Automated Construction of Reasonable Environment for Java Components

Pavel Parizek, Jiri Adamek, Tomas Kalibera

Distributed Systems Research Group
Department of Software Engineering, Charles University in Prague
{parizek,adamek,kalibera}@dsrg.mff.cuni.cz

Abstract

In software component verification, one of the challenges is model checking of isolated components. The environment of an isolated component is unknown, and therefore a part of an input to a model checker is missing. This problem can be addressed via automated generation of an artificial environment — component and its environment form a complete program that can be verified using common model checkers.
Focusing on concurrency errors in Java components, we propose to automatically generate a reasonable artificial environment that makes efficient detection of concurrency errors with Java PathFinder possible. Such an environment executes in parallel those component's methods that interact via concurrency constructs of Java and thus likely contain concurrency errors. We employ static code analysis to identify sets of methods to be executed in parallel and a metric to order the sets according to the degree of interaction. Benefits of the technique are illustrated on results of experiments performed on real-life Java components.

Keywords: Software components, model checking, environment modeling, state explosion, concurrency errors, software metric

1 Introduction

Development of reliable and robust software systems often involves decomposition of the whole system into components, and application of formal methods, such as model checking [4], for discovery of bugs in components' code. In this paper, we focus on components implemented in Java — we use the term *component* to denote all kinds of open systems in Java that have a well-defined interface in the form of a set of public methods (provided by one or more classes), like libraries and plug-ins.

To support typical development process of component software, it is important to be able to verify not only the whole component application, but also an *isolated software component*. The reason is that the components can be developed independently — the *real environment* of a component (e.g. the rest of an application) can be unknown at the component development time. Moreover, modular verification — checking one component at a time — helps to increase scalability of model checking, since a component typically has a smaller state space than the whole system.

Nevertheless, it is in general not possible to apply model checking directly to the code of an isolated software component, since it is an open system, while model checking works only for closed systems. Component's behavior depends on the context in which it is used, i.e. on the way the real environment interacts with the component. Therefore, a part of the information needed for the verification is missing. We call this issue the *problem of missing environment*. In case of Java components the problem can be illustrated by the fact that common model checkers for Java (JPF [22], Bogor [18]) work only on complete Java programs with the `main` method — in particular, they cannot be applied to isolated components that do not contain `main`.

A common solution to the problem of missing environment is to construct an *artificial environment* [15]. In the context of verification of Java components, the artificial environment takes a form of a fragment of Java program that simulates the behavior of a specific real environment — the artificial environment together with the component's code form a complete Java program (closed system) that can be model checked. The artificial environment typically consists of a driver that calls methods of the component and stubs for dependencies of the component.

Although the Java code of an artificial environment can be written by hand [10], the preferred way is to generate it automatically from a high-level model of its behavior [20][14]. The model can be written by hand or constructed automatically, e.g., based on the behavior of the component subject to checking [5].

The key challenge of automated environment generation is how to define the model of behavior of the artificial environment — generation of Java code from the model is straightforward [20][14]. The important aspects of the behavior model are: coverage of the component's code, liability of the verification to state explosion [4], and automation of the model construction. More specifically, it is in general hard to construct an artificial environment that (i) exercises the component in a way that triggers most of the errors in component's code and thus allows the model checker to find them, and (ii) is simple enough to allow the model checker to detect at least some of the errors before it runs out of memory due to state explosion [21].

In this paper, we aim at efficient detection of concurrency errors (deadlocks, race conditions) in Java components using the Java PathFinder model checker (JPF) [22]. Therefore, in our case the challenge is to decide what the behavior of the artificial environment should be in order to make it possible for JPF to find at least some concurrency errors before it runs out of available memory or reasonable time due to state explosion. Note that although state explosion is typically caused by a high number of parallel threads in the program and/or by big data domains, we focus only on the level of parallelism, since big data domains are not a problem in our case (see the end of Sect. 1 for details).

Focusing on detection of concurrency errors in Java code, we have identified the following approaches to automated construction of the behavior model of an artificial environment, which can be employed using the existing techniques:

- *Universal environment* [6] is the most general possible environment: it calls each method of a component for an arbitrary finite number of times, and in parallel with other calls of all the methods (including parallel calls of the same method). It allows JPF to find all the errors, and can be easily constructed in an automated way. On the other hand, it is not possible to use such an environment in most cases, since verification with JPF is then prone to state explosion due to high number of parallel threads, even if the well-known approaches for addressing state explosion are used.

- *Random selection of method pairs* is based on the fact that race conditions and deadlocks can be triggered using only two parallel threads in many cases. The behavior model of an artificial environment defines a sequence of pairs of methods that must be executed in parallel (a sequence of *parallel method pairs*), and the generated environment executes all the pairs in the sequence, one at a time. A parallel method pair is defined for each pair of component's methods and the pairs are ordered randomly — this way, JPF attempts to check parallel execution of all pairs of component's methods. The advantages of this approach are (i) relatively high coverage — all errors triggered by parallel execution of two threads can be found — and (ii) easy automated generation of the environment's behavior model. Checking with JPF is less prone to state explosion than if the universal environment is used, but traversing the state space of many parallel pairs may take more time than traversing the state space of a single tuple. Moreover, it is not certain that an error will be detected by JPF in reasonable time, since the parallel method pairs are ordered randomly — those involving errors may be at the end of a sequence of pairs.
- *Selection of method pairs via search for suspicious patterns* is an approach for construction of the environment's behavior model that we first proposed in [13]. It is based on heuristic static analysis of code that searches for specific suspicious concurrency-related patterns in the Java byte code of pairs of component's methods. The environment's behavior model is defined as a sequence of parallel method pairs, which involves only those method pairs that contain potential concurrency errors according to the static analysis. The technique typically makes the detection of concurrency errors in Java code with JPF possible in reasonable time. However, it also has a drawback — it supports only patterns that involve pairs of Java methods and, naturally, the supported patterns do not cover all possible concurrency errors in Java. Therefore, the coverage is lower than in the case of universal environment and random method pairs — some errors are detected in low time, but other are not found at all.

All three approaches have certain drawbacks with respect to coverage of component's code, liability of checking with JPF to state explosion, and automation of model construction. For example, checking with universal environment is prone to state explosion, while use of patterns allows to find only some concurrency errors.

Therefore, we propose a technique for automated construction of the reasonable model of environment's behavior that addresses the drawbacks of the approaches described above. The key idea of the technique is construction of the environment's behavior model directly from the component's Java code via combination of:

(i) a simple static analysis that acquires specific information about interaction among component's methods via concurrency-related constructs of Java (e.g. accesses to shared variables), and

(ii) a software metric that measures the degree of interaction among methods.

To be more specific, only those pairs (and sets) of methods, whose parallel

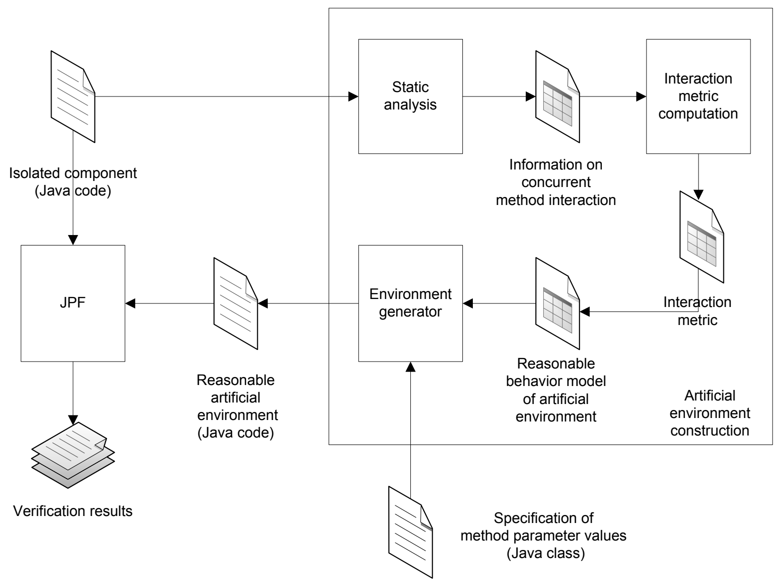
Isolated component (Java code)
Static analysis
Information on concurrent method interaction
Interaction metric computation
Interaction metric
JPF
Reasonable artificial environment (Java code)
Environment generator
Reasonable behavior model of artificial environment
Artificial environment construction
Verification results
Specification of method parameter values (Java class)

Fig. 1. The whole verification process — the big box on the right side represents the construction of reasonable artificial environment, and the smaller boxes inside it show the concrete approach to the construction.

execution should be checked with JPF for the presence of concurrency errors, are selected by the static analysis and metric — the behavior model then defines a sequence of the selected parallel pairs and sets.

Technically, we express the behavior model of the reasonable artificial environment in behavior protocols [16][14], which is a formalism for component behavior specification developed in our group. Values of component's method parameters have to be provided by the user in a Java class that works as a container for the values (see [14] for details). The user is responsible for defining the smallest possible sets of parameter values that cover all control-flow paths in component's code — this way, big data domains (one of the causes of state explosion) can be avoided. Having the behavior model defined in behavior protocols and the specification of parameter values in a Java class, the *environment generator* [12] is used to generate the Java code of the artificial environment from the model, and then JPF is applied on the complete program composed of the component and its artificial environment in order to find actual concurrency errors in the component's code (in case there are some). The whole process is shown in Fig. 1.

To summarize, the main contribution of this paper is a technique for efficient detection of concurrency errors in Java code of isolated components with JPF, which involves (i) automated generation of a reasonable artificial environment (Java code) from its behavior model defined in behavior protocols, and (ii) automated construction of the behavior model via static analysis and software metric.

The rest of the paper is structured as follows. Section 2 introduces a simple example that is used for illustration of the key ideas of the technique. Section 3 presents the details of the proposed technique — in particular the software metric that measures degree of interaction among Java methods via concurrency constructs. Section 4 provides an evaluation of the technique. The rest of the paper contains related work and conclusion.

2 Running Example

Throughout the whole paper, we illustrate various aspects of our technique on Java code of a simple component `OrderProcessor` (Fig. 2) — its code is free of concurrency errors for the purpose of simplicity.

```
public class OrderProcessorImpl implements IOrderProcessor {
  private Map orders = new HashMap();
  private Map invoices = new HashMap();

  public int placeOrder(int count) {
    synchronized (orders) {
      Order o = new Order(getId(), count, false);
      orders.put(o.id, o);
      return o.id;
    }
  }

  public double confirmOrder(int id) {
    synchronized (orders) {
      synchronized (invoices) {
        Order o = (Order) orders.get(id);
        o.shipped = true;

        Invoice i = new Invoice(o.id, o.count);
        invoices.put(i.orderId, i);
        return i.price;
      }
    }
  }

  public void payOrder(int id, double money) {
    synchronized (invoices) {
      Invoice i = (Invoice) invoices.get(id);
      if (i.price <= money) i.paid = true;
    }
  }
}
```

Fig. 2. Java code of the `OrderProcessor` component

Note that the `placeOrder` and `payOrder` methods do not interact via concurrency constructs of Java, since they access different shared variables and do not use a common lock object — the former accesses the `orders` variable, while the latter accesses only `invoices`.

3 Construction of the Behavior Model of a Reasonable Environment

The idea behind our technique is that a concurrency error can occur during parallel execution of two or more Java methods only if the methods interact via concurrency-related constructs of Java, e.g. if they access the same shared variable. Extending this observation to a heuristic, we assume that high interaction among methods makes concurrency errors more likely — Java code of the methods with high interaction contains a high number of concurrency-related constructs that might be used in a wrong way.

We define the *degree of interaction* among Java methods as a rigorous assessment (via a software metric) of mutual interaction among the methods via concurrency-related constructs of Java. Technically, Java methods feature mutual interaction, if their implementations use the following constructs:

- read/write accesses to shared variables — objects, fields, local variables and array elements,
- mutual exclusion via `synchronized` blocks and methods on the same objects (monitors), and
- synchronization via calls of the `wait` and `notify` methods on the same monitors.

Nevertheless, both `synchronized` blocks and calls of `wait`/`notify` methods can be considered as special cases of accesses to shared variables, since in Java byte code, locking and unlocking operations (corresponding to a `synchronized` block), as well as the `wait` and `notify` calls, involve accesses to the shared object that represents the monitor. Therefore, it is possible to generalize the concurrency-related interaction among Java methods to read/write accesses to shared variables.

The obvious idea is then to assess the degree of interaction among Java methods via a metric that takes into account the number of accesses to shared variables in the byte code of the methods. We propose a new software metric, *shared variable accesses* (*SVA*), for a set of Java methods.

In general, the algorithm for construction of the behavior model of a reasonable environment takes as an input a list of methods, in which each method of the given component occurs at least twice — this is necessary to allow application of the SVA metric on two instances of the same method and thus to allow detection of concurrency errors in parallel execution of two or more instances of the same method. However, for simplicity we explain the algorithm only for a set S of methods. Generalization to a list is straightforward, e.g. via adding different suffixes to elements that correspond to the same method. The actual process of model construction consists of the following four steps:

1) *Identification of shared variable accesses*: Simple static analysis is used to acquire all necessary information about shared variable accesses in Java byte code of methods in S, i.e. to provide inputs for the SVA metric — details in Sect. 3.1.

2) *Metric value computation*: The value of the SVA metric is computed for each subset S_i of the set S (Sect. 3.2) — subsets of S that contain methods featuring no interaction (identified by the zero value of the metric) are dropped, since absence

of any interaction eliminates the possibility of a concurrency error.

3) *Ordering and pruning of subsets*: Remaining subsets of S are sorted according to their SVA metric's values in descending order, forming an ordered list L — the order of application of JPF to the subsets, with the purpose of checking parallel execution of methods in each subset, is determined this way. The list L is then pruned: an element L_i corresponding to S_i is removed from L if there exists an element L_j (corresponding to S_j) with higher metric's value such that $S_i \subseteq S_j$, since (i) checking of parallel execution of methods in L_j would find all the errors that would be detected by checking parallel execution of methods in L_i, and (ii) L_i would be checked after L_j because of the ordering.

4) *Generating environment's behavior model and code*: The pruned and ordered list L is expressed in the formalism of behavior protocols, yielding a model of environment's behavior, and the model is used as an input for the environment generator that generates the corresponding Java code.

The rationale behind selection of several subsets of S instead of using their union is that, in general, checking of parallel execution of all methods in the union may not be realistic due to state explosion, while checking of some of the subsets may be realistic. Ordering prioritizes those method subsets that have a high value of the metric and therefore more likely feature a concurrency error, with the goal of detection of the error in low time and memory. In particular, parallel execution of methods in the subset with the highest SVA metric value is checked first by JPF — in an ideal case, this subset contains the methods whose parallel execution leads to a concurrency error (assuming there is an error in the component's code).

In case of the simplified `OrderProcessor` component (Sect. 2), which provides methods in the set S_{OP} = {`placeOrder`, `confirmOrder`, `payOrder`}, the result of the steps 1-3 can be e.g. the list L_{OP} = ({`placeOrder`, `confirmOrder`, `payOrder`}, {`placeOrder`, `confirmOrder`}, {`confirmOrder`, `payOrder`}) that contains selected subsets of S_{OP} — the actual order of elements in L_{OP} depends on the metric. Nevertheless, the set {`placeOrder`, `payOrder`} is not included in L_{OP}, since these two methods do not interact via Java concurrency constructs.

3.1 Identification of Shared Variable Accesses via Static Analysis

The purpose of static analysis is to provide all the information about Java byte code of component's methods, which the SVA metric needs as an input. Specifically, the analysis provides:

- the set of component's methods, whose size is expected to be in the interval $[2, \infty)$, since parallel execution of less than two methods does not make sense,
- the list of shared variables, i.e. variables that are accessed by any two or more methods from the set (size of the list is in $[0, \infty)$), and
- the number of accesses to each shared variable in each method from the set — obviously, the number has to be in $[0, \infty)$.

Given a Java class that implements a component, the analysis is applied separately on each method. It involves (i) construction of the control-flow graph of the method in a similar way to [23], and (ii) a linear scan of Java byte code instructions in each control-flow path, which identifies all shared variable accesses in the paths. An access to a shared variable is represented by any instruction that either reads the content of a variable from the shared memory or writes to such a variable. Since the number of shared variable accesses may differ over control-flow paths, we define the number of shared variable accesses for a method as an average of the numbers for all control-flow paths. Typically, the numbers for different control-flow paths are roughly the same and thus the average is a good approximation.

Note, however, that the static analysis is in general imprecise, since it works only with static types and names of the shared variables. There are two reasons for that: (1) the ASM library [1], which is used by the analysis, provides no additional information on runtime values of the variables, and (2) we decided not to perform any complex data-flow analysis for the purpose of efficiency. A consequence of this imprecision is that the analysis may report "spurious interactions", i.e. it may report that two methods interact via accesses to a common shared variable when they actually access different variables with the same types. On the other hand, the analysis is conservative — it won't miss any interaction via concurrency constructs of Java, since use of types results in an over-approximation of the actual behavior.

3.2 Metric for Degree of Interaction between Concurrent Methods

When designing the SVA metric, we have identified the following three choices, which determine whether and to what extent the metric will prioritize (i.e. produce higher values for):

(C1) low or high (or any) number of methods in a set;

(C2) extreme sharing of a specific variable over uniform sharing of the variable, where *extreme sharing* means high number of accesses to the selected variable in one method and low number of accesses to the variable in other methods, and *uniform sharing* means relatively high number of accesses to the variable in all the methods;

(C3) extreme sharing of a single variable by all methods over uniform sharing of all variables by all the methods.

Nevertheless, we believe that realization of the choices cannot be done at the design time of the metric and without knowledge of the component subject to checking, since different options of the choices may be appropriate for different components — it should be possible to realize all three choices (C1, C2, C3) at run-time. Therefore, we have decided to implement the SVA metric via a parameterized function.

The inputs of the function are information about Java byte code of all methods in a given set (acquired by static analysis), and the output of the function is a single real number in the interval $[0, \infty)$ that expresses the degree of interaction among methods in the set.

In order to reflect the choices C1, C2 and C3, the function implementing the SVA metric was designed to have three independent parameters (α, β, and γ), which allow to configure the metric to a great extent. To be more specific,

- the value of the parameter α determines whether extreme sharing (for $\alpha > 0.5$) or uniform sharing (for $\alpha < 0.5$) of a variable is preferred and to what degree,
- the value of the parameter β determines whether extreme sharing of one variable (for $\beta > 0.5$) or uniform sharing of all variables (for $\beta < 0.5$) is preferred, and

- the value of the parameter γ determines whether low or high (or any) number of methods is preferred, and to what degree.

Regarding the parameters' domains, we decided to use the interval $[0,1]$ for all three parameters to allow easy tuning of metric's behavior for different components.

Important aspect of the choices C2 and C3 is that they are of a similar nature — specifically, both of them involve selection of a representative element from a given set according to the value of a specific parameter (α or β), where the elements of the set quantify a certain phenomenon — the number of accesses to a shared variable in our case — and the selected value characterizes the whole set. For example, the selected value can be median (maximum, minimum) of all elements of the set. In both cases (C2 and C3), we decided to use the *quantile function*. Technically, $P-$quantile over an ordered data set identifies a value that divides the data set in two parts: one part contains the low $100P$ % of the data set and the other part contains the rest. In particular, $0.5-$quantile corresponds to median, $0-$quantile to minimum and $1-$quantile to maximum. By changing the values of both α and β, it is possible to configure the SVA metric to prefer either extreme or uniform sharing.

As for the choice C1, we had to find an expression that allows increasing and decreasing of the metric's value according to the number of methods and value of the parameter γ. Specifically, we were looking for an injective and continuous function that maps the number of methods in $[2,\infty)$ and value of γ in $[0,1]$ to a real number in the interval $[0,\infty)$. From the many functions satisfying this requirements we picked $|M|^{\tan(2\gamma-1)}$, where M stands for the given set of methods. If $\tan(2\gamma-1) < 0$ then low numbers of methods are preferred, and vice versa.

Nevertheless, despite all the possibilities of the SVA metric's configuration, the following should always hold: if the value of the SVA metric for a method set M_1 is higher than the metric's value for a set M_2 (assuming the same configuration is used in both cases), then methods in the set M_1 involve higher degree of interaction than the methods in the set M_2.

Following all the reasoning above, the function implementing the SVA metric has the following form, where q_X stands for $X-$quantile, M stands for the given set of Java methods, V stands for the set of all variables shared by at least two methods in the set M, and $A_{i,j}$ denotes the number of accesses to the variable V_i in the method M_j:

$$q_\beta\left(q_\alpha\left(A_{i,j}, j = 1..|M|\right), i = 1..|V|\right) \cdot |M|^{\tan(2\gamma-1)} \tag{1}$$

The function is evaluated via a three-step process. At first, a representative number of accesses is selected for each variable V_i from the ordered list of numbers of accesses to V_i in all the methods in M via the $\alpha-$quantile of $A_{i,j}, j = 1\ldots|M|$. Then, a representative number of accesses to any shared variable from the set V in any method is selected via $\beta-$quantile of the ordered list of representative numbers of accesses to individual variables. Finally, the intermediate result computed so far is multiplied by the sub-expression $|M|^{\tan(2\gamma-1)}$ in order to take the size $|M|$ of the method set and the value of γ into account.

When the value of the SVA metric is computed for each subset of the given set of methods, the ordered list of method subsets is created — subsets are ordered according to the metric's value in descending way. Note that the values of the SVA metric's parameters (α, β, and γ) affect only the order of method subsets in the list, not the inclusion or exclusion of a specific subset. E.g. in case of the `OrderProcessor` component (Sect. 2), use of the values $\alpha = 1$, $\beta = 1$, and $\gamma = 0.107301$ ($\tan(2\gamma - 1) = -1$ in this case) results in the order ({`placeOrder`, `confirmOrder`}, {`confirmOrder`, `payOrder`}, {`placeOrder`, `confirmOrder`, `payOrder`}), while use of the values $\alpha = 0.5$, $\beta = 0.5$, and $\gamma = 0.5$ ($\tan(2\gamma - 1) = 0$ in this case) results in the order ({`placeOrder`, `confirmOrder`, `payOrder`}, {`placeOrder`, `confirmOrder`}, {`confirmOrder`, `payOrder`}) — the main reason behind the difference is that smaller sets of methods are preferred in the first case due to the value of γ.

4 Evaluation

In order to perform evaluation of the proposed technique, we have developed a prototype tool that integrates our implementation of the static analysis and SVA metric with JPF and the Environment Generator for JPF [12].

The prototype tool works in a fully automated way. As an input, it accepts

- Java code of the component subject to checking (e.g. a group of Java classes),
- list of public methods of the component that can be executed in parallel (*parallel methods*),
- a sequence of methods that have to be called during initialization of the component; this is empty in many cases (when no explicit initialization is necessary),
- specification of methods' parameter values in a Java class that works as a container for the values [14], and
- the values of the SVA metric's parameters.

For the given component, our tool creates the reasonable artificial environment (Java code) and then runs JPF on the complete Java program composed of the component and its environment. Since the technique aims at detection of concurrency errors, we have configured JPF to search only for deadlocks and race conditions.

Among the questions that should be answered by experimental evaluation of the SVA metric, we have identified the following three as the most important:

1) Does the proposed technique really make efficient detection of concurrency errors in Java components with JPF possible ?

2) Does the proposed technique give better results than the alternative approaches described in Sect. 1 — random selection of method pairs and search for suspicious patterns — and if so, in what respect ?

3) What combinations of values of the SVA metric's parameters should be used in order to minimize time and memory requirements of detection of concurrency errors in Java code with JPF ?

To answer these questions, we have performed a number of experiments on two real-life Java components. We have used the `ConcurrentHashMap` class (2000 lines of code (loc) in Java) from the implementation of the `java.util.concurrent` package in GNU Classpath (version 0.97.1) [8] and the Daisy file system [17] (800 loc), which was used as an assignment for the challenge problem at the CAV/ISSTA 2004 event

— the `DaisyDir` class is used as an entry point.

The Daisy file system contains concurrency errors prepared by author (S. Qadeer) in advance for the purpose of evaluation of various verification tools. In case of `ConcurrentHashMap`, we have manually injected an error (race condition) via removal of calls of `lock()` and `unlock()` in the `size` method.

4.1 Configuration of Experiments

We have performed experiments for the following kinds of artificial environments, all based on parallel execution of particular sets or pairs of methods:

- *random pairs* — pairs of methods selected randomly,
- *patterns* — pairs of methods selected via search for suspicious patterns [13],
- *ordered pairs* — pairs of methods selected and ordered via static analysis and the SVA metric, and
- *ordered lists* — lists of methods selected and ordered via static analysis and the SVA metric.

Selection and ordering of methods in both ordered pairs and ordered lists is performed via the algorithm described in Sect. 3 — the difference is that use of ordered lists means selection of lists of arbitrary size, while use of ordered pairs means selection of lists of size 2.

In case of ordered pairs and ordered lists, we have performed several experiments for different configurations of the SVA metric in order to find the effect of different values of the SVA metric's parameters on time and memory complexity of JPF checking. We have identified several combinations of values of the SVA metric's parameters that represent reasonable defaults. Name of each combination of parameter values consists of three letters that denote the selected values: the first letter corresponds to the parameter α, the second corresponds to β and the third to γ. We selected the following three values for the parameters α and β:

- the value of 0 corresponding to `U` (for preference of uniform sharing),
- the value of 1 corresponding to `E` (for preference of extreme sharing), and
- the value of 0.5 corresponding to `C` (standing for central, i.e. for something between extreme and uniform sharing).

For γ, we selected only two values:

- the value of 0.5 corresponding to letter `N` (for no dependency on the size of a method list; $\tan(2 \times 0.5 - 1) = 0$ in Expr. 1), and
- the value of 0.107301 corresponding to `S` (for preference of smaller lists of methods; $\tan(2 \times 0.107301 - 1) = -1$ in Expr. 1).

For illustration, the name `EES` means that the value of α is 1, value of β is 1 and the value of γ is 0.107301. Note that only the value of 0.5 (letter `N`) is used for the parameter γ in case of ordered pairs, since the number of methods is always equal to 2 — there is no need to express the dependence on the size of a list of methods.

Since the proposed technique aims at making detection of concurrency errors possible in reasonable memory and time, we have configured the experiments in such a way that the time and memory available for JPF are both limited — the limits are set to 1024 MB and 4 hours. We have set these limits on the basis of initial experiments.

All the experiments were performed on the following configuration (HW & SW): PC with 2xDualCore CPU (Intel) at 2.3GHz and 4GB RAM, Gentoo Linux with all the latest updates available in Oct 2007, Sun Java SDK build `1.6.0_02-b05` (64-bit, mixed mode). Regarding Java PathFinder, we used the current version as of Oct 2007 and turned partial order reduction on for each experiment.

4.2 Results of Experiments

Results of the experiments that we have performed are listed in Table 1.

The characteristics of each experiment that we measured are total time in seconds and amount of memory (in MB) needed by JPF. Total time includes the time needed to construct the artificial environment (and its behavior model, i.e. time needed to perform static analysis and compute the SVA metric) and time needed by JPF to find an error.

The table contains also the column '> Limit', which contains the ratio of the number of JPF runs that exceeded one of the limits to the number of all JPF runs (e.g., 9/24) or 0 if no JPF runs exceeded the limits. The columns 'Time' and 'Memory' contain average results for successful JPF runs — the value is set to `n/a`, if no JPF runs were successful (i.e. all runs exceeded one of the limits). If no error was detected in a specific experiment, e.g. due to very aggressive selection of method pairs or lists, then the symbol "-" is used, and time and memory are provided in parentheses.

4.3 Discussion of Results

The results of experiments (Table 1) show that the proposed technique — use of an artificial environment constructed on the basis of the SVA metric — may significantly reduce the time and memory needed for detection of concurrency errors in Java components with JPF with respect to the alternative approaches (random selection of method pairs and search for suspicious patterns), as can be seen in the case of the Daisy file system. Nevertheless, since our technique is based on heuristic, it is not realistic to expect that it will give better results in all cases — e.g. checking with the artificial environment based on random pairs needs slightly less time and memory than ordered pairs (and significantly less than ordered lists) for `ConcurrentHashMap`.

As for the SVA metric's parameters, the results of experiments show (i) that use of different parameter values may yield very different results, and (ii) that the best option with respect to efficient detection of concurrency errors would be to use ordered pairs or ordered lists with those combinations of values of the SVA metric parameters that prefer extreme sharing (i.e. `EES` and `EEN`). Again, since the SVA metric is based on a heuristic, it is not realistic to expect that the combinations `EES` and `EEN` will give the best results for all components.

	Daisy File System			ConcurrentHashMap		
	Time	**Memory**	**> Limit**	**Time**	**Memory**	**> Limit**
Ordered pairs						
EEN	1	49	0	94	313	0
CCN	1	51	0	95	306	0
UUN	9783	1023	0	89	303	0
Ordered lists						
EEN	1	52	0	411	369	0
EES	1	49	0	4332	580	0
CCN	1	47	0	4343	411	0
CCS	1	47	0	4505	622	0
UUN	n/a	n/a	2/2	414	368	0
UUS	9782	1023	0	64	229	0
Patterns	1	51	0	- (57)	- (165)	0
Random pairs	1715	310	9/24	65	169	0

Table 1
Results of experiments

5 Related Work

Although lot of research has been done in model checking of software components implemented in mainstream languages like Java and C (see, e.g., [7] and [3]) and construction of artificial environment for model checking of components [15][20][21], we are not aware of any other approach that attempts to make model checking — and, in particular, detection of specific errors in code — feasible and possible in limited time and memory via automated reduction of environment's complexity.

Well-known approaches to addressing state explosion, when it is caused by a high number of parallel threads, include partial order reduction [4] and heuristics for state space traversal [9] — however, these techniques can be applied only during model checking of complete programs, not to the construction of an artificial environment for an isolated component.

The approach to environment construction described in [21] uses Bandera Environment Generator (BEG) [20] for automated generation of a driver (Java code) from the behavior model provided by the user and automated extraction of stubs from existing implementation of real environment (if available) via static analysis. The key difference from our approach is that we support completely automated generation of reasonable behavior model of the artificial environment, while in case of [21] the reasonable behavior model has to be specified by hand at least for the driver – the hard part would be to manually identify potential concurrency errors and define model of such an environment that would (i) trigger all the errors and (ii) involve reasonably low number of parallel threads.

Similarly, we are not aware of any concurrency-related complexity metrics for modern object-oriented programming languages (C++, Java). Although there exist several approaches to measuring the complexity of concurrent programs in Ada ([11], [19]), none of them could be used instead of our SVA metric. In particular, they either measure the control-flow complexity of concurrent programs [11] or aim at Ada-specific constructs like active rendezvous [19].

6 Conclusion

We proposed a new technique for efficient detection of concurrency errors in Java code of software components using the Java PathFinder model checker (JPF). The key idea of the technique is automated generation of a reasonable artificial environment for isolated components on the basis of static analysis of Java byte code and the new software metric (SVA) for sets of Java methods. The static analysis identifies the sets of component's methods that interact via concurrency-related constructs of Java (e.g. accesses to shared variables) and the SVA metric orders the sets according to the degree of interaction among methods. The artificial environment is then generated in such a way that parallel execution of methods in the set with the highest SVA metric's value is checked first by JPF in order to find concurrency errors in less time and memory.

Our experiments with several real-life Java components show that the proposed technique may give significantly better results than the alternative approaches (random pairs of methods, search for suspicious patterns) for some components — in particular, it makes detection of concurrency errors with JPF possible in limited time and memory, if JPF is applied to the complete program composed of the component and the reasonable environment constructed on the basis of the SVA metric. It is, however, not realistic to expect that the technique will give better results for all components, since it is based on a heuristic. Note also that the technique does not perform exhaustive verification and therefore can only be used to show presence of bugs in the code, not their absence — in particular, it will not discover concurrency errors that are caused by a specific complex sequence of method calls.

Although the technique proposed in this paper aims at components implemented in Java, it can be ported to another programming language (e.g., C or C#), provided a model checker that supports verification of multi-threaded programs written in the language is available.

In future, we plan (i) to employ more precise static analysis of Java code (e.g. points-to and shape analyses) in order to provide more accurate assessment of interaction among methods, and (ii) to design several versions of the SVA metric using different functions and compare all the versions on several components. We would also like to perform more extensive evaluation of the proposed approach on a greater number of components, provided we will have complex components that contain some concurrency errors — usage of fault seeding techniques (e.g. [2]) is one of the options to get such components.

Acknowledgments. This work was partially supported by the Czech Academy of Sciences (project 1ET400300504).

References

[1] ASM: Java byte code manipulation framework, http://asm.objectweb.org

[2] Bradbury, J. S., J. R. Cordy, and J. Dingel, *Mutation Operators for Concurrent Java (J2SE 5.0)*, In Proceedings of the 2nd Workshop on Mutation Analysis (Mutation 2006), 2006

[3] Chaki, S., E. Clarke, A. Groce, S. Jha, and H. Veith, *Modular Verification of Software Components in C*, IEEE Transactions on Software Engineering, **30**(6), June 2004

[4] Clarke, E., O. Grumberg, and D. Peled, "Model Checking", MIT Press, Jan 2000

[5] Cobleigh, J. M., D. Giannakopoulou, and C. S. Pasareanu, *Learning Assumptions for Compositional Verification*, In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2619, 2003

[6] Giannakopoulou, D., C. S. Pasareanu, and H. Barringer, *Assumption Generation for Software Component Verification*, In Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE), IEEE CS, 2002

[7] Giannakopoulou, D., C. S. Pasareanu, and J. M. Cobleigh, *Assume-guarantee Verification of Source Code with Design-Level Assumptions*, In Proceedings of 26th ICSE, IEEE CS, 2004

[8] GNU Classpath, http://www.gnu.org/software/classpath/

[9] Groce, A., and W. Visser, *Heuristics for Model Checking Java Programs*, Proceedings of the 9th International SPIN Workshop on Model Checking of Software, LNCS, vol. 2318, 2002

[10] Hughes, G., S. P. Rajan, T. Sidle, and K. Swenson, *Error Detection in Concurrent Java Programs*, In Proceedings of SoftMC 2005, ENTCS, **144**(3)

[11] Mathews, M. E., and S. Tu, *Metrics Measuring Control Flow Complexity in Concurrent Programs*, Proceedings of the 13th PNSQC, 1995

[12] Parizek, P., Environment Generator for Java PathFinder, http://dsrg.mff.cuni.cz/projects/envgen

[13] Parizek, P., and F. Plasil, *Partial Verification of Software Components: Heuristics for Environment Construction*, In Proceedings of the 33rd EUROMICRO SEAA conference, IEEE CS, 2007

[14] Parizek, P., and F. Plasil, *Specification and Generation of Environment for Model Checking of Software Components*, In Proceedings of FESCA 2006, ENTCS, 176(2), 2007

[15] Pasareanu, C. S., M. Dwyer, and M. Huth, *Assume-Guarantee Model Checking of Software: A Comparative Case Study*, In Proceedings of the 6th SPIN Workshop on Model Checking of Software, LNCS, vol. 1680, 1999

[16] Plasil, F., and S. Visnovsky, *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering, **28**(11), 2002.

[17] Qadeer, S., Daisy File System, Joint CAV/ISSTA special event on specification, verification and testing of concurrent software, 2004, http://research.microsoft.com/~qadeer/cav-issta.htm

[18] Robby, M. B. Dwyer, and J. Hatcliff, *Bogor: An extensible and highly-modular model checking framework*, In Proceedings of the 9th European Software Engineering Conference, ACM, 2003

[19] Shatz, S. M, *Towards Complexity Metrics for Ada Tasking*, IEEE Trans. Sw. Eng., **14**(8), 1988

[20] Tkachuk, O., M. B. Dwyer, and C. S. Pasareanu, *Automated Environment Generation for Software Model Checking*, In Proceedings of ASE'03, IEEE CS, 2003

[21] Tkachuk, O., and S. P. Rajan, *Application of Automated Environment Generation to Commercial Software*, In Proceedings of ISSTA'06, ACM Press, 2006

[22] Visser, W., K. Havelund, G. Brat, S. Park, and F. Lerda, *Model Checking Programs*, Automated Software Engineering Journal, **10**(2), 2003

[23] Zhao, J., *Analyzing Control Flow in Java Bytecode*, Tech. report, Fukuoka Inst. of Technology, 2000