Parallel Many-Objective Search for Unit Tests

Verena Bader
Chair of Software Engineering II
University of Passau, Germany

José Campos
University of Washington
WA, USA

Gordon Fraser
Chair of Software Engineering II
University of Passau, Germany

***Abstract*—Meta-heuristic search algorithms such as genetic algorithms have been applied successfully to generate unit tests, but typically take long to produce reasonable results, achieve sub-optimal code coverage, and have large variance due to their stochastic nature. Parallel genetic algorithms have been shown to be an effective improvement over sequential algorithms in many domains, but have seen little exploration in the context of unit test generation to date. In this paper, we describe a parallelised version of the many-objective sorting algorithm (MOSA) for test generation. Through the use of island models, where individuals can migrate between independently evolving populations, this algorithm not only reduces the necessary search time, but produces overall better results. Experiments with an implementation of parallel MOSA on the EvoSuite test generation tool using a large corpus of complex open source Java classes confirm that the parallelised MOSA algorithm achieves on average 84% code coverage, compared to 79% achieved by a standard sequential version.**

***Index Terms*—unit test generation, search-based test generation, parallel genetic algorithm, EvoSuite**

I. Introduction

Automated test generation techniques support testers and developers in the difficult and tedious task of selecting effective test cases. Meta-heuristic search algorithms, for example Genetic Algorithms (GAs), have shown promise in the context of unit test generation for object-oriented software. These algorithms are typically used in order to generate test suites that maximise code coverage while minimising the number of test cases generated. Even though these approaches have been shown to achieve good levels of code coverage, they are far from optimal. In particular, achieving high levels of code coverage requires running the search for impractically long time, and the stochastic nature means that the quality of the results varies between individual runs.

A promising approach to improve evolutionary search algorithms is parallelisation: The common availability of multi-core processors means that computationally expensive aspects of the search algorithms can immediately be parallelised. However, parallelisation of GAs goes even further than this: By independently evolving separate sub-populations in parallel in island models, and occasionally allowing migration of candidate solutions between the sub-populations, parallel GAs can achieve overall better results, not just cut down time. To date, there has been very little exploration of parallel GAs in the context of unit test generation. It is not obvious that results of parallel GAs from other domains can be carried over to test generation, since test generation uses specialised search algorithms, such as the many-objective sorting algorithm (MOSA) [21]. Thus, there is the necessity to investigate whether parallel GAs can be used to improve the performance and applicability of search-based test generation.

In this paper, we describe a parallelised version of the many-objective sorting algorithm (MOSA) for test generation. MOSA is the state-of-the-art search algorithm for unit test generation. It casts test generation as a many-objective search problem, where each coverage goal (e.g., branch in the source code) is represented as a distinct optimisation goal, and a population of unit tests (i.e., sequences of API calls) is evolved to cover as many goals as possible. The parallelisation consists of creating an island model, where multiple independent sub-populations of MOSA evolve independently. The sub-populations are linked by a scarce migration of individuals, where some individuals are selected from one population, and sent to another one, chosen based on a ring topology. The migration needs to be carefully balanced in order to avoid premature convergence of the sub-populations and to maintain the benefits of parallel evolution. Migrants are integrated into the populations when MOSA decides on individuals for the next generation, and the final test suite is created from a combination of all sub-populations.

We have implemented this parallel model in the EvoSuite test generator [13], which provides a sequential implementation of MOSA. To fully de-couple the evolution of the sub-populations aside of migration, our implementation of parallel MOSA executes each island on a separate process. This has the additional benefit that it overcomes inherent technical challenges of Java dependency handling, and accommodates for the computational needs of independent evolution. Furthermore, this approach seems particularly suitable in the face of ubiquitously available multi-core architectures.

In detail, the contributions of this paper are:

- A parallel extension of the MOSA algorithm (pMOSA) using an island model and migration strategy.
- An implementation of the parallel MOSA variant in the EvoSuite test generation tool.
- A detailed tuning study to determine the optimal parameters of the parallel MOSA.
- An empirical evaluation of the effects and improvements caused by the parallelisation.

We have applied our implementation of pMOSA to a large corpus of complex open source Java classes. Our experiments confirm that the parallelised MOSA algorithm achieves higher code coverage than the sequential version. While running MOSA with a single client and a single population achieves

an average code coverage of 79%, increasing the number of parallel evolving sub-populations to eight increases coverage to 84%. To some extent, the coverage increase can be attributed to the increased resources, but our experiments confirm that the migration strategy also positively influences coverage.

II. Background

A. Search-based Unit Test Generation

A common objective in automated test generation is code coverage; search-based testing is well suited for optimising test suites that achieve the highest possible code coverage. A code coverage criterion can be interpreted as a set of distinct coverage goals, such as branches in the program control flow. For each such coverage goal it is possible to derive an estimate of how close a given program execution is to covering it. This estimate serves as fitness function during a search for test cases.

Most coverage-oriented fitness function are based on the approach level and branch distance metrics [19], [25]. The approach level approximates the distance between the execution path of a test case t and a target execution covering a coverage goal $x \in X$ (for any given set of coverage goals X, defined by a coverage criterion). The approach level $\mathcal{A}(t, x)$ is defined as the minimal number of control dependent edges in the control dependency graph between the target goal x and the control flow path represented by the test case t. The branch distance $d(t, x)$ heuristically quantifies how far a branch in the control flow graph is from being evaluated to true or to false. The overall fitness function is usually a combination of approach level and branch distance, and can be further adapted for specific coverage criteria. For example, for branch coverage the fitness function to minimise the approach level and branch distance between a test t and a branch coverage goal x is defined as:

$$f(t, x) = \mathcal{A}(t, x) + \nu(d(t, x)) \quad (1)$$

where ν is any normalising function in the range $[0, 1]$ [3].

More recently, there is a trend to optimise for multiple coverage criteria at the same time [23]. For example, the EvoSuite test generation tool by default now optimises for (1) method coverage, (2) method coverage without exceptions, (3) line coverage, (4) branch coverage, (5) branch coverage with method direct calls, (6) exception coverage, (7) weak mutation coverage, and (8) output coverage.

In unit test generation, a test case is a sequence of calls on instances of a class under test, and coverage (and thus fitness) is measured on this class. The representation of a unit test for search-based generation consequently is a variable length sequence of calls, where individual statements in the sequence can define values that other statements use (e.g., as parameters). Mutation consists of inserting, deleting, or changing some of these calls; crossover consists of cutting and merging two sequences, while maintaining the validity of value dependencies. Further details of the search operators can be found in the literature [13].

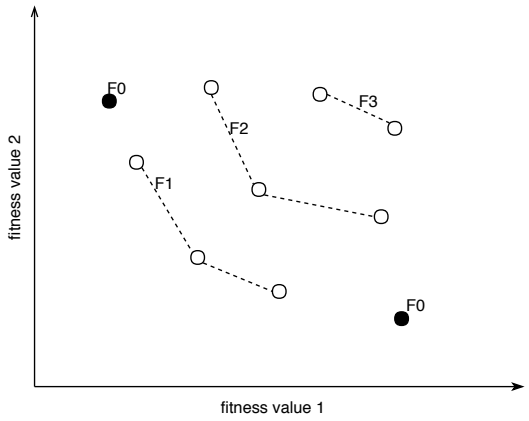

Figure 1: Ranking of individuals by their vector of fitness values into non-dominating fronts $F0$-$F3$.

To avoid the challenges of selecting an order in which to handle individual coverage goals and deciding how much of the overall available resources to invest in each goal, whole test suite optimisation was introduced, where a set of test cases is optimised for all coverage goals at the same time. More recently, however, it has been shown that the reformulation as a many-objective search problem is the most effective way to generate test suites [21].

B. Many-Objective Sorting Algorithm MOSA

The many-objective approach considers each coverage goal as an individual objective. Since there is now more than one fitness value for one individual, each individual gets assigned a fitness vector of m values for m objectives. To compare the fitness vectors and to find the individuals with the best vectors, MOSA uses the method of *Pareto dominance* and *Pareto optimality* [11]. As the fitness values should be minimised, a test case x dominates another test case y if and only if there exists at least one objective function whose value is smaller using x than using y. Assuming that all other objective functions do not have a greater value when using x.

Figure 1 shows a set of individuals with a vector of values of two objectives each. The individuals are arranged according to their fitness vector. The *Pareto dominance* arranges the individuals into subsets of non-dominating fronts where the individuals from different fronts dominate the other but the individuals of the same front are indifferent. The individuals assigned to $F0$ and $F1$ create the first front. They are the *Pareto optimum* of this set, as there is no individual with a better fitness vector according to the defined dominance criterion.

An additional preference criterion is introduced to focus the search effort on individuals that are closer to uncovered fitness goals. A test case x is preferred over another test case y if and only if the values of the objective function for a coverage goal b_i are smaller for x than for y. This preference creates a subset of the first front which is depicted as the set $F0$ in Figure 1. As the algorithm searches for individuals that contribute to maximise the total coverage, these individuals of

Algorithm 1 Main evolutionary loop of MOSA

Input: Population size n, Stopping condition C
Output: Archive of fittest individuals A

```
1:  iteration ← 0
2:  P ← initialise with random population size n
3:  A ← update with P
4:  while ¬C do
5:      P_next ← breed next generation
6:      P_next ← P_next ∪ P
7:      R ← preference sorting of P_next
8:      P = { }
9:      P ← select best n from P_next with ranking R
10:     A ← update with A ∪ P
11:     iteration ← iteration + 1
12: end while
13: return A
```

$F0$ are possible candidates. A secondary preference criterion is the test case length to keep the test cases as short as possible.

Algorithm 1 shows the main steps of the algorithm. MOSA starts with an initial population of randomly generated test cases. The population evolves through succeeding iterations by producing a new population or generation every iteration. Individuals for a new population are chosen from the set of current population combined with the set of offspring which are generated by combining two selected test cases using crossover and mutations.

To decide which test cases are selected for the new generation, a ranking for the combined test cases is calculated using the previously described *Pareto dominance* and the preference criteria. The algorithm only creates a total of two fronts. The test cases that are chosen for the first front $F0$ have the lowest objective score for each uncovered fitness goals and are given the highest chance of being selected for the next generation by assigning them to rank 0. The remaining test cases are ranked according to the traditional sorting algorithm used by NSGA-II [10] starting with rank 1. The new population is filled first with the test cases from $F0$, then with the remaining test cases respecting the assigned rank, if the configured population size is not yet achieved.

MOSA uses an archive [24] to store the best and shortest test cases that cover the fitness goals of the software under test. After each iteration the archive includes test cases that cover previously uncovered fitness goals and exchanges an already stored test case with another if the new one is shorter. This archive forms the resulting test suite when the algorithm has terminated.

C. Parallel Genetic Algorithms

There are several ways to parallelise a genetic algorithm. Cantú-Paz describes a categorisation [9] that covers the basic possibilities. Firstly, the algorithms can be distinguished by the number of populations involved. Consequently, the three main types of parallel genetic algorithms are: single population master-slave, single population fine-grained, and multiple population coarse-grained genetic algorithms. There is also the possibility of hybrids of these three general models.

A master-slave genetic algorithm consists of a single population. The master contains the population and executes all operations of the genetic algorithm. As crossover and selection consider the entire population these operations need to be done globally in the master. The fitness evaluation considers each individual on its own which is why this step can be executed in parallel. Each slave gets assigned a few individuals for which it calculates the fitness values. This can be an improvement for a genetic algorithm when the calculation of the fitness function is computation-intensive.

The second type is a fine-grained genetic algorithm where all evolutionary steps are parallelised. It also uses a single population but it is spatially structured into small groups of individuals. Every group has a fixed set of neighbouring groups with which it shares a few individuals that lie in the edge area to create overlapping regions. Selection and crossover are restricted to the individuals within the group with the only interaction to other groups coming from the overlap. This type is best suited for massively parallel systems, e.g., in grid computing.

A coarse-grained genetic algorithm consists of several sub-populations which exchange individuals occasionally, which is called migration. Selection and crossover individually take place in the different sub-populations. This form of parallelisation is often called island model which has its origins in population genetics.

Another important difference between the types is whether the parallelisation introduces changes to the genetic algorithm. Master-slave genetic algorithms do not change the behaviour of the original genetic algorithm, as they always consider the whole population and do not restrict the individuals in their choice of a crossover partner. In fine- or coarse-grained genetic algorithms selection and crossover is restricted to a subset of the individuals.

D. Island Model – Coarse-Grained Parallelisation

The island model mimics the natural evolution by distributing a large population among a number of semi-isolated sub-populations [2]. Each island maintains its own population and in each island all genetic operators (selection, mutation, and crossover) are performed locally [18]. All island populations are completely isolated which means that each island can search in its own local search space and can develop in different directions with respect to the other populations. Additionally the islands can start their search in different parts of the search space. This helps covering more of the possible global search space and preserves genetic diversity [27].

Occasionally, migration occurs where a few selected individuals, called immigrants, are sent to a selected neighbouring island. There are two main types of migration: blocking and non-blocking. Both forms are initiated at a predefined constant interval, but blocking migration waits for the receiver to accept the individuals whereas non-blocking continues with the next task [1]. Figure 2 shows the island populations with their

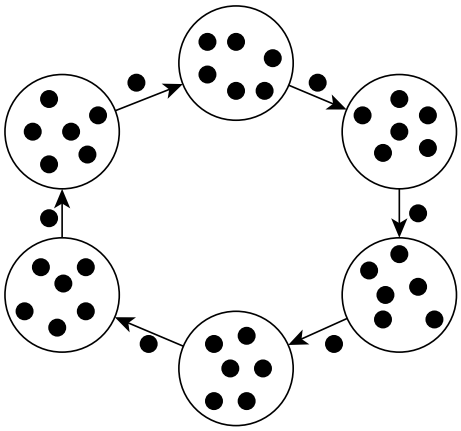

Figure 2: Schematic depiction of the island model; Each island maintains its own population while individuals can migrate to a neighbouring island.

respective sets of individuals and the communication of the immigrants. The displayed communication channels in this figure are logical and can be mapped to a physical network, for example a ring or a hypercube topology, as a part of the migration strategy. After termination the best set of individuals ever found is the resulting end population and the solution for the parallel genetic algorithm [2].

When using this model there is always a fine line between premature convergence and complete isolation of the sub-populations [16]. On one hand, if migrations are too frequent the risk of hitting a local optimum is increased. On the other hand, if the migration occurs too rarely the population might drift aimlessly without converging [8]. A migration policy that is too aggressive, by producing too many immigrants, having an excessive frequency or selecting immigrants that are too "good", could negatively affect the solution quality by converging too fast [8].

E. Migration Parameters

The island model is affected by several parameters due to the concept of migration. As simple as it is to integrate the model into a genetic algorithm, it is difficult to find an optimal migration strategy that suits the problem due to the number of possible parameter combinations [9]. For instance, the topology that defines the communication and migration from one island to another, the migration rate that controls the amount of immigrants, the migration frequency that defines the interval between migrations, etc.

The communication topology defines the neighbours an island can communicate with. Previous work [2], [9], [16] states that ring and hypercube topologies are the most commonly used static topologies. The alternative is using dynamic topologies which select the receiving island anew for every communication. The simplest dynamic topology is choosing a random receiver for every communication. The topology influences the number of connections between islands and as such the communication costs. Therefore, it is important to choose a topology that allows the islands to develop separate populations but also spreads "good" solutions among the populations [16].

The migration frequency decides on the point in time an exchange of individuals takes place. Its value defines the interval between two of those exchanges [18]. There are two possibilities to define and measure the interval, either using real time or iterations. Using the time as a constant interval is not an option in the context of test generation, because during the evaluation of the individuals some test cases can take longer to finish, e.g., when they wait for a timeout to occur. This entails that the defined moment will be missed or that the same individuals are sent again because the next generation has not been evolved yet. Therefore, the number of iterations (i.e., GA generations) is the chosen measurement for the migration frequency.

The migration rate defines the number of (copies of) individuals that are selected and exchanged every time migration takes place. In theory there is no upper bound, but in practice the migration rate is bound by the population size [1]. This leaves the question of selecting and integrating immigrants. Cantú-Paz showed that the selection strategy has a greater impact on the convergence than the replacement strategy when integrating the new individuals [8]. Selecting the individuals with the highest fitness value as immigrants causes a faster convergence than selecting the immigrants at random or using, for instance, a linear rank selection. The received immigrants can, e.g., replace random individuals or the ones with the worst fitness value. Bravo et al. [6] showed that replacing the worst individuals of a population with the worst ones from the neighbouring island has the best positive influence on the diversity.

III. Parallel Many-Objective Sorting Algorithm for Test Generation

In this section we describe the combination of the island model (see Section II-D) with the MOSA algorithm. In general, each island executes its own instance of the MOSA algorithm and the islands communicate with each other to exchange migrants. To enable communication within MOSA the concept of migration has to be added to the algorithm.

To describe the needed changes to MOSA in more detail, one has to look again at the steps of the algorithm. Algorithm 2 shows MOSA covering the main steps to compute the generations. This adapted version includes the new migration parameters frequency F and rate K, the receiving and integration of immigrants (lines 8 and 9), and the selection and sending of emigrants (lines 14 and 15). At the start of a generation, possibly received immigrant sets are added to the new population. After selecting the best individuals for the next generation, depending on the communication frequency, a set of emigrants is selected to be sent to the neighbouring island. If the islands are arranged in a ring topology (as represented in Figure 2), every island has exactly one neighbour who it receives migrants from and one who it sends migrants to.

The selection of emigrants to send to the neighbour island can be performed with usual selection functions, e.g., selection of the best k individuals, selection of k individuals at random, or

Algorithm 2 MOSA extended with migration

Input: Population size n, Stopping condition C, Migration frequency F, Migration rate K
Output: Archive of fittest individuals A

```
1:  iteration ← 0
2:  immigrants ← from neighbouring island
3:  P ← initialise with random population size n
4:  A ← update with P
5:  while ¬C do
6:      P_next ← breed next generation
7:      P_next ← P_next ∪ P
8:      if immigrants ≠ { } then
9:          P_next ⟵ P_next ∪ immigrants
10:     end if
11:     R ← preference sorting of P_next
12:     P = { }
13:     P ← select best n from P_next with ranking R
14:     if (iteration + 1) mod F == 0 then
15:         emigrate selected K from P
16:     end if
17:     A ← update with A ∪ P
18:     iteration ← iteration + 1
19: end while
20: return A
```

rank selection. Best k selection chooses the k individuals with the highest fitness value from the recently evolved population. Random k selection chooses randomly k individuals from the new population. The rank selection proposed by Whitley [26] takes a random value $r \in [0, 1]$ and computes the index of the selected individual as:

$$\frac{s}{2 \cdot (b-1)} \cdot (b - \sqrt{b^2 - 4 \cdot (b-1) \cdot r})$$

where s represents the population size and $b \in [1, 2]$ the selection pressure. Individuals with a higher fitness value have a higher chance of being selected than individuals with a lower fitness value. Note that individuals can be selected more than once during the migrant selection as both random selection and rank selection depend on the probability of generating a different r value.

After receiving a set of immigrants from the neighbouring island, these individuals have to be integrated in the current population. The immigrants need to be integrated into the system of fronts created by the ranking described in Section II-B. As the replacement strategy has a smaller impact on the genetic algorithm [8], the strategy is fixed to suppressing the individuals with the worst value to keep the variable parameters manageable. The immigrants with a higher fitness value suppress the individuals with a lower fitness by getting sorted in higher ranked fronts.

In general, as already described, after termination the best set of individuals ever found is the resulting end population and the solution for the parallel genetic algorithm [2]. However, when using a many-objective algorithm, one cannot compare sets of individuals that easily because the set as a whole has no assigned fitness value. The only possibility to decide which is the best set of individuals, is to compare all individuals with each other. To address this problem the solution sets from all islands are collected to perform one last evolutionary step. During this step all individuals are compared and the best ones are collected in a final solution. This solution is the end result of the parallelised MOSA.

IV. Implementation

We have implemented the parallelised MOSA algorithm as part of the EvoSuite test generator. EvoSuite is a search-based tool for automatic test suite generation for Java code [12]. It produces JUnit test suites with high code coverage, integrated assertions and a minimal set of tests. EvoSuite implements several algorithms and techniques to generate a test suite for a given Java project such as dynamic symbolic execution [14] and whole test suite generation [13]. Furthermore the many-objective sorting algorithm (MOSA) is implemented in EvoSuite [21].

EvoSuite is separated into different modules of which the master and the client module are the main components participating in the parallelisation. The master module processes the user input, starts a client, monitors the client and collects statistics. The client module performs the instrumentation and the test suite generation using the chosen algorithm. It also handles the post processing tasks, like minimising the test suite or placing assertions, and writes the test suite to disk.

A few changes needed to be introduced to EvoSuite to make parallelisation possible. The first step is to start and maintain more than one client. Since the master module already starts the client process it is its task to manage the new set of clients. The clients are started as a group which means that either all clients are running or all are stopped. To be able to distinguish the clients each gets assigned a unique id which is added to its name. Also parameters like the communication frequency and rate are introduced to EvoSuite.

The next step is to adapt the MOSA implementation to perform communication and integration of selected individuals. To generate a final solution, client 0 takes on a special role to collect all end results of all other clients. An additional iteration in client 0 produces the final end result. Because of this distribution of roles client 0 is the only client post processing the resulting test suite. The same applies for saving generated tests on disk and writing statistics as all other clients only have tests and statistics for an intermediate result.

In the context of parallelising MOSA, asynchronous (non-blocking) migration is used to migrate individuals. The island on the receiving side may be at a different generation than the sending one, because the duration of each iteration depends on the runtime of the generated individuals. To prevent any unnecessary waiting time, the islands do not have a synchronised generation counter. The asynchronous communication requires a buffer that can accept transferred sets of immigrants which will later be fetched by the receiving client or island. To implement this synchronisation approach a listener is introduced

in each client which creates the communication link to the running genetic algorithm. The listener accepts incoming sets of immigrants and stores them in a queue. The elements of the queue will be processed and integrated one by one by the genetic algorithm at the start of a new generation. One could think that a queue is not necessary because older sets can be discarded and the newer set contains fitter individuals. This is true if the best k individuals are selected for migration. But if k individuals are randomly chosen it can not be guaranteed that the newer set has a higher fitness. For that case it might be interesting to evaluate the difference between discarding and integrating older sets.

The implementation of the parallelised MOSA algorithm includes several migrant selection functions: Random k, best k, and linear rank selection. Rank selection was already implemented in EvoSuite, to select the individuals for the next generation of the GA, and it was proposed by Alba and Troya [2] as migrant selection function. EvoSuite provides other selection functions that are only used to select the individuals for the next generation and can be easily used for the selection of migrants. As previously described in Section III, the migrant selection functions return one selected index of an individual in the population at a time. When filling the set of emigrants there is a possibility that indices, which represent the individuals, are selected more than once. The implementation uses the Java `Set<>` to collect the selected emigrants which eliminates duplicates of selected emigrants. It can occur that the set of emigrants is smaller than the configured migration rate.

Since the arriving sets of immigrants are stored in a queue, they can be integrated whenever the algorithm starts a new generation of individuals. The potential individuals for the next generation are a combination of individuals from the previous iteration and newly bred individuals. However, when a new set of immigrants is available, they are included with other candidates to form the next generation. This way the immigrants are ranked with the other individuals which allows them to get sorted into one of the two fronts and assigned a rank.

When the algorithm terminates, client 0 waits for all the other clients to send their end solution (which has been collected as a set of individuals in each client's archive). Note that this is the only synchronisation point of the proposed parallelised MOSA algorithm which forces client 0 to need at least as much time as the slowest client. Once all clients have sent their end solutions, client 0 combines all received solutions and recomputes the fitness value of each individual. Individuals that do not cover previously uncovered goals or are not better (e.g., shorter) than the ones in client 0's archive are discarded, otherwise they are added to client 0's archive. In the end, the updated archive of client 0 contains the final result of the parallel run of MOSA.

V. Evaluation

The empirical evaluation focuses on two main goals. As the parallelisation introduces new parameters for the communication between islands and the selection of immigrants, the first goal is to find a combination of parameters that maximise the achieved results. As a second goal the resulting coverage of the MOSA algorithm is compared to the parallelised version of MOSA to assess if the parallel MOSA presents an improvement or not. Consequently, we aim to answer the following research questions:

RQ1: Does parallel MOSA improve over sequential MOSA?
RQ2: Does migration contribute to better performance?
RQ3: Does parallelisation reduce the overall runtime to achieve coverage?

A. Experimental Setup

1) Study Objects: For the empirical evaluation, we used the set of 346 Java classes used to evaluate MOSA [21] and its variant DynaMOSA [22]. These classes have been selected with the goal to be as diverse as possible, and to cover different levels of complexity and functionality [21].

2) Methodology: We conducted two empirical experiments: First we performed a tuning experiment to find the best combination of migration parameters; then, we conducted an experiment comparing the tuned parallel MOSA with the sequential MOSA. A randomly selected subset of 34 classes (10% of all classes) was used to conduct the tuning experiments which have the goal to find the best combination of migration parameters. The rest of the classes were used for the comparison of the parallel and the sequential MOSA. We only applied tuning to the parameters introduced for the parallelisation; for all other parameters we relied on EvoSuite's default values, which were empirically determined in previous tuning studies [5], and some MOSA-specific parameters that were determined in a more recent study [7], in particular a population size of 25 individuals.

For the tuning experiments, the chosen sets of values for the migration parameters are: number of clients $= \{2, 4, 8\}$, migration frequency $= \{1, 5, 10, 25\}$, migration rate $= \{1, 5, 10, 15, 20, 25\}$, and migrant selection function $= \{randomk, bestk, rank\}$. For each class all combinations that emerge from the cross product of these sets are run to collect the metric. Additionally, each class was executed with different number of clients and no communication to analyse if communication makes a difference or if an improvement just results from the use of multiple populations. To identify the "best" parameter settings for the parallel MOSA, we performed a pairwise comparison of the overall coverage achieved by using any number of clients, migration frequency, rate, and selection function. The configuration for which the parallel MOSA achieved a significantly higher overall coverage more often was selected as the best.

After the tuning study, we ran parallel MOSA using the optimal parameters, as well as non-parallel MOSA, on the remaining classes. In order to answer RQ2, we also ran parallel MOSA with migration deactivated, but all other parameters unchanged. For all experiments we used a search budget of 60 seconds, similar to previous experiments [7]. However, to answer RQ3 we additionally ran sequential MOSA with different search budgets (60s – 180s).

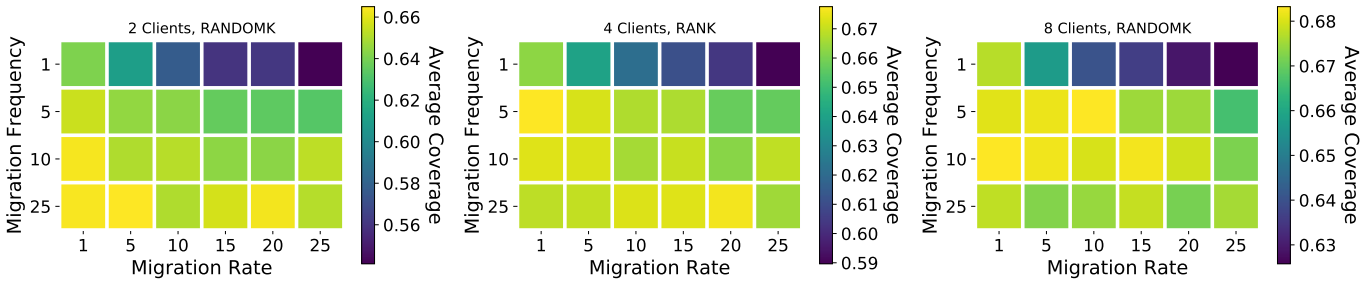

Figure 3: Tuning results: For two clients, random selection of 20 individuals for migration every 25 generations achieves the overall best coverage; for four clients, rank selection of one individual every 5 generations achieves the overall best coverage; for eight clients random selection of one individual every 10 generations achieves the overall best coverage. In general, lower migration rates are better, and the frequency depends on the number of clients, with larger numbers of clients requiring more frequent migration.

As is common when evaluating randomised algorithms, each individual configuration was repeated 30 times with different seeds for the random number generator.

3) Analysis: To compare different parameter settings and the parallel MOSA vs. the sequential MOSA, we used coverage as a measure. EvoSuite uses a combination of seven coverage criteria (line, branch, weak mutation, output, method, method without exceptions considering only normal behaviour and context, and branch coverage), and coverage values refer to the average over these criteria [23].

Using raw coverage values for parameter setting comparisons would be noisy, since most branches are always covered regardless of the chosen parameter setting, while many others are simply infeasible. We therefore use *relative coverage* r calculated using the following normalisation [5], given the coverage c achieved in an execution of EvoSuite for a class under test CUT:

$$r(c, CUT) = \frac{c - min_c}{max_c - min_c} ,$$

where min_c is the worst coverage obtain in *all* the experiments for that class CUT, and max_c is the maximum coverage obtained in all experiments. If $min_c == max_c$, then $r = 1$.

We statistically analysed all results following standard guidelines [4]. Effect sizes were measured using the Vargha-Delaney $\hat{A}_{12}$ effect size, and statistical tests were performed using the Wilcoxon-Mann-Whitney U-test with a 95% confidence level.

4) Threats to Validity: There are factors that might be threats to the *internal validity*. Although the framework and the new implementation was tested thoroughly, the risk of leftover faults remains. Additionally the algorithm depends on randomness which can lead to unexpected outliers. To deal with this all experiment executions were repeated 30 times and the results were statistically analysed.

Threats to *construct validity* come from what measure we chose to evaluate the success of our techniques. To measure improvements on testing effectiveness, we considered the achieved coverage, using EvoSuite's default set of coverage criteria. However, code coverage does not tell us how easy it will be for the final user to understand the generated test cases, and the link between fault detection ability and code coverage is an ongoing point of discussion in the research community.

Threats to *external validity* regard the generalisation of the experiment results. Although the classes that served as experiment subjects where chosen with the goal to introduce a degree of diversity, further experiments on a larger set of classes would improve the generalisation of the achieved results. Aside from that can the chosen parameters from the tuning experiment potentially threaten a generalisation. Different parameters for the genetic algorithm could lead to different results. The results for the migration parameters from the tuning experiment might only be valid for the chosen set of parameters of the genetic algorithm.

B. Parameter Tuning

There are previous studies on how communication parameters like frequency, rate or migrant selection influence a genetic algorithm in general, or in a specific application. For example, Luque and Alba [18] formulated models that approximate the influence of these parameters on a parallel genetic algorithm, and other studies [1], [6], [8] focus on evaluating the communication or migration parameters. Unfortunately there are no experiences yet with the combination of parallel genetic algorithms and test generation, which is why these parameters need further investigation in the context of test generation.

According to our tuning study, for 2 and 8 clients, the overall best selection strategy is random selection; for 4 clients the overall best selection strategy is rank selection. Figure 3 visualises the results of the tuning experiments for these selection strategies as heatmaps, where the coverage for each parameter combination is indicated by the shading of the cell. Generally, if migration is done too frequently this negatively affects code coverage, and in particular the first row of the heatmap, where migration is done for each generation, shows the lowest code coverage. Similarly, lower migration rate generally seems to be better, and so the lowest code coverage can be found in the upper right parts of the heatmaps. Comparing the three heatmaps suggests that migration should be done more frequently the more sub-populations there are;

Figure 4: Average code coverage achieved using different numbers of sub-populations.

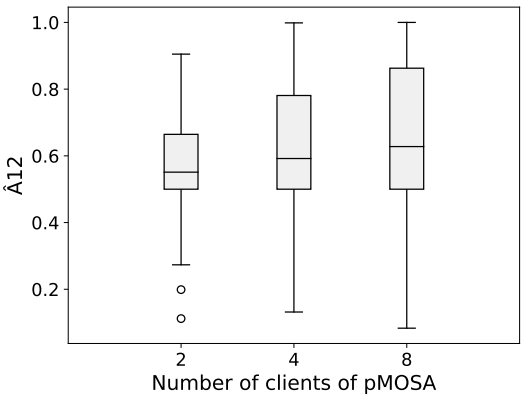

Figure 5: $\hat{A}_{12}$ effect size for the comparison between single-population MOSA vs. different numbers of sub-populations.

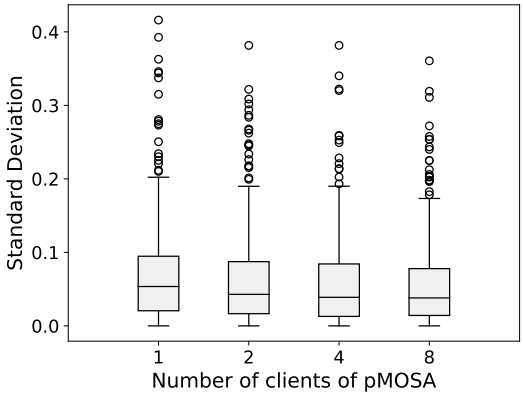

Figure 6: Standard deviation using different numbers of sub-populations.

for 8 clients the best configurations are with migration rates of 5–10, whereas for 2 clients the best configuration performs migration only every 25 configurations.

C. RQ1: Improvement over sequential MOSA

Figure 4 shows the average code coverage achieved by MOSA and the parallelised versions, using 2, 4, and 8 clients. The plot suggests a steady increase of code coverage with increasing number of clients (average coverage of 79% for MOSA vs. 81%, 82%, and 84% for parallelised MOSA). Statistical analysis confirms this trend: Figure 5 summarises the $\hat{A}_{12}$ effect sizes comparing standard MOSA and parallelised MOSA; values of $\hat{A}_{12} > 0.5$ mean that the parallelised version was better. Using 2 clients, pMOSA achieved significantly higher coverage on 80 classes; with 4 clients, pMOSA achieved significantly higher coverage on 133 classes; and with 8 clients pMOSA achieved significantly higher coverage on 142 classes. This increase is substantial, considering that MOSA already achieves very high (and often optimal) code coverage on many of the classes.

Overall, pMOSA with eight clients was significantly worse on 21 classes, compared to standard MOSA. Although for all the 21 classes the reduction in coverage is very small ($<$ 1%), we nevertheless investigated these classes in detail to understand the reasons for this reduction. The main reason seems to be problems with memory consumption when using multiple processes; indeed multiple Java processes may lead to substantial memory consumption, and may thus slow down execution. Another reason is that migration may have a negative impact on the search time, depending on how suitable the configuration is for the class under test; for example, we observed a reduction of the number of generations of the GA run by up to half on some classes. Consequently, one way to improve parallelisation of MOSA might be to *adaptively* select migration frequency and rate depending on the specifics of the class under test.

To see whether the parallelisation not only increases coverage, but also reduces the variation in the results that is common with stochastic algorithms, Figure 6 shows the standard deviation for each of the configurations. Although the difference is small, there is a decrease: Whereas default MOSA has a standard deviation of 7.2%, this reduces with increasing number of clients to 5.9% for eight clients.

> ***RQ1:*** *In our experiments, the average code coverage increased from 79% to 84% using parallelisation with eight clients with one minute search time.*

D. RQ2: Influence of migration

To understand whether the code coverage increase observed for RQ1 is caused simply by the increased computational time by using multiple processes, or whether the island model contributes beyond the basic parallelisation, we compared each of the parallel configurations with a configuration of MOSA using the same number of clients, but no migration. Figure 7 shows the effects on coverage, and Figure 8 summarises the effects using the $\hat{A}_{12}$ effect size measurement. The coverage values in Figure 7 suggest that migration achieves slightly better coverage depending on the number of clients: For two clients, migration makes almost no difference, with average effect size

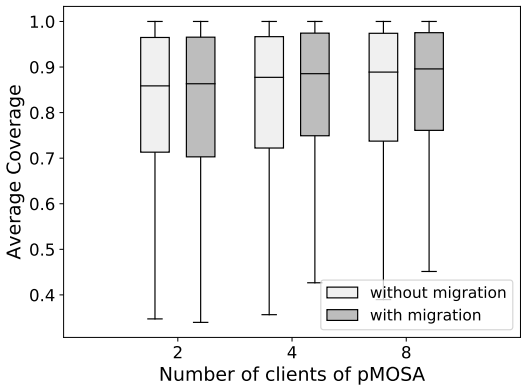

Figure 7: Average code coverage with/without migration.

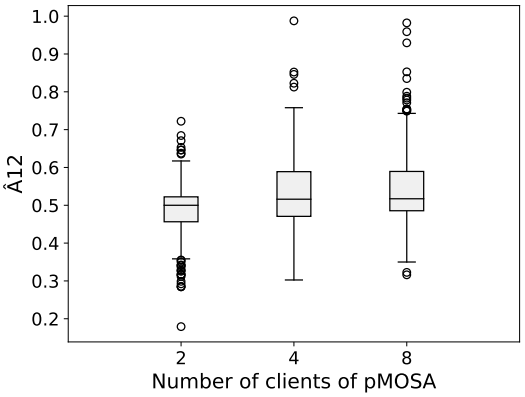

Figure 8: $\hat{A}_{12}$ effect size comparing parallel runs with different numbers of sub-populations with/without migration.

of 0.49 (suggesting there even are cases where it is worse), for four clients migration leads to significantly higher coverage on 39 classes (average $\hat{A}_{12}$ = 0.53), and for eight clients migration leads to significantly higher coverage on 39 classes (average $\hat{A}_{12}$ = 0.54). Overall, these are small effect sizes, suggesting that the improvement over MOSA observed in RQ1 to some extent is a result of the multiplication of computational time, but it is additionally improved by migration. It is not surprising that migration has a larger effect for larger numbers of clients, since it is done less frequently with fewer clients (e.g., only every 25 generations for 2 clients vs. every 10 generations for 8 clients).

RQ2: *Our experiments confirmed that migration has a positive effect on code coverage; this effect is larger the more sub-populations are independently evolved.*

E. RQ3: Runtime reduction

Although the main incentive for applying parallelisation in the context of test generation is to increase the resulting code coverage, a similar objective from a practical point of view would be to reduce the overall test generation time without affecting coverage. To see how the effects of the parallelisation compare to the runtime of a purely sequential, single-population algorithm, Figure 9 shows the $\hat{A}_{12}$ values comparing pMOSA with 8 clients run for 60 seconds with default MOSA with different values for the search budget. Up to 80 seconds, all parallelised versions of MOSA run for 60 seconds achieve the same or higher coverage than MOSA. After more than twice the runtime of the parallelised MOSA with 8 clients does the default configuration catch up, and median $\hat{A}_{12}$ suggests that after around 150s the default configuration achieves higher coverage.

While the parallelisation does make it possible to reduce the overall runtime, what we can see from Figure 9 is that the parallelisation does not lead to a 1:1 mapping of net computational time: Running pMOSA for 60 seconds with two clients does not achieve the same result as running MOSA with a single client for 2 $\times$ 60 seconds. Consequently, while parallelisation can serve as a boost to regular evolution, it cannot *replace* regular evolution. Our conjecture is that the evolutionary search in EvoSuite will initially spend some time with an initial exploration where test cases grow in length, e.g., until all methods are called and all the easy branches are covered, etc. Using our parallelisation model, this initial exploration needs to be done by each of the clients, meaning that there is some redundancy before the parallelisation can deliver some net benefits. On one hand, this redundancy is necessary since it leads to the higher diversity, which ultimately leads to higher coverage. Consequently, an interesting direction for follow-up experiments would be to run pMOSA with larger search budgets, and study whether the net benefits over sequential MOSA increase over time. On the other hand, it is conceivable that the parallelisation could be improved by making it adaptive to the runtime and coverage achieved. For example, until a kind of equilibrium coverage state is reached, maybe fewer parallel clients might be needed.

RQ3: *Our experiments show that parallelisation can be used to reduce the overall runtime of the search, but cannot completely replace the sequential effects of evolution.*

VI. Related Work

While both, research on search-based test generation and research on parallel genetic algorithms, are quite mature, there is not much existing work on using parallel search-based approaches for automated test case generation.

Whitley et al. [27] introduced the island model and its parameters, and Alba and Troya [1] and Cantú-Paz [8] analysed migration for this model further. Other studies [2], [9], [16] provide an overview over the different parallel models for genetic algorithms, specifically for island models. A modified island model was introduced by Kurdi [17]: Each island uses different evolution methods, for example for selection, instead of all islands using the same methods like in the traditional island model. An additional modification is that individuals that are the least adapted to the environment migrate first. The proposed model is described as a more realistic model of the nature. In our experiments we used a classical island model, where each island is evolved using the same parameters.

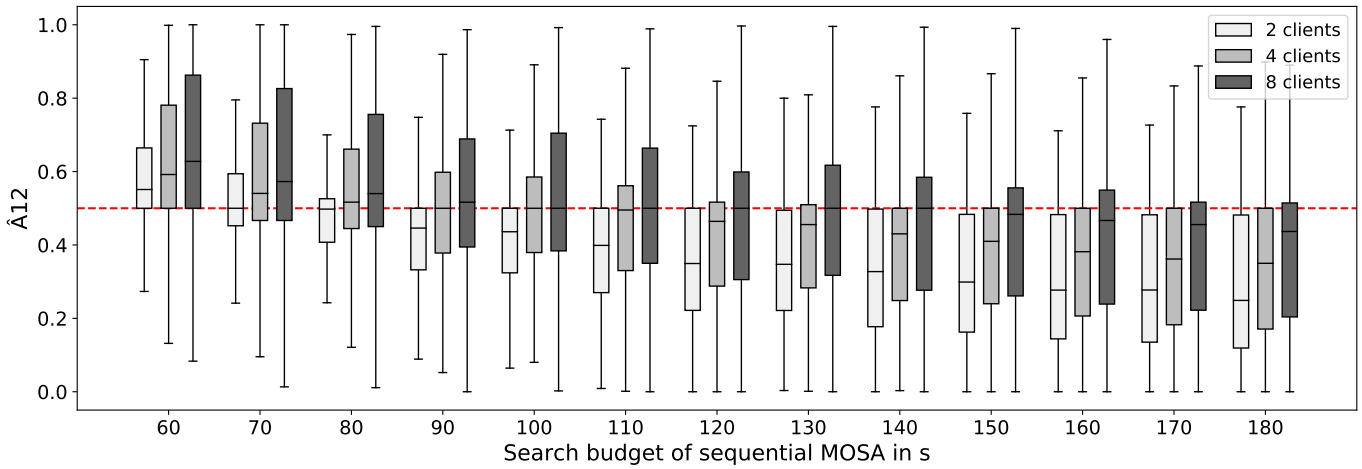

Figure 9: $\hat{A}_{12}$ effect size comparing parallelised MOSA run for 60 seconds with default MOSA run for different search budgets.

However, it is conceivable that an improvement is possible by using different evolution strategies on different clients, such as, using different coverage criteria for different islands.

There exist a few parallel search-based models for automated test case generation. The work of Geronimo et al. [15] focuses on a faster execution of a genetic algorithm for test case generation. Their parallel genetic algorithm is implemented by using Hadoop MapReduce. The model is similar to a classical master-slave genetic algorithm. The time consuming fitness evaluation is parallelised with the MapReduce model while all other parts of the genetic algorithm are executed globally. In contrast, our approach uses a coarse-grained parallelisation that does not simply aim to reduce run-time, but to also improve the results of the evolution.

Pachauri and Srivasatava [20] created a parallel hybrid master-slave model for test data generation for branch coverage. The model is structured in slaves that run a genetic algorithm respectively and a master that selects the branches that need to be covered. The master uses an extended path prefix strategy to select the branches. The branches are distributed to the slaves where a search to cover the assigned branches is conducted. Similar to the previous study, the main goal is to speed up the test generation process. While our approach uses migration between the sub-populations among others to improve diversity, the slaves in this model cannot communicate with each other.

VII. Conclusions

Improving search-based test generation algorithms is an important step towards achieving better software quality. Traditionally, search-based testing applies standard genetic algorithms, in which a single population is evolved sequentially. In this paper, we considered the parallel evolution of multiple populations by extending the MOSA search algorithm in the EvoSuite test generation tool. Experiments on a set of complex Java classes confirms that the parallel evolution leads to higher coverage, and is supported by models where individuals migrate between independent islands.

While our findings demonstrate the feasibility and benefits of parallel search, there is ample opportunity to further improve the algorithms and coverage of the test suites they produce:

- Our current implementation uses a ring-topology; as part of our future work we plan to implement and analyse different topologies, such as hypercubes or random assignment.
- In our experiments we assumed a 1:1 mapping of islands to CPU cores, to enable true parallelisation of the evolution on independent islands. As part of our future work, we plan to evaluate whether island models are beneficial in a sequential setting, where evolution steps for sub-populations would be performed sequentially.
- In our experiments, all islands evolved tests for the same algorithm, with the same parameters. However, it is conceivable that each island applies different search algorithms or strategies [17].
- We parallelised the MOSA algorithm [21], which was originally introduced to optimise for branch coverage only; however, recently the DynaMOSA [22] algorithm was introduced as an extension that can handle the larger number of coverage criteria and objectives used in practice [23]. In particular, there is opportunity to further improve MOSA/DynaMOSA by configuring different islands to optimise for different criteria. For example, one island might optimise for branch coverage, while the other might optimise for mutation testing, thus further increasing diversity.

The proposed parallelised MOSA algorithm has been integrated in EvoSuite and it is freely available for download at `https://github.com/EvoSuite/evosuite`.

References

[1] E. Alba and J. M. Troya, "Influence of the migration policy in parallel distributed gas with structured and panmictic populations," *Applied Intelligence*, vol. 12, no. 3, pp. 163–181, May 2000.

[2] ——, "A survey of parallel distributed genetic algorithms," *Complexity*, vol. 4, no. 4, pp. 31–52, Mar 1999.

[3] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability (STVR)*, vol. 23, no. 2, pp. 119–147, 2013.

[4] A. Arcuri and L. Briand, "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering," *Software Testing, Verification & Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, May 2014.

[5] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, Jun 2013.

[6] Y. Bravo, G. Luque, and E. Alba, "Migrants selection and replacement in distributed evolutionary algorithms for dynamic optimization," in *Distributed Computing and Artificial Intelligence*. Springer International Publishing, 2013, pp. 155–162.

[7] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An Empirical Evaluation of Evolutionary Algorithms for Unit Test Suite Generation," *Information and Software Technology*, 2018.

[8] E. Cantú-Paz, "Migration policies, selection pressure, and parallel evolutionary algorithms," *Journal of Heuristics*, vol. 7, no. 4, pp. 311–334, Jul 2001.

[9] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, no. 2, pp. 141–171, 1998.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Trans. Evol. Comp*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[11] K. Deb, *Multi-Objective Optimization*. Boston, MA: Springer US, 2005, pp. 273–316.

[12] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, Sept 2011, pp. 416–419.

[13] ——, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[14] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 360–369.

[15] L. D. Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Apr 2012, pp. 785–793.

[16] D. S. Knysh and V. M. Kureichik, "Parallel genetic algorithms: a survey and problem state of the art," *Journal of Computer and Systems Sciences International*, vol. 49, no. 4, pp. 579–589, Aug 2010.

[17] M. Kurdi, "An effective new island model genetic algorithm for job shop scheduling problem," *Computers and Operations Research*, vol. 67, pp. 132 – 142, 2016.

[18] G. Luque and E. Alba, *Parallel Genetic Algorithms: Theory and Real World Applications*. Springer Publishing Company, Incorporated, 2013.

[19] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun 2004.

[20] A. Pachauri and G. Srivasatava, "Towards a parallel approach for test data generation for branch coverage with genetic algorithm using the extended path prefix strategy," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, Mar 2015, pp. 1786–1792.

[21] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Apr 2015, pp. 1–10.

[22] ——, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, Feb 2018.

[23] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation," in *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*, M. Barros and Y. Labiche, Eds. Cham: Springer International Publishing, 2015, pp. 93–108.

[24] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A Detailed Investigation of the Effectiveness of Whole Test Suite Generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, Apr. 2017. [Online]. Available: https://doi.org/10.1007/s10664-015-9424-2

[25] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[26] D. Whitley, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," in *Proceedings of the 3rd International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1989, pp. 116–121.

[27] D. Whitley, S. Rana, and R. Heckendorn, "The island model genetic algorithm: On separability, population size and convergence," vol. 7, Dec 1998.