CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Boosted Decision Trees for Behaviour Mining of Concurrent Programs

R. Avros[2], V. Dudka[1], B. Křena[1], Z. Letko[1], H. Pluháčková[1],
S. Ur[1], T. Vojnar[1], Z. Volkovich[2]

[1] *IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Brno, CZ*
[2] *Ort Braude College of Engineering, Software Engineering Department, Karmiel, IL*

SUMMARY

Testing of concurrent programs is difficult since the scheduling non-determinism requires one to test a huge number of different thread interleavings. Moreover, repeated test executions that are performed in the same environment will typically examine similar interleavings only. One possible way how to deal with this problem is to use the noise injection approach, which influences the scheduling by injecting various kinds of noise (delays, context switches, etc.) into the common thread behaviour. However, for noise injection to be efficient, one has to choose suitable noise injection heuristics from among the many existing ones as well as to suitably choose values of their various parameters, which is not easy. In this paper, we propose a novel way how to deal with the problem of choosing suitable noise injection heuristics and suitable values of their parameters (as well as suitable values of parameters of the programs being tested themselves). Here, by suitable, we mean such settings that maximize chances of meeting a given testing goal (such as, e.g., maximizing coverage of rare behaviours and thus maximizing chances to find rarely occurring concurrency-related bugs). Our approach is, in particular, based on using data mining in the context of noise-based testing to get more insight about the importance of the different heuristics in a particular testing context as well as to improve fully automated noise-based testing (in combination with both random as well as genetically optimized noise setting).

Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Automated testing, concurrent programs, noise injection, data mining, AdaBoost, genetic algorithms.

1. INTRODUCTION

Testing of concurrent programs is known to be rather difficult since concurrently running threads can be executed in many different interleavings out of which only a small fraction can contain errors. A single execution of available tests used in traditional unit and integration testing usually exercises a limited subset of all possible interleavings only and hence can easily miss possibly present concurrency-related errors. Moreover, repeated executions of the same tests in the same environment usually exercise similar interleavings [3, 8], and so they will not help much to reveal errors hiding in rare interleavings.

One of the main approaches that have been proposed to cope with the above problem is *noise injection* [8, 13]. This approach is based on disturbing thread scheduling (e.g., by injecting, removing, or modifying delays, forcing context switches, or halting selected threads) with the aim of increasing the number of interleavings witnessed within repeated executions of a given concurrent program. Noise injection can drive the execution of a program into scenarios that are legal but

Copyright © 0000 John Wiley & Sons, Ltd.

less probable under common scheduling conditions, which can increase the chance of spotting concurrency-related errors that often hide in rare behaviours.*

However, as we have shown in our previous works, e.g., [35, 21, 23, 13], the efficiency of noise injection highly depends on the type of the generated noise, on the strength of the noise (which are both determined using some *noise seeding heuristics*), as well as on the program locations and instants of program executions where some noise is injected (which is determined using *noise placement heuristics*). Many different noise seeding and noise placement heuristics have been proposed and experimentally evaluated. Unfortunately, choosing the right heuristics for the given context is not easy. Therefore, random selection of noise seeding and noise placement heuristics as well as random setting of their parameters is often used. As an improvement on this practice, applications of single-objective as well as multiple-objective genetic algorithms have been proposed in [23, 21] to optimize the noise setting for a given context.

In this paper, based on our preliminary work [1], we propose a novel approach to choose suitable noise seeding and noise placement heuristics as well as suitable values of their parameters (and possibly also values of parameters of the tested programs themselves). The aim is to maximize chances of meeting a given testing goal (such as, e.g., maximizing coverage of rare behaviours and thus maximizing chances to find rarely occurring concurrency-related bugs). Our approach is, in particular, based on using *data mining*, applied on a sample of test runs of a given concurrent program, to derive *classifiers* capable of distinguishing which test and noise settings are suitable and which unsuitable for the given testing goal. To be more precise, we use *decision trees* and the *AdaBoost* machine learning algorithm, which is a well-known technique for building high-quality classifiers.

We show how AdaBoost can be applied to gain new knowledge about efficient noise-based testing of a given concurrent program with a given testing goal (or even more generally for a class of programs and/or testing goals). Subsequently, we show how the results obtained by data mining can be used to fully automatically improve testing based on randomly set up noise injection. This is achieved by either filtering out unsuitable randomly chosen settings or by narrowing down the random generation to suitable ranges of noise and/or test case parameters. Moreover, we also show that the obtained results can be used to guide and consequently speed up an automated search-based process of finding suitable values of test and noise parameters. For that purpose, we combine the process of mining of suitable settings of noise-based testing with a subsequent genetic optimization restricted to the values considered as suitable by data mining.

In order to show that the proposed approach can indeed be useful, we apply it for optimizing the process of noise-based testing for two particular testing goals on a set of several benchmark programs. Namely, we consider the testing goals of *reproducing known errors* and *covering rare interleavings* which are likely to hide so far unknown bugs. Our experimental results confirm that the proposed approach can discover useful knowledge about the influence and suitable values of test and noise parameters, which we show in two ways: (1) We manually analyse information hidden in the classifiers, compare it with our long-term experience from the field, and use knowledge found as important across multiple case studies to derive some new recommendations for noise-based testing. (2) We show that the obtained classifiers can be used—in a fully automated way—to significantly improve efficiency of noise-based testing using a random selection of test and noise parameters as well as to be successfully combined with finding suitable noise settings by genetic optimization.

Plan of the paper. The rest of the paper is structured as follows. Section 2 provides a brief introduction to the techniques that our approach builds on—in particular, noise injection, concurrency coverage metrics, the AdaBoost machine learning algorithm, and genetic algorithms. Section 3 presents our proposal to use data mining in noise-based testing of concurrent programs, and Section 4 provides results of experiments with the proposed approach. Section 5 summarizes the related work. Finally, Section 6 concludes the paper.

*Noise injection is, of course, not much appropriate when studying real-time or performance aspects of a program, which can be influenced by noise injection.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

2. PRELIMINARIES

In this section, we introduce the basics of several areas that are important for a proper understanding of the rest of the paper—namely: (1) *noise injection* for testing concurrent software and its various parameters whose suitable values we search through the new approach proposed in this paper, (2) *concurrency coverage metrics* that are used to steer and evaluate noise-based testing, (3) the *AdaBoost* approach to machine learning that is at the heart of our approach to finding suitable values of noise parameters, and (4) *genetic algorithms* that were used for finding suitable noise settings in our previous work.

2.1. Noise Injection

As we have already said, noise injection disturbs the common scheduling of concurrently executing threads in order to allow for testing less common (but legal) schedules. In Figure 1, we illustrate two of the possible effects that noise injection can have. Figure 1(a) illustrates a scenario in which the usual order in which two threads execute some events is swapped by noise injection (e.g., by an inserted delay). This can uncover a bug that happens only if the events happen in the swapped order. Note that if the swapped order can happen with noise injection, then the programmer did not exclude it using any synchronization means, and it can happen even without noise injection. If there was some synchronization in place, noise injection could not overcome it. This is, no new behaviour is introduced; just without noise injection, the probability of the events happening in the swapped order may be very low. Figure 1(b) then shows a situation where noise injection prolongs the time spend by a thread in a critical section, which can lead to another thread executing its critical section in parallel with the first one, possibly causing some concurrency error. As before, if such an error happens, it is a real error since the programmer did not prevent the situation by using any synchronization means, which noise injection would not be able to overcome. Thus, the situation can happen even without noise injection, though perhaps with a much lower probability.

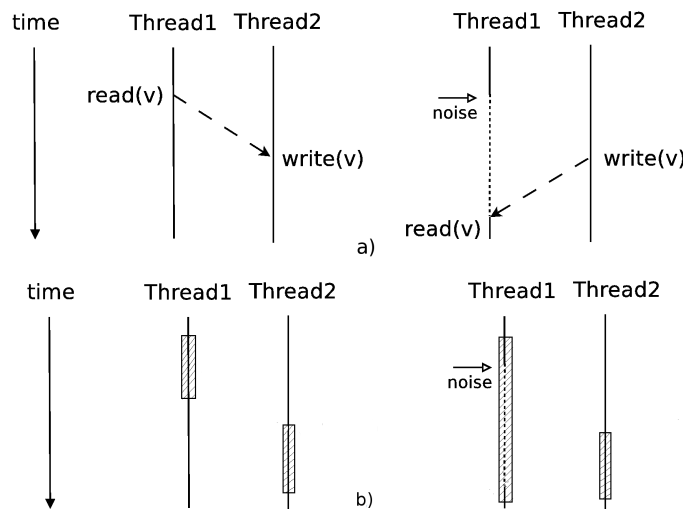

Figure 1. Two examples of the effect of noise injection: (a) reordering of the common order of two events in a concurrent program execution and (b) prolongation of the time spent by a thread in a critical section, leading to an overlapped execution of two critical sections.

We now provide some more technical details on noise injection. A thorough discussion of the technique can be found, e.g., in [13]. Noise injection heavily depends on two kinds of heuristics—namely, *noise seeding heuristics* and *noise placement heuristics*. The noise seeding heuristics determine the type and strength of the generated noise whereas the noise placement heuristics determine at what instants of program executions the noise gets injected.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

In the experiments conducted in this work with concurrent Java programs, we consider six basic and two additional noise seeding heuristics that are all commonly used in noise-based testing. It is assumed that one always uses one of the basic heuristics, which can but need not be combined with one or both of the additional heuristics.

The basic noise seeding heuristics are: *yield*, *sleep*, *wait*, *busyWait*, *synchYield*, and *mixed*. The *yield* and *sleep* heuristics inject calls of the `yield()` and `sleep()` methods, respectively. In the case of the *wait* heuristic, the concerned threads must first obtain a special shared monitor, then call the `wait()` method, and finally release the monitor. The *synchYield* heuristic combines the *yield* heuristic with obtaining the monitor as in the case of the *wait* heuristic. The *busyWait* heuristic inserts a busy-waiting loop that is executed for some time. Finally, the *mixed* heuristic randomly chooses one of the five other basic heuristics at each noise injection location.

The additional noise seeding heuristics are: *haltOneThread* and *timeoutTamper*. The *haltOneThread* technique occasionally stops one thread until any other thread cannot run. The *timeoutTamper* heuristic randomly reduces the time-outs used when calling `sleep()` in the tested program (to test that programmers do not try to synchronize their threads by explicitly delaying some events).

All the above mentioned seeding techniques are parameterised by the so-called *strength of noise*. In the case of the *sleep* and *wait* heuristics, the strength gives the time to wait. In the case of the *yield* heuristic, the strength says how many times the `yield()` method should be called.

The noise placement heuristics are: the *random* heuristic, the *sharedVarNoise* heuristic, and the *coverage-based* heuristic. The *random* heuristic injects noise with some probability before every concurrency-related event in the program execution. The *sharedVarNoise* heuristic allows one to focus noise primarily at accesses to shared variables. There are two versions of this heuristic: *sharedVarNoise-all* which targets all accesses to shared variables and *sharedVarNoise-one* which targets accesses to a single randomly chosen shared variable in each test execution. Moreover, for both of these heuristics, one can decide whether the noise should be inserted solely when accessing shared variables or also at synchronisation operations such as locking (the so-called *nonVariableNoise* heuristic).

The *coverage-based* heuristic is based on collecting information about pairs of subsequent accesses to a shared variable from different threads and on inserting noise before further executions of the program instruction by which the given variable was accessed first (or before acquiring the shared lock that guards the given access provided there is such a lock). This is motivated by trying to reverse the ordering in which threads access variables.

As we have mentioned already above, the noise placement heuristics inject noise at the selected points of program executions with some probability. This probability is determined by the *noise frequency* parameter. The values of this parameter range from never inserting a noise to always inserting it. Additionally, the *coverage-based* heuristic can be extended by another heuristic (denoted as the *coverage-based-frequency* heuristic that monitors the frequency with which a program location is visited during testing and injects noise at the given program location with a probability adjusted according to this frequency—the more often a program location is executed the lower probability is used.

2.2. *Concurrency Coverage Metrics*

Coverage metrics play a crucial role in testing as they allow one to estimate how well a program has been tested and thus to control the testing process. However, coverage metrics successfully used for testing of sequential programs (like statement coverage) are not sufficient for testing of concurrent programs because they do not reflect concurrency-related aspects of executions. In particular, they do not reflect in any way how many interleavings or—even more importantly—how many interleavings that are important from some point of view (e.g., detection of some kind of error) have been witnessed.

Various kinds of concurrency coverage metrics have been proposed in the literature, cf., e.g., [2, 34]. In this work, we, in particular, use the *GoldiLockSC* coverage metric* proposed in [34]. We chose this metric due to our positive experience with its behaviour obtained in our previous

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

experiments. This metric does usually not suffer from shoulders, it does not immediately jump to almost full coverage, it does not keep increasing forever, and it uses an advanced mechanism to distinguish interleavings from the point of view of the involved synchronization. The metric measures how many internal states the *GoldiLock* data race detector with the fast short circuit checks [10] would reach for distinct threads when monitoring the given program. This detector is one of the most advanced data race detectors which combines the use of lock sets with computing the happens-before relation. The control of the checker can thus distinguish in a quite involved way different classes of interleavings seen so far.

2.3. The AdaBoost Machine Learning Algorithm

The core idea of our approach is to apply AdaBoost in noise-based testing to derive classifiers capable of distinguishing suitable and unsuitable settings of noise parameters as well as parameters of the programs under test (and consequently to facilitate searching for suitable test and noise settings). The AdaBoost algorithm, introduced in 1995 by Freund and Schapire [17, 18, 19], is a widespread machine learning technique based on improving ("boosting") the strength of multiple weak classifiers. This is achieved by weighting outputs of the weak classifiers and combining them into a single strong classifier. A weak classifier is any classifier that behaves better than random guessing (i.e., its error degree is less than 0.5 in the binary classification case).

AdaBoost works in iterations. In each iteration, the method aims at producing a new weak classifier in order to improve the precision of the so far constructed strong classifier. To construct the new classifier, objects in the training set are assigned weights. Initially, the weights are distributed uniformly. In each iteration, weights of wrongly classified objects are enlarged, which is then used in the next round to derive and add a new weak classifier focusing on the hard examples in the training set, hence improving the precision of the strong classifier.

In the binary classification case, the input of AdaBoost is a set $\mathcal{X} = \{(\overline{x}_1, y_1), \ldots, (\overline{x}_n, y_n)\}$ where each $\overline{x}_i$ is an object from some space $\mathbb{X}$ of objects that we might want to classify as having or not having some property of interest, and each label y_i belongs to the set $\mathbb{Y} = \{1, -1\}$, which says whether $\overline{x}_i$ does or does not have the property of interest. The input set $\mathcal{X}$ is then commonly split to two subsets—the training set $\mathcal{T}$ and the validation set $\mathcal{V}$. The training set is used to get a classifier while the validation set is used for evaluating the precision of the obtained classifier.

The final strong classifier is obtained in the form

$$F(\overline{x}) = sign\left(\sum_{i=1}^{T} w_i r_i(\overline{x})\right)$$

where $\overline{x} \in \mathcal{X}$, T is the number of boosting iterations, r_i is the weak classifier produced at the i-th iteration of the algorithm (producing decisions from the set $\mathbb{Y}$), and w_i is a non-negative weight expressing confidence in the i-th weak classifier.

Subsequently, precision of the obtained classifier should be evaluated on the validation set $\mathcal{V}$. For that, one can use the notions of *accuracy* and *sensitivity*, based on the following quantities [31]:

- The number TP of *true positives* which is the number of correctly classified positive examples, i.e., those objects $\overline{x}$ where $(\overline{x}, 1) \in \mathcal{V}$ and $F(\overline{x}) = 1$.
- The number FP of *false positives* which is the number of wrongly classified negative examples, i.e., those objects $\overline{x}$ where $(\overline{x}, -1) \in \mathcal{V}$ but $F(\overline{x}) = 1$.
- The number TN of *true negatives* which is the number of correctly classified negative examples, i.e., those objects $\overline{x}$ where $(\overline{x}, -1) \in \mathcal{V}$ and $F(\overline{x}) = -1$.
- The number FN of *false negatives* which is the number of wrongly classified negative examples, i.e., those objects $\overline{x}$ where $(\overline{x}, 1) \in \mathcal{V}$ but $F(\overline{x}) = -1$.

Accuracy then gives the probability of a successful classification and can be computed as the fraction of the number of correctly classified items and the total number of items, i.e.:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

On the other hand, sensitivity (also called as the true positive rate or TPR) expresses the fraction of correctly classified positive results and can be computed as the number of the items that were correctly classified positively divided by the sum of the correctly positively and incorrectly negatively classified items (see, e.g., [57]):

$$sensitivity = \frac{TP}{TP + FN}$$

Moreover, in order to avoid over-fitting and to increase confidence in the obtained results, the process of choosing the training and validation set and of learning and validating the classifier can be repeated several times, allowing one to judge the average values and standard deviation of accuracy and sensitivity. If the obtained classifier is not validated successfully, one can repeat the AdaBoost algorithm with more boosting iterations and/or a larger input set $\mathcal{X}$.

2.4. *Genetic Algorithms*

Next, we present a brief introduction to genetic algorithms whose application for finding suitable values of test and noise parameters was proposed in [21, 23]. In this work, we compare the efficiency of these approaches with our approach based on data mining. Moreover, we also show that the approaches based on genetic algorithms and our approach based on data mining can be very well combined, leading in many cases to further improved efficiency.

Genetic algorithms [62] are metaheuristic search techniques which try to find the best solution by sampling the space of possible (candidate) solutions, usually denoted as *individuals* in this context. A finite set of individuals is called a *population*. Each individual is given by a set of properties of the solution it represents—denoted as *genes* in the *genome* of the individual. Individuals (or , more precisely, their genomes) are usually encoded as vectors. In the context of noise-based testing of concurrent programs, an individual encodes a concrete combination of the noise seeding and noise placement heuristics to be used, the concrete values of the different parameters with which these heuristics are to be applied, as well as values of the various parameters of the program under test (provided that the program—or, more precisely, its tests—are parameterized).

Genetic algorithms start with an initial population and evaluate all its members using a *fitness function*. Based on this evaluation, some of the individuals are chosen by *selection operators* to become *parents* of new individuals called *children*. The new individuals are usually obtained by *crossing* pairs of selected parents followed by a *mutation*. This process, called *breeding*, proceeds until the new child population is completed. New generations are gradually created till a sufficiently good solution is found or some maximum number of generations is reached.

Selection operators. From the current population, parents for breeding can be chosen using different techniques. For instance, the *fitness-proportionate selection* selects individuals proportionally to their fitness—individuals with a higher fitness have a higher probability to be selected for breeding than individuals with a lower fitness [40]. The *tournament selection* is based on a tournament—a specific number of individuals is randomly selected from the current population and the one with the highest fitness is taken for breeding [40].

Crossover. When two parents are selected for breeding, a crossover takes place—two new individuals are created by a recombination of the genomes of the parents (i.e., by exchanging parts of the vectors encoding them). The most common crossover techniques are *one-point*, *two-point*, and *uniform crossover* [40]. When the one-point crossover is applied, the crossing occurs at one place only. A position c of the crossing is chosen between 1 and the length of the genome l. New individuals are obtained by exchanging gens of parents from the position c to the end of their genomes. For a two-point crossover, two positions of crossing c_1 and c_2 are chosen, both between 1 and l, and new individuals are obtained by exchanging gens of the parents just between the positions c_1 and c_2. The uniform crossover technique goes through the whole genomes and exchanges a pair of corresponding gens with some predefined probability.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Mutation. Mutation is applied on a single individual—each gen from the individual's genome is replaced by any value permissible for this gen with some predefined probability.

Single-Objective Genetic Algorithm (SOGA). In single-objective optimization [21], the set of candidate solutions needs to be totally ordered according to the values of the fitness function. However, SOGA can be used to solve multi-objective problems, such as the problem of finding optimal test and noise parameters considered in this work, too. The traditional approach to solve such problems using SOGA is to bundle all objectives into a single scalar fitness function using a weighted sum of objectives. For instance, for three objectives, one can use a fitness function of the form:

$$fitness = w_1 * \frac{metric_1}{metric_1^{max}} + w_2 * \frac{metric_2}{metric_2^{max}} + w_3 * \frac{metric_3}{metric_3^{max}}.$$

Here, the objectives are measured using metrics $metric_i$ with maximum values $metric_i^{max}$ for $i \in \{1, 2, 3\}$. The efficiency of this approach heavily depends on the selected weights, which are sometimes not easy to determine.

Multi-Objective Genetic Algorithms (MOGA). Multi-objective optimization [22, 23] treats objectives separately and compares candidate solutions using the *Pareto dominance* relation: A solution is Pareto-dominant with respect to another solution if it improves on the other solution in all considered objectives. A multi-objective genetic algorithm searches for non-dominated individuals, i.e., individuals that are better than the others in at least one objective while being worse in other objectives. Such individuals are called *Pareto-optimal*, and they represent the best available trade-offs among the considered objectives (with none of the Pareto-optimal solutions being clearly better than the others). There usually exists a set of such individuals which form the *Pareto-optimal front*. There exist several algorithms for multi-objective optimization that use different ways of evaluating the individuals, but all of them exploit the non-dominated sorting. In this paper, we, in particular, consider the *Non-Dominated Sorting Genetic Algorithm II* (NSGA-II) [6], which provided us with the best results in our previous work [22, 23].

3. CLASSIFICATION-BASED DATA MINING IN NOISE-BASED TESTING

In this section, we describe our proposal of using a particular kind of AdaBoost classifiers for discovering which test and noise parameters and which of their values are the most influential for a given program under test and a given testing goal (or, even in general, across different programs under test and/or testing goals). We first describe the concrete kind of AdaBoost classifiers that we propose to be used in noise-based testing, and we provide a generic approach for deriving such classifiers. We then concretise the method for two concrete testing goals common in practice—namely, for finding rare behaviours in which so far unknown bugs may reside and for reproducing known errors. Subsequently, we discuss how the derived AdaBoost classifiers can be used to draw some conclusions about which test and noise configurations are the most influential in the given setting. Finally, we discuss three ways of using the derived classifiers in fully automated testing.

3.1. Combining Data Mining Based on AdaBoost with Noise-based Testing

For our application of data mining with the aim of finding suitable settings of noise-based testing of concurrent programs, we propose using data mining based on *binary classification*. Methods that have been used for binary classification in the literature include decision trees, Bayesian networks, support vector machines, or neural networks [57]. In this work, we, in particular, choose *decision trees*. This is motivated by the fact that one can easily understand and further exploit information hidden in decision trees obtained by machine learning, which we leverage in the following.

Decision trees, such as those shown in Fig. 2, can be viewed as hierarchically structured decision diagrams whose nodes are labelled by Boolean conditions on the items to be classified and whose

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

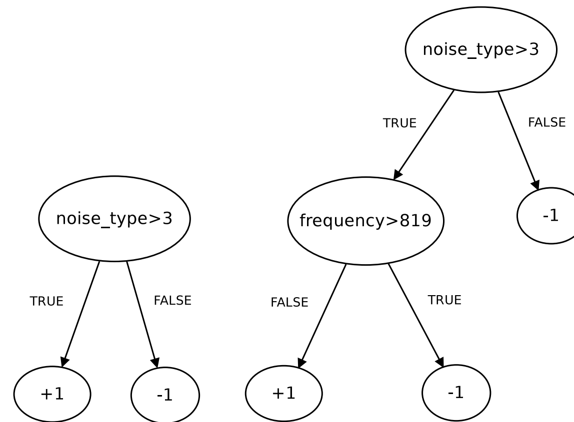

Figure 2. Examples of decision trees.

leaves represent classification results (in our case, $+1$ is used to denote a positive result, while -1 denotes a negative result). The decision process starts in the root node by evaluating the condition associated with the root on the item to be classified. According to the evaluation of the condition, a corresponding branch is followed into a child node. This descent, driven by the evaluation of the conditions assigned to the encountered nodes, continues until reaching a leaf node, and hence a decision. Decision trees are usually employed as a predictive model constructed via a decision tree learning procedure, which uses a training set of classified items.

In order to reduce the natural tendency of decision trees to be unstable (meaning that a minor data oscillation can lead to a large difference in the classification), we combine them with using the AdaBoost approach described in Section 2.3. Decision trees, with the classification result being 1 or -1, are used as the weak classifiers. The resulting strong classifier then consists of a set of weighted decision trees that are all applied on the item to be classified, their classification results are weighted by the appropriate weights, summarized, and the sign of the result provides the final decision.

In order to be able to apply AdaBoost in noise-based testing, one has to first define some *testing goal* expressible as a binary *test property* that can be evaluated over test results such that both positive and negative answers are obtained. The test property will typically be based on some non-binary *test quantity* such as the number of discovered error occurrences, number of covered tasks of some metric, testing time, or a (weighted) combination of such quantities. The binary test property can then be obtained by taking the median value of the test quantities obtained throughout the test runs and by classifying test and noise settings to those that lead (or do not lead) to results above the median.

Example *3.1*
So, a binary test property can, e.g., look like $coverage > C \wedge time < T$ *where* $coverage$ *measures coverage under the chosen coverage metric,* C *is the median coverage obtained in the so far performed test runs,* $test$ *measures the time of executing a test, and* T *is the median testing time in the so far performed runs.*

The requirement of having both positive and negative results can be a problem in some cases, notably in the case of discovering rare errors where getting positive results is—naturally—very rare. In such a case, one has to use a property that approximates the target test property (e.g., by replacing the discovery of rare errors by discovering any rare program behaviours even when they do not contain an error) and provides both positive and negative answers sufficiently often. Of course, once some testing goal is satisfied (e.g., once testing aimed at rare behaviours manages to find some error), another testing goal can become more urgent—e.g., that of repeatedly reproducing the same error for debugging purposes or finding other similar errors. The training process is then to be repeated, possibly using the newly available test results found by previously conducted test runs.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Further, note that, in the context of testing concurrent programs, the test property will typically not be defined over results of particular test runs but rather on results of multiple test runs performed under the same test and noise setting. The reason is the need of minimizing the influence of *scheduling nondeterminism*. The results obtained in several test runs can be summarised by taking, e.g., the median or cumulative value of the considered test quantity.

Once the test property representing the chosen testing goal is defined, a number of test and noise configurations is to be generated at random. Several test runs are to be performed for each of these configurations, and the test property is to be evaluated on each of the series of the test runs performed with the same test and noise configuration. For each of the considered test and noise configurations, a couple $(\overline{x}, y)$ is formed where $\overline{x}$ is a vector recording the test and noise configuration used and $y \in \{1, -1\}$ is the result of evaluating the test property. This way, we obtain the set $\mathcal{X} = \{(\overline{x}_1, y_1), \ldots, (\overline{x}_n, y_n)\}$ to be used as the input of AdaBoost as described in Section 2.3.

***Example** 3.2*
An example of a couple, which can appear in the set $\mathcal{X}$ if we consider three noise parameters, e.g., noise frequency, strength of noise, and type of noise, can be $((839, 28, 1), -1)$. *It says that for the values* 839*,* 28*, and* 1 *of the noise frequency, strength of noise, and type of noise, respectively, the test property evaluated negatively.*

In Section 3.2, we illustrate and further concretise the above ideas by proposing concrete test properties and ways of evaluating them for two testing goals common in practice: namely, *finding rare behaviours* and repeatedly *reproducing known errors*.

Once the set $\mathcal{X}$ is obtained, the AdaBoost algorithm can be applied and the result validated as described in Section 2.3. A successfully validated classifier can subsequently be analysed to get some insight which test and noise parameters are influential for testing the given program and which of their values are promising for meeting the defined testing goal. Such knowledge can then in turn be used by testers when thinking of how to optimize the testing process. We propose a way how such an analysis can be done in Section 3.3, and we experiment with it in Section 4.4. Moreover, the obtained classifier can also be used to fully automatically improve performance of noise-based testing: we propose three approaches how this can be done (two of these approaches based on filtering randomly generated test and noise settings and one based on a combination with genetic optimization) in Section 3.4. Experiments with these approaches are then described in Section 4.5.

3.2. Finding Rare Behaviours and Reproducing Known Errors

We now concentrate on two concrete testing goals: namely, (1) *repeatedly finding known errors*, which is useful for debugging purposes, and (2) *finding rare behaviours*, which is useful for finding bugs missed by common testing runs. For these two different goals, we propose concrete test properties and a way of evaluating them that turned out as suitable in our experiments for deriving input sets for AdaBoost such that AdaBoost in turn produces appropriately trained classifiers for the given testing goals.

In the case of trying to repeatedly reproduce a known error, the test property of interest is simply the *error manifestation property* that indicates whether an error manifested during the performed test executions or not. When deriving the input set $\mathcal{X}$ for AdaBoost that should in turn produce a classifier suitable for reproducing the given error, we generate a number of random test and noise configurations, perform several test runs with each of the configurations[†], and compute the number of test runs in which the error has been found. Then, we compute the median value of the number of runs in which an occurrence of the given error has been found for the different considered test and noise configurations. Configurations that reached a number of error occurrences above the median are marked as positive whereas the remaining ones are marked as negative. This will give us the set $\mathcal{X}$ that will be split into a testing set and a validation set. The testing set will be used as the input for AdaBoost, which will then produce an appropriately trained classifier for the error manifestation property.

[†]In our experiments, we, in particular, use five runs.

Example *3.3*
For an example of getting an input set for AdaBoost according to the above description, see Table I. In particular, we consider five combinations of values of three noise parameters, namely, noise frequency, strength of noise, and type of noise. Assume that when we perform five testing runs with each of the settings, we get the number of error manifestations shown in the fourth column of the table—with the median number of error manifestations being 0*. Then the classification results will be those given by the fifth column of the table. This gives us the set* $\mathcal{X} = \{((839, 28, 1), 1), ((114, 36, 5), -1), ((724, 48, 4), -1), ((895, 12, 0), 1), ((234, 8, 4), -1)\}$ *that will be split into a training and validation set for AdaBoost.*

Table I. An example of constructing an input for AdaBoost for the error manifestation property.

noise frequency	*strength of noise*	*type of noise*	*number of error manifestations*	*classification result*
839	28	1	2	1
114	36	5	0	-1
724	48	4	0	-1
895	12	0	5	1
234	8	4	0	-1

Once a classifier is derived, its precision and stability are tested on the validation set. In particular, we let the generated configurations be classified by the derived classifier as suitable or unsuitable for reproduction of the known errors, and, subsequently, we check correctness of the classification through repeated test runs under these configurations. The concrete numbers of test runs considered to get the training and validation sets in our experiments are provided in Sections 4.3 and 4.5.

Next, we consider the case of finding test and noise configurations suitable for testing rare behaviours in which so far unknown bugs might reside. In order to achieve this goal, we use classification according to a *rare events property* that indicates whether a test execution covers at least one rare coverage task of a suitable coverage metric—in our experiments, the *GoldiLockSC** metric [10] is used for this purpose. To distinguish rare coverage tasks, we collect the tasks that were covered in at least one of the performed test runs (i.e., both from the training and validation sets), and, for each such coverage task, we count the frequency of its occurrence in all of the considered runs. We define the rare tasks as those that occurred in less than 20 % of the test executions.

Furthermore, when learning the classifier, we want to avoid the scenario where we find some test and noise configurations that are capable of finding some behaviours that are rare in normal test runs, but they lead to discovering the same behaviours in each noised test run again and again. This is, we ideally want to keep finding different rare behaviours in each test run. To stress this goal, we focus on the *cumulative number* of covered rare tasks, not only on coverage in individual executions. In our experiments, we, in particular, use cumulation from five test runs. This is, we randomly generate a number of test and noise configurations. With each of them, we execute five test runs, and we cumulate (i.e., unite) the sets of covered rare tasks.

Subsequently, as we consider the time needed for testing to be also important, we take the sizes of the cumulated sets of covered rare tasks and divide them by the time needed to perform the considered five test executions. We take as positive the test and noise configurations whose cumulated number of covered rare tasks divided by the needed test time is above the median value of this combined test quantity. We then derive the AdaBoost classifier and test its precision and stability. The concrete numbers of test runs considered to get the training and validation sets in our experiments are again provided in Sections 4.3 and 4.5.

Example *3.4*
Table II gives an example of obtaining an input set for AdaBoost for the rare behaviours property according to the above description. Namely, we consider three combinations of three noise

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

parameters (noise frequency, strength of noise, and type of noise as before). To shorten the example, we assume that three testing runs were performed with each of these configurations only. Further, we assume that the rare tasks that were covered in the testing runs are as shown in the fourth column. The fifth column gives the time we assume to be consumed for the testing runs. The sixth column then gives the corresponding cumulative coverage of rare tasks divided by the total consumed time. Finally, the last column gives the appropriate classification result (due to the median coverage being $3/7$*). The value from the last column is to be used together with the values in the first three columns to derive the input set for AdaBoost:* $\mathcal{X} = \{((83, 28, 1), -1), ((451, 44, 3), 1), ((729, 32, 3), -1)\}$.

Table II. An example of constructing an input for AdaBoost for the rare behaviours property.

noise freq.	*noise strength*	*noise type*	*covered rare tasks*	*testing time*	*cumulative coverage per time*	*classification result*
83	28	1	run 1: $\{a, c, d\}$	3	$(\lvert\{a, c, d\}\rvert = 3)/7$	-1
			run 2: $\{a, d\}$	2		
			run 3: $\{c, d\}$	2		
451	44	3	run 1: $\{a, d\}$	2	$(\lvert\{a, c, d, e\}\rvert = 4)/5$	1
			run 2: $\{c, e\}$	2		
			run 3: $\{d, e\}$	1		
729	32	3	run 1: $\{c, e\}$	3	$(\lvert\{c, e\}\rvert = 2)/8$	-1
			run 2: $\{c\}$	2		
			run 3: $\{e\}$	3		

3.3. Analysing Information Hidden in Classifiers

In order to be able to easily analyse information hidden in the classifiers generated by AdaBoost, we have decided to restrict the height of the basic decision trees used as weak classifiers to one. Moreover, our experiments showed us that increasing the height of the weak classifiers does not lead to significantly better classification results.

A decision tree of height one consists of a root labelled by a condition concerning the value of a single test or noise parameter and two leaves that correspond to the cases when the condition is or is not satisfied and that are labelled as leading to either positive or negative classification. AdaBoost provides us with a set of such trees, each with an assigned weight. For better understanding which parameters are important for testing, we convert this set of trees into a set of rules such that we get a single rule for each test or noise parameter that appears in at least one decision tree. The rules consist of a condition and a weight. In particular, the conditions have the form of a conjunction of interval constraints, and the weights are real numbers from the range between zero and one.

The rules are obtained as follows. First, decision trees with negative or zero weights are omitted because they correspond to weak classifiers with the weighted error greater or equals to 0.5. Next, the remaining decision trees are grouped according to the parameter about whose value they speak. To illustrate the above, assume that AdaBoost gives us, e.g., the ten decision trees with positive weights that are shown in Fig. 3. For each obtained group of the trees, a single rule is produced by taking the disjunction of the interval constraints associated with the grouped decision trees[‡]. Intuitively, taking the disjunction corresponds to the fact that each of the intervals was found to be relevant for the given testing goal. The weight of the rule is computed by summarising the weights of the trees from the concerned group and normalising the result by dividing it by the sum of the weights of all trees from all groups. This is, if all decision trees with positive weights created

[‡]In particular, the interval constraint of the tree is taken as is when the true branch of the decision tree leads to the $+1$ leaf. Otherwise, its complement must be taken.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

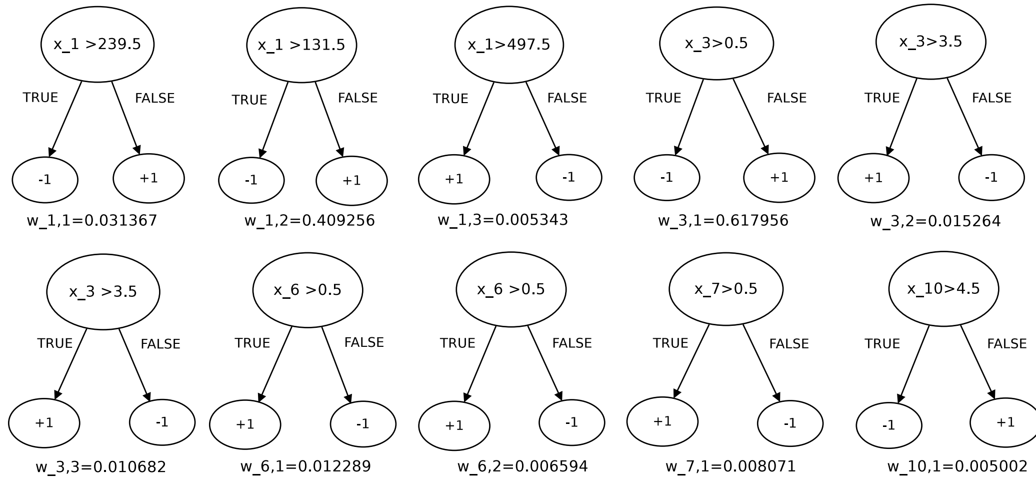

Figure 3. An example of several decision trees with conditions over parameters x_1, x_3, x_6, x_7 and x_{10} created by the AdaBoost algorithm.

by AdaBoost are $w_1, \ldots, w_m$, and the concerned group G consists of $n \leq m$ trees with weights $w_{i_1}, \ldots, w_{i_n}$ where $\forall 1 \leq j \leq n : \ 1 \leq i_j \leq m$, then the weight of the rule created from G will be computed as the fraction $\frac{\sum_{j=1}^{n} w_{i_j}}{\sum_{k=1}^{m} w_k}$.

In our example, we focus on the importance of the different parameters. We start with parameter x_1. For this parameter, when we take the disjunction of the interval constraints associated with the trees corresponding to x_1 (i.e., the first three trees in Fig. 3), we obtain the condition $x_1 \leq 239.5 \vee x_1 \leq 131.5 \vee x_1 > 497.5$, which can be simplified to $x_1 \leq 239.5 \vee x_1 > 497.5$. The weight of this rule is given by the sum of the three concerned trees divided by the sum of the weights of all the trees in the figure, which gives the (rounded) weight $w_{x_1} = 0.398$. If we process the other parameters in the same way, we get the following weights: $w_{x_3} = 0.574$, $w_{x_6} = 0.017$, $w_{x_7} = 0.007$, $w_{x_{10}} = 0.004$. Note that the weights of the parameters satisfy the constraint $\sum_i w_{x_i} = 1$. Clearly, parameters x_3 and x_1 appear to have the highest importance in the given setting; parameters x_6, x_7, and x_{10} appear to have at least some significance; while parameters such as x_2 are of no importance (since they did not even appear in any of the decision trees with positive weights).

From the rules obtained as described above, we can easily identify the parameters that most affect testing of the given program with the given testing goal. For that, we can simply take the parameters that are associated with the rules with the highest weights. In case we want to derive more general results—spanning over multiple testing goals and/or multiple tested programs, we can do that by looking for parameters (or values) that appear among the most influential ones among all (or most) of the considered test cases. Alternatively, one can also unite the training sets obtained for the different testing goals and/or programs under test, and then apply AdaBoost to the combined training set. In our example, the parameter which most affects the testing process is the parameter x_3 that has the highest weight.

Moreover, we can also see which concrete values of the different parameters are the most influential. In particular, assume that the condition of the rule derived for some parameter was created from a set $\mathcal{I} = \{I_1, ..., I_n\}$ of interval constraints where the decision trees that were associated with these intervals had weights $w_1, ..., w_n$. We identify all maximum subsets $\mathcal{J} = \{I_{i_1}, ..., I_{i_m}\} \subseteq \mathcal{I}$ of intervals with non-empty intersections (i.e., such that $\cap_{j \in \{1,...,m\}} I_{i_j} \neq \emptyset$) and assign each such set a weight $w_{\mathcal{J}}$ given by the sum of the weights of its elements, i.e., $w_{\mathcal{J}} = \sum_{j \in \{1,...,m\}} w_{i_j}$. Intuitively, the weights of all the decision trees whose interval constraints overlap contribute to the weight of their overlapping part. The most influential values of the given parameter are then given by the sets $\mathcal{J}$ with the highest weights—namely, by the union $\cup_{\mathcal{J}} \cap_{I \in \mathcal{J}} I$ of the intersections of the intervals I belonging to the subsets $\mathcal{J}$ with the highest weights $w_{\mathcal{J}}$.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

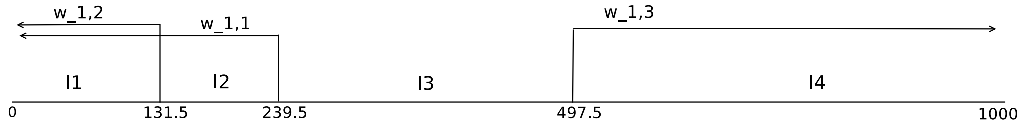

Figure 4. Division of values of parameter x_1 to intervals associated with the decision trees from Fig. 3.

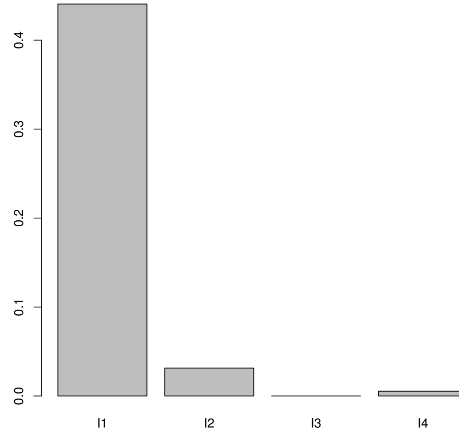

Figure 5. Histogram of weights of values of parameter x_1 derived from the decision trees in Fig. 3.

Thus, in the example, we have a look at the most influential values of some of the parameters from Fig. 3. In particular, we concentrate on parameter x_1. The parameter is associated with three decision trees and hence three interval constraints, which are illustrated in Fig. 4. From the illustration, we see that there are two maximum subsets of the interval constraints with non-empty intersections, namely, $\mathcal{J}_1 = \{x_1 \leq 239.5, x_1 \leq 131.5\}$ and $\mathcal{J}_2 = \{x_1 > 497.5\}$. The corresponding intersections are $x_1 \leq 131.5$ and $x_1 > 497.5$ with the (rounded) weights $w_{\mathcal{J}_1} = 0.441$ and $w_{\mathcal{J}_2} = 0.005$. Clearly, values of x_1 less than or equal to 131.5 are the most influential. In case one would like to have a finer look at the influence of the different values, one can take all subsets of the set of intervals associated with the given parameter, compute the corresponding intersections of the constraints and their weights (as in the case of the maximum subsets), and obtain a histogram of the weights—such as the one shown in Fig. 5 for the parameter x_1.

3.4. Using AdaBoost in Fully Automated Testing

We now present several approaches of using AdaBoost for fully automated noise-based testing. First, we describe two ways of combining AdaBoost with random generation of test and noise parameters. Second, we show how it can be combined with genetic algorithms for finding the most suitable values of test and noise parameters.

3.4.1. AdaBoost-Improved Random Testing In practice, noise-based testing is often used with randomly generated test and noise configurations. The simplest way of using AdaBoost to improve on this practice is the following. When performing repeated test runs of a given program to meet a given testing goal, one can run the program with randomly generated test and noise configurations, but use only those randomly generated configurations that get classified as suitable by an AdaBoost classifier derived for the given program and testing goal as described in Subsection 3.2. This idea, considered already in our preliminary work [1], is rather simple, but it can provide quite nice results as we illustrate through our experiments presented in Subsection 4.5.

While the above approach can provide useful results, we now propose yet another way of combining AdaBoost with random generation of test and noise configurations, which was not considered in [1]. This approach is motivated by our observation that, in many of the case studies that we conducted and which we report later on, some test and noise parameters were

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

significantly more important than others, even though the latter parameters were still influential. In such cases, however, the above proposed use of AdaBoost can include among useful test and noise configurations even some of those configurations where the less important parameters are set in a rather unsuitable way, which is tolerated due to the much higher weight of the more important parameters.

To improve on the above situation, we propose to build on the method for determining the most suitable values of each parameter, which is described at the end of Section 3.3. We then derive the test and noise configurations to be used by independently choosing the value of each of the parameters at random but from the most suitable range of its values only. For instance, assume that we have test and noise parameters x_1, x_2, and x_3, and the approach of Section 3.3 tells us that their most influential values are from intervals I_1, I_2, and I_3, respectively. Then, every time we need a test and noise configuration for a repeated test run, we generate it as a three-tuple whose first item is randomly chosen from the interval I_1, the second item is randomly chosen from I_2, and the third one is randomly chosen from I_3. Our experiments presented in Section 4.5 show that this approach can indeed provide significantly better results than the first mentioned approach.

3.4.2. Combination of Genetic Algorithms and AdaBoost Finally, we also propose a combination of using AdaBoost and the genetic algorithms that we considered for finding suitable test and noise configurations in our previous works [21, 23]. This approach is motivated as follows. Our previous works showed that genetic algorithms can achieve very good results in finding suitable test and noise configurations, especially when trying to increase the achieved concurrency coverage, but they need to execute a huge number of test runs to get these configurations. The reason of this is that the genetic algorithms start with random initial configurations in the first generations and slowly create configurations with better results in the next generations. Our idea is to accelerate this process by restricting the range of possible values of the different test and noise parameters in which the genetic algorithms will search. In particular, we restrict the range of the parameters to the most influential values found through AdaBoost and the approach described at the end of Section 3.3. Thus, essentially, we use AdaBoost to get coarse knowledge on the suitable values of the test and noise parameters, and then we refine this knowledge using genetic algorithms. Our experiments presented below confirm that this approach can often significantly outperform all the other mentioned approaches.

4. EXPERIMENTAL EVALUATION

In this section, we describe the experiments that we conducted to evaluate the approaches proposed above. We first provide a brief description of the benchmark programs that we used in our experiments. Next, we briefly characterize the accuracy and sensitivity of the AdaBoost classifiers that we were able to obtain for our case studies and testing goals. Subsequently, we analyse the knowledge hidden in the classifiers that we obtained, compare it with our experience obtained in other ways, and derive several new insights about the importance of the different test and noise parameters. Finally, we proceed to experiments illustrating that AdaBoost combined with genetic algorithms can also be quite successfully used in fully automated noise-based testing.

4.1. Case Studies

For our experimental evaluation, we used the following multi-threaded programs. The first five of them contain known concurrency-related errors, and so they are suitable for experiments with reproduction of known bugs for debugging purposes. The remaining programs do not contain any known errors, and so they are added to the first five case studies within our experiments targeted at increasing coverage of rare behaviours[§].

[§]The case studies we present in this paper do not include large programs due to we need to perform a rather large number of experiments with different test and noise settings: Already with the use cases we consider, the experiments presented

Airlines. The size of the test case is *0.3 kLOC*, *8 classes*. It is a small test case containing an *atomicity violation* error. It simulates an airline reservation system with three parameters X, Y, and Z: The system creates a flight whose capacity is Z (number of available seats). Then, X seller threads are executed, and they are periodically trying to get a seat on the flight. The parameter Y controls how many iterations of an idle loop are done (and hence how much time is spent) between two successive attempts to book a ticket.

Animator. The size of the test case is *1.5 kLOC*, *31 classes*. It is a program containing a *data race* and an *atomicity violation*. Animator is our short name for the XTANGO animation program [52] which is a general-purpose system for algorithm animation that allows programmers to create colorful, real-time, 2 & 1/2 dimensional, smooth animations of their algorithms and programs. The focus of the system is on ease of use—programmers using this system need not be graphics experts to develop their own animations.

Crawler. The size of this test case is *1.2 kLOC*, *19 classes*. The test case includes an *atomicity violation*. The program is taken from an IBM repository. It represents a skeleton of an IBM crawler product with a test environment simulating real usage of the system. Namely, the system creates a given number of threads waiting for a connection. If a connection is established, a worker thread serves it. Afterwards, when a given global time limit occurs, a shutdown sequence is initiated. This means that the working threads are not accepting new tasks, and, after finishing the current task, they die [32]. The bug present in the program manifests itself during the shutdown sequence but very rarely (roughly 15 times per 10,000 runs).

Elevator. The size of this test case is *1.2 kLOC*, *12 classes*. The program contains a *data race* and an *atomicity violation*. It implements a real-time discrete-event simulation. The application is used as an example in a course on concurrent programming. Elevators are modeled as individual threads that poll directives from a central control board. Communication through the control board is synchronized through locks. The configuration used for our experiments simulates four elevators [45]. This benchmark has one parameter which controls the number of threads used.

Rover. The size of the test case is *5.4 kLOC*, *82 classes*. Rover contains an *atomicity violation* and a *deadlock*. The K9 Rover from NASA Ames is an experimental platform for autonomous wheeled vehicles for exploration of a planetary surface such as Mars. The rover executive software prototype monitors executions of actions and performs responses and cleanup when the execution fails. In the configuration used in our experiments, eight threads are launched in the system [43]. This benchmark has one parameter which selects one of the available test scenarios.

Cache4j. The size of this test case is *1.7 kLOC*, *66 classes*. Cache4j does not contain any known error. It is an LRU (Least Recently Used) lock-based cache implementation. The implementation is based on two internal data structures, a tree and a hash-map. The tree manages the LRU while the hash-map holds the data. The implementation is based on a single global lock [28].

HEDC. The size of the test case is *12.7 kLOC*, *747 classes*. The program does not contain any known error. It represents an application kernel that implements a meta-crawler for searching multiple Internet archives in parallel. In our benchmark configuration, four principal threads issue random queries to two archives each. The individual queries are handled by a short random sleep interval of 0-200 ms; this ensures that the principal threads work out of sync. The application employs a library for concurrent programming by Doug Lea—in particular, the Pooled-Executor pattern. The workload and memory access pattern of this application kernel are typical for Internet server applications and similar to applications based on alternative mechanisms such as Java Servlets [45, 48].

below took approximately 5,592 core hours, i.e., 233 core days. However, works such as [7] show that noise-based testing can be successfully used even on programs with millions of lines of code and can find previously unknown errors in complex industrial code.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

MonteCarlo. The size of the test case is *1.4 kLOC*, *22 classes*. It does not contain any known error. MonteCarlo is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates multiple time series with the same mean and fluctuation as a series of historical data. This benchmark has one parameter which controls the number of threads used for the computation [51].

Raytracer. The size of the test case is *1.0 kLOC*, *22 classes*. It is without any known error. This benchmark measures the performance of a 3D ray tracer. The rendered scene contains 64 spheres, and it is rendered with a resolution of $N \times N$ pixels. The outermost loop (over rows of pixels) has been parallelised using a cyclic distribution for load balancing. This benchmark has one parameter controlling the number of threads used for the computation [42, 51].

SOR. The size of the test case is *7.2 kLOC*, *256 classes*. The program does not contain any known error. SOR (Successive Over-Relaxation over a 2D grid) synchronizes its threads using a barrier rather than locks. It implements an iterative method for solving discretized Laplace equations on a grid data structure. In particular, it performs multiple passes over a rectangular grid until the values in the grid change less than a certain threshold, or a pre-defined number of iterations has been reached. The new value of a grid point is computed using a stencil operation, which depends only on the previous value of the point itself and its four neighbors in the grid. The program has two parameters: the number of iterations and the number of threads [42, 45].

TSP. The size of this test case is *0.4 kLOC*, *8 classes*. It is without any known error. TSP (Travelling Salesman Problem) is a travelling salesman application which computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. The program is parallelized by distributing the search space over different processors. Because the algorithm performs pruning, the amount of computation needed for each subspace is not known in advance and varies between different parts of the search space. Therefore, dynamic load balancing between the processors is needed. This benchmark has two parameters: the number of threads and a given input file with a TSP instance [42, 46].

4.2. Considered Test and Noise Parameters

In Section 3.1, we said that our input set $\mathcal{X}$ for AdaBoost will consist of couples $(\overline{x}, y)$ where $\overline{x}$ is a vector recording the test and noise configuration used and $y \in \{1, -1\}$ is the result of evaluating the considered test property. In our experiments, we—in particular—consider vectors $\overline{x}$ of test and noise parameters consisting of 12 entries, i.e., $\overline{x} = (x_1, x_2, \ldots, x_{12})$.

In our vectors of test and noise parameters, the parameter $x_1 \in \{0, \ldots, 1000\}$ represents the *noise frequency*, the parameter $x_2 \in \{0, \ldots, 100\}$ is the *strength of noise*, the parameter $x_3 \in \{0, \ldots, 5\}$ selects one of the six available basic noise seeding heuristics. The parameters $x_4, x_5 \in \{0, 1\}$ disable or enable the additional noise seeding heuristics *haltOneThread* and *timeoutTamper*, respectively.

The parameter $x_6 \in \{0, 1, 2\}$ controls the way how the *sharedVarNoise* noise placement heuristic behaves—namely, whether it is disabled ($x_6 = 0$), it applies the *sharedVarNoise-one* strategy injecting the noise at accesses to one randomly selected shared variable ($x_6 = 1$), or it applies the *sharedVarNoise-all* strategy inserting the noise at accesses to all shared variables ($x_6 = 2$). The parameter $x_7 \in \{0, 1\}$ disables or enables the *nonVariableNoise* heuristic. The parameters $x_8, x_9 \in \{0, 1\}$ disable or enable the *coverage-based* noise placement heuristic and the related *coverage-based-frequency* heuristic, respectively.

Finally, we summarize the parameters used by the above test cases (on top of the parameters of the noise injection technology itself) and explain in more detail their encoding in our experiments. These parameters are encoded as the parameters $x_{10} \in \{1, \ldots, 10\}$ and $x_{11}, x_{12} \in \{1, \ldots, 100\}$ in the experiments. In particular, *Animator*, *Cache4j*, *HEDC*, and *Crawler* are not parametrised, and hence x_{10}, x_{11}, x_{12} are not used with them. In the *Airlines*, *Elevator*, *Montecarlo*, and *Raytracer* test cases, the x_{10} parameter controls the number of the threads used. In the Rover test case, the $x_{10} \in \{1, \ldots, 7\}$ parameter selects one of the available test scenarios. The *Sor* and *TSP* test cases have two test parameters. The x_{10} parameter is the number of iterations for *Sor* while it selects one

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

of the available test scenarios for *TSP*. The x_{11} parameter controls the number of the threads used for both of these test cases. The *Airlines* test case uses the x_{11} and x_{12} parameters where the x_{11} controls how many cycles the test does and the x_{12} parameter indicates the flight capacity.

The total number of noise configurations that one can obtain from the above can be computed by multiplying 1001 values of *noise frequency*, by 101 possible values of *noise strength*, the number of the basic noise seeding heuristics, which is six, by two to reflect whether *haltOneThread* is or is not used, two to reflect whether *timeoutTamper* is used, two to reflect whether the *nonVariableNoise* heuristic is used, two to reflect whether the *coverage-based* noise placement is used, two to reflect whether the *covergage-based-frequency* heuristic is used, and three to reflect the possible use case scenarios of the *sharedVarNoise* heuristic. This gives a rough estimate of about 58.2 million combinations of noise settings when we simplify the situation by ignoring the fact that some of the settings do not make sense when used together (for instance, enabling *coverage-based-frequency* heuristic has no effect when *coverage-based* heuristic is disabled). Of course, the state space of the test and noise settings then further grows with the possible values of parameters of the test cases and the testing environment [23].

4.3. Accuracy and Sensitivity of the Classifiers

We now present data about the accuracy and sensitivity of the AdaBoost classifiers that we derived for the above test cases. For the first five of them that contain known concurrency errors, we have considered both the testing goal of reproducing a known error as well as the goal of increasing coverage of rare behaviours. For the remaining test cases, we have considered the latter goal only.

In our experiments, we used the implementation of AdaBoost available in the GML AdaBoost Matlab Toolbox¶. We set it to use decision trees of height restricted to one and to use 10 boosting phases. When deriving the classifiers, we proceeded as described in Section 3.2. When deriving classifiers for the error manifestation property, we used 2000 random test and noise configurations. For the rare events property, due to a higher time consumption of the experiments, we used 200 random test and noise configurations. To obtain data allowing us to derive the accuracy and sensitivity of the derived classifiers, 100 different random divisions of the randomly generated configurations to training and validation sets were considered.

Tables III and IV summarise the average accuracy and sensitivity of the derived AdaBoost classifiers and their standard deviations. One can clearly see that both the average accuracy and sensitivity are quite high for the error reproduction test goal—with the average values being 0.8837 and 0.9586, respectively. For the testing goal of finding rare behaviours, both of the statistics have smaller values. However, the experiments presented in Section 4.5 show that the method works nicely even in their case. Moreover, the standard deviation is very low in all cases, which indicates that we always obtained results that provide meaningful information about our test runs.

4.4. Analysis of the Knowledge Hidden in the Obtained Classifiers

We now employ the approach described in Section 3.3 to interpret the knowledge hidden in the classifiers that we inferred for our test cases. From these classifiers, using the approach of Section 3.3, we derived the rules shown in Tables V and VI for the error manifestation property and the rare behaviours property, respectively. For each test case, the tables contain a row whose upper part contains the condition of the rule (in the form of an interval constraint), and the lower part contains the appropriate weight from the interval $(0, 1)$.

In order to interpret the obtained rules, we first focus on rules with the highest weights (corresponding to parameters with the biggest influence). Then we look at the parameters which are present in rules across the test cases (and hence seem to be important in general) and parameters that are specific for particular test cases only. Next, we pinpoint parameters that do not appear in any of the rules and therefore seem to be of a low relevance in general.

¶http://graphics.cs.msu.ru/en/science/research/machinelearning/AdaBoosttoolbox

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Table III. The average and standard deviation of the accuracy and sensitivity of the AdaBoost classifiers derived for the test cases containing known errors.

	Error reproduction				*Rare behaviours*			
	Accurancy		Sensitivity		Accurancy		Sensitivity	
CaseStudies	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Airlines	0.7488	0.0163	0.8917	0.0250	0.6601	0.0508	0.6880	0.0900
Animator	0.8353	0.0154	0.9489	0.0195	0.8503	0.0489	0.9006	0.0549
Crawler	0.9916	0.0026	0.9948	0.0018	0.7453	0.0437	0.7549	0.0740
Elevator	0.9568	0.0056	0.9965	0.0034	0.7161	0.0439	0.7327	0.0797
Rover	0.8859	0.0142	0.9611	0.0088	0.6108	0.0406	0.6330	0.0950
Average	0.8837	0.0108	0.9586	0.0117	0.7165	0.0456	0.7418	0.0787

Table IV. The average and standard deviation of the accuracy and sensitivity of the AdaBoost classifiers derived for the test cases without known errors.

	Rare behaviours			
	Accurancy		Sensitivity	
CaseStudies	Mean	Std	Mean	Std
Cache4j	0.8454	0.0671	0.8963	0.0907
HEDC	0.7819	0.0443	0.7797	0.0758
Montecarlo	0.6692	0.0607	0.6702	0.1230
Raytracer	0.6298	0.0713	0.6380	0.1114
Sor	0.7807	0.0457	0.8203	0.0797
TSP	0.6420	0.0674	0.6587	0.1179
Average	0.7248	0.0594	0.7439	0.0998

Table V. Inferred rules for the error manifestation property with the most influential intervals marked out.

Airlines

Rules	$x_1 \leq 275$	$\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3$	$x_6 \leq 1.5$	$2.5 < x_{10}$	$73.5 < x_{12}$
Weights	0.16	0.50	0.04	0.18	0.12

Animator

Rules	$705 < x_1$	$2.5 < x_3 \leq 3.5$	$x_6 \leq 0.5$
Weights	0.19	0.55	0.26

Crawler

Rules	$x_1 \leq 215$	$15 < x_2$	$1.5 < x_3 \leq 3.5$ or $\mathbf{4.5 < x_3}$	$0.5 < x_4$	$x_5 \leq 0.5$	$x_6 \leq 1.5$
Weights	0.32	0.1	0.38	0.05	0.08	0.07

Elevator

Rules	$x_1 \leq 5$	$\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$	$x_7 \leq 0.5$	$8.5 < x_{10}$
Weights	0.93	0.04	0.01	0.02

Rover

Rules	$515 < x_1$	$2.5 < x_3 \leq 3.5$	$0.5 < x_4$	$x_6 \leq 0.5$
Weights	0.21	0.48	0.08	0.23

As for the error manifestation property (i.e., Table V), the most influential parameters are x_3 in four of the test cases and x_1 in the *Crawler* test case. This indicates that the selection of a suitable noise type (x_3) or noise frequency (x_1) is the most important decision to be done when testing these programs with the aim of reproducing the errors present in them. Another important parameter is

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Table VI. Rules inferred for the rare behaviours property.

Airlines

Rules	$\mathbf{x_1 \leq 295}$ or $745 < x_1 \leq 925$	$x_2 \leq 35$	$0.5 < x_5$	$61.5 < x_{12} \leq 91.5$
Weights	0.52	0.06	0.1	0.32

Animator

Rules	$\mathbf{0.5 < x_3 \leq 3.5}$ or $4.5 < x_3$	$0.5 < x_6 \leq 1.5$
Weights	0.80	0.20

Crawler

Rules	$\mathbf{0.5 < x_3 \leq 3.5}$ or $4.5 < x_3$	$0.5 < x_4$	$0.5 < x_5$	$0.5 < x_6 \leq 1.5$
Weights	0.46	0.08	0.20	0.26

Elevator

Rules	$0.5 < x_3 \leq 3.5$ or $\mathbf{4.5 < x_3}$	$0.5 < x_4$	$0.5 < x_5$	$1.5 < x_6$	$\mathbf{1.5 < x_{10} \leq 4.5}$ or $7.5 < x_{10}$
Weights	0.22	0.05	0.20	0.06	0.47

Rover

Rules	$\mathbf{2.5 < x_3 \leq 3.5}$ or $4.5 < x_3$	$x_4 \leq 0.5$	$x_6 \leq 0.5$	$0.5 < x_7$
Weights	0.46	0.26	0.16	0.12

Cache4j

Rules	$\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$	$x_5 \leq 0.5$	$1.5 < x_6$	$x_9 \leq 0.5$
Weights	0.92	0.02	0.05	0.01

HEDC

Rules	$x_1 \leq 279$	$49.5 < x_2$	$\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$	$1.5 < x_6$
Weights	0.03	0.02	0.89	0.06

Montecarlo

Rules	$x_1 \leq 548.5$	$\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3$	$x_5 \leq 0.5$	$0.5 < x_6$	$x_9 \leq 0.5$	$3.5 < x_{10} \leq 5.5$
Weights	0.09	0.30	0.05	0.18	0.09	0.29

Raytracer

Rules	$20.5 < x_2 \leq 53.5$ or $\mathbf{75.5 < x_2}$	$0.5 < x_5$	$x_6 \leq 0.5$	$0.5 < x_7$	$x_{10} \leq 1.5$ or $\mathbf{4.5 < x_{10}}$
Weights	0.29	0.09	0.15	0.06	0.41

Sor

Rules	$x_1 \leq 144.5$	$x_3 \leq 1.5$ or $\mathbf{3.5 < x_3}$	$0.5 < x_6$	$x_7 \leq 0.5$	$x_{10} < 13$
Weights	0.26	0.32	0.07	0.07	0.28

TSP – part1

Rules	$x_1 \leq 691$	$x_2 \leq 26$	$\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$	$x_5 \leq 0.5$
Weights	0.07	0.11	0.48	0.06

TSP – part2

Rules	$0.5 < x_6$	$0.5 < x_8$	$x_9 \leq 0.5$	$x_{10} \leq 18.5$
Weights	0.06	0.06	0.07	0.09

x_6 controlling the use of the *sharedVarNoise* heuristic. Moreover, the parameters x_1, x_3, and x_6 are considered important in all of the rules, which suggests that, for reproducing the considered kind of errors, they are of a general importance.

In two cases, namely, *Crawler* and *Rover*, the *haltOneThread* heuristic (x_4) turns out to be relevant. In these test cases, the *haltOneThread* heuristic should be enabled in order to detect an error. This behaviour fits into our previous results [35] in which we show that, in some cases, this unique heuristic (the only heuristic which allows one to exercise thread interleavings which are normally far away from each other) considerably contributes to the detection of an error. Finally, the presence of the x_{10} and x_{12} parameters in the rules derived for the *Airlines* test case indicates that the number of threads (x_{10}) and the number of cycles executed during the test (x_{12}) pays an important role in the noise-based testing of this particular test case. The x_{10} parameter (i.e., the number of threads) turns out to be important for the *Elevator* test case too, indicating that the number of threads is of a more general importance.

Finally, we can see that the x_8, x_9, and x_{11} parameters are not present in any of the derived rules. This indicates that the *coverage-based* noise placement heuristics are of a low importance in general, and the x_{11} parameter specific for *Airlines* is not really important for finding errors in this test case.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Next, for the case of classifying according to the rare behaviours property, the obtained rules are shown in Table VI. The highest weights can again be found in rules based on the x_3 parameter (*Animator*, *Crawler*, *Rover*, *Cache4j*, *HEDC*, *Montecarlo*, *Sor*, *TSP*) and on the x_1 parameter (*Airlines*). However, in the case of *Elevator* and *Raytracer*, the most contributing parameter is now the number of threads used by the test (x_{10}). Moreover, the x_{10} parameter is also important in the *Montecarlo*, *Sor*, and *TSP* test cases. This suggests that choosing the right number of threads is quite important to maximize the chances to spot rare behaviours, and that it is not necessarily the case that the higher number of threads is used the better. Further, the generated sets of rules often contain the x_3 parameter controlling the type of noise (all test cases except for *Airlines* and *Raytracer*) and the x_6 parameter which controls the *sharedVarNoise* heuristic. These parameters thus appear to be of a general importance for the rare behaviours property.

The parameter x_{12}, i.e., the number of test cycles, does again turn out to be important in the *Airlines* test case. Finally, the x_8 parameter is shown only in one test case (*TSP*), x_9 shows up in the rules generated for two test cases (*Cache4j* and *TSP*), and the x_{11} parameter does not show up in any of the rules, and hence seem to be of a low importance in general for finding rare behaviours (which is the same as for reproduction of known errors).

Overall, the obtained results confirmed some of the facts we discovered during our previous experimentation such as that different goals and different test cases may require a different setting of noise heuristics [35, 23, 21] and that the *haltOneThread* noise injection heuristics (x_4) provides in some cases a dramatic increase in the probability of spotting an error [35]. More importantly, the analysis revealed (in an automated way) some new knowledge as well. Mainly, the type of noise (x_3) and the setting of the *sharedVarNoise* heuristic (x_6) as well as the frequency of noise (x_1) are often the most important parameters (although the importance of x_1 seems to be a bit lower). Further, it appears to be important to suitably adjust the number of threads (x_{10}) whenever that is possible.

4.5. Fully Automated Noise-based Testing with AdaBoost

We now present experimental results showing usefulness of the ways of applying AdaBoost in fully automated noise-based testing that we proposed in Section 3.4. We consider both the combination of AdaBoost and random noise injection as well as the combination of AdaBoost and genetic algorithms. We start by considering the case of repeated reproduction of a known concurrency error and then proceed to the case of coverage of rare tasks.

4.5.1. Repeated Error Manifestation Within our experiments aimed at repeated reproduction of known concurrency-related errors, we compare noise-based testing under test and noise configurations generated in the following ways:

- Purely random generation (referred to as *Random* below).
- Generation based on single-objective and multiple-objective genetic algorithms proposed in our earlier work and briefly described in Section 2.4 (denoted as *SOGA* and *MOGA* below).
- Random generation filtered through the classic AdaBoost approach as described in the first part of Section 3.4.1 (referred to as *AdaBoost* in what follows).
- Random generation restricted to the AdaBoost-recognised most influential values of parameters described in the second half of Section 3.4.1 (denoted as *AdaBoost2* below).
- Generation based on the single-objective and multiple-objective genetic algorithms restricted to the AdaBoost-recognised most influential values of parameters as proposed in Section 3.4.2 (referred to as *SOGA2* and *MOGA2* below).

We run 5000 executions in the learning phase of those approaches that need some training. To compare capabilities of the obtained test and noise configurations in repeatadly finding the known errors, we then run 20 executions for 20 best configurations obtained through each of the approaches (apart from the random approach where we simply run 400 executions).

For experiments with the genetic algorithms, one has to choose the fitness function to be used. In particular, for the *SOGA* and *SOGA2* experiments, based on the experience we gained in our

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

previous work, we have chosen the following fitness function:

$$fitness = \frac{Error}{Error_{max}} * 10 + \frac{Warning}{Warning_{max}} + \frac{GoldiLockSC^*}{GoldiLockSC^*_{max}} + \frac{time_{max} - time}{time_{max}}$$

Here, the *GoldiLockSC** coverage metric is used since it has good properties for measuring general coverage of concurrency behaviour. The value $GoldiLockSC^*$ used in the fitness function gives the cumulative number of tasks covered in a series of five test runs performed with the given test and noise parameter values while $GoldiLockSC^*_{max}$ gives the maximal cumulative number of covered tasks across all so far performed series of test runs. However, since we want the fitness function to steer the search towards error discovery, we add to the fitness function information about the number of detected errors and error warnings. In particular, $Error$ gives the number of error manifestations detected in the given series of five runs by looking for unhandled exceptions, and $Error_{max}$ gives the maximal number of error manifestations so far seen in some series of five test runs. $Warning$ gives the number of warnings detected in the given series of five test runs through the $Avio$ checker [34] which detects atomicity violations over one variable. This metric has been chosen because atomicity violations are present in all the case studies considered in this experiment. Again, $Warning_{max}$ gives the maximum $Avio$ coverage obtained in the so far performed series of test runs. Finally, as we want to reflect the time needed for the test runs, we add it into the fitness function in such a way that lower amounts of time needed for the test runs are preferred[‖].

For the *MOGA* and *MOGA2* experiments, we have let the multi-objective genetic algorithm work with the same objectives as those summarized in the fitness function of the *SOGA* and *SOGA2* approaches, i.e., the number of detected error manifestations, the $Avio$ coverage, the *GoldiLockSC** coverage, and the needed testing time. In all our experiments with the genetic algorithms, we used the following settings: the probability of mutation was set to 0.5, the number of individuals in one population was 20, and each individual was evaluated by using the cumulative value from five executions of one configuration. We used the two-point crossover and the tournament selection operator (which provided us with the best results in our previous work [22]). For each case study, we repeat each experiment ten times.

Table VII compares results obtained using the above described approaches. In particular, the table presents numbers and percentages of the executions that managed to find an error in those of our benchmark programs that contain a known error. As we can see, the single-objective genetic algorithm restricted to the AdaBoost-selected most influential parameter values (i.e., *SOGA2*) has achieved the best results on average. However, random generation of test and noise parameter values restricted to the AdaBoost-selected most influential parameter values (*AdaBoost2*) and the combination of the multi-objective genetic algorithm and AdaBoost (*MOGA2*) have also achieved very good results.

Table VII. An experimental comparison of various fully automated approaches to noise-based testing in the context of reproducing a known error. The best results are highlighted in bold.

CaseStudies	*Random* error/ %	*SOGA* error / %	*MOGA* error/ %	*AdaBoost* error/ %	*AdaBoost2* error/ %	*SOGA2* error/ %	*MOGA2* error/ %
Airlines	132.93/33.23	313.25/78.31	272.25/68.06	323.50/80.88	351.80/87.95	**371.80/92.95**	332.7/83.13
Animator	106.75/26.69	220.20/55.05	131.00/32.75	144.80/36.20	252.40/63.10	**350.30/87.58**	241.25/60.31
Crawler	0.00/0.00	0.50/0.13	0.50/0.13	0.80/0.20	1.00/0.25	**2.40/0.60**	0.80/0.20
Elevator	59.25/14.81	**133.25/33.31**	116.75/29.19	80.40/20.10	36.60/9.15	105.00/26.25	86.80/21.70
Rover	17.00/4.25	143.00/35.75	88.25/22.06	57.40/14.35	48.4/12.65	**324.80/81.20**	203.30/50.83
Average	/15.80	/40.51	/30.44	/19.11	/34.62	**/57.72**	/43.24
ASD	/6.01	/5.50	/7.91	/7.44	/4.91	/4.89	**/2.58**

[‖]Here, one could be tempted to divide the fitness values by the time needed. We do not use this approach since our previous experience [22] showed that this often leads to significant degeneration of the search (producing configurations that produce very low coverage in extremely short time).

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

It must be noted that 14 generations were used for the *SOGA* and *MOGA* experiments, and 7 generations were used for the *SOGA2* and *MOGA2* experiments, which are very small numbers only. The reason for using such small numbers of generations is that we wanted to compare the different approaches while giving them the same time for the learning phase. The *MOGA2* approach had the lowest standard deviation on average. This means that the *MOGA2* approach gives good results with a high probability.

4.5.2. Coverage of Rare Concurrent Behaviours In the second part of our experiments, we concentrate on increasing coverage of rare concurrent behaviours. Compared with the experiments of the previous section, we consider all of our benchmark programs since we do not need them to contain an error. For the *SOGA* and *SOGA2* approaches, we use the following simplified fitness function:

$$fitness = \frac{GoldiLockSC^{*}}{GoldiLockSC^{*}_{max}} + \frac{time_{max} - time}{time_{max}}$$

From the fitness function, we have left out information about errors and warnings since we now do not focus on occurrences of any known errors. The *MOGA* and *MOGA2* approaches are based on the same objectives as *SOGA* and *SOGA2*, i.e., $time$ and $GoldiLockSC^{*}$. As in the experiments of the previous section, the probability of mutation was set to 0.5, and each individual was evaluated using cumulative coverage obtained in five runs. Each generation had 20 individuals.

For the random approach, we executed 1000 test runs with randomly generated test and noise configurations. For the other approaches, we used the same number of test runs, which we divided into 500 runs to train the approaches and the remaining 500 runs to execute the test cases with the configurations obtained from the training phase. When training the AdaBoost-based approaches, we took as positive (i.e., suitable for testing) 50 configurations with the highest results of cumulative coverage obtained from five runs and the other configurations as negative. For the approaches based purely on genetic algorithms, i.e., *SOGA* and *MOGA*, we used five generations in the training phase. For the combination of AdaBoost and genetic algorithms, i.e., *SOGA2* and *MOGA2*, we used 250 runs for training AdaBoost and three generations for the subsequent training of the genetic algorithms. For each case study, we repeated each experiment ten times.

Table VIII. A comparison of average cumulative numbers of rare tasks over the time needed to cover them.

CaseStudies	Rand. rareTasks/ %	SOGA rareTasks/ %	MOGA rareTasks/ %	AdaBoost rareTasks/ %	AdaBoost2 rareTasks/ %	SOGA2 rareTasks/ %	MOGA2 rareTasks/ %
Airlines	0.6566/ 41.4	1.2950/ 81.6	1.5462/ 97.4	0.4768/ 30.0	0.9298/ 58.6	**1.5876/ 100.0**	1.1216/ 70.6
Animator	7.0193/ 4.6	145.8694/ 95.3	**153.0821/ 100.0**	87.3576/ 57.1	136.5519/ 89.2	114.9578/ 75.1	110.4470/ 72.1
Cache4j	0.0165/ 38.9	0.0167/ 39.4	0.0413/ 97.4	0.0292/ 68.9	0.0194/ 45.8	0.0389/ 91.7	**0.0424/ 100.0**
Crawler	3.0415/ 51.1	4.7546/ 79.9	3.1230/ 52.5	3.6581/ 61.5	5.8669/ 98.6	4.1439/ 69.6	**5.9502/ 100.0**
Elevator	9.0015/ 48.1	13.5446/ 72.4	16.9801/ 90.8	17.4073/ 93.1	**18.7019/ 100.0**	14.9516/ 79.9	17.1540/ 91.7
HEDC	0.3605/ 22.1	0.9909/ 60.7	0.7595/ 46.5	0.9754/ 59.7	1.1568/ 70.8	1.3836/ 84.7	**1.6334/ 100.0**
Montecarlo	0.1469/ 59.9	0.2158/ 88.0	**0.2453/ 100.0**	0.1482/ 60.4	0.1780/ 72.5	0.1664/ 67.8	0.1823/ 74.3
Raytracer	0.0009/ 7.7	0.0003/ 2.6	0.0003/ 2.6	0.0006/ 5.1	0.0052/ 44.4	**0.0117/ 100.0**	0.0104/ 88.9
Rover	1.1532/ 42.1	1.7713/ 64.6	1.5623/ 57.0	1.4008/ 51.1	1.3018/ 47.5	1.9877/ 72.5	**2.7411/ 100.0**
Sor	0.0497/ 25.4	0.0742/ 37.9	0.0860/ 44.0	0.1088/ 55.6	0.1154/ 59.0	0.1855/ 94.8	**0.1956/ 100.0**
TSP	0.0381/ 36.9	0.0659/ 63.9	0.0971/ 94.1	0.0520/ 50.4	0.0642/ 62.2	0.0867/ 84.0	**0.1032/ 100.0**
Average	/ 34.4	/ 62.4	/ 71.1	/ 55.6	/ 67.7	/ 83.6	**/ 90.7**
ASD	/ 17.6	/ 26.9	/ 32.5	/ 20.7	/ 20.5	**/ 11.8**	/ 12.4

In Table VIII, we present results of the above experiments (which took in total approximately 6,939 core hours, i.e., 289 core days). In particular, the entries of the table contain—for the different programs and different approaches—the obtained coverage of rare tasks over the time needed to obtain the coverage. We divide the obtained coverage by the needed time in order to better see which of the approaches is better to quickly obtain a high coverage of rare tasks. Moreover, the obtained coverage over the testing time is followed by its interpretation in per cent. Namely, the approach with one hundred per cent is the winning one, and, for the others, the percentage shows how far they are from the winning approach in terms of the achieved coverage over time. As we can see, the combinations of AdaBoost with the genetic approaches (i.e., *MOGA2* and *SOGA2*) have the best results on average, and they are also more stable than the other methods.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

5. RELATED WORK

Below, we divide related works to two areas: First, we discuss works representing alternative approaches to noise-based testing of concurrent programs. Second, we discuss works where data mining is applied in testing. None of them, however, is going in the same direction as this paper.

5.1. Testing and Analysis of Concurrent Programs

As we have already indicated above, finding errors in concurrent programs is a particularly difficult problem due to the fact that a multi-threaded program can generate many executions differing in the interleavings of its threads, out of which, however, usually a very small fraction leads to errors only. Moreover, naïvely repeated test executions are likely to repeatedly execute similar interleavings and hence miss the errors [3, 8]. One of the ways to cope with this problem is the approach of noise-based testing [8, 13] whose further improvement we propose in this paper.

Among the main alternatives to noise-based testing, we first mention the so-called *systematic testing* [20, 3, 58, 38, 25, 24] that controls the scheduling of threads and systematically enumerates their different interleavings. Unlike noise-based testing, systematic testing provides better guarantees that a concurrency-related error will be found if present, and it can avoid re-execution of the same schedules. On the other hand, despite many heuristic optimizations that have been proposed, due to a need to systematically enumerate different schedules, systematic testing is still more heavy-weight than noise-based testing. Moreover, systematic testing can have problems with programs containing sources of non-determinism such as user input, external client requests, etc.

Coverage-driven testing as proposed in [61] and implemented in the Maple tool attempts to influence the scheduling such that the obtained coverage of several important synchronization idioms (called iRoots) is maximized. These idioms capture several important memory access patterns that are shown to be often related with error occurrences. Maple uses several heuristics to likely increase the coverage of iRoots. The technique provides lower guarantees of finding an error than systematic testing, but it is more scalable. The approach of Maple does not support some kinds of bugs (e.g., value-dependent bugs or some forms of deadlocks). Interestingly, multiple of the heuristics it uses are based on randomization. Maple can thus be viewed as being in between of systematic testing and noise-based testing (note that some of our noise placement heuristics are based on maximizing coverage too). An interesting question for future work is thus whether an approach for finding suitable values of noise parameters, such as the one we propose in this paper, could be combined with the heuristics used in Maple too.

Another approach to increase efficiency of discovering concurrency-related errors is that of *extrapolating dynamic analysis*. Such an analysis is trying to discover symptoms of concurrency-related errors (e.g., wrong locking order, use of different locks to guard the same variable, etc.), and it can warn about a possible error even if it is not witnessed in a given run. Many extrapolating dynamic analyses have been proposed, including, e.g., analyses targeting *data races* [49, 10, 16], *deadlocks* [4], *atomicity violations* [14, 39, 15], etc. Note that extrapolating dynamic analyses can be combined with noise injection as these approaches are to a large degree complementary.

Yet another possibility how to look for concurrency-related errors is that of using techniques of *static analysis*, often with deep *formal roots*. There exist many static analysis techniques, including, e.g., software model checking, data flow analysis, abstract interpretation, or searching for error-patterns. Some of these techniques have problematic scalability (e.g., model checking), some suffer from false alarms, and/or have problems to capture all forms of concurrency-related errors (e.g., error patterns). On the other hand, some of them can provide very high guarantees of finding errors or be very fast. These techniques can be viewed as complementary to testing and dynamic analysis. Their deeper discussion is beyond the scope of this paper—for an overview, see, e.g., [36].

Finally, various combinations of the above approaches have been studied in the literature. In *active testing*, which is considered, e.g., in [50, 44, 27], some bug detector based on static analysis or extrapolating dynamic analysis is used to detect possible concurrency errors and then some form of noise-based testing, directed by information from the first phase, is used to check whether the

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

detected error is real. In [12], an approach combining noise-based testing and extrapolating dynamic analysis in the first phase was combined with bounded software model checking along the (partially) recorded trace from the first phase and in its neighbourhood.

Note that it is not a goal of this paper to argue that any single of the above mentioned approaches is clearly better than the rest. Each of the approaches has advantages and disadvantages, and hence, in practice, the best way is to use several of these approaches and/or some of their combinations such as those mentioned above. The goal of this paper is to significantly advance one of the approaches, namely, noise-based testing. In particular, we proposed a new solution to the problem of finding suitable settings of the many parameters that control noise-based testing. For that, previous works used random noise setting [8] or applied genetic algorithms [21, 23]. Our experiments show that our approach based on data mining can overcome both of the previous approaches in some cases (in general, there is no single best approach). Moreover, data mining and genetic algorithms can be combined, yielding an approach which produced the best results in most of our experiments.

5.2. Related Works from the Area of Data Mining

Most of the existing works on obtaining new knowledge from multiple test runs of concurrent programs focus on gathering debugging information that helps to find the root cause of a failure [9, 54]. In [54], a machine learning algorithm is used to infer points in the execution such that the error manifestation probability is increased when noise is injected into them. It is then shown that such places are often involved in the erroneous behaviour of the program. Another approach [9] uses a technique similar to data mining, more precisely, a feature selection algorithm, to infer a reduced call graph representation of the system under test, which is then used to discover anomalies in the behaviour of the system under test within erroneous executions.

None of the works above, and, to the best of our knowledge, no other existing work has applied data mining for finding values of test and noise parameters suitable for noise-based testing of concurrent programs. The only exception is our preliminary work [1], on which this paper is based. However, compared with [1], the present paper provides (1) a significantly improved presentation of the idea, (2) it proposes a new way of exploiting the results from data mining for fully automated noise-based testing, (3) a combination of data mining with genetic approaches, and (4) it provides a significantly improved experimental evaluation of the approach.

Naturally, there is much richer literature and tool support for data mining test results without a particular emphasis on concurrent programs. The existing works study different aspects of testing, including identification of test suite weaknesses [2], optimisation of the test suite [60], or error localization [11]. Adler et al [2] show that a substring hole analysis is used to identify sets of untested behaviours using coverage data obtained from testing of large programs. Contrary to the analysis of what is missing in coverage data and what should be covered by improving the test suite, other works focus on what is redundant. Yoo et al [60] show that a clustering data mining technique is used to identify tests which exercise similar behaviours of the program. The obtained results are then used to prioritise the available tests. Erman et al [11] show that clustering of similar test case failures is used to help the analyst to identify the underlying causes of the failures and thus to make it easier to deal with huge numbers of test results obtained due to test automation.

Further, data mining techniques are, of course, used in many other areas of software engineering than testing. An exhaustive list of such applications is beyond the scope of this paper, and so we mention just a few examples. For instance, in the recent result [59], machine learning is used to extract design knowledge allowing one to improve assignment of responsibilities to classes, which is a vital task in object-oriented design. Cheung et al [5] show that clustering is used to detect smells in spreadsheet cells, which are susceptible to contain errors. Rubinič et al [47] show that machine learning is applied for software defect prediction, using ensembles of genetic classifiers to deal with imbalanced data sets. Luo et al [41], data mining is used for automatically identifying code changes that may potentially be responsible for a performance regression. Next, Kreutzer et al [30] show that a clustering algorithm is used in combination with two syntactical similarity metrics to automatically detect groups of similar code changes. Liang et al [37] focus on improving the precision of code mining with the aim of error detection by carefully preprocessing the source

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

code. Tantithamthavorn et al [53] show that an automated parameter optimization technique has been applied to obtain prediction models in the form of classifiers trained to identify defect-prone software modules. Wang et al [55] show that machine learning is used to automatically learn a semantic representation of programs from their source code.

6. CONCLUSIONS AND FUTURE WORK

In the paper, we have proposed a novel application of data mining in the context of noise-based testing of concurrent programs. In particular, we have employed data mining based on binary classification, decision trees, and the AdaBoost machine learning algorithm. We have shown how to use these technologies for finding a suitable set up of noise injection, i.e., selecting suitable noise injection heuristics out of the many known ones and finding suitable values of their various parameters, with the aim of maximizing chances of meeting a given testing goal. We have illustrated our approach on two concrete testing goals in the context of concurrent programs, namely, reproduction of known errors for debugging purposes and covering rare behaviours, which are more likely to contain so far unknown bugs than common behaviours. We have shown how data mining can be used to gain more insight into the suitability of the different noise heuristics and their parameters, allowing testers to choose the right ones for the given context, as well as how to use data mining to improve fully automated noise-based testing. For the latter case, we have combined our approach both with noise-based testing on a random basis as well as with genetically optimized noise-based testing. For all the proposed approaches, we have illustrated on a number of case studies that they can indeed improve the process of noise-based testing of concurrent programs.

In the future, we would like to apply in the context of testing of concurrent programs other approaches to data mining than AdaBoost and binary classification that we considered in this paper. This could include approaches such as outliers detection, clustering, or association rules mining. We would also like to look for other applications of data mining than setting up noise injection in a suitable way. For example, many of the concurrency coverage metrics based on dynamic detectors contain a lot of information on the behaviour of the tested programs, and when mined, this information could be used for debugging purposes. One could also think of generalising the various existing works devoted to detection of untested behaviour or to eliminating tests of similar behaviour of sequential programs (cf. Section 5) for the case of concurrent programs.

Acknowledgement. This work was supported by the Czech Science Foundation project 14-11384S, the EU/Czech IT4Innovations Excellence in Science project LQ1602, and the internal project of FIT BUT FIT-S-14-2486.

REFERENCES

1. R. Avros, V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, S. Ur, T. Vojnar, and Z. Volkovich. Boosted Decision Trees for Behaviour Mining of Concurrent Programs. In *Proc. of MEMICS'14*, pages 15–27. NOVPRESS, 2014.
2. Y. Adler, N. Behar, O. Raz, O. Shehory, N. Steindler, S. Ur, and A. Zlotnick. Code Coverage Analysis in Practice for Large Systems. In *Proc. of ICSE'11*, pages 736–745. ACM, 2011.
3. T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer. Preemption Sealing for Efficient Concurrency Testing. In *Proc. of TACAS'10*, volume 6015 of LNCS, pages 420–434. Springer-Velrlag, 2010.
4. S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *In Proc. of PADTAD'05*, pages 208–223. Springer-Verlag, 2005.
5. S.-Ch. Cheung, W. Chen, Y. Liu, and Ch. Xu. CUSTODES: Automatic Spreadsheet Cell Clustering and Smell Detection using Strong and Weak Features, In *Proc. of ICSE'16*, IEEE/ACM, 2016, Austin, TX, USA.
6. K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley paperback series. Wiley, 2009.
7. R. Dias, C. Ferreira, J. Fiedor, J. Lourenço, A. Smrčka, D.G. Sousa, and T. Vojnar. Verifying Concurrent Programs using Contracts. In *Proc. of ICST'17*. IEEE CS, 2017.
8. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499. Wiley, 2003.
9. F. Eichinger, V. Pankratius, P. W. L. Große, and K. Böhm. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. In *Proc. of TAIC PART'10*, volume 6303 of LNCS, pages 56–71. Springer-Verlag, 2010.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

10. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, pages 245–255. ACM, 2007.
11. N. Erman, V. Tufvesson, M. Borg, A. Ardö, and P. Runeson. *Navigating Information Overload Caused by Automated Testing – A Clustering Approach in Multi-Branch Development*, ICST'15, IEEE, 2015.
12. J. Fiedor, V. Hruba, B. Krena, T. Vojnar. DA-BMC: A Tool Chain Combining Dynamic Analysis and Bounded Model Checking, In *Proc. of RV'11*, LNCS 7186. Springer-Verlag, 2012.
13. J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-based Testing of Concurrent Software. In Software Testing, Verification and Reliability, 25(3):272–309. Elsevier, 2015.
14. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, Jan. 2004.
15. C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 43(6):293–303, 2008.
16. C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proc. of PLDI'09*, pages 121–133, New York, NY, USA, 2009. ACM.
17. Y. Freund. Boosting a weak learning algorithm by majority. In *Information and Computation*, 121(2):256–285,1995.
18. Y. Freund and R. E. Schapire. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting*. Journal of Computer and System Sciences, 55(1): 119–139, Academic Press, Inc., Orlando, FL, USA, August 1997.
19. Y. Freund and R. E. Schapire. A Short Introduction to Boosting. In *In Proc. of IJCAI'99*, pages 1401–1406. Morgan Kaufmann, 1999.
20. S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proc. of ISSTA'12*. ACM, 2012.
21. V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*, volume 7515 of LNCS, pages 152–167. Springer-Velrlag, 2012.
22. V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, and T. Vojnar. *Testing Concurrent Programs Using Multi-objective Genetic Algorithms*, FIT-TR-2013-05, Brno, 2013.
23. V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, and T. Vojnar. Multi-objective Genetic Optimization for Noise-based Testing of Concurrent Software. In *Proc. of SSBSE'14*, volume 8636 of LNCS, pages 107–122. Springer-Verlag, 2014.
24. G.-H. Hwang, H.-Y. Lin, S.-Y. Lin, Ch.-S. Lin. Statement-Coverage Testing for Concurrent Programs in Reachability Testing, In *Journal of Information Science and Engineering*, Volume 30, number 4, pages 1095-1113, 2014.
25. G.-H. Hwang, Ch.-S. Lin, T.-S. Lee, Ch. Wu-Lee. A model-free and state-cover testing scheme for semaphore-based and shared-memory concurrent programs, In *Software Testing, Verification and Reliability*, Volume 24, Issue 8, pages 706737, December 2014.
26. H. Ishibuchi and Y. Shibata. *A Similarity-based Mating Scheme for Evolutionary Multiobjective Optimization*, Lecture Notes in Computer Science, 2003, pages 1065–1076, Springer
27. P. Joshi, M. Naik, C.-S. Park, K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs, In *Proc. of CAV'09*, LNVS 5643. Springer-Verlag, 2009.
28. G. Korl and N. Shavit and P. Felber. *Noninvasive concurrency with Java STM*.
29. S. B. Kotsiantis. *Supervised Machine Learning: A Review of Classification Techniques*. Informatica 31:249 – 268, 2007.
30. P. Kreutzer, G. Dotzler, M. Ring, b. M. Eskofier, and M. Philippsen. Automatic Clustering of Code Changes. In *Proc. of MSR'16*, IEEE/ACM, 2016, Austin, TX, USA.
31. W. J. Krzanowski and D. J. Hand. *ROC Curves for Continuous Data*, Monographs on Statistics and Applied Probability 111 2009, Chapman & Hall/CRC.
32. B. Křena, Z. Letko, Y. Nir-Buchbinder, R. Tzoref-Brill, S. Ur, and T. Vojnar. A Concurrency Testing Tool and Its Plug-ins for Dynamic Analysis and Runtime Healing. In *Proc. of RV'09*, LNCS 5779, Springer-Verlag, 2009.
33. B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of PADTAD'10*, ACM, 2010.
34. B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of RV'11*, volume 7186 of LNCS, pages 177–192. Springer-Velrlag, 2012.
35. B. Křena, Z. Letko, and T. Vojnar. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'11*, volume 7119 of LNCS, pages 123–131, Springer-Velrlag, 2012.
36. B. Křena and T. Vojnar. Automated Formal Analysis and Verification: An Overview In International Journal of General Systems, 42(4):335–365. Taylor and Francis, 2013.
37. B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Can. AntMiner: Mining More Bugs by Reducing Noise Interference. In *Proc. of ICSE'16*,IEEE/AMC, 2016, Austin, TX, USA.
38. Ch.-S. Lin and G.-H. Hwang. State-cover Testing for Nondeterministic Concurrent Programs with an Infinite number of Synchronization Sequences, In *SCIENCE OF COMPUTER PROGRAMMING*, Volume 78, Issue 9, 1 September 2013, Pages 12941323.
39. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*. ACM, 2006.
40. S. Luke. *Essentials of Metaheuristics*, first, 2011.
Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/
41. Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining Performance Regression Inducing Code Changes in Evolving Software. In *Proc. of MSR'16*, IEEE/ACM, 2016, Austin, TX, USA.
42. R. V. van Nieuwpoort *Efficient Java-Centric Grid Computing*, 2003, Rob van Nieuwpoort. Available for free at https://books.google.cz/books?id=fRmZ6aC4eDwC

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

43. Y. Nir-Buchbinder, R. Tzoref, S. Ur. *Deadlocks: From Exhibiting to Healing*, In Third Workshop on Runtime Verification (RV03), volume 89(2) of Electronic Notes in Theoretical Computer Science, 2004.
44. S. Park, S. Lu, Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places, In *Proc. of ASPLOS'09*, ACM Press, 2009.
45. Ch. von Praun and T. R. Gross. *Object Race Detection*, SIGPLAN Not., 2001, volume 36, number 11, pages 70 –82, New York, NY, USA, ACM.
46. Ch. von Praun and T. R. Gross. *Static detection of atomicity violations in object oriented programs*, Runtime Verification, 2008, pages 104–118, Springer Berlin Heidelberg.
47. E. Rubinić, G. Mauša, and T. Galinac Grbac. *Software Defect Classification with a Variant of NSGA-II and Simple Voting Strategies*, In *Proc. of SSBSE'15*, LNCS 9275, pp.347–353,2015.
48. P. Saint-Hilaire, Ch. von Praun, E. Stolte, G. Alonso, A. O. Benz and T. Gross. *The RHESSI Experimental Data Center*, Solar Physics 210: 143–164, 2002.
49. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*. ACM, 1997.
50. K. Sen. Race Directed Random Testing of Concurrent Programs, In *Proc. of PLDI'08*, ACM Press, 2008.
51. L. A. Smith, J. M. Bull, J. Obdržálek. A Parallel Java Grande Benchmark Suite. In *Proc. of Supercomputing'01*, ACM, 2001.
52. S. D. Stoller. *Testing Concurrent Java Programs using Randomized Scheduling*, Proc. Second Workshop on Runtime Verification (RV), 2002, series: Electronic Notes in Theoretical Computer Science, volume 70(4), Elsevier.
53. Ch. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models, In *Proc. ICSE'16*, IEEE/ACM, 2016, Austin, TX, USA.
54. R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proc. of ISSTA'07*, pages 27–38. ACM, 2007. ACM.
55. S. Wang, T. Liu, and L. Tan. Automatically Learning Semantic Features for Defect Prediction, In *Proc. of ICSE'16*, IEEE/ACM, 2016, Austin, TX, USA.
56. D. White. Software Review: The ECJ Toolkit. *Genetic Programming and Evolvable Machines*, 13, 2012.
57. I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition, 2011.
58. J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*. ACM, 2012.
59. Y. Xu, P. Liang, and M. A. Babar. *Introducing Learning Mechanism for Class Responsibility Assignment Problem*, In *Proc. of SSBSE'15*, LNCS 9275, pp. 311–317, 2015.
60. S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proc. of ISSTA'09*, pages 201–212. ACM, 2009.
61. J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proc. of OOPSLA'12*. ACM, 2012.
62. E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, ETH Zurich, 1999.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe