Efficient Detection of Errors in Java Components Using Random Environment and Restarts

Pavel Parizek and Tomas Kalibera

Distributed Systems Research Group, Department of Software Engineering,
Faculty of Mathematics and Physics, Charles University in Prague
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
`{parizek,kalibera}@dsrg.mff.cuni.cz`

Abstract. Software model checkers are being used mostly to discover specific types of errors in the code, since exhaustive verification of complex programs is not possible due to state explosion. Moreover, typical model checkers cannot be directly applied to isolated components such as libraries or individual classes. A common solution is to create an abstract environment for a component to be checked. When no constraints on component's usage are defined by its developers, a natural choice is to use a universal environment that performs all possible sequences of calls of component's methods in several concurrently-running threads. However, model checking of components with a universal environment is prone to state explosion.

In this paper we present a method that allows to discover at least some concurrency errors in component's code in reasonable time. The key ideas of our method are (i) use of an abstract environment that performs a random sequence of method calls in each thread, and (ii) restarts of the error detection process according to a specific strategy. We have implemented the method in the context of Java components and the Java PathFinder model checker. We have performed experiments on non-trivial Java components to show that our approach is viable.

1 Introduction

The current practice in the application of model checking to real-world programs is that model checkers are used mostly as tools for detection of specific types of errors in the code (e.g. concurrency errors like deadlocks and race conditions), since exhaustive verification of complex programs is not possible due to state explosion. In this paper we focus on the detection of concurrency errors in Java components using the Java PathFinder model checker (JPF) [17]. We use the term *component* to denote an open Java program that has a well-defined interface — this includes, for example, Java libraries and individual Java classes.

One of the problems in the application of JPF to Java components is that it accepts only a runnable Java program with the `main` method on input, but a Java component typically does not contain `main`. Behavior of a Java component depends on the context (environment) in which it is used, e.g. on the order that component's methods are called by its actual environment. A common solution is to create an abstract environment (a model of an actual environment) for the component subject to checking and apply a model checker (JPF) to a runnable Java program composed of the component and its abstract environment. An abstract environment for a Java component typically has the form of a fragment of a Java program that contains the `main` method. The abstract environment performs various sequences of calls of component's methods with various combinations of method parameters' values — the goal is to cover as many control-flow paths in the component's code as possible, and, when the focus is on the detection of errors, to trigger as many errors in the component's code as possible.

When no constraints on the order of calls of component's methods are defined by the developers and no knowledge about the target environment (where the component will be deployed) is available, then a natural choice is to use an abstract environment that runs several threads concurrently and performs all possible sequences of calls of component's methods with many different input values in each thread — a *universal environment*. Such an environment exercises the component very thoroughly and therefore triggers a high percentage of errors in the component's implementation (if there are some). Nevertheless, model checking of a non-trivial component with a universal environment is typically infeasible due to state explosion, even if only a few threads (2-3) are run in parallel by the environment. JPF typically runs out of available memory quite soon (in the order of minutes).

We propose to address this problem by model checking a component with an abstract environment that (i) performs a randomly selected sequence of method calls in each thread and (ii) runs exactly two threads in parallel — we use the term *random-sequence environment* to denote such an abstract environment. We restrict the number of threads to two for the reason of feasibility of model checking, and also because a recent study [10] showed that a great majority of concurrency errors in real-world programs involve only two threads.

The motivation behind this approach is to discover at least some errors in the component in reasonable time, when model checking with a universal environment is not feasible. We show that although the use of a random-sequence environment helps to reduce the time and memory needed to find an error with JPF in most cases, still JPF can run for a very long time for some components and random-sequence environments due to state explosion. The cause is that time and memory requirements of checking with JPF depend very much on the specific random-sequence environment that is used. Moreover, a result of the random choice of a sequence environment determines whether an error in the component is found by JPF, since some random-sequence environments would not trigger any errors in the component. The whole state space of the program composed of the component and a particular random-sequence environment is traversed by JPF in the case of an environment that does not trigger any error, and therefore the running time of JPF can be very long.

In order to avoid very long running times of JPF and to ensure that errors are found (assuming there are some in the component), we also propose to apply restarts of the error detection process. The key idea is that if the running time of JPF for a particular random-sequence environment exceeds a predefined limit, then (1) JPF is stopped, (2) a new random-sequence environment is generated, and (3) JPF is started again on the Java program composed of the component and the new random-sequence environment. This is repeated until JPF finds an error in the component with some random-sequence environment. Our approach is greatly inspired by existing work on restart strategies for various long-running software processes in general [13] and for search tree traversal in SAT solvers specifically [5, 8] — the goal of restarts is to improve performance (e.g., to decrease the response time). We show that the application of restarts to the error detection process significantly reduces the time and memory needed to find an error in a component, and, in particular, helps to avoid the long running times of JPF. Using our approach, errors in components' code are discovered by JPF in a reasonable time.

The rest of the paper is structured as follows. In Section 2 we describe Java components used for experiments and in Section 3 we provide information relevant to all experiments that we performed. We provide technical details of checking with universal environment and present the results of experiments in Section 4. Then we present the technical details and experimental results for checking with random-sequence environments and for application of restarts, respectively, in Sections 5 and 6. We evaluate our approach in Section 7, and then we discuss related work (Section 8) and conclude in Section 9.

2 Example Components

We have used three Java components of different complexity for the purpose of experiments: `AccountDatabase`, `ConcurrentHashMap`, and `GenericObjectPool`. All the three components contained known concurrency errors — either already present in the code or manually injected by us before the experiments. A short description of each component follows.

The `ConcurrentHashMap` component (2000 loc in Java) is a part of the `java.util.concurrent` package from the standard Java class library, as implemented in GNU Classpath (version 0.98) [20]. The component is an implementation of a map data structure that allows concurrent accesses and updates. We have manually injected a race condition into the Java code of the component.

The `GenericObjectPool` component (500 loc in Java) is a part of the Apache Commons Pool library (version 1.4) [19]. It represents a robust and configurable pool for arbitrary Java objects. Again, we have manually injected a race condition into the component's Java code.

The `AccountDatabase` component (170 loc in Java) is a part of the demo component application developed in the CRE project [1]. It works as a simple in-memory database for user accounts. The code of the component already contained a race condition.

3 General Notes on Experiments

Here we provide information that applies to all the experiments whose results are presented in Sections 4-7.

For each experiment, we provide total running time of the error detection process in seconds and, if it is relevant for the experiment, also the number of runs of the process and memory needed by the process in MBs. We have repeated each experiment several times to average out the effects of randomness. The values of numerical variables in the tables (except the number of runs) have the form *M +- CI*, where *M* stands for the mean of measured data and *CI* is the half-width of the 90% confidence interval.

All the experiments were performed on the following configuration (HW & SW): PC with 2xQuadCore CPU (Intel Xeon) at 2.3 GHz and 8 GB RAM, Gentoo Linux, Sun Java SE 6 Hotspot 64-bit Server VM. We have used the current version of Java PathFinder as of June 2009 and we limited the available memory for verification to 6 GB.

4 Checking Components with Universal Environment

In our approach, we have used a restricted form of a universal environment where only two threads run concurrently. Each thread performs a potentially infinite loop (termination of the loop depends on non-deterministic choice) and calls a non-deterministically selected method of the component in each iteration. The Java code of each thread corresponds to the template in Figure 1a. Since JPF explores the options of a non-deterministic choice in a fixed order from the lowest to the highest (from 1 to N in case of code on Figure 1a, where N is the number of component's methods), we eliminate the dependence of results of checking with JPF on a specific order of component's methods by randomization — the method to be called for a particular value of the non-deterministic choice (via `Verify.getInt(X)`) is determined randomly during generation of the environment's code.

Input data for the components (e.g., method parameters) are specified in a Java class that works as a container for the data values [15]. The environment then retrieves the parameter values from the Java class when it calls methods of the component. A user has to create the specification of data values manually such that for each method m of the component, all paths in the control-flow graph (CFG) of m are covered (explored by JPF) — for each path p in the CFG of m, at least one combination of values of m's parameters should be specified that triggers p when m is called by the environment.

The results of experiments for checking components with a universal environment of the restricted form, where only two threads run concurrently, are listed in Table 1. JPF run out of available memory (6 GB) in all experiments and therefore it found no errors — this clearly illustrates that JPF checking even

```
while (Verify.getBoolean())
{
  int idx = Verify.getInt(X);
  if (idx == 1) comp.method1(..);
  if (idx == 2) comp.method2(..);
  ...
  if (idx == N) comp.methodN(..);
}
```

a)

```
 int len = Random.getInt(2*N);
 for (int i = 1; i <= len; i++) {
   int idx = Random.getInt(N);
   if (idx == 1) comp.method1(..);
   ...
   if (idx == N) comp.methodN(..);
}
```

b)

Fig. 1. Fragment of Java code of a single thread (a) in a universal environment and (b) in a random-sequence environment

with the restricted universal environment is not feasible for non-trivial Java components. We present only the running times of JPF in the table to show how fast it run out of memory.

Component	**JPF running time**
AccountDatabase	921 ± 121 s
ConcurrentHashMap	1426 ± 377 s
GenericObjectPool	1034 ± 308 s

Table 1. Results for checking components with a universal environment

5 Random-Sequence Environments

Similarly to a universal environment, a random-sequence environment calls methods of a component in two concurrently-running threads and retrieves method parameter values from the Java class provided by the user. The key difference is that, in the case of a random-sequence environment, each thread performs a randomly selected sequence of calls of the component's methods. The length of the sequence is a random number from the interval $[1, 2 \times |M|]$, where M stands for the set of component's methods — we set the maximal length of the sequence to $2 \times |M|$ to ensure that the sequence contains multiple calls of several methods with a high probability. The Java code of each thread corresponds to the template in Figure 1b, where N is the number of component's methods. The need for randomness in the selection of a sequence environment is motivated by the absence of any knowledge about the component's implementation and expected usage — in particular, it is not known in advance which sequence environments trigger an error and which do not, and therefore it is not possible to select only such sequence environments that trigger errors.

The results of experiments for checking components with random-sequence environments are listed in Table 2. Since some random-sequence environments do not trigger any errors, we distinguish between two groups of results based on whether JPF found an error in the component's code (value "yes" in the "Error found" column), or traversed the whole state space and found no error (value "no" in the "Error found" column). Note that JPF did not run out of available memory in any experiment with random-sequence environments.

Component	**Error found**	**Runs**	**Time**	**Memory**
AccountDatabase	yes	23	1040 ± 802 s	334 ± 134 MB
	no	17	12420 ± 8861 s	1784 ± 738 MB
ConcurrentHashMap	yes	37	173 ± 108 s	104 ± 19 MB
	no	3	14 ± 10 s	70 ± 12 MB
GenericObjectPool	yes	22	1934 ± 2208 s	186 ± 78 MB
	no	18	10230 ± 7979 s	1136 ± 789 MB

Table 2. Results for checking components with random-sequence environments

The experimental results show that running times of JPF vary to a great degree independently of whether JPF found an error or not. Although the experimental results suggest that running times of JPF generally tend to be shorter when JPF finds an error and much longer when JPF is applied to a component in an environment that does not trigger any error (this is especially visible for `AccountDatabase` and `GenericObjectPool`), the results show that JPF can run very long even when it finds an error in the component. The Figures 2, 3 and 4 show graphs of the empirical cumulative distribution function for the experimental results for each component. A point $[t, p]$ in a graph means that the running time of JPF (regardless of whether JPF finds an error or not) will be shorter than t with the probability p. The time axis has a logarithmic scale in each graph. The graphs indicate that if JPF is running for a long time and has not found an error yet, then the chance that it will find an error (or terminate with no error found) in reasonable time is significantly decreasing. Solving this issue was our primary goal in the application of restarts to the error detection process.

6 Restart Strategies

Based on [13], we define a *restart strategy* as a sequence $(t_1, t_2, t_3, \ldots)$ of times at which the error detection process is restarted — i.e. as a sequence of *restart times*. The key idea is that if in the run n JPF either (a) does not finish in time t_n or (b) traverses the whole state space in a time shorter than t_n and does not find any error or (c) runs out of memory, then the whole error detection process is restarted with time limit t_{n+1} for the JPF run, and so on. We say that a run of the error detection process involves one or more runs of JPF (*iterations* of the

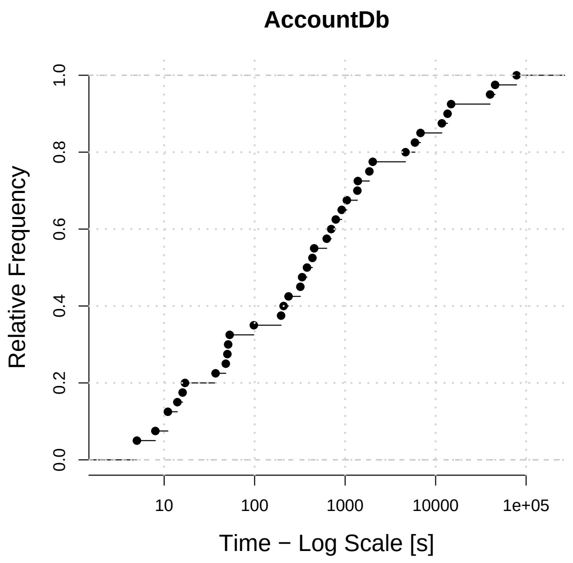

Fig. 2. Graph of the empirical cumulative distribution function for results of checking `AccountDatabase` with random-sequence environments

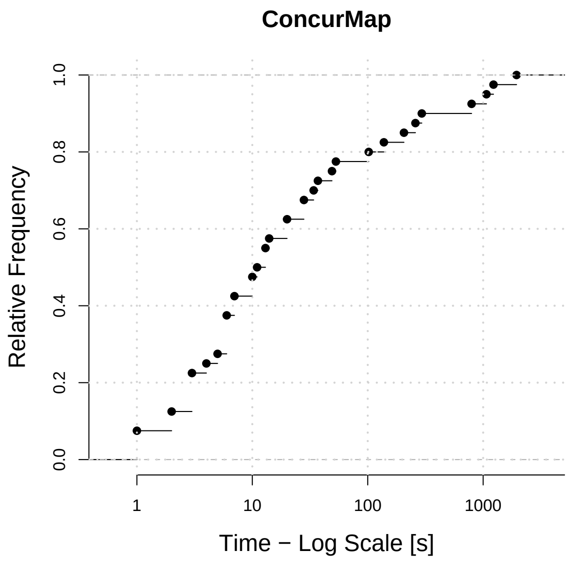

Fig. 3. Graph of the empirical cumulative distribution function for results of checking `ConcurrentHashMap` with random-sequence environments

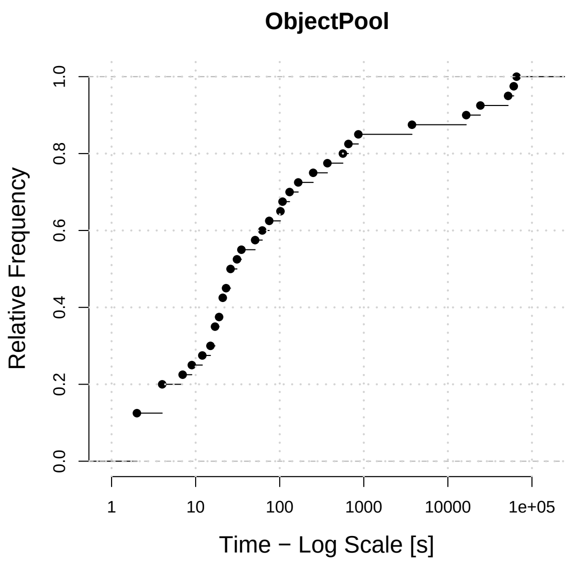

Fig. 4. Graph of the empirical cumulative distribution function for results of checking `GenericObjectPool` with random-sequence environments

run-stop-generate-restart loop) and terminates when a specific JPF run finds an error. For all iterations except the last one, it holds that either JPF traverses the whole state space before the restart time (and finds no error) or JPF runs out of the time limit (restart time). Restart of the error detection process involves three steps: (i) terminating the current run of JPF, (ii) generating a new random-sequence environment, and (iii) starting a new run of JPF on the Java program composed of the component and new environment.

The key challenge is to determine the best possible restart strategy. In this paper, we focus on the use of a predefined application-independent strategy, which is the typical approach of SAT solvers. Another possible approach would be to compute the strategy on the basis of a metric of component's code or state space traversal process, where the metric can be static, i.e. measured before a JPF run, or dynamic, i.e. measured on-the-fly during JPF checking — we discuss this approach in more detail in Section 9.

We have identified three restart strategies, which are widely and successfully used in state-of-the-art SAT solvers (e.g., [3, 4]) and also in search problems of other kinds: fixed strategy, Luby strategy, and Walsh geometric strategy. We performed experiments for all the three strategies to find which gives the best results in the case of error detection with JPF.

Fixed strategy [5] is a constant sequence $S = t, t, t, \ldots$, where t represents the fixed restart time.

Luby strategy [12] is a sequence $S = k_1 u, k_2 u, k_3 u, \ldots$, where u is a restart time unit and k_i is computed using the following expression:

$$k_i = 2^{n-1}, \text{ if } i = 2^n - 1$$
$$k_i = k_{i-2^{n-1}+1} \text{ if } 2^{n-1} \leq i < 2^n - 1$$

The first few elements of the sequence k_i are $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8$.

Walsh geometric strategy [18] is a sequence $S = u, ru, r^2 u, r^3 u, \ldots$, where u is a restart time unit and $r > 1$ is a ratio of the geometric sequence. In our case we used $r = \sqrt{2}$.

The Luby strategy was proposed in [12] for speedup of randomized algorithms of the Las Vegas type with unknown probability distribution of running time. However, the theoretical results presented in [12] (e.g., the bound on the running time with respect to optimal time) are not applicable in our case of model checking components with random-sequence environments, since our algorithm is not strictly of the Las Vegas type — in our case, an input of the algorithm is different for each run, since different sequence environments are randomly selected. Similarly, the model for restart strategies proposed in [13] cannot be used in our case, since the probability distribution of JPF running times for all random-sequence environments is not known in advance. Knowledge of the probability distribution is one of the requirements of the model.

We present results of experiments for all combinations of the three restart strategies — fixed, Luby and Walsh — and six different values of restart time unit — 1 second, 3 seconds, 10 seconds, 30 seconds, 60 seconds, and 600 seconds. We selected these values of restart time unit in order to cover a wide range of situations, including corner cases such as too early restarts and too late restarts. Restart time unit of 1 second is used only for the Luby and Walsh strategies, since it is too small for the fixed strategy which does not extend the restart time adaptively — initialization would form a significant part of JPF's running time in that case.

The results of experiments are listed in Table 3 for `AccountDatabase`, in Table 4 for `ConcurrentHashMap`, and in Table 5 for `GenericObjectPool`. Values in the "Time" column represent the total running time of the error detection process, i.e. the time needed to detect an error. Total running time of a single run of the error detection process equals to the sum of JPF running times in individual iterations. Similarly, values in the "Memory" column represent the memory needed by the error detection processes. Memory needed by a single run of the error detection process equals to the maximal value over all iterations in the run. Note that JPF did not run out of memory in any experiment for restarts of the error detection process.

Experimental results show that extremely long running times of JPF can be avoided by restarts of the error detection process. On average, the time needed to detect an error is the lowest when the fixed strategy with a small restart time

Strategy	Unit time	Time	Memory
Fixed	3 s	197 ± 47 s	72 ± 1 MB
	10 s	153 ± 44 s	108 ± 4 MB
	30 s	152 ± 46 s	129 ± 11 MB
	60 s	287 ± 81 s	169 ± 14 MB
	600 s	1212 ± 488 s	430 ± 71 MB
Luby	1 s	215 ± 43 s	116 ± 8 MB
	3 s	240 ± 62 s	130 ± 11 MB
	10 s	158 ± 46 s	134 ± 14 MB
	30 s	244 ± 77 s	158 ± 14 MB
	60 s	323 ± 111 s	172 ± 20 MB
	600 s	609 ± 172 s	385 ± 78 MB
Walsh	1 s	216 ± 111 s	133 ± 19 MB
	3 s	663 ± 529 s	241 ± 90 MB
	10 s	284 ± 85 s	174 ± 23 MB
	30 s	270 ± 90 s	179 ± 27 MB
	60 s	387 ± 135 s	211 ± 37 MB
	600 s	1250 ± 545 s	486 ± 148 MB

Table 3. Experimental results for error detection with restarts for `AccountDatabase`

Strategy	Unit time	Time	Memory
Fixed	3 s	19 ± 4 s	57 ± 1 MB
	10 s	13 ± 3 s	69 ± 6 MB
	30 s	29 ± 9 s	79 ± 7 MB
	60 s	26 ± 8 s	78 ± 8 MB
	600 s	136 ± 75 s	97 ± 15 MB
Luby	1 s	21 ± 4 s	57 ± 3 MB
	3 s	15 ± 3 s	61 ± 3 MB
	10 s	18 ± 5 s	71 ± 5 MB
	30 s	18 ± 6 s	70 ± 7 MB
	60 s	43 ± 12 s	89 ± 9 MB
	600 s	73 ± 37 s	82 ± 10 MB
Walsh	1 s	18 ± 7 s	59 ± 4 MB
	3 s	15 ± 4 s	64 ± 5 MB
	10 s	24 ± 5 s	75 ± 5 MB
	30 s	22 ± 7 s	72 ± 6 MB
	60 s	33 ± 9 s	87 ± 9 MB
	600 s	61 ± 36 s	83 ± 12 MB

Table 4. Experimental results for error detection with restarts for `ConcurrentHashMap`

Strategy	Unit time	Time	Memory
Fixed	3 s	28 ± 6 s	53 ± 1 MB
	10 s	29 ± 7 s	77 ± 3 MB
	30 s	79 ± 20 s	93 ± 6 MB
	60 s	120 ± 29 s	127 ± 14 MB
	600 s	549 ± 182 s	310 ± 66 MB
Luby	1 s	43 ± 9 s	64 ± 4 MB
	3 s	33 ± 12 s	65 ± 5 MB
	10 s	58 ± 16 s	83 ± 6 MB
	30 s	82 ± 24 s	103 ± 12 MB
	60 s	89 ± 37 s	117 ± 19 MB
	600 s	594 ± 278 s	300 ± 63 MB
Walsh	1 s	50 ± 21 s	74 ± 9 MB
	3 s	47 ± 12 s	80 ± 9 MB
	10 s	104 ± 73 s	102 ± 24 MB
	30 s	75 ± 26 s	105 ± 18 MB
	60 s	413 ± 398 s	173 ± 39 MB
	600 s	1197 ± 674 s	391 ± 100 MB

Table 5. Experimental results for error detection with restarts for `GenericObjectPool`

unit (1, 3, 10 or 30 seconds) is used. However, the results also show that use of too small a restart time unit (1 or 3 seconds) may actually increase the time needed to detect an error. The error detection process is restarted too early for JPF to find an error in such a case.

7 Evaluation

Results of all experiments that we performed show that the combination of checking with random-sequence environment and restarts of the error detection process has three main benefits: (1) errors are discovered in very short time in most cases and in reasonable time in the other cases, (2) extremely long running times of JPF are avoided, and (3) JPF does not run out of memory. Compared to checking with random-sequence environments only, the use of restarts always leads to discovery of an error (assuming there are some errors in the component) and an error is found in shorter time in most cases. Some random-sequence environments do not trigger any errors and thus none can be discovered by JPF, when such an environment is used. Nevertheless, when the checked component does not contain any errors, then the error detection process would be restarted again and again — it is up to the user to terminate the process after a reasonable time (when no error is found after several restarts). Compared to checking with a universal environment, an error is on average found in shorter time using random-sequence environments and restarts than it takes JPF to run out of memory when a universal environment is used.

As for the choice of a restart strategy and restart time unit, best results are achieved using the fixed strategy and short restart times. However, it is not true that the shortest restart time always provides the best result. Optimal restart time most probably depends on whether concurrency errors in the component are "shallow" or "deep". Shallow errors exhibit themselves in many thread interleavings (on many state space paths) and therefore can be found "early in the search" by JPF, while deep errors occur only in rare corner cases (for specific thread interleavings) and thus it takes JPF more time to find them. Use of short restart times would give better results in discovery of shallow errors than for deep errors.

Table 6 summarizes the results of experiments with different approaches described in this paper and also presents the results of application of a technique described in [14] on the same components. The technique described in [14] is our previous work in automated construction of abstract environment for Java components with the goal of efficient detection of concurrency errors. It is based on a combination of static analysis and a software metric — static analysis is used to identify method sets whose parallel execution may trigger a concurrency error, and the metric is used to order the sets by the likeliness that an error will really occur. Table 6 provides the following information for each component:

– the time it takes JPF to run out of memory when checking the component with a universal environment (the "Univ env" column),
– the time needed to find a concurrency error in the component when only a random-sequence environment is used (the "Random env" column),
– the time to find an error using a combination of checking with a random-sequence environment and restarts of the error detection process (in the "Restarts" column) — the lowest time over all restart strategies and restart time units is presented, and
– the time to detect an error using the technique described in [14] (the "Prev work" column) — the lowest time over all configurations of the metric is presented in the table.

Component	**Univ env**	**Random env**	**Restarts**	**Prev work**
AccountDatabase	921 ± 121 s	1040 ± 802 s	152 ± 46 s	114 s
ConcurrentHashMap	1426 ± 377 s	173 ± 108 s	13 ± 3 s	64 s
GenericObjectPool	1034 ± 308 s	1934 ± 2208 s	28 ± 6 s	1590 s

Table 6. Summary and results for a technique proposed in previous work

Results in Table 6 show that the method proposed in this paper is an improvement over our previous work [14]. The proposed method gives significantly better results for the `ConcurrentHashMap` and `GenericObjectPool` components, while both methods give comparable results in case of the `AccountDatabase` component.

8 Related Work

Significant amount of work has been done in various optimizations aiming towards more efficient search for errors in program code via model checking. The existing approaches include heuristics for state space traversal, context-bounded model checking, and a combination of model checking with runtime analysis.

Heuristics for state space traversal are typically used to address state explosion with the goal of detection of specific errors in reasonable time and memory — for discovery of concurrency errors, a heuristic that prefers aggressive thread scheduling [6] can be used.

The idea behind context-bounded model checking [2, 16] is to check only those executions of a given program that involve bounded number of thread context switches. The bound can apply to all threads together [16] or to each thread separately [2].

An example of a technique based on the combination of runtime analysis with model checking is [7]. The key idea of [7] is that runtime analysis is performed first with the goal of detecting potential concurrency errors in a program, and then a model checker (JPF) is run on the same program, using counterexamples provided by the runtime analysis as a guide during state space traversal.

A common characteristic of the approaches described above is that, like the method proposed in this paper, they sacrifice completeness of checking for the purpose of efficient detection of errors. Nevertheless, the existing approaches are complementary to the proposed method — they could be applied during model checking of a runnable program, i.e. during a single run of the error detection process, to reduce the time needed to find an error even further.

9 Summary and Future Work

We have proposed a method for efficient detection of concurrency errors in Java components with Java PathFinder, which is based on random-sequence environments and restarts of the error detection process according to a predefined application-independent strategy. Results of experiments that we performed show that the application of the proposed method significantly reduces the time and memory needed to find errors in components' code. In particular, JPF does not run out of memory as in the case of checking with a universal environment and extremely long running times of JPF, which occur in checking with random-sequence environments only, are also avoided by using restarts.

Although the proposed method is promising, there is a large space for improvements and optimizations that may further reduce time needed to find errors in the code. Moreover, we focused only on sequential and static restart strategies in this paper, but it is possible to use also other kinds of restart strategies. We will investigate some of the following approaches in the future:

- Use of dynamic restart strategies, e.g. such as proposed in [9], in which case the restart time could be determined dynamically during a JPF run using a heuristic. The heuristic could be based on the time JPF is already running

or on the (estimated) size of the already traversed part of the state space (on the number of explored branches).
- Use of parallel restart strategies, e.g. based on the ideas and results published in [11]. The key idea would be to increase the chance that an error is found in shorter time by running several instances of the error detection process in parallel.
- Use of metrics of component's code to determine statically, i.e. before the start of the error detection process, the restart strategy and restart time.

Variants of the proposed method could be applied also to detection of other kinds of errors. For example, errors like null pointer exceptions or assertion violations often occur only for specific inputs (method parameters) — the idea would be to create an abstract environment that calls component's methods with randomly selected parameter values. We also plan to evaluate the proposed method on multiple larger case studies.

Acknowledgments. This work was partially supported by the Grant Agency of the Czech Republic project 201/08/0266 and by the Ministry of Education of the Czech Republic (grant MSM0021620838). We also thank Nicholas Kidd for his valuable comments and suggestions.

References

1. J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Component Reliability Extensions for Fractal Component Model, 2006, `http://kraken.cs.cas.cz/ft/public/public_index.phtml`.
2. M.F. Atig, A. Bouajjani, and S. Qadeer. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads, In Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), LNCS, vol. 5505, 2009.
3. A. Biere, PicoSAT Essentials, In Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, 2008.
4. N. Een and N. Sorensson. An Extensible SAT-solver, Proceedings of 6th International Conference On Theory and Applications of Satisfiability Testing, LNCS, vol. 2919, 2003.
5. C.P. Gomes, B. Selman, and H.A. Kautz. Boosting Combinatorial Search Through Randomization, Proceedings of AAAI'98.
6. A. Groce and W. Visser. Heuristics for Model Checking Java Programs, International Journal on Software Tools for Technology Transfer, vol. 6, no. 4, 2004.
7. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs, In Proceedings of the 7th SPIN Workshop on Model Checking of Software, LNCS, vol. 1885, 2000.
8. J. Huang. The Effect of Restarts on the Efficiency of Clause Learning, Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), 2007.
9. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic Restart Policies, In Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02), AAAI Press, 2002.

10. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics, In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), ACM, 2008.
11. M. Luby and W. Ertel. Optimal Parallelization of Las Vegas Algorithms, In Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS'94), LNCS, vol. 775, 1994.
12. M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas Algorithms, Information Processing Letters, 47(4), 1993.
13. A.P.A. van Moorsel and K. Wolter. Analysis of Restart Mechanisms in Software Systems, IEEE Transactions on Software Engineering, vol. 32, no. 8, 2006.
14. P. Parizek, J. Adamek, and T. Kalibera. Automated Construction of Reasonable Environment for Java Components, To appear in Proceedings of International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2009), ENTCS, 2009.
15. P. Parizek and F. Plasil. Specification and Generation of Environment for Model Checking of Software Components, In Proceedings of International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006), ENTCS, 176(2), 2007.
16. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software, In Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), LNCS, vol. 3440, 2005.
17. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs, Automated Software Engineering Journal, vol. 10, no. 2, 2003.
18. T. Walsh. Search in a Small World, In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 99), 1999.
19. Apache Commons Pool, `http://commons.apache.org/pool/`.
20. GNU Classpath, `http://www.gnu.org/software/classpath/`.