Boosted Decision Trees for Behaviour Mining of Concurrent Programs

Renata Avros[2], Vendula Hrubá[1], Bohuslav Křena[1], Zdeněk Letko[1], Hana Pluháčková[1], Tomáš Vojnar[1], Zeev Volkovich[2], and Shmuel Ur[1]

[1] IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Brno, CZ
{ihruba, krena, iletko, ipluhackova, vojnar}@fit.vutbr.cz,
shmuel.ur@gmail.com
[2] Ort Braude College of Engineering, Software Engineering Department, Karmiel, IL
{r_avros, vlvolkov}@braude.ac.il

Abstract. Testing of concurrent programs is difficult since the scheduling nondeterminism requires one to test a huge number of different thread interleavings. Moreover, a simple repetition of test executions will typically examine similar interleavings only. One popular way how to deal with this problem is to use the noise injection approach, which is, however, parameterized with many parameters whose suitable values are difficult to find. In this paper, we propose a novel application of classification-based data mining for this purpose. Our approach can identify which test and noise parameters are the most influential for a given program and a given testing goal and which values (or ranges of values) of these parameters are suitable for meeting this goal. We present experiments that show that our approach can indeed fully automatically improve noise-based testing of particular programs with a particular testing goal. At the same time, we use it to obtain new general insights into noise-based testing as well.

1 Introduction

Testing of concurrent programs is known to be difficult due to the many different interleavings of actions executed in different threads to be tested. A single execution of available tests used in traditional unit and integration testing usually exercises a limited subset of all possible interleavings. Moreover, repeated executions of the same tests in the same environment usually exercise similar interleavings [2, 3]. Therefore, means for increasing the number of tested interleavings within repeated runs, such as *deterministic testing* [2], which controls threads scheduling and systematically enumerates different interleavings, and *noise injection* [3], which injects small delays or context switches into the running threads in order to see different scheduling scenarios, have been proposed and applied in practice.

In order to measure how well a system under test (SUT) has been exercised and hence to estimate how good a given test suite is, testers often collect and analyse coverage metrics. However, one can gain a lot more information from the test executions. One can, e.g., get information on similarities of the behaviour witnessed through different tests, on the behaviour witnessed only within tests that failed, and so on. Such information can be used to optimize the test suite, to help debugging the program, etc. In order to get such information, *data mining* techniques appear to be a promising tool.

In this paper, we propose a novel application of data mining allowing one to exploit information present in data obtained from a sample of test runs of a concurrent program to optimize the process of noise-based testing of the given program. To be more precise, our method employs a data mining method based on *classification* by means of *decision trees* and the *AdaBoost* algorithm. The approach is, in particular, intended to find out which parameters of the available tests and which parameters of the noise injection system are the most influential and which of their values (or ranges of values) are the most promising for a particular testing goal for the given program.

The information obtained by our approach can certainly be quite useful since the efficiency of noise-based testing heavily depends on a suitable setting of the test and noise parameters, and finding such values is not easy [8]. That is why, repeated testing based on randomly chosen noise parameters is often used in practice. Alternatively, one can try to use search techniques (such as genetic algorithms) to find suitable test and noise settings [8, 7].

The classifiers obtained by our data mining approach can be easily used to fully automatically optimize the most commonly used noise-based testing with a random selection of parameter values. This can be achieved by simply filtering out randomly generated noise settings that are not considered as promising by the classifier. Moreover, it can also be used to guide and consequently speed up the manual or search-based process of finding suitable values of test and noise parameters (in the latter case, the search techniques would look for a suitable refinement of the knowledge obtained by data mining). Finally, if some of the noise parameters or generic test parameters (such as the number of threads) appear as important across multiple test cases and test goals, they can be considered as important in general, providing a new insight into the process of noise-based testing.

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 (which are, of course, to be validated in the future on more case studies). (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.

Plan of the paper. The rest of the paper is structured as follows. Section 2 briefly introduces the techniques that our paper builds on, namely, noise-based testing of concurrent programs, data mining based on decision trees, and the AdaBoost algorithm. Section 3 presents our proposal of using data mining in noise-based testing of concurrent programs. Section 4 provides results of our experiments and presents the newly obtained insights of noise-based testing. Section 5 summarizes the related work. Finally, Section 6 provides conclusions and a discussion of possible future work.

2 Preliminaries

In our previous works, e.g., [8, 10], we have used noise injection to increase the number of interleavings witnessed within the executions of concurrent program and thus to increase the chance of spotting concurrency errors. Noise injection is a quite simple technique which disturbs thread scheduling (e.g., by injecting, removing, or modifying delays, forcing context switches, or halting selected threads) with the aim of driving the execution of a program into less probable scenarios.

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 program executions into which some noise is injected (which is determined using some *noise placement heuristics*). Multiple noise seeding and noise placement heuristics have been proposed and experimentally evaluated [10]. Searching for an optimal configuration of noise seeding and noise placement heuristics in combination with a selection of available test cases and their parameters has been formalized as the *test and noise configuration search problem* (TNCS) in [7, 8].

To assess how well tests examine the behaviour of an SUT, error manifestation ratio and coverage metrics can be used. Coverage metrics successfully used for testing of sequential programs (like statement coverage) are not sufficient for testing of concurrent programs as they do not reflect concurrent aspects of executions. Concurrency coverage metrics [1] are usually tailored to distinguish particular classes of interleavings and/or to capture synchronization events that occur within the execution. Some of the metrics target concurrency issues from a general point of view while some other metrics, e.g., those inspired by particular dynamic detectors of concurrency errors [9], concentrate on selected concurrency aspects only (e.g., on behaviours potentially leading to a deadlock or to a data race). In this work, we, in particular, use the *GoldiLockSC** *coverage metric* which measures how many internal states of the GoldiLock data race detector with the fast short circuit checks [5] have been reached [9].

The data mining approach proposed in this paper is based on *binary classification*. Binary classification problems consist in dividing items of a given collection into two groups using a suitable classification rule. Methods for learning such classifiers include decision trees, Bayesian networks, support vector machines, or neural networks [12]. The use of decision trees is the most popular of those because they are known for quite some time and can be easily understood. A *decision tree* can be viewed as a hierarchically structured decision diagram whose nodes are labelled by Boolean conditions on the items to be classified and whose leaves represent classification results. The decision process starts in the root node by evaluating the condition associated with it 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 a leaf node, and hence a decision, is reached. 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 the paper, we—in particular—employ the advanced classification technique called *Adaptive Boosting* (shortly, AdaBoost) [6] which reduces the natural tendency of decision trees to be unstable (meaning that a minor data oscillation can lead to a large differ-

ence in the classification). This technique makes it possible to correct the functionality of many learning algorithms (so-called weak learners) by weighting and mixing their outcomes in order to get the output of the boosted classifier. The method works in iterations (phases). In each iteration, the method aims at producing a new weak classifier in order to improve the consistency of the previously used ones. In our case, AdaBoost uses decision trees as the weak learners with the classification result being -1 or $+1$. In each phase, the algorithm adds new weighted decision trees obtained by concentrating on items difficult to classify by the so far learnt classifier and updates weights of the previously added decision trees to keep the sum of the weights equal to one. The resulting advanced 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.

3 Classification-based Data Mining in Noise-based Testing

In this section, we first propose our application of AdaBoost in noise-based testing. Subsequently, we discuss how the information hidden in the classifier may be analysed to draw some conclusions about which test and noise parameters are important for particular test cases and test goals or even in general. Finally, we describe two concrete classification properties that are used in our experiments.

3.1 An Application of AdaBoost in Noise-based Testing

First, in order to apply the proposed approach, one has to define some testing goal expressible as a binary property that can be evaluated over test results such that both positive and negative answers are obtained. The requirement of having both positive and negative results can be a problem in some cases, notably in the case of discovering rare errors. In such a case, one has to use a property that approximates the target property of interest (e.g., by replacing the discovery of rare errors by discovering rare behaviours in general). Subsequently, once testing based on settings chosen in this way manages to find some behaviours which were originally not available (e.g., behaviours leading to a rare error), the process can be repeated on the newly available test results to concentrate on a repeated discovery of such behaviours (e.g., for debugging purposes or for the purpose of finding further similar errors).

Once the property of interest is defined, a number of test runs is to be performed using a random setting of test and noise parameters in each run. For each such run, the property of interest is to be evaluated and a couple $(\overline{x}, y)$ is to be formed where $\overline{x}$ is a vector recording the test and noise settings used and y is the result of evaluating the property of interest. This process has to be repeated to obtain a set $X = \{(\overline{x}_1, y_1), \ldots, (\overline{x}_n, y_n)\}$ of such couples to be used as the input for learning the appropriate classifier.

Now, the AdaBoost algorithm can be applied. For that, the common practice is to split the set X to two sets—the training set and the testing set, use the training set to get a classifier, and then use the testing set for evaluating the precision of the obtained classifier. To evaluate the precision, one can use the notions of *accuracy* and *sensitivity*.

Accuracy 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. Sensitivity (also called as the negative predictive value or NPV) expresses the fraction of correctly classified negative results and can be computed as the number of the items correctly classified negatively divided by the sum of correctly and incorrectly negatively classified items (see e.g. [12]). Moreover, in order to increase confidence in the obtained results, this 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 phases and/or a bigger set X of data.

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 a knowledge can then in turn be used by testers when thinking of how to optimize the testing process. We discuss a way how such an analysis can be done in Section 3.2 and we apply it in Section 4.3. Moreover, the obtained classifier can also be directly used to improve performance of noise-based testing based on random selection of parameters by simply filtering out the settings that get classified as not meeting the considered testing goal. The fact that such an approach does indeed significantly improve the testing process is experimentally confirmed in Section 4.4.

3.2 Analysing Information Hidden in Classifiers

In order to be able to easy analyse the 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 preliminary 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 corresponding to positive and negative classification. AdaBoost provides us with a set of such trees, each with an assigned weight. We convert this set of trees into a set of rules such that we get a single rule for each parameter that appears in at least one decision tree. The rules consist of a condition and a weight, and they are obtained as follows. First, decision trees with negative weights are omitted because they correspond to weak classifiers with the weighted error greater than 0.5.[3] Next, the remaining decision trees are grouped according to the parameter about whose value they speak. For each group of the trees, a separate rule is produced such that the conjunction of the decision conditions of the trees from the group is used as the condition of the rule. 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.

The obtained set of rules can be easily used to gain some insight into how the test and noise injection parameters should be set in order to increase efficiency of the testing

[3] Note that the AdaBoost methodology suggests that the employed weak classifiers should not be of this kind, but they can appear in practical applications.

process—either for a given program and testing goal or even in general. In particular, one can look for parameters that appear in rules with the highest weights (which speak about parameters whose correct setting is the most important to achieve the given testing goal), for parameters that are important in all or many test cases (and hence can be considered to be important in general), as well as for parameters that do not appear in any rules (and hence appear to be irrelevant).

3.3 Two Concrete Classification Properties

In the experiments described in the next section, we consider two concrete properties according to which we classify test runs. First, we consider the case of finding TNCS solutions suitable for repeatedly finding known errors. In this case, the property of interest is simply the *error manifestation property* that indicates whether an error manifested during the test execution or not.

Subsequently, we consider the case of finding TNCS solutions 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** 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 1 % of the test executions.

4 Experimental Evaluation

In this section, we first describe the test data which we used for an experimental evaluation of our approach. Then, we describe the precision of the classifiers inferred from this data. Subsequently, we analyse the knowledge hidden in the classifiers, compare it with our previously obtained experience, and derive some new insights about importance of the different test and noise parameters. Finally, we demonstrate that a use of the proposed data mining approach does indeed improve (in a fully automated way) the process of noise-based testing with random setting of the parameters.

4.1 Experimental Data

The results presented below are based on 5 multi-threaded benchmark programs that contain a known concurrency error. We use data collected during our previous work [7]. Namely, our case studies are the *Airlines* (0.3 kLOC), *Animator* (1.5 kLOC), *Crawler* (1.2 kLOC), *Elevator* (0.5 kLOC), and *Rover* (5.4 kLOC). For each program, we collected data from 10,000 executions with a random test and noise injection setting. We collected various data about the test executions, such as whether an error occurred during the execution (used as our *error manifestation property*) and various concurrency coverage information, including the *GoldiLockSC** coverage used for evaluating the *rare events property*.

In our experiments, we consider vectors of test and noise parameters having 12 entries, i.e., $\overline{x} = (x_1, x_2, \ldots, x_{12})$. Here, $x_1 \in \{0, \ldots, 1000\}$ represents the noise frequency which controls how often the noise is injected and ranges from 0 (never) to 1000 (always). The $x_2 \in \{0, \ldots, 100\}$ parameter controls the amount of injected noise and ranges from 0 (no noise) to 100 (considerable noise). The $x_3 \in \{0, \ldots, 5\}$ parameter selects one of six available basic noise injection heuristics (based on injecting calls of `yield()`, `sleep()`, `wait()`, using busy waiting, a combination of additional synchronization and `yield()`, and a mixture of these techniques). The $x_4, x_5, x_7, x_8, x_9 \in \{0, 1\}$ parameters enable or disable the advanced injection heuristics *haltOneThread*, *timeoutTampering*, *sharedVarNoise*, *nonVariableNoise*, *advSharedVarNoise1*, and *advSharedVarNoise2*, respectively. The $x_6 \in \{0, 1, 2\}$ parameter controls the way how the *sharedVarNoise* advanced heuristic behaves (namely, whether it is disabled (0), injects the noise at accesses to one randomly selected shared variable (1) or at accesses to all such variables (2)). A more detailed description of the particular noise injection heuristics can be found in [3, 7, 8, 10].

Furthermore, $x_{10} \in \{1, \ldots, 10\}$ and $x_{11}, x_{12} \in \{1, \ldots, 100\}$ encode parameters of some of the test cases themselves. In particular, *Animator* and *Crawler* are not parametrised, and x_{10}, x_{11}, x_{12} are not used with them. In the *Airlines* and *Elevator* test cases, the x_{10} parameter controls the number of used threads, and in the *Rover* test case, the $x_{10} \in \{0, \ldots, 6\}$ parameter selects one of the available test scenarios. The *Airlines* test case is the only one that uses the x_{11} and x_{12} parameters, which are in particular used to control how many cycles the test does.

4.2 Precision of the Classifiers

In our experiments, we used the implementation of AdaBoost available in the GML AdaBoost Matlab Toolbox[4]. We have set it to use decision trees of height restricted to one and to use 10 boosting phases. The algorithm was applied 100 times on randomly chosen divisions of the test data into the training and validation groups.

Table 1 summarises the average accuracy and sensitivity of the learnt AdaBoost classifiers. One can clearly see that both the average accuracy and sensitivity are quite high, ranging from 0.61 to 0.99. Moreover, the standard deviation is very low in all cases. This indicates that we always obtained results that provide meaningful information about our test runs.

4.3 Analysis of the Knowledge Hidden in the Obtained Classifiers

We now employ the approach described in Section 3.2 to interpret the knowledge hidden in the obtained classifiers. Tables 2 and 3 show the inferred rules and their weights 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 interval constraints) and the lower part contains the appropriate weight from the interval $(0, 1)$.

[4] http://graphics.cs.msu.ru/en/science/research/machinelearning/AdaBoosttoolbox

Table 1. Average accuracy and sensitivity of the learnt AdaBoost classifiers.

	Error manifestation				*Rare behaviours*			
	Accurancy		Sensitivity		Accurancy		Sensitivity	
CaseStudies	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Airlines	0.7695	0.0086	0.6229	0.0321	0.9755	0.0056	0.9964	0.0021
Animator	0.937	0.0054	0.9866	0.0052	0.7815	0.0054	0.9071	0.0217
Crawler	0.9975	0.00076	0.999	0.00077	0.7642	0.0402	0.9741	0.0765
Elevator	0.8335	0.0038	0.9982	0.0016	0.6566	0.0051	0.6131	0.027
Rover	0.9714	0.0031	0.9912	0.0012	0.8737	0.1092	0.9687	0.137

Table 2. Inferred weighted rules for the error manifestation classification property.

Airlines

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

Animator

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

Crawler

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

Elevator

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

Rover

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

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.

As for the error manifestation property (i.e., Table 2), 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 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 advanced *haltOneThread* heuristic (x_4) turns out to be important. In the *Crawler* and *Rover* test cases, this heuristic should be enabled in order to detect an error. This behaviour fits into our previous results [10]

Table 3. Inferred weighted rules for the rare behaviours classification property.

Airlines

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

Animator

Rules	$0.5 < x_3 < 3.5$ or $4.5 < x_3$	$0.5 < x_6 < 1.5$
Weights	0.8	0.2

Crawler

Rules	$0.5 < x_3 < 3.5$ or $4.5 < x_3$	$0.5 < x_4$	$0.5 < x_5$	$0.5 < x_6 < 1.5$
Weights	0.46	0.08	0.2	0.26

Elevator

Rules	$0.5 < x_3 < 3.5$ or $4.5 < x_3$	$0.5 < x_4$	$0.5 < x_5$	$1.5 < x_6$	$1.5 < x_{10} < 4.5$ or $7.5 < x_{10}$
Weights	0.22	0.05	0.2	0.06	0.47

Rover

Rules	$2.5 < x_3 < 3.5$ or $4.5 < x_3$	$x_4 < 0.5$	$x_6 < 0.5$	$0.5 < x_7$
Weights	0.46	0.26	0.16	0.12

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 *advSharedVarNoise* noise 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.

For the case of classifying according to the rare behaviour property, the obtained rules are shown in Table 3. We can again find the highest weights in rules based on the x_3 parameter (*Animator*, *Crawler*, *Rover*) and on the x_1 parameter (*Airlines*). However, in the case of *Elevator*, the most contributing parameter is now the number of threads used by the test (x_{10}). The rule suggests to use certain numbers of threads in order to spot rare behaviours (i.e., it is important to consider not only a high number of threads). The generated sets of rules often contain the x_3 parameter controlling the type of noise (all test cases except for *Airlines*) and the x_6 parameter which controls the *sharedVarNoise* heuristic. These parameters thus appear to be of a general importance in this case.

Next, the parameter x_{12} does again turn out to be important in the *Airlines* test case, and the x_{10} parameter is important in the *Elevator* test case. This indicates that even for testing rare behaviours, it is important to adjust the number of threads or test cycles to suitable values. Finally, the x_8, x_9, and x_{11} parameters do 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).

Table 4. A comparison of the random approach and the newly proposed AdaBoost approach.

	Error manifestation				*Rare behaviours*			
CaseStudies	Rand.	AdaBoost	Pos.	Impr.	Rand.	AdaBoost	Pos.	Impr.
Airlines	56.26	75.43	1,612	1.34	1.94	1.64	2,444	0.85
Animator	14.81	54.05	901	3.65	39.53	57.95	3,258	1.47
Crawler	0.18	0.25	2,806	1.39	22.41	31.26	1,513	1.39
Elevator	16.75	27.66	1,410	1.65	52.77	59.51	1,398	1.13
Rover	6.65	36.25	822	5.45	10.76	23.21	1,620	2.16

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 [10, 7, 8] and that the *haltOneThread* noise injection heuristics (x_4) provides in some cases a dramatic increase in the probability of spotting an error [10]. 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.

4.4 Improvement of Noise-based Testing with Random Parameters

Finally, we show that the obtained classifiers can be used to fully automatically improve the process of noise-based testing with randomly chosen values of parameters. For that, we reuse the 7,500 test runs out of 10,000 test runs recorded with random parameter values for each of the case studies. In particular, we randomly choose 2,500 test runs as training set for our AdaBoost approach to produce classifiers. Then, from the rest of the test runs, we randomly choose 5,000 test runs to compare our approach with the random approach.

From these 5,000 test runs, we first select runs that were performed using settings considered as suitable for the respective testing goals by the classifiers that we have obtained. Then, we compute what fractions of all the runs and what fractions of all the selected runs satisfy the testing goals for the considered case studies, which shows us the efficiency of the different testing approaches.

In Table 4, the columns *Pos.* contain the numbers of test runs (out of the considered 5,000 runs) classified positively by the obtained classifiers for the two considered test goals. The columns *Rand.* give the percentage of runs out of the 5,000 runs performed under purely randomly chosen values of parameters that met the considered testing goals (i.e., found an error or a rare behaviour, respectively). The columns *AdaBoost* give this percentage for the selected runs (i.e., those whose number is in the columns *Pos.*). Finally, the columns *Impr.* present how many times the efficiency of testing with the selected values of parameters is better than that of purely random noise-based testing (i.e., it contains the ratio of the values in the *AdaBoost* and *Rand.* columns).

The improvement columns clearly show that our AdaBoost technique often brings an improvement (with one exception described bellow), which ranges from 1.13 times in the case of the rare behaviours property and the *Elevator* test case to 5.45 times in the case of the error manifestation property and the *Rover* test case. In the case of the Airlines test case and the rare behaviours property, our technique provided worse results (impr. 0.85). This is mostly caused by the simplicity of the case study and hence lack of rare behaviours in the test runs. Therefore, our approach did not have enough samples to construct a successful classifier. Nevertheless, we can conclude that our classification approach can really improve the efficiency of testing in majority of studied cases.

5 Related Work

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 [4, 11]. In [11], 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 [4] uses a data mining-like technique, more precisely, the feature selection algorithm, to infer a reduced call graph representation of the SUT, which is then used to discover anomalies in the behaviour of the SUT within erroneous executions.

There is also rich 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 [1] and optimisation of the test suite [13]. In [1], 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. In [13], 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.

6 Conclusions and Future Work

In the paper, we have proposed a novel application of classification-based data mining in the area of noise-based testing of concurrent programs. In particular, we proposed an approach intended to identify which of the many noise parameters and possibly also parameters of the tests themselves are important for a particular testing goal as well as which values of these parameters are suitable for meeting this goal. As we have demonstrated on a number of case studies, the proposed approach can be used to fully automatically improve the noise-based testing approach of a particular program with a particular testing goal. Moreover, we have also used our approach to derive new insights into the noise-based testing approach itself.

Apart from validating our findings on more case studies, there is plenty of space for further research in the area of applications of data mining in testing of concurrent

programs. One can ask many interesting questions and search for the answers using different techniques, such as outliers detection, clustering, association rules mining, etc. 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 adjusting the above cited works on detecting untested behaviour or on eliminating tests of similar behaviour for the case of concurrent programs.

Acknowledgement. The work was supported by the bi-national Czech-Israel project (Kontakt II *LH13265* by the Czech Ministry of Education and *3-10371* by Ministry of Science and Technology of Israel), the EU/Czech IT4Innovations Centre of Excellence project *CZ.1.05/1.1.00/02.0070*, and the internal BUT projects *FIT-S-12-1* and *FIT-S-14-2486*. Z. Letko was funded through the EU/Czech Interdisciplinary Excellence Research Teams Establishment project *CZ.1.07/2.3.00/30.0005*.

References

1. Yoram Adler, Noam Behar, Orna Raz, Onn Shehory, Nadav Steindler, Shmuel Ur, and Aviad Zlotnick. Code Coverage Analysis in Practice for Large Systems. In *Proc. of ICSE'11*, pages 736–745. ACM, 2011.
2. Thomas Ball, Sebastian Burckhardt, Katherine E. Coons, Madanlal Musuvathi, and Shaz Qadeer. Preemption Sealing for Efficient Concurrency Testing. In *Proc. of TACAS'10*, volume 6015 of LNCS, pages 420–434. Springer-Velrlag, 2010.
3. Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499. Wiley, 2003.
4. Frank Eichinger, Victor Pankratius, Philipp W. L. Große, and Klemens 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-Velrlag, 2010.
5. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, pages 245–255. ACM, 2007.
6. Yoav Freund and Robert E. Schapire. A Short Introduction to Boosting. In *In Proc. of IJCAI'99*, pages 1401–1406. Morgan Kaufmann, 1999.
7. Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, and Tomáš 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.
8. Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Shmuel Ur, and Tomáš Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*, volume 7515 of LNCS, pages 152–167. Springer-Velrlag, 2012.
9. Bohuslav Křena, Zdeněk Letko, and Tomáš 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.
10. Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'11*, volume 7119 of LNCS, pages 123–131, Springer-Velrlag, 2012.
11. Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proc. of ISSTA'07*, pages 27–38. ACM, 2007. ACM.
12. Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition, 2011.

13. Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proc. of ISSTA'09*, pages 201–212. ACM, 2009.