Randomized Backtracking: Next Steps

Pavel Parízek, Ondřej Lhoták
David R. Cheriton School of Computer Science, University of Waterloo
{pparizek,olhotak}@uwaterloo.ca

***Abstract*—The use of randomized backtracking in state space traversal is a technique for efficient detection of errors that we proposed recently. In this paper we summarize the basic approach and results of our initial experiments, and then we discuss possible extensions and optimizations.**

***Keywords*-state space traversal, randomization, backtracking, error detection, Java Pathfinder**

I. Introduction

There exist many techniques based on state space traversal that aim to find errors in a reasonable time and avoid state explosion — directed search with heuristics that navigate towards the error state [6] [4] [5], random and parallel search [3] [7] [16], and context-bounded model checking [15] [9].

We proposed *randomized backtracking* [13] as yet another way to detect errors quickly with state space traversal. It allows backtracking also from states that still have some unexplored outgoing transitions, and uses random number choice to decide whether to backtrack early (and thus prune a part of the state space) or continue forward along some unexplored transition. Our experiments with a prototype implementation of the approach in Java Pathfinder (JPF) show that use of randomized backtracking significantly improves performance over existing techniques for many benchmark programs, i.e. fewer states are explored before some error is found. Still, much work remains to be done before the approach can be successfully applied to large concurrent programs.

In this paper we summarize the basic approach introduced in [13], outline possible extensions and optimizations of the basic approach, and discuss our future plans.

II. State Space Traversal with Randomized Backtracking

We introduced randomized backtracking only for explicit state depth-first traversal. We assume that each state is a snapshot of program variables and threads at one point on one execution path, and that each transition is a sequence of instructions executed by one thread where the last instruction is associated with a non-deterministic choice (e.g., thread scheduling choice).

A. Algorithm

Figure 1 shows the overall algorithm. Differences from the standard algorithm for depth-first state space traversal are highlighted by underlining.

The function enabled returns a set of transitions enabled in the state s that must be explored. A typical basic implementation of this function under our assumptions returns a set that contains (a) one transition for each thread runnable in the given state s in the case of a scheduling choice or (b) one transition for each non-deterministically selected data value (all of them being associated with the same thread). The function order implements the search order — it takes the given set of transitions and returns a list that determines the order in which the transitions are explored.

```
DFS_RB(threshold, strategy, ratio):
  visited = {}
  stack = [ ]
  push(stack, s0)
  explore(s0)

procedure explore(s)
  if error(s) then
    counterexample = stack
    terminate
  end if
  transitions = order(enabled(s))
  for tr ∈ transitions do
    depth = size(stack)
    if depth >= threshold then
      if rnd(0,1) > ratio(depth) return
    end if
    s' = execute(tr)
    if s' ∉ visited then
      visited = visited ∪ s'
      push(stack, s')
      explore(s')
      pop(stack)
      if backtrackAgain(strategy) return
    end if
  end for
end proc
```

Fig. 1. Algorithm for state space traversal with randomized backtracking

For each transition tr from the state s, the algorithm uses random number choice to decide whether (a) to move forward and execute the transition or (b) backtrack early to some previous state on the current path and ignore the remaining unexplored transitions from s. The function call rnd(0,1) returns a random number from the interval $\langle 0, 1\rangle$. After each backtracking step, i.e. after the recursive call to explore returns, the backtrackAgain procedure is used to determine whether the algorithm should backtrack further.

The process of state space traversal with randomized backtracking is controlled by three parameters: threshold, strategy, and ratio. Specific values of the parameters make a *configuration* of randomized backtracking.

B. Parameters

The value of the *threshold* parameter determines the least search depth (i.e., the number of transitions between the initial state and the current state) at which early backtracking is enabled. If the current search depth is smaller than the value of the threshold parameter, then backtracking is possible only when all outgoing transitions have already been explored.

The *strategy* parameter determines the length of backtrack jumps. When the algorithm decides to backtrack early from the state s, it can jump back over any number of previous transitions on the current path. We considered three strategies in our initial work: fixed, random, and Luby. Using the fixed strategy, the algorithm always backtracks over a single transition. When using the random strategy, a backtrack jump has a random length. When the Luby strategy [8] is used, the length of a backtrack jump with the index N (overall) is the number at the corresponding position in the Luby sequence $l_1, l_2, \ldots$, which is defined by the following expression:

$$\begin{aligned} l_i &= 2^{n-1}, \text{ if } i = 2^n - 1 \\ l_i &= l_{i-2^{n-1}+1} \text{ if } 2^{n-1} \leq i < 2^n - 1 \end{aligned}$$

The *ratio* parameter expresses the general preference for going forward along some unexplored transition over early backtracking. When the current state s has some unexplored outgoing transitions, then the algorithm backtracks early from s only if the randomly selected number from the interval $\langle 0, 1 \rangle$ is greater than the ratio R. The value of the ratio parameter can be defined as a constant number, or as a function of the search depth that is represented by an expression $R = 1 - d/c$, where d is the current search depth and c is an integer constant. The algorithm decides about early backtracking separately for each transition from s. A consequence is that each transition from s has a different probability of being explored — for the first transition in the list returned by the order function for s, the probability of being explored is equal to the ratio R, while for a transition with the index i the probability is R^i.

C. State Space Pruning

The use of randomized backtracking implies that parts of the state space are pruned by early backtracking from states with unexplored outgoing transitions, and thus only an incomplete traversal is performed. Error states may be reached faster due to early backtracking, if the algorithm prunes (skips) large state space fragments without any error states and avoids spending a lot of time exploring them. On the other hand, the algorithm can also prune state space fragments that contain error states, but that does not matter if at least some other error states are reached. It is not guaranteed that an error state will be reached when the randomized backtracking is used.

Setting the threshold parameter to a specific non-zero value prevents backtracking too early, so that an interesting part of the state space that may contain some errors is always reached. If the ratio is defined as a function of the search depth, then early backtracking (state space pruning) becomes more likely from states with a greater depth.

III. Initial Experiments

We implemented the depth-first state space traversal with randomized backtracking in JPF. The only change of the standard JPF is the use of a custom search driver, which performs the algorithm shown in Figure 1. Parameter values are specified using the configuration mechanism of JPF. All the strategies are hardwired in the search driver.

For our initial experiments, we have used seven small multi-threaded Java programs: the Daisy file system [14], the Elevator benchmark from the PJBench suite [11], and five small programs that are publicly available in the CTC repository [1] — Alarm Clock, Linked List, Producer Consumer, RAX Extended, and Replicated Workers. Basic characteristics of the programs are provided in Table I.

TABLE I
Benchmark programs

Program	Source code lines	Number of threads
Daisy file system	1150	2
Elevator	320	4
Alarm Clock	180	3
Linked List	185	2
Producer Consumer	135	7
RAX Extended	150	5
Replicated Workers	430	6

Only the Linked List benchmark already contained a concurrency error (race condition) that JPF could detect, so we manually injected concurrency errors into all the other benchmark programs. We created race conditions in all benchmarks except Linked List and Daisy by modifying the scopes of synchronized blocks. In the case of Daisy, we inserted assertions that are violated as a consequence of complex race conditions that already existed in the code but JPF could not detect them directly. Some benchmarks contained hard-to-find errors with a low density of error paths (Daisy, Elevator), while for others there was a high percentage of state space paths leading to error states.

The results of our initial experiments [13] show that usage of randomized backtracking significantly improves performance over existing techniques for many benchmark programs. We measure performance by the number of states that are explored before an error state is reached. However, the performance and ability to find errors both depend very much on the selected configuration and the outcome of the random number choice — there is a great variability of performance between different configurations and also between different JPF runs for a single configuration. Different configurations yield the best performance for individual benchmarks, but reasonably good performance for all benchmarks was achieved with the configuration (H, 0.9, random) for some program-specific threshold value H.

As an illustration, Table II shows selected results for these benchmarks: Elevator, Linked List, RAX Extended, and Replicated Workers. Each row contains values of the following metrics: the number of states processed before an error was found (mean μ, minimum, and maximum), and the percentage

TABLE II
SELECTED EXPERIMENTAL RESULTS

Configuration	Processed states			Error found
	μ	min	max	
Elevator				
(1) default search order	143373			100 %
(2) random search order	2399	1062	3833	100 %
(3) (50, random, $1 - d/100$)	270	255	293	100 %
(4) (50, fixed, 0.99)	358	253	424	100 %
(5) (100, fixed, $1 - d/50$)	2263	2227	2338	100 %
(6) (50, random, 0.9)	290	261	312	100 %
(7) (20, random, 0.9)	-	-	-	0 %
Linked List				
(1) default search order	328			100 %
(2) random search order	186	15	234	100 %
(3) (10, fixed, $1 - d/50$)	112	51	215	100 %
(4) (5, fixed, $1 - d/50$)	74	38	133	56 %
(5) (20, fixed, 0.9)	235	170	408	100 %
(6) (50, random, 0.9)	276	275	279	100 %
(7) (20, random, 0.9)	197	171	266	100 %
RAX Extended				
(1) default search order	1617			100 %
(2) thread interleavings	104			100 %
(3) (10, luby, 0.99)	97	86	113	100 %
(4) (5, fixed, 0.5)	60	8	313	67 %
(5) (50, fixed, 0.5)	1617	1617	1617	100 %
(6) (50, random, 0.9)	1617	1617	1617	100 %
(7) (20, random, 0.9)	441	401	491	100 %
Replicated Workers				
(1) default search order	9881			100 %
(2) context bound (10)	6585			100 %
(3) (50, fixed, 0.9)	148	95	278	100 %
(4) (50, fixed, 0.5)	339	71	1282	100 %
(5) (50, luby, $1 - d/20$)	6258	139	19190	100 %
(6) (50, random, 0.9)	1774	146	5891	100 %
(7) (20, random, 0.9)	-	-	-	0 %

of JPF runs for the given configuration that found an error. If no error was found by any JPF run for a given configuration, then columns for metrics related to the number of processed states contain the character "-". Rows of the table associated with each benchmark provide results for: (1) traversal with the default search order in JPF, (2) the existing technique with the best performance out of those currently implemented in JPF, (3) the configuration of randomized backtracking with the best average performance out of those where 100% of JPF runs found an error, (4) the configuration for which some JPF run achieved the best performance on the given benchmark over all configurations of randomized backtracking and all JPF runs for these configurations, (5) the configuration for which some JPF run achieved the worst performance on the given benchmark over all configurations of randomized backtracking and all JPF runs for these configurations, (6) configuration (H, random, 0.9) with the threshold $H = 50$, and (7) configuration (H, random, 0.9) with the threshold $H = 20$.

For example, the data in rows 1–4 for the benchmarks Elevator and Replicated Workers show the performance improvement over existing techniques. Rows 4–5 for Elevator and Replicated Workers show the variability between configurations and also between different JPF runs for one configuration. The data in row 4 for Linked List and RAX Extended show that the ability to find errors with randomized backtracking also varies over configurations, i.e. that some JPF runs do not find any error. Row 4 for RAX Extended also shows that better performance can be achieved when only at least 50 % of JPF runs are required to find some error.

Our implementation and the complete set of experimental results are available at the web site http://plg.uwaterloo.ca/~pparizek/jpf/spin11/.

When using our current algorithm with the configuration (H, random, 0.9), the selection of a good threshold value H with regard to the given program is very important. It influences the chance that JPF will find an error and also its performance. If the threshold is too low, almost no JPF run will find an error due to backtracking too early. If the threshold is too high, randomized backtracking will have almost no effect during traversal, because JPF will find an error at a search depth lower than the threshold value. These two cases are captured by row 7 for Elevator and row 6 for RAX Extended, respectively.

Although the use of randomized backtracking does not guarantee that an error is discovered, results of our experiments show that some error is discovered by all JPF runs (or by a very high percentage of JPF runs), as long as the threshold value is not too small. The values of the other two parameters of randomized backtracking (ratio and strategy) have much less influence on the ability to find errors than threshold.

IV. NEXT STEPS

Our initial experiments gave promising results, but much work still remains to be done before the approach can be successfully used to detect errors in large concurrent programs. The possible improvements that we have identified can be divided into the following categories: general extensions and optimizations, determining parameter values, integration with existing techniques, extension towards complete verification, and empirical evaluation on more complex programs. In the rest of this section we discuss our plans in each category.

A. General Extensions and Optimizations

Randomized backtracking is currently defined and implemented only for depth-first explicit state traversal, but it may work equally well or even better with other search algorithms. We will introduce randomness in a similar manner to breadth-first search (BFS) and mixed search algorithms, and perform experiments to see how useful these variants really are.

In the case of BFS, we will use random numbers in the selection of the next state to be explored from the queue. The parameters of randomized backtracking will most probably have a different meaning for BFS.

In the case of depth-first search with randomized backtracking, we will create a new version of the existing algorithm where the threads associated with transitions are considered in the decision whether to backtrack early or continue forward. This means, for example, to define two values of the ratio parameter — one for the case when the next transition according to the search order is associated with a different thread than the previous transition, and one for the case when

the same thread is associated with both transitions. We might also try different strategies for the length of backtrack jumps.

We could even try a mixed search that involves both DFS and BFS. The state space fragment with depth smaller than the given threshold would be explored using BFS, and then DFS would be performed from any state with depth greater than the threshold.

B. Determining Parameter Values

Much space for future research is in possible ways to determine useful parameter values and their combinations. We plan to investigate the following:

- automated setting of reasonable threshold values for the strategy (H, random, 0.9) based on some heuristics and static analysis,
- setting threshold and ratio based on (1) the shape of the state space and (2) some properties of execution paths (e.g., number of runnable threads), and
- dynamically changing parameter values during a JPF run based on some properties of the given program and its state space.

Our general goal is to try to find configurations of randomized backtracking which satisfy these properties: (1) they yield consistently good performance for many benchmark programs and (2) the results of individual JPF runs have small variability. Use of such configurations could lead to higher predictability of our approach with respect to error detection in a reasonable time. The configuration (H, random, 0.9) was picked based on our initial experiments and it works for the benchmark programs that we currently have, but it might not yield good performance for some other programs.

We also plan to explore the relation between parameters of randomized backtracking and parameters of other approaches. For example, there might be some relation between threshold values and bounds on the number of thread preemptions in context-bounded model checking.

C. Integration with Existing Techniques

Most existing techniques for efficient detection of errors based on state space traversal are implemented using custom versions of the functions order and enabled. Randomized backtracking does not require any changes in these two functions, and thus it can be easily combined with existing techniques.

We will evaluate combinations of randomized backtracking with selected other techniques to see how efficient they are and to identify the best combinations. The general issue is that randomized backtracking may negatively impact some existing techniques, so that they cannot be effectively combined. We will consider the following existing techniques: traversal with random search order [3] [16], directed search with a heuristic that prefers thread switches [6], and maybe some others.

Besides that, we will also try a combination of randomized backtracking with restarts of the state space traversal process (see our earlier work in [12]). An obvious possibility is to reinitialize the random number generator at each restart to ensure that a different sequence of random choices is explored after the restart.

D. Extension towards Complete Verification

The use of randomized backtracking does not guarantee that all errors will be found because of state space pruning. However, providing certain guarantees about soundness and coverage is very important for practical usefulness of any error detection approach. A popular example is bounded verification, where the algorithm (tool) guarantees finding all errors up to a certain bound (e.g., on the number of context switches [9] or the search depth [17]).

We plan to extend the current algorithm towards complete verification by (1) remembering all states from which JPF backtracked early and (2) exploring pruned transitions from those states after the random traversal finishes. This idea is similar to the approach proposed in [2], where the verification process is divided into two steps: a heuristic-based search is performed first that aims to find many errors very quickly, followed by an exhaustive traversal (as a long-running background job) that provides coverage guarantees. It is very hard to measure the coverage achieved by a run of JPF with randomized backtracking, i.e. how large a fragment of the state space was explored before an error was found, as the state space is constructed dynamically and therefore its size is not known a priori (or at any moment during the actual search).

Another possibility is to use random number choice also to control later exploration of pruned transitions. In each state s, the algorithm would choose between processing the current state s or going back to some other partially explored state (that was skipped before and has unexplored outgoing transitions). If the algorithm decides to go back, then the state to explore next could also be picked randomly from the list of states skipped before.

The main difference of our approach from existing techniques is that we would target stateful model checking, i.e. not stateless search like CHESS [10] and its extensions (e.g. [2]).

E. Empirical Evaluation

We would like to evaluate randomized backtracking with the proposed improvements on large and more complex Java programs, both sequential and concurrent.

The main challenge that we now face is to find large Java programs with interesting behaviors that we could use as benchmarks. We are looking for more complex benchmark programs than we currently have, most notably ones where each thread executes much more instructions. They should be large Java programs with errors, or at least variants of the Java programs with errors that really existed in the given original program at some point. We can inject some errors manually, but injected errors are often artificial. It is also difficult to inject hard-to-find errors, for whose discovery model checking tools are most useful compared to other approaches such as testing. On the other hand, the programs should not use Java libraries that are not fully supported by JPF (e.g., networking and graphics). This problem can be addressed by manual

simplification (removing library calls), but the result may be very different from the original program in terms of possible behaviors.

A good solution would be to have a public repository similar to [1], but one that contains more complex programs or at least models of complex programs.

Using the more complex benchmarks, we would evaluate state space traversal with randomized backtracking, and we would also compare randomized backtracking with other existing approaches to fast error detection that are not implemented in JPF (e.g., iterative context bounding [9]).

ACKNOWLEDGEMENT.

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Concurrency Tool Comparison repository, https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison
[2] K.E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. In PPoPP 2010, ACM.
[3] M.B. Dwyer, S.G. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. In ICSE 2007, IEEE CS.
[4] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed Explicit-State Model Checking in the Validation of Communication Protocols. International Journal on Software Tools for Technology Transfer, 5(2-3), 2004.
[5] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, and H. Aljazzar. Survey on Directed Model Checking. In 5th Workshop on Model Checking and Artificial Intelligence (MoChArt), LNCS, vol. 5348, 2009.
[6] A. Groce and W. Visser. Heuristics for Model Checking Java Programs. Journal on Software Tools for Technology Transfer, 6(4), 2004.
[7] G.J. Holzmann, R. Joshi, and A. Groce. Swarm Verification. In ASE 2008, IEEE CS.
[8] M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas Algorithms. Information Processing Letters, 47(4), 1993.
[9] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In PLDI 2007, ACM.
[10] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In OSDI 2008, USENIX.
[11] Parallel Java Benchmarks, http://code.google.com/p/pjbench
[12] P. Parizek and T. Kalibera. Efficient Detection of Errors in Java Components Using Random Environment and Restarts. In TACAS 2010, LNCS, vol. 6015.
[13] P. Parízek and O. Lhoták. Randomized Backtracking in State Space Traversal. In SPIN 2011, LNCS, vol. 6823.
[14] S. Qadeer. Daisy File System. Joint CAV/ISSTA special event on specification, verification and testing of concurrent software, 2004.
[15] S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In TACAS 2005, LNCS, vol. 3440.
[16] N. Rungta and E. Mercer. Generating Counter-Examples Through Randomized Guided Search. In SPIN 2007, LNCS, vol. 4595.
[17] A. Udupa, A. Desai, and S. Rajamani. Depth Bounded Explicit-State Model Checking. In SPIN 2011, LNCS, vol. 6823.