

Searching for Safety Violations using Estimation of Distribution Algorithms

Jan Staunton

Department of Computer Science
University of York
York, United Kingdom YO10 5DD
Email: jps@cs.york.ac.uk

John A. Clark

Department of Computer Science
University of York
York, United Kingdom YO10 5DD
Email: jac@cs.york.ac.uk

Abstract—Using aspects of model checking to analyse multi-threaded software is a promising method for finding common concurrent errors such as deadlock. Traditional model checking tools exhaustively search the state space of a concurrent system in order to find faults. Unfortunately, model checking suffers from the state space explosion problem, limiting the applicability of the approach to commercial software. Metaheuristic search mechanisms have been used in an attempt to overcome this issue with good results. Techniques such as Genetic Algorithms (GAs) and Estimation of Distribution Algorithms (EDAs) focus the search of the state space on areas that are more likely to contain errors. In this work, a novel EDA-based approach to exploring the state space of a model is outlined. Experiments are performed on an implementation using the Java PathFinder (JPF) model checker and the ECJ toolkit. The EDA-based approach is shown to perform well against standard search procedures such as depth-first search, whilst also outperforming random search on a benchmark problem. On larger problems, the EDA is shown to be the only effective technique of those compared.

I. INTRODUCTION

In recent years, prominent CPU manufacturers have abandoned the race for higher clock speeds, and are instead focusing on increasing the number of processing cores per chip. Consequently, software developers are forced to use concurrent programming paradigms in order to exploit this trend. Developers building single-threaded applications have come to rely upon a sophisticated set of debugging tools. When developing multi-threaded or concurrent applications, however, tools for detecting common problems such as deadlock and data races in practical software are the subject of research [Eytani et al., 2008].

One promising avenue of research is to use aspects of *model checking* techniques to analyse concurrent programs. Model checking analyses the state space of a model to verify a set of formalised properties. The model can be directly extracted from programs written in standard industry languages such as Java. The state space of realistic models/programs, however, can become very large and this is known as the state space explosion problem. There are a number of techniques for reducing the state space of a model, but even after applying these techniques, many state spaces are intractable.

This work describes a novel way of searching the state space of a model using a form of N-gram GP [Poli and McPhee, 2008] and Java PathFinder (JPF) [Visser et al., 2003].

JPF builds models from Java bytecode allowing analysis of ordinary Java programs. N-gram GP is used to focus the search of a state space on areas where concurrent errors are more likely to occur, thus making some large models tractable. The focus of this work is on finding deadlock in multithreaded Java programs.

This paper is structured as follows: Section II gives an overview of model checking. Section III gives a short overview of Search Based Software Engineering (SBSE). Section IV describes the N-gram GP based approach to searching the state space of a model. Section V describes empirical evaluation of the approach against a benchmark problem. Section VI provides a summary and potential avenues of future work.

II. MODEL CHECKING

Model Checking is a technique for analysing reactive concurrent systems [Clarke et al., 2000]. A model checking tool can automatically verify that a given specification is satisfied by a model of a system. A model of a system can be expressed in a number of ways, or be automatically extracted from common industry languages such as Java. From the model of a system, a state space can be generated. The state space of a multi-threaded Java program, for instance, can be constructed using the product of the state spaces of each thread. Not all states will be reachable from the initial state, however, given locking and other phenomena. The state of a thread consists of the current program counter, the values of any variables local to the thread and any shared memory. The *actions* of a single thread alter the state of that thread, and consequently the state of the entire system. The actions of a thread are said to cause *transitions* between global states in the state space. The state space generated from a model of a system, or indeed source code, is referred to as a *transition system* [Baier et al., 2008]. The state space of a transition system is typically visualised as a digraph, where the nodes are unique global states and the edges are labelled with the actions that caused the transitions between them. Figure 1 shows an example digraph.

The specification is given as a set of formalised properties. The types of properties that can be verified are divided into two categories: *safety* properties and *liveness* properties [Baier et al., 2008]. Safety properties describe requirements of the form “something bad does not happen”, an example of which

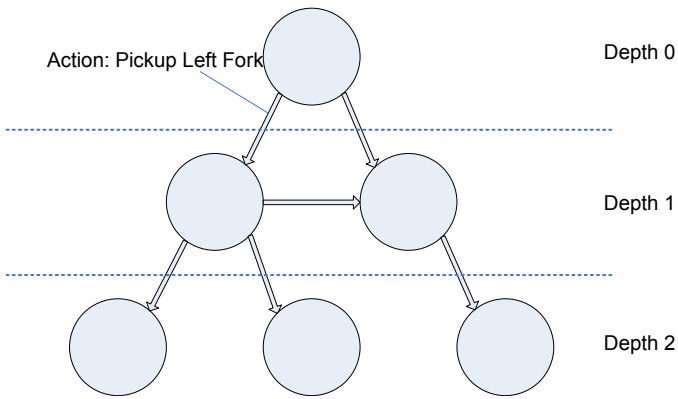


Fig. 1. Illustration of a state space digraph. Actions cause transitions between states. States are nodes, arcs are actions. The action labelled is an example from the dining philosophers problem.

is the “the system does not deadlock”. Liveness properties describe requirements of the form “something good eventually does happen”, an example of which is “the server always responds after being signalled”.

In order to verify that a specification is satisfied, a model checker systematically examines all of the states of the model [Clarke et al., 2000]. Each state is checked for compliance with the properties of the specification. If no violations are found after an exhaustive check, then the model is proven to satisfy the specification. When a violation of a property in the specification is detected, a *counterexample* is returned to the user consisting of the error and the execution path that led to that error [Clarke et al., 2000]. When checking for safety property violations, the model checker must search the state space for a state or execution path fragment that violates that property. When checking a liveness property, the model checker must find an infinite path containing the violation (an infinite path being a path with a cycle). Counterexamples with shorter execution paths are preferred to ease the debugging process.

Model checking tools typically employ an exhaustive search technique to enumerate the state space. The most common mechanisms are depth-first search and breadth-first search. Resource constraints, however, may place a limit on the size of model that can be verified using these techniques. The size of a state space for some trivial examples can be huge, and the size typically increases exponentially with the number of concurrent components in this system. Size is measured in the number of states. This issue is known as the *state space explosion* problem [Clarke et al., 2000] and is one of the major obstacles in model checking practical commercial software.

Efforts have been made to reduce the state space of a model. Techniques such as partial order reduction [Peled, 1998, Valmari, 1991] and symbolic model checking [Anand et al., 2007, McMillan, 1993] can reduce state space sizes significantly. In some situations, it may suffice to show the presence of a violation rather than exhaustively check a model. In this situation, one can use a search algorithm that aims to search parts of the state space that are more likely to contain an

error. Such algorithms rely on heuristics to guide the search process to promising parts of the state space. An example of such an algorithm is best-first search, which examines states in an order determined by a heuristic. In an effort to produce shorter counterexamples, heuristic algorithms such as A* search can be used to optimise the length of the execution path. Clarke et al. [2000] and Baier et al. [2008] are excellent texts regarding the topic of model checking, describing all of the issues mentioned above and more.

Recently, there has been some work on applying metaheuristic search techniques to the model checking problem. Early work reported by Alba and Troya [1996] and Godefroid and Khurshid [2004] study the use of Genetic Algorithms (GAs) to find errors in models. Recent work [Alba and Chicano, 2007, Chicano and Alba, 2008b] details the use of Ant Colony Optimisation for finding safety and liveness errors in large models, along with work exploiting partial order reduction [Chicano and Alba, 2008a].

III. SEARCH-BASED SOFTWARE ENGINEERING

Search-Based Software Engineering (SBSE) is the practice of applying metaheuristic search techniques to software engineering tasks. Examples of metaheuristic search techniques include simulated annealing [Kirkpatrick et al., 1983], genetic algorithms [Goldberg, 1989], ant colony optimisation [Dorigo and Stützle, 2004] and probabilistic model-building genetic algorithms [Pelikan et al., 2002]. SBSE has been applied to a variety of software engineering tasks including project management, automatic programming and software testing.

SBSE requires that the target software engineering task be reformulated as a search problem [Clark et al., 2003]. In order to achieve this reformulation, a solution space encoding must be described along with an objective function that ranks potential solutions. Solutions can be encoded in a number of ways and the encoding is often problem specific. For instance, when generating test data, a solution may consist of a vector of inputs to a procedure. The objective function takes a solution and returns a measure of how good the solution is. When generating test data for instance, an objective function could take a program and a solution (a vector of inputs) and return some measure of how well those inputs tested a program. An example of such a measure could be how many branches the input vector covered. Clark et al. [2003] give an excellent overview of SBSE and the reformulation of software engineering tasks to search problems.

A. Model Checking as an SBSE problem

The problem of model checking can be cast as an SBSE problem. Typically, the objective of any model checking “run” is to find a counterexample to a given property. A counterexample is a path through the state-space of a system/program in which the given property does not hold. A path is a sequence of actions/states. The following formulation can be used to transform the model checking problem into an SBSE problem. The solution space is a particular representation of the paths

through a state-space. The objective function must measure how “close” a particular path is to violating a property.

IV. EDA-BASED MODEL CHECKING

A. Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) are population-based probabilistic search techniques that search solution spaces by learning and sampling probabilistic models [Pelikan et al., 2002]. EDAs iterate over successive populations or bags of candidate solutions to a problem. Each population is sometimes referred to as a generation. To construct a successor population, EDAs build a probabilistic model of promising solutions from the current population and then sample that model to generate new individuals. The newly generated individuals replace individuals in the current population to create the new population according to some replacement policy. An example replacement policy is to replace half of the old population with new individuals. The initial population can be generated randomly, or seeded with previously best known solutions. The algorithm terminates when a termination criterion is met, typically when a certain number of generations are reached or a good enough solution has been found.

Algorithm 1 Pseudocode for basic EDA

```
P = InitialPopulation();
evaluate(P);
while not(termination_criterion) do
  S = SelectPromisingSolutions(P);
  M = UpdateModelUsingSolutions(S);
  N = SampleFromModel(M);
  P = ReplaceIndividuals(N);
  evaluate(P);
end while
```

The pseudocode of a basic EDA algorithm is shown in Algorithm 1. Readers who are familiar with Genetic Algorithm (GA) literature can view EDAs as similar to a GA with the crossover and mutation operators replaced with the model building and sampling steps. EDAs can be seen as strategically sampling the solution space in an attempt to find a “good” solution whilst learning a model of “good” solutions along the way. EDAs are sometimes referred to as Probabilistic Model-Building Genetic Algorithms (PMBGAs), a full overview of which can be found in Pelikan et al. [2002].

Unfortunately for researchers in this field, there is an awkward collision in terminology regarding EDAs and model checking centered around the word model. From this point onwards, references to “EDA model”, “probabilistic model” or just “model” relate to the model stored and manipulated by an EDA algorithm. The state space of a concurrent system and actions/transition within that state space are referred to as a transition system.

B. Model and Solution Space

A popular solution space for EDAs are bit strings that are interpreted as solutions via some encoding [Pelikan et al., 2002]. A popular and simple example problem is the *MaxOnes()* problem. The solution space for the *MaxOnes()* problem is the set of bit strings of length n . The objective function, *MaxOnes(c)*, returns the number of bits set to 1 in bit string c . The fittest solution for the *MaxOnes()* problem is a bit string with all bits set to 1. The aim is to maximise the objective function. The *MaxOnes()* problem is used as an illustrative example throughout this section.

The simplest model for the bit string solution space is a univariate model over each bit, as implemented in the UMDA algorithm [Mühlenbein and Paaß, 1996]. This is sometimes referred to as a probability vector. For each bit $i < n$, the model stores a probability of bit i being a 1. In the context of an EDA, the vector or model constitutes the current hypothesis on the values of variables in the fitter solutions in the solution space.

The solution space of paths in a transition system requires a novel modelling and encoding mechanism. There are a number of choices available when encoding and modelling paths in a transition system. When making this choice, a number of factors must be considered. One of the major factors when choosing a modelling and solution representation is the ability to represent paths of arbitrary length. In some situations, knowing the length of a counterexample may be unfeasible. The majority of EDA algorithms can effectively deal only with representations of a fixed or known size.

In order to encode paths in a transition system, a simple string representation is used. A path in a transition system can be viewed as a sequence of actions causing transitions between states. The alphabet of the string representation used in this work is the set of actions that can be executed in the transition system. The choice of action set is problem dependent and can be at any level of abstraction. In this work, we retrieve the alphabet from Java PathFinder APIs [Visser et al., 2003], where each alphabet member represents a choice of action in the transition system. Each alphabet member consists of Java file names, line numbers and byte code instructions.

Examples of the typical alphabet members used in this work are shown in Figure 2. All alphabet members are of the same format, apart from a few special instructions. In the alphabet member on line 6 of Figure 2, `<examples/DiningPhil.java:38>` refers to the filename and line number currently being executed. `< synchronized(left) {>` is the contents of line 38 in `DiningPhil.java` and `<monitorenter>` is the instruction being executed. Note that the alphabet is thread independent in an effort to scale well with symmetrical problems, like dining philosopher systems. A typical path in a system is shown in Figure 2. In the path, there are a few special instructions at the start of the path that do not follow the format described. These instructions are special Java instructions relating to the creation of the initial objects and

```

1 <null transition>
2 <[synthetic] [clinit]<clinit>><[synthetic] [clinit]<clinit>><invokeclinit>
3 <[java/lang/ThreadGroup.java:859]><[java/lang/ThreadGroup.java:859]><monitorexit>
4 <[examples/DiningPhil.java:38]>< synchronized (left) {<runstart>
5 <[examples/DiningPhil.java:38]>< synchronized (left) {<getfield>
6 <[examples/DiningPhil.java:38]>< synchronized (left) {<monitorenter>
7 <[examples/DiningPhil.java:33]>< start();><invokevirtual>
8 <[java/lang/ThreadGroup.java:844]><[java/lang/ThreadGroup.java:844]><monitorenter>
9 <[examples/DiningPhil.java:39]>< synchronized (right) {<getfield>
10 <[examples/DiningPhil.java:39]>< synchronized (right) {<monitorenter>
11 <[examples/DiningPhil.java:41]>< }><monitorexit>
12 <[java/lang/ThreadGroup.java:859]><[java/lang/ThreadGroup.java:859]><monitorexit>
13 <[examples/DiningPhil.java:38]>< synchronized (left) {<runstart>
14 <[examples/DiningPhil.java:42]>< }><monitorexit>
15 <[examples/DiningPhil.java:33]>< start();><invokevirtual>
16 <[examples/DiningPhil.java:38]>< synchronized (left) {<getfield>
17 <[examples/DiningPhil.java:41]>< }><monitorexit>
18 <[examples/DiningPhil.java:42]>< }><monitorexit>

```

Fig. 2. A typical trace/string/path from JPF on the Dining Philosopher problem with 2 philosophers. This trace does not include a deadlocked state. Note the lack of references to specific threads.

are at the beginning of every path in JPF.

This technique is not limited to analysing Java bytecode exclusively. When searching over other forms of models, such as Promela models or C code, a suitable alphabet must be chosen. The rationale for the alphabet used in this work is as follows. The alphabet must be fine grained enough to include some reference to the bytecode instruction being executed. Simply referring to the line number in a Java source code file may ignore a lot of crucial detail, as many bytecode instructions can be executed when executing a statement of Java. However, referring to bytecodes alone is not enough as it is highly likely a particular bytecode instruction is executed as a part of many statements in a given Java program. Good results were obtained using a combination of the statement being executed as well as the bytecode instruction during small-scale empirical evaluation.

To model strings in the solution space, a variant of N-gram GP is used [Poli and McPhee, 2008]. An n-gram is a subsequence of length n from a longer sequence. N-gram GP learns the joint probabilities of fit string subsequences of length n . In this work, the n-grams represent recent histories of n actions, and the distributions associated with these N-grams are sampled to determine the best action to choose next. N-gram GP has the advantage of not being limited to a fixed size representation. One can sample from an N-gram GP model until some stopping criterion is met. This will be discussed in the model sampling section.

C. Learning the Model

In the UMDA univariate bit string model approach discussed earlier, a simple frequency count is performed on the selected individuals which is then normalised to obtain the probability vector distribution. The model learning step used in this work is a simple frequency count of actions after each unique n-gram. A set of fitter paths or strings S is selected from the current population. Then, a count of frequencies is performed on all the unique n-grams over all the strings in S . The frequency count is then normalised to obtain a probability distribution. An illustration of this process can be found in

Figure 3, where the A s and B s maps onto alphabet members like those shown in Figure 2. In addition to learning all the n-gram distributions, the distributions for (n-1)-grams and (n-2)-grams and so on are also learned down to 1-grams.

Current N-gram	Observed next choice	Frequencies
Step q : B A B A B B		B A: B = 1
Step $q + 1$: B A B A B B		A B: A = 1 B A: B = 1
Step $q + 2$: B A B A B B		A B: A = 1 B A: B = 2
Step $q + 3$: B A B A B B		A B: A = 1, B = 1 B A: B = 2

Fig. 3. Illustration of the N-gram learning process (2-grams in this case). A frequency count is performed for each unique N-gram in the selected set of strings. The boxes represent a basic sliding window algorithm, with frequency counts display on the right.

In order to select the individuals from which the model is learned, a selection mechanism is used. The selection mechanism used by an EDA is typically biased toward the fitter individuals in a population, according to the fitness function. The fitness function is discussed later in this work. A common selection mechanism used in recent EDA literature is truncation selection. Truncation selection simply selects the fittest n individuals from the population. In this work, truncation selection was used as the selection mechanism.

When learning the model, there is a choice between building an entirely new model from the selected individuals or updating the existing model using the selected individuals. In this work, the entirely new model approach was taken as it proved more effective when evaluated empirically. After each generation, the model is destroyed and a new one constructed using the procedure described above.

There is a small issue with regard to the substrings at the start of each string/path. At the beginning of each path, there are substrings that are less than the required n-gram length n . This is dealt with in the same way as the work in Poli

and McPhee [2008]. A distribution is learned for all of the n -grams up to length n at the start of the strings only. From then onwards, the method outlined above is used.

The model learned using this method can be described as a strategy for traversing the transition system. Given a recent history of actions of length n , the n -gram model can be queried to obtain a distribution that can be used to choose the best next action. Due to the thread independence of the alphabet, this model has the potential of describing concise solutions for highly symmetrical problems. The model is also agnostic to the type of property being verified.

D. Model Sampling

When sampling to create the next generation, UMDA sets a bit i in a particular individual to 1 according to the respective probability in the probability vector model. The sampling mechanism used in this work is slightly more sophisticated. To generate an individual/path in the transition system, the method initially starts with an empty string p representing the history of actions and the initial state i of the transition system. Then, an algorithm called *MakeAChoice()* is executed, which takes a state s (initially i) and the history of actions p as arguments and returns an action/transition a that is possible from s . The transition a is then chosen in the model checker, yielding a new state i' . The action a is appended to p to accurately reflect the history of actions on the current path. *MakeAChoice()* is then called with i' and p as arguments to yield the next state. This process is repeated until either a state that has previously been encountered on this path, or an end state is reached.

The *MakeAChoice()* algorithm can be broken down into a number of stages. The first stage is *distribution acquisition*, in which the model is queried for a relevant distribution given a recent history of actions. From the input string p , the n most recent actions are obtained from the end of p . This n -gram r is then used to query the model to obtain a distribution. If no distribution is known for r , r' is created using the most recent $n - 1$ actions in p and the model is queried again. This shortening of the n -gram is repeated until a distribution is found, or until the n -gram length is 0. If no distribution can be found for the n -gram, or any shorter n -gram, then a “blank” distribution is obtained. If a blank distribution is returned from the model, this indicates that the n -gram was not observed during the model learning phase of the EDA.

Once a distribution is obtained, the *distribution sampling* stage is executed. *MakeAChoice()* takes a state s as an argument. From s , a set of transitions T are available. The goal of the distribution sampling stage is to choose one of the transitions available from s . Each transition in T represents the progress of a component within the transition system, in this case a thread of a multithreaded program. Each transition in T has an associated action that caused that transition. Therefore, there is a set of actions A possible from state s . A number of transitions may be caused by concurrent components taking the same action. For instance, a group of threads may be competing to obtain a lock on an object. Thus

in some instances, the cardinality of the set of possible actions A may be less than T .

In this work, the distributions in the model are distributions over the possible actions from a state rather than the transitions. The rationale behind this is to aid in the scalability of the approach on symmetrical problems, such as the dining philosophers coordination problem, whilst gracefully degrading for non-symmetrical problems. If n threads (where $n > 1$) are poised to take an action, and that action is chosen, then one of the n threads is chosen at random.

When the set of actions A are the same actions as those represented by the obtained distribution, then a choice is made according to that distribution. However, when A is not perfectly described by the obtained distribution, a few special cases which must be handled during the *MakeAChoice()* procedure. The first is when there are actions in the distribution that are not in A . When this occurs, the excess actions in the distribution are culled and the distribution is normalised. Then an action is selected according to the distribution.

The second case is when one or more actions in A are not in the distribution. This raises an interesting question. The actions in the distribution have been observed in the fitter paths from the previous generation. By this logic, actions not in the distribution were not observed in the fitter paths from the previous generation, and may even have been in the paths that are discarded. By this reasoning, it would be prudent to give these actions a low probability of being selected, instead favouring actions that produce fitter paths. However, since the algorithm is still searching for a deadlock path, the deadlock paths are precisely those that have not yet been observed. By this reasoning, it would be desirable to favour these actions in the hope of discovering fitter paths.

In this work, the first option is used, giving low probability of selection to those actions that are not in the distribution. The actions in A that are in the distribution are given their respective probabilities, whilst those that are not in the distribution are given half the lowest probability. Then the distribution is normalised and sampled to yield a choice of action.

A third case also exists. If no distribution can be found for a recent history, a blank distribution is returned and a uniformly random choice is made between the actions in A . In addition, to generate the initial population, the model is seeded with a blank distribution for every n -gram. Therefore, when generating the initial population, all individuals are generated at random.

E. Fitness Function

An EDA samples a solution space in a meaningful way in order to find good solutions, using a fitness function as a guide. The truncation selection mechanism described earlier uses the fitness function to sort the population of solutions. Once the population has been sorted, the selection mechanism then chooses the top n solutions. The selected solutions are then used to learn the n -gram model.

Metaheuristic search algorithms can be viewed as function optimisers, aiming to minimise or maximise a particular ob-

jective function. The solution space in this work is the set of all possible paths in the transition system of the code under test. Good solutions in this solution space are those that end with a state of deadlock and are short. Solutions in the solution space that are “closer” to leading to a deadlocked state must rank higher than those that are “far away”.

Determining how close a path is to a path ending in a deadlocked state is an open and challenging question. For instance, the sequence of actions and states leading up to a deadlocked state in a commercial software system may be very different from a deadlock in a poor dining philosophers coordination policy. Defining a general deadlock finding fitness function is likely impossible, with each problem requiring a custom fitness function with domain specific knowledge to be effective. A possible avenue of research is to automatically derive helpful metrics for this purpose using analysis of the system under test.

Algorithm 2 Fitness function used to rank individuals. Individuals that are “closer” to deadlock are favoured.

Require: A, B are Individuals;

```

if  $A.error\_found \neq B.error\_found$  then
  return IndividualWithErrorFound(A,B);
else if  $A.error\_found$  and  $B.error\_found$  then
  return IndividualWithShortestPath(A,B);
else
  return IndividualWithHighestBlockMetric(A,B);
end if

```

Algorithm 3 Blocked threads metric algorithm.

Require: I is an Individual;

```

aggregateBlocked = 0;
for all States  $s \in I.Path$  do
  aggregateBlocked += s.NumberOfBlockedThreads;
end for

```

In this work, a fitness function (Algorithm 2) was constructed manually using a basic rationale and information from small-scale empirical evaluation. The function is a path comparison function, comparing two paths and returning the more desirable path. The rationale for the fitness function is that increased blocking of threads may lead to a deadlocked state. The fitness function is structured simply, and the functions called in the algorithm perform the actions indicated by their respective names. *IndividualWithShortestPath(A, B)*, for instance, returns the individual with the shortest path. The fitness function will always favour a path that contains an error state to one that does not. If both paths contain an error state, then the shortest path is favoured. If neither path contains an error state, then the path that has the highest blocked threads metric is favoured. The blocked threads metric algorithm is described in Algorithm 3. The algorithm simply sums the number of blocked threads for each state on the path.

The fitness function described in Algorithm 2 depends on deadlock being defined as all threads in the program being

blocked, rather than a subset. Although the fitness function described above is only potentially useful for finding full system deadlock, defining fitness functions for finding other kinds of errors is also possible. Several heuristics for finding liveness errors, for instance, have been defined by the directed model checking community [Edelkamp et al., 2001] and have been used in conjunction with metaheuristic search techniques [Chicano and Alba, 2008b]. The EDA based technique proposed in this work could easily be applied to finding liveness errors using similarly defined fitness functions.

F. Other Parameters and Features

A common feature of EDA implementations is a mutation operator, similar to that used in genetic algorithms. The purpose of such an operator is to introduce new genetic material into a population. In the UMDA algorithm, this can be achieved by flipping each bit in each individual in the population with some probability. In this work, mutation occurs when choices are made. When making a choice during the path generation phase of the algorithm, with probability m , a uniformly random choice is made from the available actions. If m is set to 1.0, all choices are made at random yielding a random path search mechanism. The probability of mutation in the majority of implementations of EDAs or GAs is typically set to a low value as a high value can be disruptive to the search process.

A common feature in Evolutionary Algorithms is the notion of *elitism*, and is sometimes implemented in EDA work. Elitism is the practice of copying the best n individuals from one generation to the next without mutation or change, whilst the rest of the population are generated in the usual way. In this work, elitism was implemented and seemed to give better results than an EDA without elitism in small-scale experimentation.

V. EXPERIMENTATION

In order to experiment with this mechanism, we use an implementation based upon Java PathFinder (JPF) [Visser et al., 2003] and the ECJ evolutionary toolkit [Luke et al., 2007]. Both tools are popular in their respective fields. The implementation consists of a bridge between the two frameworks. Interfacing with JPF is a relatively trivial matter with many examples of previously implemented search mechanisms available. However, ECJ required a great deal more effort as ECJ focuses on solution interaction rather than examining the population as a whole.

In order to benchmark this mechanism, the Dining Philosophers problem is used. The dining philosophers problem is a coordination problem described by Dijkstra [1968]. A group of n philosophers are sat around a circular table to share some food at the center. Between each pair of adjacent philosophers is a fork, making n forks in total. Each philosopher alternates according to some schedule between a state of *thinking* and *eating*. In order to eat from the food on the table, a philosopher must acquire the fork to the immediate left and right. There are no prerequisites for thinking.

The task is to coordinate the use of forks such that deadlock does not occur, whilst maximising the thinking time of each philosopher. This implies the minimisation of waiting for forks so that a philosopher can eat. If the four conditions for deadlock hold [Coffman and Elphick, 1971], then it is possible for deadlock to occur if each philosopher picks up one fork. This problem is a classic coordination problem and is a typical benchmark in model checking literature [Alba and Chicano, 2007, Alba et al., 2008, Baier et al., 2008, Godefroid and Khurshid, 2004].

In this work, a naive coordination strategy is used for benchmark. The benchmark, known in the JPF community as *DiningPhil*, implements the following policy. All philosophers first pick up the fork to the left, then the fork to right, eat, and then release the right and left fork in that order. The philosophers then terminate execution. In this coordination policy, there is ample room for deadlock, as it is possible to reach a state in which all philosophers have picked up the left fork and are blocked on acquiring the right. The implementation used in these experiments is the Java class included with JPF. The size of the state space of this particular Dining Philosopher implementation grows exponentially with the number of philosophers. Alba et al. [2008] report state space sizes of 2094 and 120544 for 3 and 4 philosophers. In this work, experiments were carried out on varying sizes of the DiningPhil problem, from 4 to 40 philosophers.

The parameters chosen for this set of experiments are the result of small-scale empirical evaluation and is kept constant for all experiments. An n-gram length of 3 was used, meaning models for 3-grams, 2-grams and 1-grams are constructed from each generation. The population size for each generation was set to 150. This means that 150 paths are sampled from the model to build each generation. The mutation parameter for these experiments is set to 0.001, meaning that on average 1 in 1000 transition choices are made randomly, disregarding the model. The elitism parameter was set to 1, meaning that the top individual from the population is copied to the next generation. In order to build the model from which the next generation is sampled, truncation selection selects the top 20% of individuals from the population. This means that the top 30 individuals from the current population are used to build the EDA model. The algorithm terminates when a solution is found. Initially, the model is a blank model meaning that all the paths evaluated during the first generation are completely random.

Comparisons of the EDA with other techniques are also shown. The other techniques are depth-first search, breadth-first search and random search. Depth-first search examines the “deepest” states before any others, whereas breadth-first search favours the “shallowest” states. The random search algorithm behaves precisely like the proposed technique in this paper, however all choices are made at random. The algorithm, starting from the initial state, chooses transitions at random until a previously found state is encountered, or an end state is reached. The algorithm repeats this process until a deadlock state is found. The random search process

is equivalent to the EDA with the mutation parameter set to 1.0. Since the EDA is a pseudo-random probabilistic process, the results reported are statistics of 50 independent executions. Each run was performed using a unique seed for the pseudo-random number generator. The results for the random search algorithm are also statistics of 50 independent runs, each with a unique seed for the pseudo-random number generator.

The results from the experiments are shown in Table I. Each of the algorithms were run on the Dining Philosopher problem using gradually increasing numbers of philosophers. Since DFS and BFS are deterministic algorithms, the result of only one execution is shown. Statistically significant differences from the EDA, with a confidence level of 0.05, are shown using (+) to indicate a significant result, and (−) is indicate otherwise. When comparing the 50 independent executions of the EDA against the single result from a deterministic algorithm, the Wilcoxon signed-rank test is used. When comparing the EDA against the random search approach, both using a sample of 50 independent executions, a Mann-Whitney rank sum test is used. Both tests are valid under general conditions, for instance the assumption of normality is not required. These tests are the non-parametric equivalents to the one and two-sided t-tests.

The “Errors / runs” row of the results table shows the number of runs which found an error. The time statistics are wall clock times measured with millisecond accuracy. Memory statistics are collected from the measurements of JPF. The number of paths evaluated results are only relevant to the random search and the EDA, since DFS and BFS operate on a state by state basis.

When testing the algorithms against the 4 philosopher problem, the DFS algorithm is not only the fastest algorithm in terms of time, but also the most efficient in terms of the number of states visited as well as memory usage. Also worth noting is the equivalent results of the random search algorithm and the EDA approach. One can see from the average number of generations required for the EDA to find an error that the average is less than 1. This indicates that the EDA approach was finding errors without performing any learning, showing that a random search is enough for such a small problem. However, the average number of states examined before finding an error is significantly less for the EDA as opposed to random search. An odd result was the memory usage reported for the BFS algorithm. BFS typically exhibits high memory usage, and the low memory usage reported by JPF for 4 philosophers may be due to a bug.

When the problem size is increased to 8 philosophers, the results start to favour the EDA. Both BFS and random search fail to find errors in the 8 philosopher problem. BFS ran out of memory (even when given an upper limit of 30GB), and random search examined 30000 paths before terminating. Whilst the time taken to find an error is equivalent between the EDA and DFS, the number of states examined before an error is found is significantly less when using the EDA. Also worth noting is that the average number of generations required to find the error was 4.96, indicating that the EDA

TABLE I
RESULTS FOR EACH OF THE ALGORITHMS

Problem Size		DFS	BFS	RS	EDA
4	Errors / runs	1/1	1/1	50/50	50/50
	Mean Time (s)	(+)0.491	(+)3.971	(-)8.277	7.356
	Min. Time (s)	0.491	3.971	3.453	4.175
	Max. Time (s)	0.491	3.971	18.165	11.933
	Mean Max. Mem. Usage (MB)	(-)482	(+)80	(+)972	843.4
	Mean States Visited	(-)312	(-)13029	(+)13510	5028
	Mean Paths Evaluated	-	-	(+)385.3	143.32
	Mean Generations	-	-	-	0.74
8	Errors / runs	1/1	0/1	0/50	50/50
	Mean Time (s)	(-)25.191	-	-	23.303
	Min. Time (s)	25.191	-	-	11.428
	Max. Time (s)	25.191	-	-	40.859
	Mean Max. Mem. Usage (MB)	(-)496	-	-	1244
	Mean States Visited	(+)238264	-	-	57470
	Mean Paths Evaluated	-	-	-	811.6
	Mean Generations	-	-	-	4.96
12	Errors / runs	1/1	0/1	0/50	50/50
	Mean Time (s)	(+)16375.392 (4h32m55s)	-	-	54.778
	Min. Time (s)	16375.392 (4h32m55s)	-	-	24.420
	Max. Time (s)	16375.392 (4h32m55s)	-	-	107.376
	Mean Max. Mem. Usage (MB)	(-)3233	-	-	2177
	Mean States Visited	(+)150841824	-	-	158134
	Mean Paths Evaluated	-	-	-	1498
	Mean Generations	-	-	-	9.64
16	Errors / runs	0/1	0/1	0/50	50/50
	Mean Time (s)	-	-	-	123.012
	Min. Time (s)	-	-	-	38.498
	Max. Time (s)	-	-	-	299.005
	Mean Max. Mem. Usage (MB)	-	-	-	3049
	Mean States Visited	-	-	-	357997
	Mean Paths Evaluated	-	-	-	2549
	Mean Generations	-	-	-	16.66

required several rounds of model learning before an error could be found.

This trend continues when analysing problems with increasing numbers of states. For the 12 philosopher problem, DFS requires an enormous amount of time to find an error. The average time required for the EDA to find an error is significantly less. The most striking difference however is the number of states examined before finding an error. Whilst the DFS results have jumped from 2.4×10^5 states for 8 philosophers to 1.5×10^8 for 12, the EDA numbers have increased more gradually. The EDA, on average, examined 1.5×10^5 states before finding an error in the 12 philosopher problem. This indicates that the EDA is focusing the search more effectively on areas of the state space that are more likely to contain deadlocked states.

For the 16 philosopher problem, the EDA is the only technique that finds deadlock when given reasonable resources. DFS ran out of memory after 24 hours of searching, using all of a 30GB memory limit. The EDA, however, found an error with an average time of roughly two minutes, using less memory.

An often cited advantage of EDAs, as opposed to standard evolutionary techniques such as GAs, is the insight into the target problem the model itself can yield. Whilst the model learned by an EDA is not itself a solution to the problem, the model can reveal important insights and provide helpful debugging information. The output of a typical evolutionary

algorithm run is a bag of solutions and nothing more. The model learned by the EDA model checking algorithm proposed in this paper could indeed provide insight into the target problem. The model structure can highlight interesting interleaving of actions between threads that lead to increased blocking, or whatever attribute the fitness function is trying to optimise.

One interesting phenomenon that was noted from the experiments is that the models from which the error paths were sampled are all similar. The model from which the solution for 8 philosophers was sampled is similar in structure and distributions to the model for 32 philosophers. This indicates that the EDA is learning a strategy for finding errors in the *DiningPhil* family of problems. The model learned from solving the 4 philosopher problem can easily be sampled from to aid in solving the 32 and higher philosopher problems. One can exploit this phenomena to exert less computational effort by using models learned from smaller variants of problems to effectively search much larger problems. This kind of approach is often used when verifying hardware designs [Clarke et al., 2000].

Figure 4 shows the results of the EDA applied to increasingly larger *DiningPhil* problems. The parameters for these runs are the same as those described earlier. The boxplots show the outliers, lower, median and upper quartiles for a sample of 50 runs. The number of paths evaluated before finding an error are plotted against the number of philosophers in the target problem, ranging from 4 to 40 philosophers. The figure

Number of paths evaluated against number of philosophers

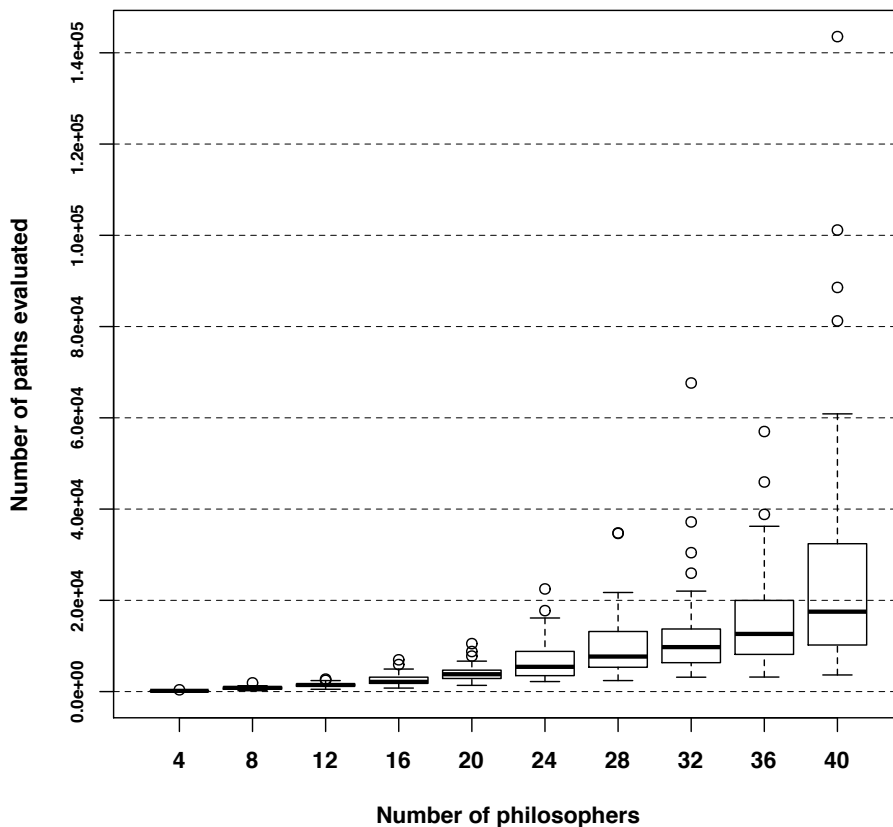


Fig. 4. Boxplots showing the number of paths evaluated before finding an error against the number of philosophers.

shows an upward trend in the number of paths evaluated before an error is found for increasing numbers of philosophers. The trend, however, is impressive given that the state space of the problem is increasing at an enormous rate.

Showing results from larger numbers of philosophers is unfortunately difficult due to the high memory usage of JPF on such large problems. The majority of the memory usage of the EDA can be attributed to JPF, as the EDA is lightweight in terms of both memory and computational effort. With a more efficient model checker or with refinements to JPF, the authors believe there is no reason why larger instances of the dining philosophers problem can not be solved.

VI. CONCLUSION AND FUTURE WORK

The authors have proposed a novel EDA-based algorithm for finding deadlock in concurrent systems. The approach was demonstrated using the Java PathFinder model checker, a popular tool that can analyse compiled Java programs. The proposed algorithm was shown to outperform common model checking algorithms depth-first search and breadth-first search, as well as a random search procedure on large instances of the Dining Philosopher problem. For larger instances of the Dining

Philosophers problem (16 philosophers and above), the EDA was the only algorithm to find deadlocked states using reasonable resources. The EDA-based approach produced results using a competitive amount of resources for smaller instances that are trivially solved using random search and deterministic techniques.

Although the work reported in this paper focuses on deadlock in the dining philosophers problem, the authors believe that the approach can be easily applied to finding other kinds of errors. Extending the EDA to search for liveness violations or race conditions is a possibility and one that the authors are planning to pursue. Fitness functions can be defined for finding liveness errors, and the EDA-based approach can be adapted to searching for infinite paths that violate a liveness property. Additionally, the authors believe that the approach described in this work can be applied to other deadlock problems. However, the fitness function used in this paper may not be effective in finding deadlock in some problems. In these cases, alternatives must be defined. A potential avenue for future work is to automatically derive heuristics from Java programs using static analysis techniques.

The authors also intend to experiment with problems rep-

representative of those faced in industry. Whilst the Dining Philosophers problem is a standard example in the academic community of deadlock in concurrent systems, showing the efficacy of techniques such as these on commercial problems may spur interest from industrial organisations.

The authors believe that there is a large amount of potential for extending the proposed EDA-based algorithm. For instance, a common approach for tackling large problems is to break the state space down into smaller pieces based on depth [Alba and Chicano, 2007]. One can envisage a separate model being learned for choices at depth 0 to 20, 20 to 40 and so on. The EDA could build separate models depending on the code scope of the current state, or mode of the system. There is a large area of opportunity for context-sensitive modelling, as opposed to the “one model fits all approach” of this work.

ACKNOWLEDGMENTS

This work is supported by an EPSRC grant (EP/D050618/1), SEBASE: Software Engineering By Automated SEArch. The authors would like to thank Simon Poulding for his help with statistical matters.

REFERENCES

- E. Alba and F. Chicano. Finding safety errors with ACO. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1066–1073. ACM Press New York, NY, USA, 2007.
- E. Alba and J.M. Troya. Genetic Algorithms for Protocol Validation. *Lecture Notes in Computer Science*, pages 870–879, 1996.
- E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding deadlocks in large concurrent java programs using genetic algorithms. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1735–1742. ACM New York, NY, USA, 2008.
- S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. *Lecture Notes in Computer Science*, 4424:134, 2007.
- C. Baier, J.P. Katoen, and I. NetLibrary. *Principles of Model Checking*. The MIT Press, 2008.
- F. Chicano and E. Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters*, 106(6): 221–231, 2008a.
- F. Chicano and E. Alba. Finding liveness errors with ACO. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence)*. *IEEE Congress on*, pages 2997–3004, 2008b.
- J. Clark, JJ Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, et al. Reformulating software engineering as a search problem. In *Software, IEE Proceedings-[see also Software Engineering, IEE Proceedings]*, volume 150, pages 161–175, 2003.
- Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000. ISBN 0262032708.
- EG Coffman and MJ Elphick. System Deadlocks. *Computing*, 3:67–78, 1971.
- Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.
- M. Dorigo and T. Stützle. *Ant Colony Optimization*. Mit Press, 2004.
- S. Edelkamp, A.L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 57–79. Springer-Verlag New York, Inc. New York, NY, USA, 2001.
- Y. Eytani, R. Tzoref, and S. Ur. Experience with a Concurrency Bugs Benchmark. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop-Volume 00*, pages 379–384. IEEE Computer Society, 2008.
- P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):117–127, 2004.
- D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.
- S. Kirkpatrick, CD Gelatt, and MP Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- Sean Luke, Liviu Panait, Gabriel Balan, and Et. Eej 16: A java-based evolutionary computation research system, 2007.
- K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
- H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature-PPSN IV*, pages 178–187, 1996.
- Doron A. Peled. Ten years of partial order reduction. *Lecture notes in computer science*, pages 17–28, 1998.
- M. Pelikan, D.E. Goldberg, and F.G. Lobo. A survey of optimization by building and using probabilistic models. *Computational optimization and applications*, 21(1):5–20, 2002.
- R. Poli and N.F. McPhee. A linear estimation-of-distribution GP system. *Lecture Notes in Computer Science*, 4971:206–217, 2008.
- A. Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification’90: Proceedings of a DIMACS Workshop, June 18-21, 1990*. American Mathematical Society, 1991.
- W. Visser, K. Havelund, G. Brat, S.J. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.