased the running time by several orders of magnitude, thus rendering the method impractical for large programs. In this work, we use the method of random injection (see Section 3.2) to tackle the problem of convergence to local maximum, thus allowing us to use a relatively high smoothing parameter (in this work we are using $0.8 \leq \alpha \leq 0.9$).

3 Algorithm for Approximate Replay of Concurrent Programs

In this section we describe the algorithm that uses cross-entropy for approximate replay of concurrent programs and discuss random injection - the main change from the traditional cross-entropy method.

3.1 Algorithm

Given a (partially) recorded execution r of a concurrent program P, the algorithm outputs a set of executions of P that are the closest to r. In each iteration, the algorithm generates a set of executions based on the probability distribution table. The probability distribution table contains probability distribution on edges and is updated at each iteration. We consider two options for the initial probability table:

1. Uniform distribution, as in other applications of cross-entropy for testing.
2. Probability distribution biased toward the recorded execution. Here, we perform preprocessing of the probability distribution, assigning higher probability to edges that participate in the recorded execution.

In the classic cross-entropy setting, the initial probability distribution on an input space is a part of the input. In our setting, the probability space is on the set of all possible executions of the concurrent system under test, and there is no predefined probability distribution on this space. In [4], we start with the initial uniform distribution over the edges of the control flow graph, since this is the distribution that most accurately reflects the situation where the program runs without any intervention. In replay, however, the situation is different because the recorded execution is a part of the input. Thus, the initial distribution of executions for ConCenter does not need to be uniform – it can be biased toward the recorded execution. The uniform initial distribution has the advantage of not requiring any special preprocessing before the start of the execution. An obvious disadvantage of a biased initial probability distribution is that it requires a preprocessing, however, as we show in Section 4.2, it significantly improves the convergence rate of the algorithm.

We assume that P is partially instrumented (this is needed for recording executions) 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) according to the probability distribution table. At each iteration, the algorithm performs the following tasks:

1. The instrumented program P is executed a number of times sufficient to collect a meaningful sample. Executions are forced to perform scheduling decisions according to the current probability distribution on edges.
2. The executions are used in order to compute the new probability distribution (see Equation 1 and Equation 2).
3. If the current iteration satisfies the criteria for random injection (see Section 3.2), the algorithm computes the probabilities of edges as a weighted average of the probabilities according to the cross-entropy method and a random injection of uniform distribution, where the weight of the injection is between 0.01 and 0.1.
4. The best execution e (the one with the maximum value of $S^r(e)$) is compared with the best execution obtained so far and a new best execution is chosen.

The algorithm terminates when the collected sample of the current iteration has a sufficiently small relative standard deviation (between 1% and 5%), or, alternatively, when we get the best replay, that is, an execution e for which $S^r(e) = 0$. We note that there can be several best replays, depending on the level of instrumentation of the program and the level of control over the executions. Step 4 of the algorithm is needed in order not to miss an exact replay if it is drawn before the best quantile of the sample converges.

3.2 Random Injection

The concept of using random injection in order to prevent convergence to local maximum is not new and is used widely, for example, in simulated annealing [11]. Essentially, without a (sporadic) random injection, the sample might become uniform too soon and not reach the global maximum. The most common case of converging to a wrong result is when the sample converges to a local maximum. In replay, due to the unique shape of the performance function with a very narrow global maximum, there is also the phenomenon of converging to some value which is neither a local nor a global maximum. In this work, we examine several ways to add random injection to the algorithm without compromising its convergence. We describe our experiments in more detail in Section 4 and compare the effect experimental of applying different methods on the convergence and the convergence rate of the program.

4 Implementation and Experimental Results

We use our cross-entropy-based testing tool ConCEnter for checking our approach to replay of concurrent programs. We briefly describe ConCEnter and the way replay is implemented in it in Section 4.1. We describe the experimental setting in Section 4.2 and present our experimental results in Section 4.3.

4.1 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.

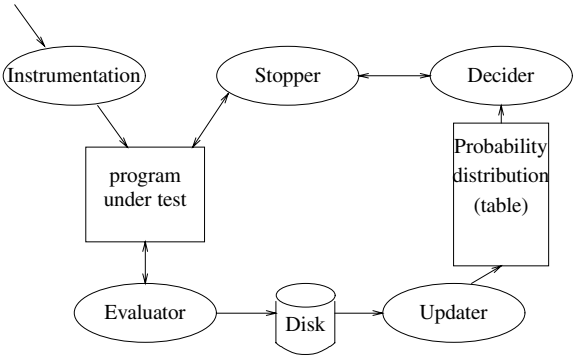

Fig. 1. Parts of ConCEnter

- **Instrumenter** is an instrumentation tool that adds callbacks at control points.
- **Decider** receives a node v of the JCG of the program under test and chooses which thread is allowed to execute 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 execute next (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 disk.

4.2 Description of the Experimental Setting

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 2.6.9-42.0.3.ELsmp. In all examples, we executed the program once and recorded it using the instrumentation provided by ConTest. We then attempted to replay this execution with ConCEnter. For each program, we repeated the experiment of recording and replay 10 times; the reported numbers are the average numbers calculated on these executions.

Tuning of parameters for ConCEnter We checked the influence of different parameters on the success and the convergence rate of replay. These parameters control the execution of ConCenter as follows:

1. Target relative standard deviation is the stopping condition: when the relative standard deviation of the sample reaches the target, the execution terminates.

Table 1. Best Parameters for Replay (After Tuning)

target relative standard deviation	0.01
smoothing parameter	0.8
number of runs per series	200
quantile size q	0.2
target weight in initial distribution	0.9
injection threshold	0.05
injection factor	0.05
change of injection	1
injection frequency	3

2. Smoothing parameter is the weight of the new probability distribution in the weighted average with the previous probability distribution, as explained in Section 2.3.
3. Number of runs per series is the number of executions drawn from the space of all executions according to the current probability distribution on edges at each iteration.
4. Quantile size q is the best quantile of the sample according to the performance function (i.e,, for example, the best 10% of the sample used to recompute the probability distribution for the next step).
5. Target weight in initial distribution is the initial bias of the distribution toward the recorded execution, as discussed in Section 3.1.
6. Injection threshold is the relative standard deviation of the best q-part of the sample that triggers random injection in the next iteration.
7. Injection factor is the weight of the random injection in the sample.
8. Change of injection is the multiplicative factor applied to the weight of the random injection at each iteration (if the change of injection is 1, then the weight of the injection is constant during the whole execution).
9. Injection frequency is x if a random injection is added to each x-th iteration, provided that it satisfies the criterion of the injection threshold.

In our setting, we performed the tuning of parameters manually, and the best values are presented in Table 1. Clearly, manual tuning of parameters is time-consuming. On the other hand, our experiments show that the best values of these parameters are approximately the same for all programs we applied our replay algorithm to, so we conjecture that in most cases tuning can be viewed as a one-time preprocessing task. It is also possible to let the algorithm to adjust these parameters automatically during the execution as described in Chapter 5 of [24].

4.3 Experimental Results

In this section we describe the concurrent programs on which we checked the convergence and performance of ConCEnter and discuss the meaning of the experimental results.

Programs under test

Toy examples We start with two toy examples:

- A standard producer-consumer program, in which the producer threads fill the buffer, and the consumer threads empty the same buffer. The program launches two threads of each kind running concurrently, and the recorded execution contains a buffer overflow. Since placing a request in the buffer and removing it from the buffer require approximately the same time, a buffer overflow is reproduced only on executions on which only the producer threads run until the buffer is full. It is easy to see that if all threads are enabled all the time, the probability of reproducing a buffer overflow in a random execution is $O(1/2^n)$, where n is the size of the buffer.
- A "Push-pop" example, in which there are two types of threads, A and B, and each thread either pushes its name on top of the stack or pops the top element of the stack depending on the value of the current top element: if the value of the top element is equal to the name of the current thread, the thread pops the value, otherwise it pushes its name. The recorded execution contains stack overflow. It is easy to see that it can only be reproduced in executions in which threads constantly alternate, and thus, similarly to the previous example, the probability of reproducing it is $O(1/2^n)$, where n is the size of the stack.

ConCEnter consistently achieved the perfect replay of recorded executions with the values of parameters as specified in Table 1. We also checked the effect of biased initial distribution on the second example (see the results below).

Java 1.4 Collection Library Our real-life example is the Java 1.4 collection library that was used as a case study in [25]. This is a thread-safe collection framework implemented as a part of `java.util` package of the standard Java library provided by Sun Microsystems. We ran our experiments for all tests in this collection. The simplest tests $ITest$ and $MTLinkedListInfiniteLoop$ converged in less than 5 iterations to 100% replay. The other tests in the library, MTListTest, MTSetTest, and MTVectorTest, converged to 90% replay after less than 5 iterations, and to 95% replay after less than 10 iterations.

The effect of biased initial distribution. We experimented with the bias in the initial distribution ranging from 1 (probability 1 to choose the recorded execution), to 0.5 in steps of 0.2 and compared the results with the uniform initial distribution (that is, no bias toward the recorded distribution). The results presented in Table 2 are the average results over 10 executions, and it is easy to see that preprocessing of the probability table that assigns higher probabilities to edges that are present in the recorded execution decreases the number of iterations and significantly improves the probability of achieving a good replay. The experiments were done on the push-pop example.

We note that while the best results were achieved with the initial distribution that gives the recorded edges probability 1, in real-life examples and especially when the program was modified before it is replayed, we should not start with such a distribution. This is because giving the recorded edges the probability 1 effectively eliminates the

Table 2. Results for biased initial distribution, push-pop example

Bias weight	total replay success	good replay ($> 95\%$) success	local maximum
1	100%	–	–
0.9	100%	–	–
0.7	90%	10%	–
0.5	100%	–	–
unbiased	30%	30%	40%

Table 3. Adding print statements before replay (Java 1.4 Collection Library)

Test name	number of iterations to 90% replay
MTListTest	$2-3$
MTSetTest	$2-4$
MTVectorTest	2

chance of choosing other edges. If the program is modified before it is replayed, the recorded execution might not be enabled at all, and thus the probability of choosing other edges should be greater than 0 to allow an approximate replay.

Changing the weight of injection. In our experiments, we checked the influence of injection on convergence of the cross-entropy based replay by varying the weight of injection from 0.01 to 0.1. We also checked the effect of different injection frequency, from 1 (each iteration) to 10. The best results, on all examples, were obtained with the weight of injection between 0.02 and 0.03, and injection frequency either 2 or 3.

Replay after program modification. In order to simulate a common debugging scenario, we introduced print statements after each action on the shared variables in the examples from Java 1.4 Collection Library. The print statements are conditional on a random bit, i.e., executed with the probability 50% at each execution, and they were introduced after the execution was recorded, which corresponds to the scenario where an engineer attempts to debug a program by introducing print statements and then reproducing the bug. Table 3 shows that a replay that is 90% close to the recorded execution was achieved after less than 5 iterations.

Comparison with ConTest. We compared our results to ConTest, where replay can be attempted by using the same random seed as in the recorded execution. We executed ConTest on the examples above, saved the seed of the random noise generator's decisions and attempted to replay the execution 100 times for each example. To compare the executions in ConTest, we checked the number of elements in the buffer after the execution terminates (that is, a looser criterion than an identity between executions). In buffer overflow, ConTest succeeded in 4 attempts, and in push-pop in 16 attempts (out of 100), even with such a permissive criterion of equivalence between executions. On the collection of program from the Java 1.4 Collection Library, ConTest did not achieve

even an approximate replay. This is hardly surprising, since the same random seed does not guarantee the same scheduling of concurrent events.

5 Conclusions and Future Work

We presented an application of cross-entropy method to replay of concurrent programs. To the best of our knowledge, this is the first approach that does not require a full recording of the concurrent events in the execution and also allows only partial control over the repeated executions. Our approach also accommodates program changes between the recording and the replay. For example, adding print statements to the program before replaying it does not interfere with the replay. In future work, we will integrate other techniques into replay, such as clustering algorithms, simulated annealing, and other methods for updating the probability distribution. We also plan to integrate techniques from genetic algorithms, as there seems to be quite a significant common ground between them and cross-entropy, especially when cross-entropy is applied to reproducing a single event, such as in replay.

References

1. Agarwal, A., Garg, V.K.: Efficient dependency tracking for relevant events in concurrent systems. Distributed Computing 19(3), 163–182 (2006)
2. Alon, G., Kroese, D.P., Raviv, T., Rubinstein, R.Y.: Application of the cross-entropy method to buffer allocation problem in simulation-based environment. Annals of Operations Research (2004)
3. Carver, R.H., Tai, K.C.: Replay and testing for concurrent programs. IEEE Software 8(2), 66–74 (1991)
4. Chockler, H., Farchi, E., Godlin, B., Novikov, S.: Cross-entropy based testing. In: Proceedings of Formal Methods in Computer Aided Design (FMCAD), pp. 101–108. IEEE Computer Society, Los Alamitos (2007)
5. Choi, J.-D., Srinivasan, H.: Deterministic replay of multithreaded java applications. In: ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT), pp. 48–59 (1998)
6. Costa, A., Jones, O.D., Kroese, D.: Convergence properties of the cross-entropy method for discrete optimization. Operations Research Letters 35(5), 573–580 (2007)
7. Dubin, U.: The cross-entropy method for combinatorial optimization with applications. Master Thesis, The Technion (2002)
8. Edelstein, O., Farchi, E., Nir, Y., Ratzaby, G., Ur, S.: Multithreaded java program test generation. IBM Systems Journal 41(3), 111–125 (2002)
9. Hamming, R.W.: Error detecting and error correcting codes. Bell System Technical Journal 26(2), 147–160 (1950)
10. Keith, J.M., Kroese, D.P.: Rare event simulation and combinatorial optimization using cross entropy: sequence alignment by rare event simulation. In: Proceedings of the 34th Winter Simulation Conference: Exploring New Frontiers, pp. 320–327. ACM, New York (2002)
11. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
12. Krause, E.F.: Taxicab Geometry: An Adventure in Non-Euclidean Geometry, Dover (1987)
13. Kullback, S., Leibler, R.A.: On information and sufficiency. Annals of Mathematical Statistics 22, 79–86 (1951)

14. Kumar, R., Garg, V.K.: Modeling and control of logical discrete event systems. Kluwer Academic Publishers, Dordrecht (1995)
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
16. Leblanc, T.J., Mellor-Grummy, J.M.: Debugging parallel programs with instant replay. IEEE Transactions on Computers 36(4), 471–481 (1987)
17. Margolin, L.: Cross-entropy method for combinatorial optimization. Master Thesis, The Technion (2002)
18. Mittal, N., Garg, V.K.: Debugging distributed programs using controlled re-execution. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 239–248 (2000)
19. Mittal, N., Garg, V.K.: Finding missing synchronization in a distributed computation using controlled re-execution. Distributed Computing 17(2), 107–130 (2004)
20. Netzer, R.H.B.: Optimal tracing and replay for debugging shared-memory parallel programs. In: Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging; also available as ACM SIGPLAN Notices 28(12), 1–11 (1993)
21. Paik, E.H., Chung, Y.S., Lee, B.S.: Chae-Woo Yoo. A concurrent program debugging environment using real-time replay. In: Proc. of ICPADS, pp. 460–465 (1997)
22. Rubinstein, R.Y.: Optimization of computer simulation models with rare events. European Journal on Operations Research 99, 89–112 (1997)
23. Rubinstein, R.Y.: 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)
24. Rubinstein, R.Y., Kroese, D.P.: The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning. In: Information Science and Statistics. Springer, Heidelberg (2004)
25. Sen, K., Agha, G.A.: Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)