Finding Short Counterexamples in Promela Models Using Estimation of Distribution Algorithms

Jan Staunton
Department of Computer Science
University of York
York, United Kingdom YO10 5GH
jps@cs.york.ac.uk

John A. Clark
Department of Computer Science
University of York
York, United Kingdom YO10 5GH
jac@cs.york.ac.uk

ABSTRACT

Model checking is an automatic technique that exhaustively checks the state space of a system/program to prove if a specification is satisfied. If an error is detected, the precise circumstances of the issue are returned to the user in the form of a counterexample. Exhaustively checking the state space of a large system, a system with many concurrent components for example, is often intractable. In this scenario, heuristic mechanisms can be employed with the task of detecting errors rather than proving the system is correct. Recently, a metaheuristic EDA-based approach to detecting deadlock in multithreaded Java software has shown great promise in this area. In this paper, we extend that work to search Promela models for counterexamples. We show that the EDA-based technique can find errors where algorithms such as A* search fail. We also show the ability of the EDA to find shorter errors than those discovered by traditional heuristic methods.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Verification, Algorithms, Experimentation

Keywords

Estimation of Distribution Algorithms, Metaheuristics, Model Checking, Safety, Liveness, HSF-SPIN, SPIN

1. INTRODUCTION

Model checking is an automatic technique for verifying concurrent reactive systems [9]. Given a description of a system and a specification of some properties, a model checking algorithm is executed that automatically determines whether the system satisfies the given specification. The model checker exhaustively checks the system, thus proving that the system satisfies the specification. If a violation of the specification is found, a counterexample is returned to the user along with a execution trace that describes the precise circumstances of the fault. This information can then be used to refine the system and/or specification.

For some systems, however, exhaustive verification may be intractable. Systems with many concurrent processes, for instance, can have a large number of states causing a model checker to fail due to lack of resources. The size of the state space typically grows exponentially with respect to the number of concurrent processes in a system. This is known as the *state space explosion problem*, and places limitations on the size of systems that can be verified by a model checker. Despite a number of sophisticated techniques to mitigate this problem whilst preserving the exhaustive property of model checking, verification of some large systems is still out of reach.

As well as exhaustive verification, model checkers can be used in a bug finding capacity. Using heuristic search techniques, a model checker can focus the exploration of a state space on areas of the state space more likely to contain an error. This can reveal errors in a model/specification without exploring the entire state space, using limited time and resources. This approach to model checking is analogous to testing the system. In addition to heuristic techniques, metaheuristic techniques have shown some promise when applied to model checking. Genetic Algorithms, Ant Colony Optimisation and most recently Estimation of Distribution Algorithms have all shown great promise within the model checking problem domain. In this work, we apply a promising EDA-based model checking technique to the HSF-SPIN model checker, a heuristic variant of the SPIN model checker, to search for violations of a variety of properties in a range of models. Some of the models used in our experiments have been derived from industrial samples. We also demonstrate the ability of the EDA to optimise the quality of counterexamples, by finding shorter error trails than other prominent methods.

This paper is structured as follows: Section 2 gives a brief overview of model checking. Section 3 gives an overview of the EDA-based model checking technique. Section 4 reports the empirical work showing the ability of the EDA to find and optimise the length of found counterexamples. Finally, a summary and conclusion is given in Section 5.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

2. MODEL CHECKING

Model Checking is a technique for analysing reactive concurrent systems [9]. 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 system with many processes, for instance, can be constructed using the product of the state spaces of each process. Not all states will be reachable from the initial state, however, given locking and other phenomena. The state of a process consists of the current "program counter", the values of any variables local to the process 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 a *transition* 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* [6]. 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. A transition system has one or more initial states, and may have zero or more terminal states (states from which no transition is possible).

A path through a transition system can be viewed as a sequence of states and/or actions and is equivalent to a potential execution of the system. The specification is given as a set of formalised properties that describe a set of paths. A path that exists in a set described by a property is said to satisfy that property. The types of properties that can be verified are divided into two categories: *safety* properties and *liveness* properties [6]. Safety properties describe requirements of the form "something bad does not happen", an example of which is the "the system does not deadlock". System invariants are also examples of safety properties. Liveness properties describe requirements of the form "something good eventually does happen", an example of which is "the server always responds after being signalled". A popular method of expressing these properties is by using Linear Temporal Logic (LTL).

In order to check for violations of a specification expressed in LTL, automata-based model checking is a typically implemented. Most model checkers (including SPIN) create (at least conceptually) a Büchi that is equivalent to the negation of the LTL formulae. A Büchi automaton differs from regular finite automaton only in the acceptance condition. Whereas regular finite automata accept strings that end in an accepting state, Büchi automata require an infinite string that visits an accepting state *infinitely often*. The negation of the LTL formulae is used because typically the LTL formulae describe the correct behaviour of the system, therefore the negation of the formulae describe behaviours that should never happen. The created Büchi automaton accepts paths which violate the LTL specification. In the SPIN model checker, the Büchi automaton is expressed using a *never claim* and can be automatically created from the negation of LTL formulae.

In order to verify that a specification is satisfied, a model checker systematically examines all of the states (and therefore paths) of the synchronous product Büchi automaton and the transition system. [9]. For safety properties, prominent mechanisms include depth-first search, breadth-first search and A*. For liveness properties, a nested depth-first is typically used. A depth-first search is performed to find an accepting state in the product automaton. Once an accepting state has been found, a second depth-first search is performed to find a cycle. 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 the system. Size is typically measured in the number of states. This issue is known as the *state space explosion* problem [9] 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 [15, 21] and symbolic model checking [5, 14] can reduce state space sizes significantly. Manual reductions in state space size, such as abstracting away from superfluous details, can also be employed.

In some situations, it may suffice to show the presence of a violation rather than exhaustively check a model, employing 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. To produce shorter counterexamples, heuristic algorithms such as A* search can be used to optimise the length of the execution path. [9] and [6] are excellent texts regarding the topic of model checking, describing all of the issues mentioned above and more.

HSF-SPIN implements a particular method that can reduce computational effort whilst model checking, and that method is exploited in this work. The detection and classification of strongly connected components within a negated LTL Büchi automata equivalent can help detect irrelevant paths in the product automata. A subset S of nodes in a graph are strongly connected if for all nodes v and u in S, there is a path from v to u, and a path from u to v. A strongly connected component (SCC) is a strongly connected set of nodes that is not linked to any other by a cycle. The SCCs of a never claim can be computed in linear time, and a library for doing so is provided in HSF-SPIN. The library can also be used to classify certain types of SCC.

[11] describes a number of useful types of SCC. The first is the N-SCC, an SCC with no accepting cycles. Second, the P-SCC (partial SCC) an SCC with at least once cycle that does not contain an accepting state, and one cycle that does contain an accepting state. Finally, the F-SCC, in which all cycles contain an accepting state. For liveness specifications that contain N-SCCs and F-SCCs, once a cycle is found and it is part of an F-SCC, then an error has been discovered due to all cycles in an F-SCC being accepting. Accepting states found in N-SCCs can be ignored because no accepting cycles exist in N-SCCs, potentially saving some computational effort. Classifying and exploiting SCCs seems to result in a higher hit rate in the ACOhg algorithm [2].

Recently, there has been some work on applying metaheuristic search techniques to the model checking problem. Early work reported by [4] and [12] study the use of Genetic Algorithms (GAs) to find errors in systems. Recent work [8, 1] details the use of Ant Colony Optimisation for finding safety and liveness errors in large models, along with work exploiting partial order reduction [7]. In addition to

```
1  (NULL TRANSITION)
2  (models/deadlock.philosophers.noloop.prm:32)(break)
3  (models/deadlock.philosophers.noloop.prm:12)(left?fork)
4  (models/deadlock.philosophers.noloop.prm:12)(left?fork)
```

Figure 1: A typical trace/string/path from HSF-SPIN on the Dining Philosopher problem with 2 philosophers. This trace ends in a deadlocked state, because all the philosophers have picked up their left fork

this, recently published work [20] shows the potential of Estimation of Distribution Algorithms for finding deadlock in multithreaded Java software.

3. EDA-BASED MODEL CHECKING

An understanding of how Estimation of Distribution Algorithms (EDAs) work at an abstract level is assumed and a full description is not given here. For a full description of a broad class of EDAs, we recommend reading [16]. What follows is a brief description of the EDA-based model checking algorithm, which is described fully here [20].

3.1 Model and Solution Space

In order to model paths in the transition system, we use a simple string representation. Paths in a transition system can be seen as a sequence of actions causing transitions between states. The alphabet of the strings used in this work is the set of all actions possible in the transition system. In this work, the alphabet consists of information gathered by HSF-SPIN whilst constructing the transition system. Examples of the alphabet members used in this work can be found in Figure 1, which shows a typical path through a Dining Philosophers transition system. The alphabet members do not refer to specific philosophers, but instead refer to actions that can be performed by any one of the philosophers in the system. By modelling paths through a transition system without referring to specific processes, sequences of actions regardless of which processes executed them are modelled. This represents a minor abstraction from modelling actions performed by specific processes, reducing the size of the alphabet and therefore reducing the solution space searched by the EDA.

3.2 Modelling Paths

In order to model paths in the transition system, we use a customised version of N-gram GP [17]. 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. The rationale is that N-gram GP is modelling a strategy to use whilst exploring a transition system. The n-grams are seen as a recent history of actions on a particular path. The distribution associated with the n-gram in the probabilistic model describes the actions that are most likely to minimise a fitness function, hopefully leading to a counterexample. The model is "queried" with n-grams during the sampling phase in order to probabilistically choose actions that are more likely to lead to a fault.

For each generation a set of "fit" paths is selected using truncation selection and the fitness function described later in this work. In order the learn the model or strategy from a set of fit paths, a simple sliding window frequency count algorithm is used. Once the paths are selected, a frequency count of actions occurring after each unique n-gram in the paths is performed. The frequency count is then normalised to obtain distributions for each n-gram observed. A simple illustration of this process can be seen in Figure 2. In addition to learning 3-gram distributions, in this work the distributions for 2-grams and 1-grams are also constructed and these additional distributions are used during the sampling phase.

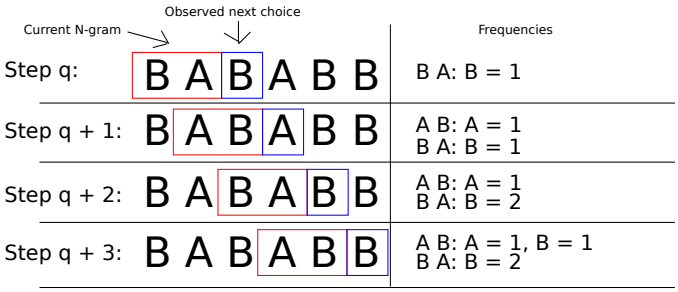

Figure 2: Illustration of the N-gram learning process (2-grams in this case). A frequency count is performed for each unique N-gram in the selected set of strings. The boxes represent a basic sliding window algorithm, with frequency counts display on the right.

3.3 Sampling the Model

Once a set of individuals are selected and a model created, the model is then sampled to create the next generation. A number of paths are generated from the model to replace individuals in the current generation according to some policy. In this work, we generate the entire new generation using newly sampled individuals. To generate a path in the transition system, the algorithm starts with the initial state of the transition system and an empty path. Then, the algorithm must choose an action to execute from the available actions in the current state. To do this, the model is queried with the most recent n moves from the path, in this case an empty string, and a distribution is returned. From the initial state, a set of actions are possible, each of which leads to a potentially new state. Using the distribution obtained from the model for the current n-gram, an action is probabilistically chosen and executed. If more than one process are in a position to execute the chosen action, then a single process is selected at random to progress. This leads to a new state s. The action chosen is appended to the current path, and the process is repeated using the new state and the new path. This process repeats until a non-accepting terminal state is reached, a non-accepting cycle is detected, or an error is detected.

N-gram GP was initially used to evolve programs, allowing any sequence of alphabet members to constitute a program. However, in this work we must generate valid paths in the transition system. If we used the process described above without modification, it is entirely possible to generate invalid paths in the transition system. We have a number of special cases when generating a path, and these are described and handled in [20].

3.4 Fitness function

In order to rank potential solution paths we use a fitness

function that ranks the solutions for truncation selection, employing a HSF-SPIN heuristic to do so. The pseudo-code of our ranking function can be found in Algorithm 1. The fitness functions compares two individuals, returning the "fitter" individual. The fitness function firstly prefers paths that contain errors to paths that do not contain an error. If both paths contain an error, then the shorter path is preferred. If neither path contains an error, then the decision falls back to the HSF-SPIN metric. The HSF-SPIN metric is described in Algorithm 2.

Algorithm 1 Fitness function used to rank individuals. Individuals that are "closer" to violating a property are favoured.

```
Require: A, B are Individuals;
  if A.error_found ≠ B.error_found then
    return IndividualWithErrorFound(A,B);
  else if A.error_found and B.error_found then
    return IndividualWithShortestPath(A,B);
  else
    return IndividualWithLowestHSFSPINMetric(A,B);
  end if
```

Algorithm 2 HSF-SPIN heuristic metric algorithm.

```
Require: I is an Individual;
  aggregateMetric = 0;
  for all States s ∈ I.Path do
    aggregateMetric += s.HSFSPINMetric;
  end for
  return aggregateMetric/LengthOfPath;
```

The algorithm described in Algorithm 2 aggregates a heuristic value calculated by HSF-SPIN for all the states of a path, and then averages that value over the path length. This calculation gives a heuristic value for the entire path which the EDA seeks to minimise. The heuristics used in this work are described in the experimental section. The fitness function described above favours shorter counterexamples to longer ones. Because of this, if the algorithm is allowed to execute further once an error is found, there is the potential for shorter counterexamples to be found.

3.5 Other Features and Parameters

In addition to the features described above, some aspects of more traditional EDAs are adopted in this work. Mutation, for instance, is implemented as taking a uniformly random choice with a fixed probability m which is typically set at a low value. To illustrate, setting m to 1.0 would cause the algorithm to make every choice randomly, Elitism is also implemented, copying n paths verbatim from the current generation to the next.

3.6 Motivation for using EDAs in the model checking domain

Estimation of Distribution Algorithms have been shown to be effective in other software engineering and, more specifically, software testing activities. This activity motivates the use of EDAs for use in the model checking domain. [19] shows how EDAs can be used to generate test data meeting coverage criteria for sequential software. [18] have shown an EDA-based framework for generating test data for object-oriented software. Lastly, [20] have shown that EDAs have great potential in the model checking domain, showing how an EDA can be used to find deadlock in multithreaded Java software.

N-gram GP can generate strings of arbitrary length, and that advantage is exploited here. Previous work involving genetic algorithms and model checking [3] used a representation that could resize during the run using the crossover operator. However, this size has to be "learned" during the course of the run of the GA and a sophisticated "memory operator" augmentation is required. The EDA based approach eliminates this learning step, allowing for the construction of paths of arbitrary length. The ACOhg algorithm described in [1] must store pheromone information for a potentially huge number of transitions/edges, and ACOhg employs complex measures in order to avoid exhausting memory. The EDA, however, is able to discard a large proportion of states expanded during the exploration of a model, making the technique memory efficient. The EDA need only store the action/state sequences discovered during the execution of the algorithm and the probabilistic model, discarding the heavy-weight data structures created by HSF-SPIN or indeed Java PathFinder. This advantage is shared with the GA-based approach described in [3].

An often cited advantage to using EDAs is the ability to analyse and reuse the models produced during the execution of the algorithm. Analysing the models produced may yield insight into the target problem and may be helpful when tuning parameters. In previous work [20], it has been suggested that the models produced by the EDA-based model checking approach could be used to verify errors in families of problems, reducing the effort required to find a violation in larger instances of a problem family. The model could also be used between revisions of a model during the debugging process, allowing for efficient checking of future revisions of the specification/system.

4. EXPERIMENTS IN FINDING AND OPTIMISING COUNTEREXAMPLES

In this section, we present the results of experiments in finding and optimising counterexamples in Promela models using the EDA-based approach. We compare the EDA-based approach to traditional deterministic algorithms included in the HSF-SPIN package. Rather than just running the EDA until an error is found, we allow the EDA to run for 200 generations allowing the EDA to potentially optimise the length of the counterexample. The approach was implemented using the ECJ toolkit [13] and HSF-SPIN [11]. This allowed us to exploit tried and tested heuristics and algorithms, as well as the breadth of Promela examples included with HSF-SPIN. In our implementation, ECJ is the driver of the search process, using HSF-SPIN as a workhorse to explore paths. Our current implementation allows us to not store previously visited states in memory, keeping memory usage to a fairly constant level of around 300MB or less depending on the Promela model. This measurement includes the Java and HSF-SPIN process. To demonstrate the capabilities of the EDA-based approach, we have selected a number of models that violate safety, assertion and liveness properties.

4.1 Example Promela Models

The HSF-SPIN software distribution includes a number

of different models and systems that violate a variety of different kinds of properties. In this work, we use the models listed in Table 1 to demonstrate the efficacy of the EDA-based approach. The table shows the name of the model, the maximum number of processes created, the number of Lines of Code in the model and the property the model violates.

Table 1: Models and the respective properties violated.

Model	Processes	LoC	Property
phil-loop$n$	$n+1$	34	Deadlock
phil-noloop$n$	$n+1$	35	Deadlock
deadlock-giop$n$	$n+6$	717	Deadlock
pots	3	453	Deadlock
leader$n$	$n+1$	117	Assertion
alter	2	64	$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
elev$n$	$n+3$	191	$\Box(p \rightarrow \Diamond q)$
ltlgiop$n$	$n+6$	740	$\Box(p \rightarrow \Diamond q)$
sgc	20	1001	$\Diamond p$

The selected models exhibit a range of property violations which include safety and liveness properties. The first two (*phil-loop* and *phil-noloop*) are Promela implementations of the Dining Philosophers coordination problem. In both of these examples, each philosopher effectively locks/picks up the left fork, then then right fork, then releases them in that order. The "locking" is implemented by reading and writing to channels that represent the forks. The first models infinitely loops this behaviour, whilst the second model terminates each philosopher after executing the fork retrieval. The *deadlock-giop* model is an implementation of the CORBA General Inter-Orb Protocol with a faulty timeout phase that can lead to deadlock. The *pots* model is an implementation of the Plain Old Telephony System which exhibits a deadlock. *leader* implements an election algorithm for nodes in a unidirectional ring configuration and violates a consistency assertion. *alter* is an implementation of the alternating bit protocol. The *elevn* models an elevator that services n floors in a building. The *ltlgiop* model is the same implementation as the *deadlock-giop* model but includes an LTL property which the model violates. Finally, the *sgc* simulates the operator protocol of a power planet. The *giop* series of models have been derived from an industrial standard, and the *phil*, *elev* and *giop* models are stated as having very large state spaces [2].

4.2 Parameters of the EDA

The parameters used in this experiment were chosen through small-scale experimentation on the selection of Promela models, using the parameters from [20] as a guide. 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/N-gram model. All individuals in the population are replaced at each generation with individuals sampled from the model. The algorithm terminates once it reaches 200 generations, allowing for the potential optimisation of counterexamples. Initially, the model is a blank model meaning that all the paths evaluated during the first generation are completely random.

4.3 Experiments

In this section, we present results from experimentation that compares the EDA-based approach against traditional deterministic approaches for finding short property violations. When searching for deadlock, we use the active processes heuristic provided by HSF-SPIN and compare against the A* algorithm. The active processes heuristic simply returns the number of active processes/enabled transitions in a particular state. If the heuristic returns 0 and the state is not a valid terminal state, then the state is a deadlocked state. When searching for the assertion violation in the leader model, we use the formula-based heuristic and A*. The formula-based heuristic is described in [10] and returns an estimate on the number of transitions from a given state s to a violation based upon satisfaction of sub-formulae in a specified property. And finally, we use the endstate of claim distance heuristic and the improved nested depth-first (INDFS) search when searching for LTL violations. The endstate of claim distance heuristic estimates the distance a given state is from an accepting state in the product Büchi automaton. In this scenario, INDFS uses the heuristic to determine the order in which new states are expanded during the depth-first expansion. The same heuristic is used for the EDA and for the respective deterministic approaches, with the deterministic approaches using the heuristic on individual states rather than paths. In order to gain statistically sound results, we run the EDA-based approach 100 times (with the exception of *ltlgiop20* due to time constraints) as the algorithm is probabilistic.

Table 2 shows the results from this experiment, along with some statistical comparison information. The performance with respect to a selection of measures of each algorithm (apart from the EDA) is compared with the performance of the EDA. These measures include the length of the error found, the generation in which the error was found and the number of states expanded before the error is found. These measurements are shown for the first error found by the EDA, and the best error found. We opted to not show wall clocks times as the number of states expanded is a more realistic measure of the algorithms performance. The vast majority of the CPU time and memory used during the course of a run is spent expanding states, as opposed to building models/evaluating statistics. Most run times, of both the EDA and the deterministic approach, range from a 4 milliseconds to a maximum of 4 hours. Statistical comparisons are indicated with plus and minus symbols. In order to compare the EDA-based algorithm against the deterministic variants, we use the Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$. We have also performed comparisons with random search. However, in the interest of parsimony, the random search results are briefly discussed rather than included in full.

4.4 Discussion of Results

Initial observations from the results table reveal that the

EDA is the only algorithm to achieve a 100% hit rate on all of the sample models. The deterministic approaches fail to find an error on the *deadlock-giop* and the *ltlgiop* examples, either hitting a 64GB memory limit or going over a time limit of 24 hours. Random search also failed to discover an error on the *phil-noloop* model family. These models are particularly large, and the results show the ability of the EDA to focus the search on promising areas of the state space revealing errors by expanding fewer states, and consequently using less memory and CPU time. The 100% hit rate on all the test cases shows that the EDA-based approach is a very promising algorithm for discovering counterexamples in large state spaces, especially with respect to robustness and sensitivity to the state space explosion problem. The ACOhg algorithm in [8, 2] shows a less than 100% hit rate on the *ltlgiop* and philosophers models, suggesting that the EDA-based approach may have an advantage on some of the larger models.

In the majority of test cases the EDA found a statistically significantly shorter counterexample than the deterministic approach, with the exception of the *pots* model and the larger *phil-loop* model. However, the difference in the length of the best paths found is a matter of 1 or 2 states on the *pots* model. The larger difference on the *phil-loop128* model is likely explained by the way the heuristic is constructed. By using the average of the heuristic values of states in a path as the heuristic value for a path in the EDA, the EDA may favour longer paths in some instances. For example, a longer path of low heuristic values may be favoured over a short path with high heuristic values. This result shows that the EDA can be sensitive to the heuristic used, and that some models may require a carefully thought out heuristic in order for the EDA to be effective. The ability to optimise counterexamples makes the EDA an appealing approach, as shorter counterexamples remove superfluous information enabling a software tester/model checking practitioner to focus on the underlying cause of an issue. To optimise the counterexample, however, more states must be expanded in order to learn a model that reflects shorter counterexamples. This is shown in the results table on the majority of the models with the exception of the *leader* model, where the EDA managed to find a shorter error by expanding far fewer states and therefore using less memory and CPU time.

In some cases, the number of states required to find the first error in an EDA run is either less than or statistically insignificantly greater than or equal to the results from the deterministic algorithms. In the cases where the EDA does expand more states to find the first error, mere milliseconds are added to the EDA search time with respect to the deterministic search time. However, this is not the case on the *phil-noloop* and *pots* models, where the states required to find the first error is far greater than the deterministic algorithm. However in the vast majority of cases, the EDA found the first error by expanding a reasonable amount of states. The results shown in this experiment show that the EDA is capable of finding errors in a short space of time, even in models (the *giop* based models) that are deemed very large by previous work [2].

In the majority of the tests, one can observe that the number of generations required to find the first error is on average zero. This indicates that the EDA algorithm found the first error using random search alone due to the EDA starting with a blank model. When given a blank model, all transitions are chosen at random. It is the case that random search is also able to find an error with a 100% hit rate in all of the test cases with the exception of the *phil-noloop* model. However in all but three cases (the *elevn*, *alter* and *sgc* models), the EDA was able to shorten the error statistically significantly when compared to the best error found by random search. The fact that random search is able to find an error with a 100% hit rate in the majority of the tests, as well being statistically equivalent to the EDA when faced with the *elev*, *alter* and *sgc* models suggests that a comparison with random search is necessary when evaluating probabilistic algorithms in the model checking domain. It also suggests that as part of a benchmark suite of tests for model checking algorithms, relevant models that defeat random search must be found. We can state, however, that the EDA-based approach achieves a 100% hit rate, whilst being able to more effectively optimise counterexamples in the majority of our selected benchmark tests when compared with random search and the most prominent deterministic algorithms.

On some of the tests, the EDA finds the best error in generation 0 100% of the time. The models in which this occurs are the *sgc* and *alter* models. The best error found in these systems is the shortest possible error that can be found. This indicates that the model is trivial for any sensible algorithm, since the shortest possible error can be found using random search without any guidance/learning. These trivial models are likely not good candidates for future model checking benchmarks and we suggest that the use of these models should be avoided.

5. CONCLUSION AND FUTURE WORK

In this paper we have presented results from experiments that show the promising ability for an EDA-based algorithm to find counterexamples in large state spaces. We compared the algorithm against prominent deterministic approaches using similar heuristics, showing that the EDA can find shorter counterexamples in the majority of the test cases. In two cases, we have shown that the EDA can find errors where traditional approaches fail due to exhaustion of resources, showing that the EDA can effectively focus the effort of the search. The optimisation of counterexamples is a key feature that model checking practitioners could find useful. This ability of the EDA has not been demonstrated before and is part of the novelty of this work. Current prominent mechanisms, namely the improved nested depth-first search algorithm, do not provide this ability. Although they can be extended to potentially do so, it is not clear how well an extended nested depth-first search algorithm would scale on large state spaces. The EDA-based approach shows promise with respect to large state spaces, with the results of the experiment showing robustness when tested with a variety of models.

In addition to previous work [20] showing the ability to find deadlock in concurrent systems, we have shown that the EDA can find violations of assertions and liveness properties, a key step in demonstrating the efficacy of an EDA-based approach. We were initially concerned that the EDA can only be effective on highly symmetric toy problems such as the Dining Philosophers problem. However, the results show that the EDA can find errors in large asymmetric systems that are derived from industrial scenarios. The results suggest that an EDA-based approach to searching a state space

can be used to find an error of any kind provided a suitable heuristic can be defined. In addition to this, the EDA could potentially be used as a counterexample optimiser as a supplement to another algorithm known to be effective on a particular model. For instance, one could use a traditional algorithm to find an error in the *phil-loop* series of models, and then use the EDA-based algorithm with a state-distance heuristic to find a shorter counterexample.

The next phase of our work is to investigate the potential for model reuse within the debugging life-cycle. The use of the models generated by EDAs is often cited as a major advantage over GAs and ACO. In the debugging life-cycle, one can envisage using an EDA to find a bug in a system, and then using the probabilistic models generated from the execution of the EDA in subsequent executions. This could potentially reduce the amount of resources required to find errors in future revisions of a system/specification by using information learned from previous debugging sessions, allowing for more debugging cycles in a set period of time. Reusing models to reduce effort in verifying the existence of errors in large models can also be implemented. Using models learned from small instances of families of problems to find errors in larger instances can be implemented. Our most tantalising prospect is reusing models during the refinement of a system. By running the EDA on an abstract version of a system, there is the possibility to use the information/-models collected from the abstract version to find errors in a refined implementation of that system. In addition to these investigations, we hope to implement the EDA-based technique as an Eclipse plugin, allowing for efficient debugging of concurrent software in more mainstream languages such as Java.

Acknowledgments

This work is supported by an EPSRC grant (EP/D050618/1), SEBASE: Software Engineering By Automated SEarch. We would also like to thank Alberto Lluch for his guidance and help whilst ironing out bugs in HSF-SPIN.

6. REFERENCES

[1] 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.

[2] E. Alba and F. Chicano. Searching for liveness property violations in concurrent systems with ACO. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1727–1734. ACM New York, NY, USA, 2008.

[3] 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.

[4] E. Alba and J.M. Troya. Genetic Algorithms for Protocol Validation. *Lecture Notes in Computer Science*, pages 870–879, 1996.

[5] 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.

[6] C. Baier and J.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[7] 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, 2008.

[8] 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, 2008.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[10] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.

[11] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2):247–267, 2004.

[12] 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.

[13] Sean Luke, Liviu Panait, Gabriel Balan, and Et. Ecj 16: A java-based evolutionary computation research system, 2007.

[14] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[15] Doron A. Peled. Ten years of partial order reduction. *Lecture notes in computer science*, pages 17–28, 1998.

[16] 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.

[17] R. Poli and N.F. McPhee. A linear estimation-of-distribution GP system. *Lecture Notes in Computer Science*, 4971:206–217, 2008.

[18] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 438–444. IEEE, 2008.

[19] Ramón Sagarna and Jose A. Lozano. On the performance of estimation of distribution algorithms applied to software testing. *Applied Artificial Intelligence: An International Journal*, 19(5):457–489, 2005.

[20] Jan Staunton and John A. Clark. Searching for safety violations using estimation of distribution algorithms. *Software Testing Verification and Validation Workshop, IEEE International Conference on Software Testing, Verification, and Validation*, 0:212–221, 2010.

[21] 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.

Table 2: Results for each of the algorithms

Statistic	EDA	Deterministic
pots		
Errors/Runs	100/100	1/1
First error:		
Length	68.08/69	67 (+)
Generation	19.34/9	-
States	44,292.95/22,926	7060 (+)
Best error:		
Length	68.64/69	67 (+)
Generation	42.15/24	-
States	92,286.66/58,298	7060 (+)
phil-noloop*64*		
Errors/Runs	100/100	1/1
First error:		
Length	258/258	258 (-)
Generation	6.35/6	-
States	324,209.42/333,886	258 (+)
Best error:		
Length	258/258	258 (-)
Generation	6.35/6	-
States	324,209.42/333,886	258 (+)
phil-noloop*128*		
Errors/Runs	100/100	1/1
First error:		
Length	514/514	514 (-)
Generation	18.93/18	-
States	1,859,306.63/1,762,271	514 (+)
Best error:		
Length	514/514	514 (-)
Generation	18.93/18	-
States	1,859,306.63/1,762,271	514 (+)
phil-loop*64*		
Errors/Runs	100/100	1/1
First error:		
Length	611.4/594	258 (+)
Generation	0/0	-
States	1,732.25/1,414	258 (+)
Best error:		
Length	268.84/258	258 (-)
Generation	29.21/9	-
States	826,669.98/312,698	258 (+)
phil-noloop*128*		
Errors/Runs	100/100	1/1
First error:		
Length	1,294.12/1,272	514 (+)
Generation	0/0	-
States	2,766.09/2,276	514 (+)
Best error:		
Length	735.32/754	514 (+)
Generation	68.62/5	-
States	4,267,972.45/708,306	514 (+)
deadlock-giop*20*		
Errors/Runs	100/100	0/1
First error:		
Length	197.4/199	-
Generation	0/0	-
States	140.44/142	-
Best error:		
Length	141/141	-
Generation	0.48/0	-
States	17,553.87/18,917	-
deadlock-giop*40*		
Errors/Runs	100/100	0/1
First error:		
Length	283.29/279	-
Generation	0/0	-
States	190.46/182	-
Best error:		
Length	221/221	-
Generation	0.39/0	-
States	19,836.35/17,704	-

Statistic	EDA	Deterministic
leader*5*		
Errors/Runs	100/100	1/1
First error:		
Length	69.17/69	58 (+)
Generation	0/0	-
States	48.17/48	5,108 (+)
Best error:		
Length	55/55	58 (+)
Generation	9.52/7	-
States	69,607.76/53,748	5,108 (+)
leader*10*		
Errors/Runs	100/100	1/1
First error:		
Length	124.24/125	88 (+)
Generation	0/0	-
States	83.24/84	4,876,999 (+)
Best error:		
Length	75.65/76	88 (+)
Generation	86.86/75	-
States	926,073.91/800,486	4,876,999 (+)
alter		
Errors/Runs	100/100	1/1
First error:		
Length	26.32/24	64 (+)
Generation	0/0	-
States	15.31/14	32 (+)
Best error:		
Length	8/8	64 (+)
Generation	0/0	-
States	202.36/170	32 (+)
elev*20*		
Errors/Runs	100/100	1/1
First error:		
Length	767.72/529	1,159 (+)
Generation	0.01/0	-
States	4,127.04/2,978	558 (+)
Best error:		
Length	378.88/381	1,159 (+)
Generation	70.78/53	-
States	1,698,200.08/1,266,285	558 (+)
elev*40*		
Errors/Runs	100/100	1/1
First error:		
Length	1,196.82/723	2,039 (+)
Generation	0.03/0	-
States	8,104.37/5,494	978 (+)
Best error:		
Length	578.98/581	2,039 (+)
Generation	55.99/39	-
States	2,254,850.07/1,603,325	978 (+)
ltlgiop*10*		
Errors/Runs	100/100	0/1
First error:		
Length	209.66/184	-
Generation	0/0	-
States	3,153.72/707	-
Best error:		
Length	73.45/74	-
Generation	111.54/113	-
States	7,291,250.65/7,297,643	-
ltlgiop*20*		
Errors/Runs	87/87	0/1
First error:		
Length	348.9/338	-
Generation	0/0	-
States	57,244.90/50,511	-
Best error:		
Length	96.58/96	-
Generation	105.75/117	-
States	44,940,817.34/49,606,760	-
sgc		
Errors/Runs	100/100	1/1
First error:		
Length	19.54/18	46 (+)
Generation	0/0	-
States	4.4/4	11 (+)
Best error:		
Length	18/18	46 (+)
Generation	0/0	-
States	6.28/4	11 (+)