Parameter Tuning for Search-Based Test-Data Generation Revisited: Support for Previous Results

Anton Kotelyanskii
Department of Computer Science
Allegheny College

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

***Abstract*—Although search-based test-data generators, like EvoSuite, efficiently and automatically create effective JUnit test suites for Java classes, these tools are often difficult to configure. Prior work by Arcuri and Fraser revealed that the tuning of EvoSuite with response surface methodology (RSM) yielded a configuration of the test data generator that did not outperform the default configuration. Following the experimental design and protocol described by Arcuri and Fraser, this paper presents the results of a study that lends further support to prior results: like RSM, the EvoSuite configuration identified by the well-known Sequential Parameter Optimization Toolbox (SPOT) failed to significantly outperform the default settings. Although this result is negative, it furnishes further empirical evidence of the challenge associated with tuning a complex search-based test data generator. Moreover, the outcomes of the presented experiments also suggests that EvoSuite's default parameters have been set by experts in the field and are thus suitable for use in future experimental studies and industrial testing efforts.**

I. Introduction

Since computers and software are used in every major economic sector, software testing—the practice of ensuring that a program works as intended—is a crucial, if challenging and time-consuming, task [1]. Developers can use automated testing tools, such as EvoSuite [2], to generate test suites that may help to both identify defects in and established a confidence in the correctness of the program under test. While test-data generation tools can automatically create a JUnit test suite for a Java class, they are often difficult to configure and may not always produce the desired output [3], [4].

Search-based testing tools that employ genetic algorithms (GAs), like the award-winning EvoSuite [2], have a wide variety of parameters—such as the population size, chromosome length, and crossover rate—that the tester normally needs to tune [3]. Since testers are usually not experts in the configuration of GAs, they often must resort to either using the default values of parameters or leveraging an optimizer that can automatically pick these values. In prior work, Arcuri and Fraser experimentally determined that parameters identified by response surface methodology (RSM) failed to significantly outperform the default configuration of EvoSuite [4].

This paper further investigates whether the negative result of Arcuri and Fraser was due to the fundamental challenge of tuning search-based test data generation methods or, rather, their choice of RSM as the parameter tuner. Following the experimental design and protocol of the previous experiments, we conducted a similar experiment to Arcuri and Fraser's, replacing RSM with the well-known sequential parameter optimization toolbox (SPOT) [5]. We chose SPOT because it worked well on similar algorithms, as mentioned in Section II, and it has an easy-to-use implementation in the readily available R language for statistical computation—thus making it easier for others to reliably replicate our experiments.

This paper's large-scale empirical study reveals that SPOT, like RSM, failed to find a configuration of EvoSuite that performed better than the data generator's default configuration. This negative result lends support to the one arrived at by Arcuri and Fraser, suggesting that, at least for the task of selecting parameter values for a GA-based test data generator, parameter tuning is fundamentally challenging. This paper's empirical outcomes also lend further credibility to the assertion that EvoSuite has been developed by experts who picked default parameter values that are likely to be suitable both for future testing experiments and industrial testing tasks.

The remainder of this paper is organized in the following fashion. First, Section II gives an overview of related work. In addition to explaining how EvoSuite and SPOT work, Section III furnishes details about the design of this paper's experiments. While Section IV reports on the outcomes of the empirical study, Section V discusses the threats to the experiments' validity. Finally, Section VI concludes with a summary of the paper's results and a proposal for future work.

II. Related Work

Arcuri and Fraser tried to tune EvoSuite using response surface methodology, which they describe as the "world's most used technique for optimizing parameters" [3]. First, they sought to manually identify an optimal configuration for a small subset of cases and parameter values, ultimately determining that, while tuning did not necessarily provide better results, it had potential. However, when they applied RSM to a large subset consisting of 609 classes from 10 different Java applications it failed to find a configuration that yielded substantially better results than the default settings [3].

While Arcuri and Fraser's empirical result suggests that parameter tuning, or at the least RSM, may not be suited to the tuning of search-based software testing tools like EvoSuite, the literature is replete with reports of tuning's efficacy. For instance, Myers et al. and Lenth report that RSM has been successfully used in a wide variety of contexts such as chemical reaction characterization and semiconductor manufacturing [6], [7]. More germane to this paper, Preuß and Bartz-Beielstein used SPOT to tune a variety of evolutionary algorithms (EA); even though they were unable to make firm conclusions about the usefulness of the EAs, they did find that tuning them with SPOT improved their performance [8]. Also, Flasch et al. demonstrated that SPOT-derived configurations of genetic algorithms quickly made accurate predictions about the behavior of environmental engineering systems [9].

This paper reports on a replication of Arcuri and Fraser's experimental study of parameter tuning for search-based test data generation. In prior work Brooks et al. asserted that, while replication in software engineering research is infrequent, it is crucially important—these authors also pointed out that exact replications of past studies are "unattainable" [10]. Furthermore, Kapfhammer suggested that a major reason for the lackluster adoption of testing techniques by industry is a dearth of large-scale empirical data supporting the efficiency and effectiveness of techniques [11]. Finally, although Clark et al. campaigned for increased replication of experiments in the field of operating systems, their key point—that additional results increase the confidence in prior work and any supporting tools—also applies to the field of software testing [12].

Much like the experimental outcomes of Arcuri and Fraser, this paper's experiments lead to the negative conclusion that parameter tuning with SPOT does not improve the efficacy of search-based test data generation. While it could be argued that it is not useful to report that a technique was ineffective, several leading experts, such as Gupta, Stopfer, and Schooler have noted that the publication of negative outcomes is an important mechanism for advancing scientific research [13], [14]. Since Ioannidis previously pointed out that most published results—the majority of which are positive—are normally refuted by suqsequent evidence [15], negative results, like those identified by both Arcuri and Fraser and this paper, clarify trends and point out future research directions [3].

III. Experimental Design and Implementation

A. Sequential Parameter Optimization Toolbox

To ensure that this paper is self contained, this section briefly reviews SPOT; for more details please reference [5], [16]. The optimization algorithm implemented in SPOT takes as input some SPOT configurations r and a parameter space P. Using random sampling from the possible parameter space P, it selects a population P' of candidate configurations p. SPOT evaluates these configurations by executing the algorithm being tuned—in this paper, the EvoSuite test data generator—and creates a prediction model F based on the results.

Using predictions made by F, SPOT generates a large set of new candidate configurations and assigns a utility value u to each one. Those with the best u are chosen to create a new population of candidate configurations, P''. The candidate configurations in P'' are then evaluated—again, in this paper by running EvoSuite—so that a total of $r_{numElite}$ of the best candidate configurations in P'' are then added to P'. Finally, SPOT updates the prediction model F with the results from the previous EvoSuite runs. SPOT repeats all of the aforementioned steps until a termination criterion becomes true; in the case of the experiments in this paper, SPOT ran for a number of candidate configuration evaluations. Ultimately, SPOT returns the best p in P', which corresponds, in this paper, to a hopefully improved configuration of EvoSuite.

The initial design, or set of candidate configurations to be evaluated, and the subsequent designs were created by SPOT with Latin Hypercube Design (LHD) [5], [16]. The initial design size was set to 30, a value determined as the best after running SPOT in preliminary small-scale experiments. The prediction model F used to pick candidate configurations for evaluation was the Kriging model, implemented based on Matlab code by Forrester et al. [17]. Both are part of the `SPOT` package available for the R language [5]. We chose LHD and the Kriging model since they are used in a demonstration problem similar to parameter tuning for EvoSuite [5].

B. Search-based Test Data Generation with EvoSuite

Again, to ensure that this paper is self contained, this section briefly reviews the fundamentals of test data generation with EvoSuite; more details are available in [2], [4], [18]: As mentioned in Section I, EvoSuite is a tool that uses a genetic algorithm to generate a JUnit test suite for a Java class. Briefly, the steps in EvoSuite's genetic algorithm are:

1) **Initialization**: A population of individuals is randomly generated. In the case of EvoSuite, an individual is a whole test suite made up of test cases that execute methods in the program and then call test oracles.
2) **Fitness**: Every individual's fitness is calculated according to the given fitness function, a quantitative measurement of an individual's overall effectiveness. Currently, EvoSuite can use either statement coverage, branch coverage, or mutation coverage, depending on the configuration—with the default being statement coverage. Intuitively, these higher-is-better coverage metrics indicate that the tests are exercising more of the program under test and are thus more likely to find defects.
3) **Crossover and Reproduction**: Parts of two individuals' chromosomes are joined together to form a new individual. For EvoSuite, this step involves combining two test suites to form a new, and hopefully improved, test suite. While all of the individuals have a chance of reproducing, those with a better fitness value are more likely to be combined to produce a new test suite.
4) **Mutation**: Without involving other members of the population, some individuals are randomly changed and kept in the population, thus preserving diversity. EvoSuite mutates an entire test suite by adding new test cases or mutating individual tests through the addition, deletion, and modification of both statements and parameters.
5) **Iteration and Termination**: The new population is evaluated and evolution continues until a stopping criterion is reached. Common stopping criteria include fitness stagnation, when little to no fitness increase is observed between generations, or the number of generations—with EvoSuite defaulting to overall execution time.

TABLE I: EvoSuite parameters subject to tuning and the minimum and maximum of their potential values, as given to SPOT.

Parameter Name	Minimum	Maximum
Population Size	5	99
Chromosome Length	5	99
Rank Bias	1.01	1.99
Number of Mutations	1	10
Max Initial Number of Tests	1	10
Crossover Rate	0.01	0.99
Probability of Using the Pool of Constants	0.01	0.99
Probability of Inserting New Test Case	0.01	0.99

TABLE II: EvoSuite's default configuration and the final parameter values resulting from optimization with SPOT.

	Default	SPOT
Population Size	50	74
Chromosome Length	40	45
Rank Bias	1.7	1.884547
Number of Mutations	1	1
Max Initial Number of Tests	10	5
Crossover Rate	0.75	0.6426859
Probability of Using Constant Pool	0.5	0.5289
Probability of Inserting Test Case	0.1	0.0355009

C. Experimental Design

Since our careful review of Arcuri and Fraser's work suggested that their methodology was sound, we based this paper's experimental design on the one they used when attempting to tune EvoSuite with response surface methodology [3]. Our choice of this prior methodology also supports a comparison of our results with those that were previously reported.

We picked a potential configuration space of eight EvoSuite parameters, with continuous intervals of potential values for each one. We also randomly chose a total of 10 projects from those available through the SF100, a set of 100 randomly selected Java projects downloaded from the file-sharing site SourceForge.net, as compiled by Arcuri and Fraser to provide representative real-world examples for evaluating the effectiveness of test data generation software [19]. Choosing these 10 projects resulted in a total of 475 Java classes for which EvoSuite would attempt to generate JUnit test suites.

SPOT was configured to run with a maximum of 280 evaluations for EvoSuite, with each evaluation requiring an execution of EvoSuite on each of the 475 classes. That is, when SPOT did an evaluation of EvoSuite, it performed test suite generation using the candidate configuration for the first class and then for the second class, and so on until the tool had attempted to generate test suites for all of the 475 Java classes. We decided that SPOT should run for 280 evaluations of EvoSuite because Arcuri and Fraser used the same number of evaluations for RSM [3]. In summary, except for the fact that we purposefully made a random selection of projects differing from those used in the prior study, our experimental design is that same as that of Arcuri and Fraser.

Table I gives the potential parameter space—that is, the parameters to be tuned and the range of their possible values—for this paper's experiments. Due to space constraints, we furnish brief and intuitive definitions of each of these parameters. Population size is the number of candidate test suites in the population pool, while the chromosome length is the maximum length of each candidate. EvoSuite can employ rank selection to ensure that candidate test suites are not selected by absolute fitness but rather by their rank in the overall population, with the best candidate having the highest numerical ranking. Ensuring that EvoSuite is not overly elitist, the rank bias parameter in Table I allows for the probability of a ranking position to be weighted.

The number of mutations is the number of genes of an individual's chromosome that are changed when a mutation event occurs. In addition, the maximum initial number of tests is the maximum number of tests that each candidate in the population can have when the population is first generated. The crossover rate is the chance that the two chosen candidates will be crossed over instead of being directly passed on to the next generation; if EvoSuite performs crossover, then the offspring resulting from this operation become a part of the subsequent population. When EvoSuite generates a test suite for a Java class, it first parses it and places all of the class constants into a pool. The probability of using the pool of constants, as given in Table I, is the probability of picking one of these constants instead of a random value or a variable. Finally, the probability of inserting a new test is the probability of inserting a new test case into a test suite during a mutation.

The metric used to guide both the SPOT parameter tuner and to evaluate EvoSuite's effectiveness was inverse branch coverage. Branch coverage is the percentage of source code branches in the Java class that are covered by the generated test suite; intuitively, higher coverage suggests a better test suite [18]. We used the inverse of this metric—that is, for branch coverage score b, the value of $(1 - b)$—because SPOT always seeks to minimize the fitness value. To collect enough data points to support a rigorous statistical analysis, we ran EvoSuite for 100 trials with the default configuration and 100 trials with the configuration returned by SPOT [3].

To evaluate the statistical significance of the results, we followed the guidelines in [20] and used the Mann-Whitney U-Test to evaluate the null hypothesis that two populations—in this case, the results from using the SPOT-derived and EvoSuite-default configurations, respectively—are the same. If the test returns a p-value less than 0.05, then we reject the null hypothesis, concluding that the two configurations are not the same; otherwise we confirm the null hypothesis.

Again adhering to the standard set in [20], we evaluate the effect size with the Vargha-Delaney $\hat{A}_{12}$ effect size measure that discerns the difference between the results from using either the SPOT-derived configuration or the configuration with EvoSuite's default settings [21]. The values of this measure range from 0 to 1; if it is < 0.29 or > 0.71, the effect is "large". If the size is < 0.36 or > 0.64, we classify the effect as "medium." If it is < 0.44 or > 0.56, then we judge the effect to be "small". Otherwise, it is labeled as "none". An effect size > 0.5 means that the results returned by the SPOT-chosen parameters are higher, while < 0.5 means that the SPOT-based results are lower. Since we are using an inverse—or lower-is-better metric—a value > 0.5 means SPOT performed worse, while < 0.5 means it performed better than the defaults.

TABLE III: Values of the Vargha-Delaney $\hat{A}_{12}$ effect size measure and the Mann-Whitney U-test, with the scores from SPOT's configuration being the first input to the statistical procedures and the default configuration being second. Since we use the lower-is-better inverse fitness metric, an $\hat{A}_{12}$ value greater than 0.5 means that the SPOT configuration performed worse than EvoSuite's default parameter setting.

	$\hat{A}_{12}$ Effect Size Measure	Mann-Whitney U-Test p-value
All results for all trials and all classes ("all results")	0.5029	0.1045
"All results" without the "easy" or "hard" classes	0.5048	0.0314
Mean of results by trial, across all classes ("all trials")	0.6085	0.0081
"All trials" without the "easy" or "hard" classes	0.6196	0.0034
Mean of results by class, across all trials ("all classes")	0.5034	0.8507
"All classes" without the "easy" or "hard" classes	0.5061	0.7886

D. Implementation

All experiments were run in R, a high-level, open-source language for statistical computing [22]. We chose R for the experimentation infrastructure both because SPOT is implemented in R and since it offers an environment in which data can be saved and commands can be run both automatically and interactively, allowing for easy experimentation and implementation. SPOT is provided by R's `SPOT` package [5].

As in Arcuri and Fraser's experiment, running the parameter tuning algorithms proved to be very computationally expensive. Performing the 280 evaluations of the candidate configurations, with 475 EvoSuite executions for all the Java classes and each run taking approximately two minutes, required $280*475*2 = 266000$ minutes—or over 184 days of computational time—only for SPOT to identify the optimized configuration. Since we also had to complete 100 trials of test data generation with EvoSuite in both the default and SPOT-derived configurations, we leveraged a cluster of computers.

The experiments were run on a cluster containing about 70 computers. Allowing for the fact that some computers might crash or otherwise be temporarily unavailable, our cluster implementation included fault tolerance mechanisms. Each cluster node ran Ubuntu 12.04 and a 3.5.0-49-generic Linux kernel. Except for EvoSuite, which is implemented in Java, all other programs used R version 3.1; EvoSuite executed in the version 1.7.0_55 of the Java SE Runtime Environment. All inputs (e.g., Java classes subject to testing and SPOT configuration files) and outputs (e.g., EvoSuite output and R result files) were stored on a network file system.

SPOT executed the program subject to tuning by invoking a user-defined wrapper function with the current candidate configuration and other relevant information. The wrapper runs EvoSuite and returns a fitness value for the chosen Java class and the provided parameter values. After SPOT found its best configuration, EvoSuite ran in an identical fashion to the tuning phase, with the exception that, for 100 trials each, it first used the tuned parameter values and then the defaults.

IV. Experimental Results

After randomly selecting the sample of 475 Java classes from the SF100, we determined that 139 of them were "easy" subjects because EvoSuite always achieved perfect coverage regardless of the configuration. In addition, we found a total of 21 "hard" subjects, or classes for which EvoSuite always achieved 0% coverage; EvoSuite crashed on most of these, citing errors regarding Java's inheritance mechanism.

As mentioned in Section III, the evaluation metric for these experiments was the lower-is-better inverse branch coverage metric; Table III furnishes the values of the $\hat{A}_{12}$ effect size measure and the Mann-Whitney U-Test for this metric. Knowing that our sample of Java classes contained "easy" and "hard" cases, we ran the statistical procedures both with and without these two types of classes. To account for different types of variability, we arranged the data in three different ways: (i) all of the individual scores for 100 trials and all of the 475 classes (i.e., 47500 observations in total) (ii) the mean score for each trial across all of the 475 classes (i.e., 100 observations in total), and (iii) the mean score for each class across all of the 100 trials (i.e., a total of 475 observations).

For the "all results" and "all classes" categories, Table III shows that, according to both $\hat{A}_{12}$ and the p-values, the SPOT-derived and default configurations are indistinguishable. Looking at the results in the "all trials" category, we see that SPOT's configuration is worse than the defaults. In summary, these results support Arcuri and Fraser's conclusion: A configuration returned by a parameter tuning algorithm yields results that are either the same as or worse than EvoSuite's defaults.

Even though our results support Arcuri and Fraser's, further exploration is useful as a means of developing an understanding of this negative phenomenon. The box-and-whisker plots in Figures 1a and 1b visualize the means from the "all trials" and "all classes" categories. In these plots the bottom and top whiskers show the minimum and maximum data values excluding outliers, while the box itself represents the interquartile range (i.e., the measure of statistical dispersion that is the difference between the first and third quartiles), the middle line represents the median value, and open circles are outliers. These plots visually confirm that SPOT's configuration either performs worse than or no better than the defaults.

Further investigation revealed that there were only 11 classes, out of the 475 in total, in which the SPOT configuration led to results that were better than the defaults in a statistically significant way. Even though we could not discern a pattern in the projects and Java classes for which the SPOT-derived parameters were better, we did note that four classes performed types of input and output (e.g., `MyInputStream` from `lagoon` and `jgaapGUI` from `jgaap`).

Figure 2 visualizes how SPOT optimized EvoSuite's parameters, with two broad strategies emerging: either quickly discarding a substantial range of potential values (as evident in Figures 2c, 2d, and 2h) or continuing to explore almost the entire range of possible values (as seen in Figures 2a, 2b, and

Figures 2e through 2g). Whether or not SPOT stops examining a large part of a parameter's range of values, the fact that half of the parameters are floats and that their ranges often contain many values prevents it from finding a good configuration. As shown by the graph in Figure 2i, after SPOT causes an initial undesirable spike in inverse fitness, the quality of the best configuration hovers in the same range. In essence, parameter tuning of search-based test data generation has a "soft floor" beyond which it is hard to make meaningful progress.

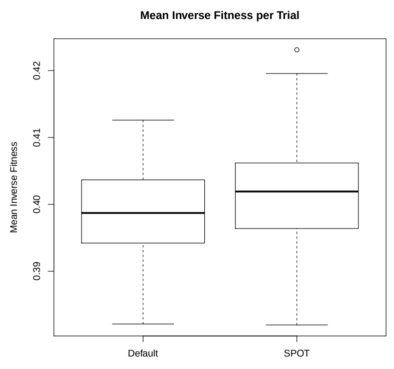
Yet, it is important to recall that Table III shows that the default configuration is only better than SPOT's by a small degree. While our results suggest that it is not likely to be the case, it is possible that, if SPOT was given a larger search budget or configured in a different manner, it might break past the soft floor and find better parameters for EVOSUITE.

V. THREATS TO VALIDITY

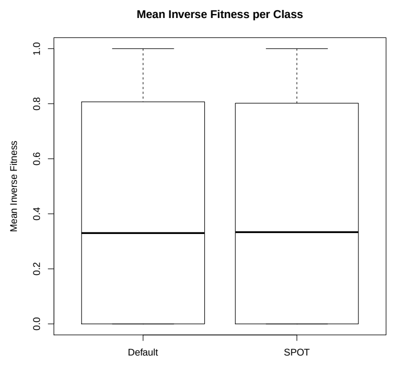
Since this paper's experimental design is largely based on that of Arcuri and Fraser [3], the threats to the validity of our experiments are similar to theirs. Threats to internal validity might come from either how we implemented the components of the experimental framework (e.g., the cluster distribution mechanism and the integration of SPOT and EVOSUITE) or how we configured SPOT. Even though we checked the correctness of our framework on several small examples, it is possible that defects may still exist in it. Since SPOT has been successfully used in other real-world applications [8], [16], it is possible that a smaller region of interest, another initial design size, alternative methods for generating the designs, different prediction models, or the choice of new EVOSUITE parameters may produce results that either reveal the efficacy of tuning or highlight different challenges.

It is also important to note that we did not use a dedicated cluster to run the experiments. Yet, evidence from running preliminary experiments during peak and non-use times suggests that both the practical and the statistical significance of the results were not influenced by this choice. Additionally, since the experiments measured the effectiveness of a test data generation tool that employs random numbers, we used different random seeds and conducted multiple trials, thus allowing us to create box-and-whisker plots in addition to employing rigorous statistical methods such as the Mann-Whitney U-test and the Vargha-Delaney effect size.

As with Fraser and Arcuri's experiment, construct validity threats may arise from the fact that branch coverage was used as EVOSUITE's fitness function, even though developers may prioritize other aspects of their test suites, such as execution time, size, or fault-finding capability. Threats to external validity might come from the chosen projects not being representative of Java programs in general. Even though they were selected randomly, it is possible that the random selection process did not pick Java classes that are emblematic of the entire population. Yet, leveraging programs from the SF100—a very large and representative set of open-source projects [19]—enabled us to make statistically sound claims and to compare our results with those of Arcuri and Fraser.



(a) **Mean of results by trial, across all classes**. For 100 trials and 475 classes, we calculated the mean inverse statement coverage score for each trial across all of the classes, with this plot showing the variability in these 100 mean values.

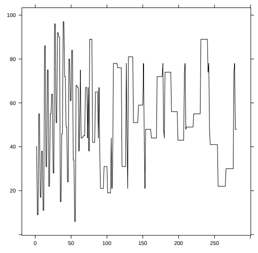


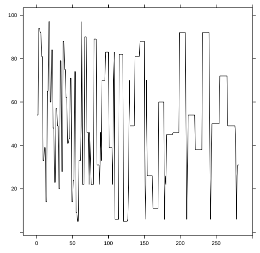
(b) **Mean of results by class, across all trials**. For 100 trials and 475 classes, we calculated the mean inverse statement coverage score for each class across all of the trials, with this plot showing the variability in these 475 mean values.

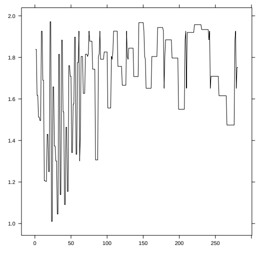
Fig. 1: Box-and-whisker plots of the lower-is-better inverse fitness.

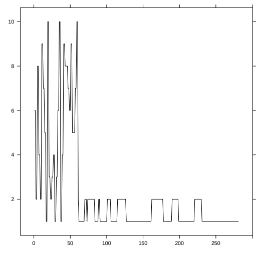
VI. CONCLUSION AND FUTURE WORK

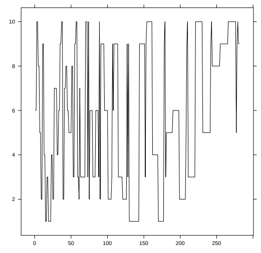
This paper revisits the challenge of parameter tuning for search-based test data generation, ultimately lending support to the prior negative results of Arcuri and Fraser: tuning EVOSUITE's parameters with the well-known SPOT optimizer does not yield configurations significantly better than the defaults. On a randomly selected set of 10 Java projects available in the SF100 repository, with 475 classes in total, the configurations returned by the parameter tuning algorithm only performed better on eleven of the classes. Since there were many Java classes in the randomly chosen experimental subset that were either "easy" (i.e., all configurations always achieved perfect coverage) or "hard" (i.e, all configurations always achieved no coverage because, in some cases, EVOSUITE could not generate any data), there were times in which SPOT could not improve on the default values. Of the remaining Java classes, the SPOT-derived configuration either performed worse than the defaults or had no statistically significant impact.

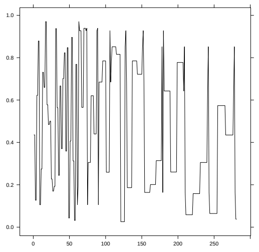

(a) Population Size

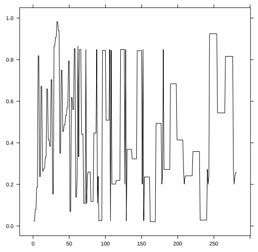

(b) Chromosome Length

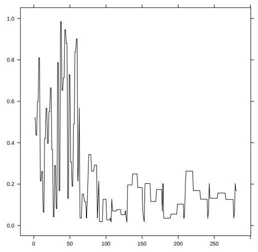

(c) Rank Bias


(d) Num. of Mutations


(e) Max. Num. Initial Tests


(f) Crossover Rate


(g) Prob. Using Const. Pool


(h) Prob. Insert New Test Case

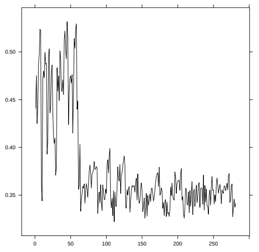

(i) Inverse Fitness

Fig. 2: SPOT's search trajectory, with the horizontal axis of the graph showing the number of evaluations and the vertical one giving the value of the parameter in the title. For descriptions of the parameters whose values are visualized in this graph, please refer to Section III.

In summary, this paper's negative result suggests that EvoSuite's default parameters have been set by experts and are thus suitable for use in future experimental studies and industrial testing efforts. In future work, we plan to extend the current experiment by using a different, and larger, random sampling of Java classes. Additionally, we will try new configurations of SPOT and select both different EvoSuite parameters and value ranges as inputs to SPOT. Finally, experiments with different parameter tuning algorithms, such as ParamILS [23], and test-data generation techniques, like Randoop [24], may yield more insight into the benefits and challenges of parameter tuning in software testing.

References

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," Planning Report 02-3, NIST, Tech. Rep., 2002.

[2] G. Fraser and A. Arcuri, "EvoSuite at the SBST 2013 tool competition," in *Proc. of SBST*, 2013.

[3] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *ESE*, vol. 18, no. 3, 2013.

[4] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *Proc. of ISSTA*, 2013.

[5] T. Bartz-Beielstein, "SPOT: An R package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization," *arXiv preprint arXiv:1006.4645*, 2010.

[6] R. V. Lenth, "Response-surface methods in R, using rsm," *JSM*, vol. 32, no. 7, 2009.

[7] R. Myers, D. Montgomery, and C. Anderson-Cook, *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, 2009.

[8] M. Preuß and T. Bartz-Beielstein, "Sequential parameter optimization applied to self-adaptation for binary-coded evolutionary algorithms," in *Parameter Setting in Evolutionary Algorithms*. Springer, 2007.

[9] O. Flasch, T. Bartz-Beielstein, A. Davtyan, P. Koch, W. Konen, T. D. Oyetoyan, and M. Tamutan, "Comparing CI methods for prediction models in environmental engineering," in *Proc. of CEC*, 2010.

[10] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller, "Replication's role in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Søjberg, Eds., 2008.

[11] G. M. Kapfhammer, "Empirically evaluating regression testing techniques: Challenges, solutions, and a potential way forward," in *Proc. of RT*, 2011.

[12] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews, "Xen and the art of repeated research," in *Proc. of FREENIX*, 2004.

[13] N. Gupta and M. Stopfer, "Negative results need airing too," *Nature*, vol. 470, no. 39, 2011.

[14] J. Schooler, "Unpublished results hide the decline effect," *Nature*, vol. 470, 2011.

[15] J. P. A. Ioannidis, "Why most published research findings are false," *PLoS Med*, vol. 2, no. 8, 2005.

[16] T. Bartz-Beielstein, C. W. Lasarczyk, and M. Preuß, "Sequential parameter optimization," in *Proc. of CEC*, 2005.

[17] A. Forrester, A. Sobester, and A. Keane, *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley, 2008.

[18] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. of ESEC/FSE*, 2011.

[19] ——, "Sound empirical evidence in software testing," in *Proc. of ICSE*, 2012.

[20] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. of ICSE*, 2011.

[21] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Jour. of Educ. and Behav. Stat.*, vol. 25, no. 2, 2000.

[22] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org

[23] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An automatic algorithm configuration framework," *JAIR*, vol. 36, no. 1, 2009.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. of ICSE*, 2007.