Journal of Software Engineering Research and Development, 2021, 9:4, doi: 10.5753/jserd.2021.816

An Approach for the Generation of Multi-Objective Algorithms Applied to the Integration and Test Order Problem

Giovani Guizzo [**University College London** | ***g.guizzo@ucl.ac.uk***]
Thainá Mariani [**Federal University of Paraná** | ***marianithaina@gmail.com***]
Silvia Regina Vergilio [**Federal University of Paraná** | ***silvia@inf.ufpr.br***]
Aurora Pozo [**Federal University of Paraná** | ***aurora@inf.ufpr.br***]

Abstract

Multi-Objective Evolutionary Algorithms (MOEAs) have been successfully applied to solve hard real software engineering problems. However, to choose and design a MOEA is considered a difficult task, since there are several parameters and components to be configured. These aspects directly impact the generated solutions and the performance of MOEAs. In this sense, this paper proposes an approach for the automatic generation of MOEAs applied to the Integration and Test Order (ITO) problem. Such a problem refers to the generation of optimal sequences of units for integration testing. The approach includes a set of parameters and components of different MOEAs, and is implemented with two design algorithms: Grammatical Evolution (GE) and Iterated Racing (`irace`). Evaluation results are presented, comparing the MOEAs generated by both design algorithms. Furthermore, the generated MOEAs are compared to two well-known MOEAs used in the literature to solve the ITO problem. Results show that the MOEAs generated with GE and `irace` perform similarly, and both outperform traditional MOEAs. The approach can reduce efforts spent to design and configure MOEAs, and serves as basis for implementing solutions to other software engineering problems.

Keywords: *Multi-Objective Evolutionary Optimization, Grammatical Evolution, Iterated Racing, Software Testing*

1 Introduction

The testing activity is usually performed in different phases. In the unit testing phase, each module (unit) is individually tested. After this, in the integration phase, the units are combined and tested in order to identify software design faults related to the interaction of units. In many cases, there are dependency relations between the units, that is, to test a unit A, another unit B is required. When dependency cycles among units exist it is necessary to break the cycle and to construct a stub for B, if B is not available. The stubbing process may be an expensive and error prone task. Hence, to reduce stubbing costs, it is very important to determine the best sequence of units to be tested and integrated. Additionally, there are some factors to take into consideration, such as the number of required methods, attributes, and parameters to be emulated in the stub, besides other ones associated with the software development (Assunção et al., 2014), making this task a multi-objective problem that cannot be easily solved by the tester in a short time. Such problem is known in the literature as Integration and Testing Order problem (ITO) and appears in different contexts such as, component based development, object-oriented development, aspect-oriented development, and software product line engineering.

The ITO problem has been investigated in the SBSE (Search-Based Software Engineering) field (Harman et al., 2012) by using different search-based algorithms (Wang et al., 2011). The most promising ones are the Multi-Objective Evolutionary Algorithms (MOEAs) (Assunção et al., 2014; Vergilio et al., 2012a), as they can solve hard real-world problems impacted by many conflicting objectives. Moreover, MOEAs are widely used for software engineering problems in general. Surveys of the SBSE field (Harman et al., 2012; Colanzi et al., 2019) report that evolutionary algorithms are the preferred and most used ones.

In spite of this preference and large use, the design of a MOEA is not always easy. MOEAs are distinguished by different components, which directly affect the generated solutions. Furthermore, they have a wide range of parameters to be configured (Eiben and Smit, 2011). For example, MOEAs have as components the search operators, the replacement and archiving procedures. They also have as parameters: population size, crossover and mutation probabilities. In this work, the word design refers to the choice and implementation of the best components and parameters configuration. In this way, the great number of alternatives makes the MOEA design a hard task. The best combination of components and parameters may depend on the problem being solved.

Moreover, it is not always possible to know a priori what is the best choice among existing MOEAs. In the SBSE literature, the best MOEA for a problem is usually determined by conducting evaluation experiments, which requires effort and increases costs. Regarding the ITO problem, Assunção et al. (2014) conducted experiments with three different MOEAs - NSGA-II, SPEA and PAES - and no one of them proved to be the best to solve the problem considering all instances and contexts.

To reduce such difficulties faced by software engineers, in this paper, we propose an approach for the automatic generation of MOEAs applied to the ITO problem. The approach includes an offline training process, performed by a design algorithm that receives as input an instance of the ITO problem reported in the literature (Assunção et al., 2014). The output is a MOEA that can be used by the tester on other instances. To allow the use of different design algorithms the approach encompasses a set of MOEA parameters and

components to be combined. It is implemented with `irace` (from Iterated Racing algorithms) (López-Ibáñez et al., 2011) and Grammatical Evolution (GE) (Ryan et al., 1998). `irace` defines a "parameter space" in which the parameters, their types, ranges and constraints are defined. GE is a type of Genetic Programming (GP) (Koza, 1992) that uses a grammar containing a set of rules and values to guide the evolutionary process in the generation of programs. In this sense, by using the grammar of GE or the parameter space of `irace`, it is possible to map the components and parameters to be used in the automatic design.

The approach is evaluated with seven systems and produces results statistically better (in terms of hypervolume) than MOEAs commonly used to solve the ITO problem in the literature. A comparison between both design algorithms shows that they present similar results. These results provide some evidence for the benefits of automating the design and configuration of algorithms in SBSE. Furthermore, our results can be seen as evidence for the effectiveness of GE and the competitive results between GE and *irace*, which are ongoing matters of discussion in the community (Whigham et al., 2017b; Ryan, 2017; Whigham et al., 2017a; O'Neill and Nicolau, 2017).

An empirical evaluation of our approach was conducted using seven testing instances of the ITO problem in order to answer the following research questions:

- RQ1 – How are the results of the MOEAs generated by our approach in comparison to the MOEAs used in the literature? and,
- RQ2 – How are the results of the MOEAs generated using GE in comparison to the results of the MOEAs generated using `irace`?

RQ1 concerns comparing MOEAs generated by our approach with the algorithms NSGA-II and SPEA, used in the ITO problem literature. RQ2 concerns comparing the design algorithms GE and `irace`. As a result, we observe that both design algorithms have similar performance and our approach generates MOEAs that are better or similar than the traditional ones, considering some statistical tests and hypervolume, the main quality indicator used in the optimization field to compare MOEAs (Zitzler et al., 2003, 2007).

Some works on automatic design of Evolutionary Algorithms (EAs) are based on GE (Lourenço et al., 2012, 2013, 2015) and `irace` (Bezerra et al., 2014, 2015). These papers use different parameters and components, which are successfully combined to evolve better algorithms. One of these papers addresses the automatic design of MOEAs specialized in some benchmark instances using `irace` (Bezerra et al., 2014, 2015). With respect to the use of such algorithms in SBSE, the work of Mariani et al. (2016) proposes a GE-based hyper-heuristic, named GEMOITO, for generating MOEAs to solve the ITO problem. The encouraging results of all the aforementioned works motivated the work herein described. In this sense, our work has the following main contributions:

- Use of a set of components and parameters for MOEAs design that includes elements not used in related work;
- Introduction of an approach that can be used to solve software engineering problems formulated as permutation problems. The main idea is to ease the design of MOEAs, reducing efforts; and
- Application and evaluation of the approach to the ITO problem. Such an evaluation reports results comparing two design algorithms: GE and `irace`. We have not found works comparing them.

This paper is organized as follows. Section 2 reviews subjects related to this work: the ITO problem, automatic design of algorithms, and MOEAs. Section 3 introduces the proposed approach, describing its main elements. Section 4 reports the conducted empirical evaluation and the discussion of the results. Section 5 contains related work. And finally, Section 6 presents the conclusion and describes some future work.

2 Background

This section introduces the ITO problem and reviews background on automatic design of algorithms and MOEAs.

2.1 Integration and Testing Order problem

A testing strategy usually includes a set of phases with distinct goals. First of all, the unit testing focuses on each unit, the smallest part of a system to be tested. After this, an integration testing phase is performed to find problems in the interaction among the units. In this phase, the units are combined and tested according to the testing plans. During such integration it is necessary to determine an order to integrate and test the units. Such an order impacts the sequence in which units are developed; the design and execution of test cases; the order in which integration faults are revealed; and the number of required stubs for the units that possibly are not available, but from which the unit being tested depends on.

The stubbing process can be expensive and error-prone. The cost can be impacted by many factors such as number of attributes and parameters to be emulated, number of return types and so on. When there is a cyclic dependency between two classes, such a dependency needs to be broken and a stub needs to be built. In this sense, it is important that the class associated with the smaller cost, given for instance by the number of attributes and operations, appears first in the integration order. Due to this, we find in the literature many strategies to solve the Integration and Testing Order Problem (ITO) (Wang et al., 2011; Briand et al., 2002a). The most promising works are based on Multi-Objective Evolutionary Algorithms (MOEAs) (Vergilio et al., 2012a; Assunção et al., 2014). The proposed strategies are generally based on graphs that represent the dependencies among the units. A dependency cycle in the graph, which needs to be broken, corresponds to a required stub. The problem is to determine the best sequence of units associated with minimal stubbing cost.

In this work, we use the formulation of the ITO problem proposed by Assunção et al. (2014), as well as the same benchmark. A static model representing the dependencies among the units and costs (associated with the number of public classes attributes and methods to be emulated) is provided

to the algorithm by using matrices. These matrices are generated previously and reused during the optimization process. If the tester wants to use a new program, this program static model must be generated and given as input (as described in Assunção et al. (2014)).

Since the ITO problem is related to permutations of units, which form testing orders, the chromosome is represented by a vector of integers where each vector position corresponds to a class. The size of the chromosome is equal to the number of units of each system. Thus, let each unit be represented by a number, an example of a valid solution for a problem with five units is $\{2, 4, 3, 1, 5\}$. In this example, the first unit to be tested and integrated would be the unit represented by number 2. The second, the unit represented by number 4, and so on. Each unit can appear only once.

We use two objective functions to evaluate the solutions (Assunção et al., 2014), which measure the dependencies between server and client units. Hence, considering that: (i) a unit is a module to be tested that can be either classes or aspects; (ii) m_i and m_j are two coupled modules; and (iii) the operation term represents class methods, aspect methods and/or aspect advice, the used coupling measures are defined as follows:

- Number of attributes (A) = The number of attributes locally declared in m_j when references or pointers to instances of m_j appear in the argument list of some operations in m_i, as the type of their return value, in the list of attributes (data members) of m_i, or as local parameters of operations of m_i (adapted from Vergilio et al. (2012a); Briand et al. (2002a)). This complexity measure counts the (maximum) number of public attributes that would have to be handled in the stub if the dependency were broken.
- Number of methods (O) = The number of operations/methods (including constructors) locally declared in m_j that are invoked by operations of m_i (adapted from Briand et al. (2002a)). This complexity measure counts the number of public operations that would have to be emulated if the dependency were broken.

In the literature, the ITO problem is usually defined in terms of the testing process cost (Assunção et al., 2014; Assunção et al., 2013). By reducing the number of attributes and operations to be emulated, the tester can reduce the overall cost of the integration testing activity, i.e., these metrics serve as surrogates for the cyanstubbing process cost. It is hard to estimate the actual human effort for emulating such items, but using metrics in such granularity, number of attributes and number of operation/methods, as opposed to number of stubs can provide the tester a better way of estimating the cost. Moreover, the ITO problem formulation, commonly found in the literature, does not address the fault revealing capability of the testing process, just the cost of stubbing.

The input data of the MOEAs consist of matrices that are read from a text file. They are matrices associated to (i) dependencies between units; (ii) measure A; and (iii) measure O. Then it is necessary to establish a trade-off between these values. The dependency matrix is used to define precedence constraints and the others to calculate the fitness of each solution, where the sum of dependencies between the classes for each measure corresponds to an objective. It is important to highlight that there are classes with a great number of operators and few attributes, and vice-versa; the goal is to minimize both objectives.

2.2 Automatic Design of Algorithms

The design of an algorithm is related to all decisions taken during its definition and considering a specific problem (Eiben and Smit, 2011). Algorithms used for solving hard optimization problems commonly have several parameters and components to be defined by the user (López-Ibáñez et al., 2011). The chosen values have a great influence on the algorithm performance, but there is no generic set of values, since the optimal set depends on the problem being solved (Eiben and Smit, 2011). These values are then usually chosen based on common wisdom of the community(López-Ibáñez et al., 2011), since finding the appropriate values for the parameters and components is one of the greatest challenges in the evolutionary computation field (Eiben and Smit, 2011).

In the EAs' context, examples of parameters can be the population size, crossover and mutation probabilities. Examples of components are the search operators, such as selection, crossover and mutation ones. In this sense, the automatic design of algorithms can be very useful, since there is a great number of alternatives for these parameters and components. To this end, we find in the literature different methods for the automatic design of EAs. More details about existing methods can be found in the surveys of Eiben and Smit (2011, 2012).

Our approach works with `irace` and GE. We chose such algorithms because they are widely used in the literature of automatic configuration of meta-heuristics (Eiben and Smit, 2011, 2012). The GE algorithm has already been used for generating EAs (Lourenço et al., 2012, 2013, 2015) and presented encouraging results. This is one of the main motivations for its usage. Another advantage of GE is that it allows a flexible and context-free definition of programs to be generated by using a grammar. On the other hand, `irace` uses a very interesting mechanism based on statistical tests that has shown good results for multi-objective algorithms (López-Ibáñez and Stützle, 2012; Bezerra et al., 2014, 2015). Both algorithms are described in the following.

2.2.1 irace

The Iterated Race algorithm (here called `irace`) works by sequentially evaluating the candidate configurations and excluding the statistically worse. It consists of three main steps, repeated until a stopping criterion is satisfied, according to López-Ibáñez et al. (2011):

1. sampling new configurations according to a particular distribution;
2. selecting the best configurations from the newly sampled ones by means of racing; and
3. updating the sampling distribution in order to bias the sampling towards the best configurations.

To sample new configurations (Step1), the irace algorithm considers two types of parameters: numerical and categorical, and the sampling distribution of each parameter depends on the parameter type. Numerical parameters have a normal distribution and categorical parameters have a discrete distribution. The update (Step 3) changes the sampling by updating the mean and standard deviation for normal distributions, and the probabilities for the discrete distributions. This update process guides the distribution in order to increase the probability of selecting the parameters used in the best configurations when generating new ones (López-Ibáñez et al., 2011).

In order to select the best configurations (Step 2), the candidate ones are evaluated at each step on a single instance. After each step, the candidates that are significantly worse than at least another one are excluded. The race is repeated with the surviving configurations, and continues until a stopping criterion is met. This criterion is generally related to a number of surviving configurations, a number of used instances or a predefined computational budget (López-Ibáñez et al., 2011).

In this work we use the irace package[1] of the R Project[2], the same one used in Bezerra et al. (2014, 2015). In order to be executed, the irace package requires three inputs: a configuration file, the set of instances to be used and the parameter space. The parameter space is where the parameters used in the automatic configuration, their types, ranges and constraints should be defined (López-Ibáñez et al., 2011). Moreover, the statistical analysis is performed using the non-parametric Friedman statistical test (Derrac et al., 2011).

Figure 1 presents an example of a parameter space defined in a parameter file for the irace package. This example is for the automatic configuration of an Ant Colony Optimization (ACO) algorithm, and is included in the irace package along with other configuration files as a usage sample.

```
### PARAMETER FILE FOR THE ACOTSP SOFTWARE

# name       switch            type values      [conditions (using R syntax)]

algorithm    "--"              c    (as,mmas,eas,ras,acs)
localsearch  "--localsearch "  c    (0, 1, 2, 3)
alpha        "--alpha "        r    (0.00, 5.00)
beta         "--beta "         r    (0.00, 10.00)
rho          "--rho  "         r    (0.01, 1.00)
ants         "--ants "         i    (5, 100)
nnls         "--nnls "         i    (5, 50)     | localsearch %in% c(1, 2, 3)
q0           "--q0 "           r    (0.0, 1.0)  | algorithm %in% c("acs")
dlb          "--dlb "          c    (0, 1)      | localsearch %in% c(1,2,3)
rasrank      "--rasranks "     i    (1, 100)    | algorithm %in% c("ras")
elitistants  "--elitistants "  i    (1, 750)    | algorithm %in% c("eas")
```

Figure 1. Example of parameter space defined for the irace package

Each parameter has a name, a switch value, a type, values and conditions (optional). The name of the parameter is an identifier for later usage in the definition of the conditions. A condition (after the "|" delimiter) is used to define constraints among the parameters. For instance, in the example, the parameter "nnls" is only used when a local search (parameter "localsearch") has received a value between 1 and 3. The type identifier "c" represents the categorical parameters, whereas "r" (real) and "i" (int) represent numerical parameters. While categorical parameters can only receive as value what is given in the "value" field, a numerical parameter can receive any value in the specified ranges in its value field. The "switch" field of a parameter is what the algorithm actually gives as argument to execute.

What the irace package does is to build a sequence of arguments and values. Each argument sequence is what forms the algorithm configuration. In this sense, a script is defined in the package configuration file to receive this argument sequence and execute the algorithm using such arguments. In the end, the irace algorithm reads the result printed by the generated algorithm and updates its state.

Because a statistical test is used to compare the algorithms of each irace run, at the end of its execution, irace returns a set of algorithms statistically equal. However, in this paper, we only use the one with the best hypervolume and execute the designed algorithm several times instead. We only choose the best algorithm because the procedure of generating MOEAs in our approach needs an output of only one algorithm.

2.2.2 Grammatical Evolution

A GE algorithm can be considered a type of GP, since it is similarly used to evolve programs (Ryan et al., 1998). However, while a conventional GP algorithm typically uses a tree as representation for an individual and applies search operators to those trees (Koza, 1992), a GE algorithm uses an array of integers or bits and evolves the solutions similarly to a conventional EA (Ryan et al., 1998). Moreover, aside from the usual parameters of an EA (e.g. population size, maximum number of fitness evaluations and others), a GE receives a grammar file, usually in BNF, to map each solution into a program. This mapping is called *genotype-phenotype mapping* (GPM) (Barros et al., 2013).

The evolution is applied to the chromosome (genotype level), but only the program (phenotype level) can be executed and evaluated by a fitness function. Therefore, the GPM procedure is needed by the GE algorithm to transform each chromosome into an executable program. One of the advantages of GPM is that the genotype space can be freely explored, maintaining the validity of the phenotype (Barros et al., 2013). Furthermore, this allows a design of phenotype neutral crossover and mutation operators, where different genotypes can be mapped to the same phenotype. Next, we present more details of how this is done, and the default structure of a GE algorithm.

The common solution representation used by the GE algorithms is an array of integers or bits. If the array of bits is used, it is first mapped to an array of integers and, then, this array of integers is mapped to a program. It is possible to skip this step and use an array of integers directly. Nevertheless, the algorithm reads the grammar file, interprets the grammatical rules and then uses the integer values of the array to decide which values are assigned to each rule. If the algorithm reaches the end of the integer array, but still needs more genes to map into production rules, then the *wrapping* process is applied. *Wrapping* consists in consuming genes (when needed) starting from the beginning of the array when the end of the array is reached. To illustrate the GPM process, the following BNF grammar (Figure 2) is given for evolving simple mathematical expressions:

[1] https://cran.r-project.org/web/packages/irace/index.html

[2] https://www.r-project.org/

⟨*expr*⟩ ::= ⟨*var*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩

⟨*var*⟩ ::= x | y

⟨*op*⟩ ::= * | / | + | -

Figure 2. Example of grammar for mathematical expressions

The items between ⟨ and ⟩ are non-terminal rules, ‖ represents the logical operator OR, ::= means that the rule can take any of the next options and the remaining items are terminal nodes. For instance, the rule $\langle var \rangle$ can take either the value x or y when mapped to a program. On the other hand, $\langle expr \rangle$ can take the value of a single $\langle var \rangle$ or a composition of $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$. The choice between each option is given by the genes of the chromosome array of each individual.

In this work we use an integer array of variable size as representation for the solutions. This is actually an existing strategy (Ryan et al., 1998) that might help the algorithm to eliminate useless genes or reinsert new genes into the chromosomes as the evolution proceeds. For this end, the GE algorithm employs two distinguishable search operators: i) gene duplication operator; and ii) gene pruning/deletion operator. The duplication operator selects a random subarray of the chromosome and copies it to the end of the chromosome. The prune operator, on the other hand, selects an index to truncate the array. These operators are usually applied with the same probability as the mutation operator and as additional steps in the evolution.

Summarizing, Algorithm 1 presents the pseudocode of a conventional GE algorithm. As the algorithm shows, it is very similar to an EA, aside from the evaluation of the population (GE has the GPM procedure) and the application of the duplication and prune operators.

Algorithm 1: Pseudocode of a GE algorithm

```
1  Input: GF – Grammar File;
2  begin
3      population ← Initialize the population;
4      programs ← Map population to programs using GF;
5      Execute programs;
6      Assign a fitness value to the solutions of population
         according to the output of their respective programs;
7      while stop condition is not achieved do
8          matingPopulation ← Select parents;
9          offspring ← Recombine matingPopulation;
10         Apply gene prune operator to the solutions of
             offspring;
11         Apply gene duplication operator to the solutions of
             offspring;
12         Apply mutation operator to the solutions of offspring;
13         programs ← Map offspring to programs using GF;
14         Execute programs;
15         Assign a fitness value to the solutions of offspring
             according to the output of their respective programs;
16         population ← Perform replacement;
17     end
18     return Best program of population;
19 end
```

2.3 Multi-Objective Evolutionary Algorithms

Multi-objective optimization problems have more than one objective to be optimized Coello et al. (2007). In this kind of problem, usually two or more objectives are in conflict and cannot be optimized at the same time, i.e., by optimizing one objective, the value of the other is degraded. Many real world problems are multi-objective. For example, a car driver might assume two objectives when taking the best route between two points: time needed to complete the route and travel cost.

In this context, Pareto dominance concept is employed (Coello et al., 2007). In a minimization problem, a solution x is said to dominate ($\prec$) a solution y if $\forall z \in Z : z(x) \leq z(y)$ and if $\exists z \in Z : z(x) < z(y)$, where z is an objective in the set of considered objectives Z. If these conditions are not satisfied for both x and y, then such solutions are said to be non-dominated. The set of all possible non-dominated solutions in the search space of the problem being optimized is called the Pareto front (PF). In most cases it is not possible to determine the (PF) for a given problem. Hence, the algorithms try to find an approximation of this front, here called best-known Pareto front (PF_{known}). Differently from a mono-objective EA that yields a single solution at the end of its execution, the result of a MOEA is a set of non-dominated solutions. Thus, engineers usually have to decide which solution better fits their needs and/or preferences.

To evaluate the performance of a MOEA the most used quality indicator is the hypervolume (Zitzler et al., 2003). The hypervolume of a PF is the area dominated by this front with respect to a reference point. Considering a two objective optimization problem, each point of the Pareto Front defines a rectangle in the search space. The hypervolume corresponds to the area formed by the sum of all rectangles.

In mono-objective EAs the engineer can easily decide which solutions are the best ones according to their fitness values in any given moment. However, MOEAs cannot do this in a straightforward way, because they have several objectives to evaluate and potentially a great number of non-dominated solutions. Therefore, some strategies may be applied to help the decisions that MOEAs take during the evolution process. Usually a fitness calculation is applied to the cyansolutions so that the comparison between them becomes possible. For instance, the SPEA2 algorithm (Zitzler et al., 2001) uses the concepts of Combined Dominance Strength (how many solutions a solution dominates and how many solutions dominate it) and K-th Nearest Neighbour (the distance to the k-th nearest neighbour) to assign a fitness value for each solution, whereas NSGA-II (Deb et al., 2002) uses the concepts of Dominance Depth (rank of the sub-front) and Crowding Distance (density estimation).

Notice that the fitness evaluation in both examples uses two kinds of evaluations: i) a convergence assessment (Combined Dominance Strength and Dominance Depth); and ii) a diversity assessment (K-th Nearest Neighbour and Crowding Distance). The convergence regards how close the solutions of a front are to a reference front (usually optimal or best-known). A good convergence in the search process can guide the algorithm on finding solutions closer to the refer-

ence Pareto front and can improve the overall performance of the algorithm. On the other hand, a good search diversity can prevent the algorithm from falling into a local optimum and provide a better exploration of the search space. The idea is to balance both factors during the optimization process in order to optimize the resulting output. However, this might be a difficult task and may come along with some drawbacks (e.g. computational cost).

In addition, there are other parameters and components that must be taken into account when designing a MOEA. Some of the most distinguishable parameters are: population size, archiving size, and mutation and crossover probabilities. The archiving size regards to the size of the archive used by some MOEAs (Coello et al., 2007) as an external population to support the evolution process. Furthermore, the MOEAs components can have different implementations. For instance, the replacement strategy can be generational or ranking based. All these details contribute to increment the algorithm complexity, and require a lot of effort from a novice and non-expert engineer in the optimization field.

Unfortunately, these details largely influence the performance of the algorithms, and their design and tuning are an optimization problem itself (Eiben and Smit, 2011). This serves as the main motivation for our approach proposed in the next section.

3 Proposed Approach

The proposed approach uses offline training to automatically design a MOEA specialized in the ITO problem. The training can be performed by using two different algorithms. Figure 3 shows how the proposed approach works[3].

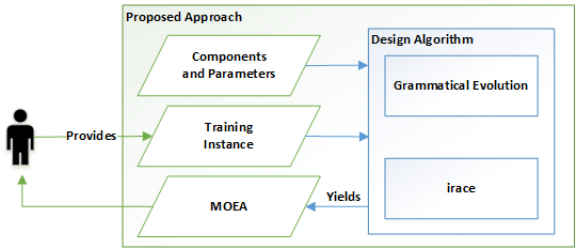

Figure 3. Proposed Approach

Two inputs are used by the approach: the training instance and the set of components and parameters, which are described in detail, respectively in Sections 3.2 and 3.3. The instance of the problem is provided by the user, who also chooses a design algorithm that can be either GE or `irace`. Another input is the set of components and parameters, which are defined in a representation compatible with the selected design algorithm (grammar or parameter space). This set is pre-defined, but can be modified or extended if desired. Then, the design algorithm is executed with the training instance, and at the end the best MOEA is returned. This MOEA is used by the tester to solve other ITO instances.

The design algorithms work with a population where each individual is a MOEA. The fitness is given by some indicator calculated by using the corresponding fronts obtained by each individual to the ITO problem. In this work we use hypervolume indicator (Zitzler et al., 2003).

It is important to emphasise that the GE and `irace` algorithms work on a higher level of the search. Instead of trying to find the solution for the problem directly, these two algorithms try to generate the best MOEA that can in turn solve the problem. Hence, these two algorithms search for MOEAs in the "MOEA space" using conventional search methods with which they were proposed (López-Ibáñez et al., 2011; Ryan et al., 1998). Next, we describe the structure of the generated MOEAs by detailing their main components and the representation used by each design algorithm.

3.1 MOEA

Algorithm 2 shows the structure of a standard MOEA manipulated and returned by the approach. The components of each step and their respective parameters are selected by the design algorithm. These steps are: initialization of the population (Line 2), evaluation of the population (Lines 3 and 9), selection of parents (Line 6), crossover operator (Line 7), mutation operator (Line 8), replacement (Line 10) and archiving of the individuals (Lines 4 and 11).

Algorithm 2: Template of a designed MOEA

```
1  begin
2      population ← Initialize the population;
3      Evaluate (population);
4      Archive (population);
5      while stop condition is not achieved do
6          matingPopulation ← Selection (population);
7          offspringPopulation ← Crossover
             (matingPopulation);
8          offspringPopulation ← Mutation
             (offspringPopulation);
9          Evaluate (offspringPopulation);
10         Replacement (offspringPopulation, population);
11         Archive (offspringPopulation);
12     end
13 end
```

We divide the fitness assignment into three independent components, one for selection, another for replacement and one for archiving. By changing these assignments separately, the design algorithms can find better MOEAs by focusing on one kind of search at each step. For instance, a selection component can focus on mating more diversified parents, whereas a replacement component can focus on making the most converged solutions to survive.

3.2 Training Instance

A training instance of the ITO problem must be given by the user, so that the design algorithms can execute the generated MOEAs. This instance must contain the matrices mentioned in Section 2. This information is later used to formulate a permutation problem, where each unit is represented by its ID. During the problem solving, the order in which the units appear in the chromosome determines the order in which they are integrated and tested. Because ITO is a permutation problem, the MOEA components and parameters used in the pro-

[3] Source code of our approach at `https://github.com/GiovaniGuizzo/jMetalGrammaticalEvolution`.

posed approach are focused on the permutation representation.

3.3 Components and Parameters

We chose the parameters and components based on experiments and tuning conducted in related work (Assunção et al., 2014; Guizzo et al., 2015; Briand et al., 2002a). For instance, in Guizzo et al. (2015) the authors used three permutation crossover operators in their online operator selection, which are also included here.

The parameters and components are categorized regarding the following MOEA steps: population initialization, selection, mating, replacement and archiving. Moreover, some fitness assignment strategies are used in the selection, replacement and archiving to guide the evolution. In this sense, the *Fitness Assignment* component is defined as a mechanism to identify which solution is the best one according to all objectives, but its three usages are completely independent and can vary according to the best outcome.

Most components and parameters are generic enough to be used for any problem and by any algorithm (e.g. crossover and selection operator, type of population initialisation), but a few components are more specific and were extracted from existing algorithms (e.g. replacement strategy, archiving of solutions, fitness evaluation mechanism). We have extracted and implemented the components and parameters from the following algorithms: NSGA-II (Deb et al., 2002), SPEA (Zitzler et al., 2001), SPEA2 (Zitzler et al., 2001), Multi Objective Genetic Algorithm (MOGA) (Fonseca and Fleming, 1993), Pareto Achieved Evolution Strategy (PAES) (Knowles and Corne, 2000), and Indicator Based Evolutionary Algorithm (IBEA) with hypervolume (Zitzler et al., 2003). Hence, our approach is able (although unlikely) to generate each of those algorithms by joining their components together during the evolutionary process. If the tester wants to adapt our approach, for instance by including the components of the their own algorithm, they just need to implement the components and add them to the grammar (explained in more details in Section 3.4). Next, we present all these elements in detail.

3.3.1 Population

This element is composed by a *Population Size* and an *Initialization* procedure. *Population Size* specifies the number of individuals in the population. *Initialization* defines how the first population is initialized. It can be done by using *Random* or *Parallel Diversification* (Talbi, 2009). The latter aims at generating diversified solutions by initializing the individuals in a way that an integer number cannot be repeated at the same position in another individual of the population.

3.3.2 Selection

The *Selection* element is related to the selection of parents to be recombined and defines the *Source* and *Selection Operator* components. *Source* specifies from where the parents are extracted. That way, the parents can be selected only from the current population, or from the archive and population combined. *Selection Operator* specifies the strategy used to select the parents. *Random* randomly selects two solutions to be recombined. *K-Tournament* performs k number of binary tournaments (comparisons) between random solutions and chooses the best two to be recombined. In such a case, the parameter *k* is also selected. *Roulette Wheel* gives a probability based on the fitness value of a parent, and performs a selection complying with the probabilities of each parent. *Ranking* classifies the solutions based on the fitness value and selects the best ones to be recombined.

The *K-Tournament*, *Roulette Wheel* and *Ranking* selection operators use the fitness of each solution to aid the selection. That way, they use the *Fitness Assignment* component.

3.3.3 Fitness Assignment

The *Fitness Assignment* element encompasses the *Convergence Strategy* component to assess the quality regarding the convergence of the solutions, and the *Diversity Strategy* component for the tie-breaking of solutions with the same convergence value. Another possibility is the usage of only one kind of strategy, either convergence or diversity for the evaluation. If no *Convergence Strategy* component is selected, then the *Diversity Strategy* component becomes the primary metric for fitness assignment.

There are four possible components for the *Convergence Strategy*. *Dominance Depth* (NSGA-II) (Deb et al., 2002) assesses the convergence quality of the solutions using Pareto fronts. The first Pareto front has all the non-dominated solutions of the population. The second Pareto has all the non-dominated solutions excluding the ones in the first Pareto front. Such process is performed until there are no more solutions. The *Dominance Depth* of a solution is the Pareto front number in which it is present, thus the lower the Dominance Depth value, the better. *Dominance Strength* (SPEA) (Zitzler et al., 2001) computes the number of solutions that a solution dominates. If a solution dominates many others (greater strength), then it may indicate that this solution dominates a great area of the objective space. *Raw Fitness* (SPEA2) (Zitzler et al., 2001) assesses the convergence quality by computing the sum of the strength values of all the solutions that dominate a solution. Thus, the lower this value, the less likely a solution is to be on an easily dominated area of the objective space. *Dominance Rank* (MOGA) (Fonseca and Fleming, 1993) computes the number of solutions that dominates a solution, thus the lower this value, the better.

Regarding the *Diversity Strategy*, there are four possibilities. *Crowding Distance* (NSGA-II) (Deb et al., 2002) is based on the distance between the neighbours solutions in the objective space. A low crowding distance value means that the solution is in a crowded area of the objective space, and possibly brings low diversity to the search (e.g. if used as a parent for recombination). *K-th Nearest Neighbour* (SPEA2) (Zitzler et al., 2001) assesses the distance from a solution to its k_{th} nearest neighbour solution in the objective space. The parameter k is defined as in (Zitzler et al., 2001): $k = \sqrt{N + \overline{N}}$, where N is the population size and $\overline{N}$ is the archive size. Similar to *Crowding Distance*, it is necessary

to maximize the value the *K-th Nearest Neighbour* diversity strategy. *Adaptive Grid* (PAES) (Knowles and Corne, 2000) divides the objective space into grids to trace the crowding degree of different regions. It is possible to diversify the non-dominated solutions and help to remove excessive non-dominated solutions located in the crowded grids. The adaptive grid value of a solution is the number of solutions in its grid, thus the lower the value, the more isolated the solution is. *Hypervolume Contribution* (Zitzler et al., 2003) is based on the hypervolume quality indicator. Briefly, the hypervolume contribution of a solution measures how much a solution strictly contributes to the hypervolume of the front in which it is contained. Therefore, the greater the hypervolume contribution of a solution, the bigger the space dominated only by this solution.

3.3.4 Mating

Mating Strategy (Talbi, 2009) defines the way and how many off-springs are created in each generation. *Generational One Child* and *Generational Two Children* generate N children at each generation. The former generates one in each recombination and the latter generates two. *Steady State* generates only one offspring at each generation. Even though *Steady State* is usually defined as a replacement strategy (Eiben and Smith, 2003), here it is defined as a reproduction strategy and the replacement component is responsible to insert the generated solution in the population according to the replacement strategy.

Mating Operators are composed by the *Crossover Operator*, *Mutation Operator* (Eiben and Smith, 2003) and their probabilities. Since in this paper we are addressing a permutation problem, all the crossover and mutation operators used here are for permutation representations.

The *Crossover Operator* creates one or more off-springs by combining the genes of two parents. In this approach, no crossover or one of four crossover operators can be selected. *Single Point Crossover* selects a random point and cuts the parents in this point. One half of each parent is merged into different children. *Two Points Crossover* has a similar strategy, but two random points are chosen to cut the parents. The sub-array inside each cutting point of a parent is copied to a solution, and the remaining genes are selected from the other parent. *Partially Mapped Crossover (PMX)* also cuts the parents in two points, but it uses a more complex cyclic procedure to select the genes of the sub-arrays. *Cycle Crossover* works by dividing the elements into cycles. A cycle is a subset of elements where each element always occurs paired with another element of the same cycle when two parents are aligned. The off-springs are created by selecting alternate cycles from each parent.

A *Mutation Operator* applies some kind of transformation in the individual. In this approach, cyanthe alternatives to that component are no mutation or one of the four following mutation operators: *Swap Mutation* randomly selects two genes and swaps their values. *Insert Mutation* randomly selects a random value and moves it to a random position in the chromosome. *Scramble Mutation* randomly selects two genes and shuffles the values between them. *Inversion Mutation* randomly selects two genes and inverts the order of the values appearing between such genes.

3.3.5 Replacement

Replacement represents the strategy to define which individuals survive and compose the next generation. The *Generational* strategy defines that the parents are always replaced by the off-springs in the next generation. This replacement takes into account the idea of elitism. Briefly, the elitism forces the survival of a predetermined number of best parents for the next generation. The *Elitism Size* is also selected by the algorithm. If the elitism is indeed selected, then the *Fitness Assignment* component is used to determine the best parents for survival. The *Ranking* replacement creates a ranking of the solutions, based on the fitness values that are measured by the *Fitness Assignment* component. That way, the best solutions are selected to survive, regardless of being parents or children.

3.3.6 Archiving

The *Archiving* element is related to the archiving procedure used to store the solutions. This is employed by some algorithms such as SPEA2 (Zitzler et al., 2001) and PAES (Knowles and Corne, 2000). Sometimes the archive is called external population, but the purpose is the same: to aid the search process with another source of solutions for reproduction, or simply to store the best solutions found so far. At each generation, the archive is updated with the newly generated solutions. In this work the *Ranking* component is always used to rank the solutions using the *Fitness Assignment* component, and the best solutions are kept in the archive. It is able to store dominated and non-dominated solutions. Moreover, the *Archive Size* parameter is used to define how many solutions are stored in the archive. In this work, a set of population size percentages is used. If such value is zero, no archive is used by the MOEA.

3.4 Design Algorithms

For the generation, execution, evaluation and selection of MOEAs the tester can choose between GE and `irace`. These algorithms have the same purpose: to generate MOEAs based on several trials. However, they require different representations for the elements and components presented in previous section and specialized for the ITO (permutation) problems. The GE algorithm generates each MOEA using the following grammar.

⟨*GA*⟩ ::= ⟨*populationSize*⟩ ⟨*initialization*⟩ ⟨*selection*⟩ ⟨*mating*⟩ ⟨*replacement*⟩ ⟨*archive*⟩

⟨*populationSize*⟩ ::= `50` | `100` | `150` | `200` | `250` | `300`

⟨*initialization*⟩ ::= `Random` | `Parallel Diversification`

⟨*selection*⟩ ::= ⟨*selectionOperator*⟩ ⟨*source*⟩ ⟨*fitnessAssignment*⟩

⟨*selectionOperator*⟩ ::= `K Tournament` ⟨*tournamentSize*⟩ | `Random` | `Roulette Wheel` | `Ranking`

⟨*tournamentSize*⟩ ::= `2` | `4` | `6` | `8` | `10`

⟨*source*⟩ ::= `Population` | `Archive and Population`

⟨*fitnessAssignment*⟩ ::= ⟨*convergenceStrategy*⟩ ⟨*diversityStrategy*⟩

⟨*convergenceStrategy*⟩ ::= λ | `Dominance Rank` | `Dominance Strength` | `Dominance Depth` | `Raw Fitness`

⟨*diversityStrategy*⟩ ::= λ | `Crowding Distance` | `K-th Nearest Neighbour` | `Adaptive Grid` | `Hypervolume Contribution`

⟨*mating*⟩ ::= ⟨*matingOperators*⟩ ⟨*matingStrategy*⟩

⟨*matingOperators*⟩ ::= ⟨*crossoverOperator*⟩ ⟨*crossoverProbability*⟩ ⟨*mutationOperator*⟩ ⟨*mutationProbability*⟩

⟨*crossoverOperator*⟩ ::= λ | `Two Points Crossover` | `Single Point Crossover` | `PMX Crossover` | `Cycle Crossover`

⟨*crossoverProbability*⟩ ::= `1.0` | `0.95` | `0.9` | `0.8` | `0.5`

⟨*mutationOperator*⟩ ::= λ | `Swap Mutation` | `Insert Mutation` | `Scramble Mutation` | `Inversion Mutation`

⟨*mutationProbability*⟩ ::= `0.01` | `0.02` | `0.05` | `0.1` | `0.2` | `0.5` | `0.7` | `0.8` | `0.9` | `1.0`

⟨*matingStrategy*⟩ ::= `Steady State` | `Generational Two Children` | `Generational One Child`

⟨*replacement*⟩ ::= `Generational` ⟨*elitismSize*⟩ ⟨*fitnessAssignment*⟩ | `Ranking` ⟨*fitnessAssignment*⟩

⟨*elitismSize*⟩ ::= `0` | `N * 0.01` | `N * 0.05` | `N * 0.1` | `N * 0.5`

⟨*archive*⟩ ::= `Ranking` ⟨*fitnessAssignment*⟩ ⟨*archiveSize*⟩

⟨*archiveSize*⟩ ::= `0` | `N` | `N * 1.5` | `N * 2`

Based on this grammar, the symbol λ means empty component. If an alternative for a component is not wanted by the tester, then it can be removed from the grammar and it will not be used by the GE algorithm. Similarly, if the tester already has a MOEA and only wants to configure some parameters, then it is necessary to set the components of the MOEA in the grammar without other alternatives. For instance, to apply the NSGA-II algorithm and automatically configure it, the tester must set the replacement rule with only the "Ranking" alternative, the fitness assignment component with only "Dominance Depth" and "Crowding Distance", and any other element that must not change.

Similarly, the parameter space of `irace` is defined with all the parameters and components, and can be changed if desired. The parameter space used by our approach is structured as follows.

```
populationSize "--populationSize "
 c(50,100,150,200,250,300)
initialization "--initialization "
c("Random","Parallel Diversified Initialization")
selectionOperator "--selectionOperator "
c("K Tournament","Random","Roulette Wheel","Ranking")
tournamentSize "--tournamentSize "
c(2,4,6,8,10) | selectionOperator == "K Tournament"
selectionSource "--selectionSource "
c("Population","Archive and Population")
selectionRankingStrategy "--selectionRanking "
c(" ","Dominance Rank", "Dominance Strength", "Dominance
Depth", "Raw Fitness") | selectionOperator != "Random"
selectionDiversityStrategy "--selectionDiversity "
c(" ","Crowding Distance", "K-th Nearest Neighbour",
"Adaptive Grid", "Hypervolume Contribution") |
selectionOperator != "Random"
crossoverOperator "--crossoverOperator "
c(" ","Two Points Crossover", "Single Point Crossover",
"PMX Crossover", "Cycle Crossover")
crossoverProbability "--crossoverProbability "
c(1.0,0.95,0.9,0.8,0.5) | crossoverOperator != " "
mutationOperator "--mutationOperator "
c(" ","Swap Mutation","Insert Mutation","Scramble
Mutation", "Inversion Mutation")
mutationProbability "--mutationProbability "
c(0.01,0.02,0.05,0.1,0.2,0.5,0.7,0.8,0.9,1.0) |
mutationOperator != " "reproduction "--reproduction "
c("Steady State","Generational Two Children","Generational
One Child")
replacement "--replacement "
c("Generational","Ranking")
elitismSize "--elitismSize "
c(0,0.01,0.05,0.1,0.5) | replacement == "Generational"
replacementRankingStrategy "--replacementRanking "
c(" ","Dominance Rank","Dominance Strength", "Dominance
Depth", "Raw Fitness") | replacement == "Ranking" ||
(replacement == "Generational" && elitismSize != "0")
replacementDiversityStrategy "--replacementDiversity "
c(" ","Crowding Distance","K-th Nearest
Neighbour","Adaptive Grid",
"Hypervolume Contribution") | replacement == "Ranking" ||
(replacement == "Generational" && elitismSize != "0")
archiveSize "--archiveSize "
c(0,1.0,1.5,2.0)
archiveRankingStrategy "--archiveRanking "
c(" ","Dominance Rank","Dominance Strength", "Dominance
Depth", "Raw Fitness") | archiveSize != "0"
archiveDiversityStrategy "--archiveDiversity "
c(" ","Crowding Distance","K-th Nearest Neighbour",
"Adaptive Grid", "Hypervolume Contribution") | archiveSize
!= "0"
```

The grammar used by GE, as well as the parameter space of `irace`, were created to ensure the generation of only valid combinations. This is one of the advantages of using GE algorithms: they can only generate what the grammar allows, making the resulting solutions always syntactically correct.

4 Empirical Evaluation

As mentioned before, the evaluation of our experimental evaluation was guided by two research questions. RQ1 compares the MOEAs generated using our approach with the traditional MOEAs used in the ITO literature. RQ2 compares both design algorithms GE and `irace`.

To answer the questions, we use eight real world systems, also explored in related work (Assunção et al., 2014; Guizzo et al., 2015). One was used for training and seven for testing. They are implemented in Java and AspectJ. Table 1 shows the number of units, dependencies and LOC of each system. The number of dependencies directly impacts the number of existing solutions for the problem. AJHSQLDB (first row) is used only for the training.

Table 1. Systems used in the empirical evaluation

Name	Units	Dependencies	LOC	Language
		Training		
AJHSQLDB	301	1338	68550	AspectJ
		Testing		
AJHotDraw	321	1592	18586	AspectJ
HealthWatcher	117	399	5479	AspectJ
TollSystems	77	188	2496	AspectJ
BCEL	45	289	2999	Java
JBoss	150	367	8434	Java
JHotDraw	197	809	20273	Java
MyBatis	331	1271	23535	Java

The empirical evaluation is performed in two phases. In the training phase, the proposed approach is executed in the training instance AJHSQLDB to automatically generate a set of MOEAs. Then, in the testing phase, the generated MOEAs are executed in all testing instances of the problem and are evaluated.

To answer RQ1, we execute two MOEAs, successfully used in related work to solve the ITO problem (Assunção et al., 2014): i) NSGA-II (Deb et al., 2002); and ii) SPEA2 (Zitzler et al., 2001). In order to perform a fair comparison, we use the proposed approach with GE to tune these algorithms. This is another advantage of this approach: it can also be used to tune algorithms and not only to design new ones. For this tuning, we adapted a grammar for each algorithm, fixing some key components and parameters of the algorithms, and only letting the other ones vary. We fixed the replacement as Ranking with Dominance Depth and Crowding Distance for the NSGA-II grammar, whereas for the SPEA2 grammar we fixed this component as Generational with no elitism. If we let the GE tune all elements of NSGA-II and SPEA2, then they could lose their main features and become totally different algorithm implementations. All the components were implemented with jMetal (Durillo and Nebro, 2011). The approach was executed once for each algorithm and for the same amount of evaluations (10,000). In the end, the best configuration was selected and then used in the testing phase for 60,000 fitness evaluations, as well as the generated MOEAs. Table 2 shows the configuration of each algorithm chosen by the approach.

Table 2. Parameters of NSGA-II and SPEA2

Parameter	NSGA-II	SPEA2
Population Size	50	50
Maximum Fitness Evaluations	60,000	60,000
Crossover Operator	PMX	PMX
Crossover Probability	100%	95%
Mutation Operator	Swap	Swap
Mutation Probability	1%	5%
Archive Size	-	50

4.1 Training Phase

The system AJHSQLDB was chosen as the training instance of our approach, because it is the biggest instance. In preliminary experiments, we observed that using an easily solvable instance usually results in a weak training. Furthermore, we use only one instance of the problem for the training because otherwise it would increase the cost of such a phase. We acknowledge that the use of several instances would increase the training quality, but this is a subject for a future work only focused on this trade-off.

We execute each design algorithm (GE and `irace`) 10 times. That way, each run returns one MOEA, for a total of 20 MOEAs. For the GE configuration, we define the parameters based on the literature (Lourenço et al., 2013). Table 3 presents such configuration.

The `irace` algorithm requires as parameter only a training budget (number of MOEA evaluations), for which we use the same amount given to GE: 10,000. In addition, each generated MOEA is executed by the algorithms for 2,000 fitness evaluations. We use few fitness evaluations for the training due to the great computational time needed for this task. Summarizing, each design algorithm is executed 10 times with 10,000 evaluations in each independent run, and each generated MOEA receives a budget of 2,000 fitness evaluations during the training phase.

Table 3. GE parameters

Parameter	Value
Population Size	100
Number of MOEA Evaluations	10,000
Number of Fitness Evaluations by MOEA	2,000
Crossover Operator	Single Point Crossover
Crossover Probability	90%
Mutation Operator	Integer Mutation
Mutation Probability	1%
Selection Operator	Binary Tournament
Pruning Operator Probability	1%
Duplication Operator Probability	1%
Minimum of Genes in the Initial Population	10
Maximum of Genes in the Initial Population	20
Replacement Strategy	Ranking

We analyse the parameters values and the components of the MOEAs returned in all runs. Based on the 20 obtained MOEAs, we compute the frequencies that the values appear. Table 4 shows, for each parameter and component and for each design algorithm, the values that appear more often in the MOEAs.

As seen in Table 4, some components and parameters are clearly dominant, since they appear in the design of the best MOEAs very often, regardless of which design algorithm is being used. For example, *Steady State* mating, *Ranking* selection operator and *Dominance Strength* for the selection procedure are used for all the 20 MOEAs. Other components and parameters such as *Parallel Diversified* initialization, 100% mutation probability, *Archive* and *Population* selection source and *Ranking* archiving are used for almost every obtained MOEA.

Even though the design algorithms obtain similar MOEAs, some differences can be noted by analysing these frequencies. For instance, using GE, the best MOEAs always use *Inversion Mutation*, whereas using `irace` the best MOEAs always use *Swap Mutation*. In addition, `irace` presents greater crossover probabilities and a convergence strategy for the archiving procedure, while the GE algorithm generates MOEAs with the lowest crossover probability 50% of the time and it does not use a convergence strategy for archiving more than half of the time.

4.2 Testing Phase

In order to answer the research questions, the 20 MOEAs generated by our approach in the training phase, and the traditional algorithms NSGA-II and SPEA2 are executed using all the seven testing instances of the problem.

For each instance and MOEA, 30 independent runs are executed for 60,000 fitness evaluations. In the comparison, we use the hypervolume indicator (Zitzler et al., 2003) to assess the quality of each front obtained after each execution. We do not know the real Pareto fronts of the problems, thus this indicator can be used because it does not require such front. Furthermore, if the hypervolume value of a front A is greater than the value of a front B, then A is not worse than B. These characteristics make hypervolume suitable for the context of this experimentation.

In order to compare the algorithms, we calculate the hypervolume value of the 30 runs of each algorithm in each

Table 4. Frequency of most used parameters values and components. Some frequencies are not multiples of 10 because in some executions the corresponding components/parameters were not applicable, since they depended on other components that were not selected.

Category	Parameter/Component	GE		irace	
		Value	**Frequency**	**Value**	**Frequency**
Population	Population Size	50	50%	50	50%
	Initialization	Parallel Diversified	80%	Parallel Diversified	80%
Mating	Mating Strategy	Steady State	100%	Steady State	100%
	Crossover Operator	Single Point	40%	Single Point	50%
		None	40%	-	-
	Crossover Probability	0.5	50%	0.8	33.33%
		-	-	0.95	33.33%
	Mutation Operator	Inversion Mutation	100%	Swap Mutation	100%
	Mutation Probability	1.0	80%	1.0	90%
Selection	Source	Archive and Population	80%	Archive and Population	70%
	Selection Operator	Ranking	100%	Ranking	100%
	Convergence Strategy	Dominance Strength	100%	Dominance Strength	100%
	Diversity Strategy	Hypervolume Contribution	40%	K-th Nearest Neighbour	30%
		-	-	Hypervolume Contribution	30%
Replacement	Convergence Strategy	Dominance Rank	40%	Dominance Strength	50%
	Diversity Strategy	Hypervolume Contribution	50%	K-Th Nearest Neighbour	40%
		-	-	None	40%
Archiving	Type	Ranking	70%	Ranking	60%
	Archive Size	N * 2	42.85%	N * 1.5	50%
	Convergence Strategy	None	57.14%	Dominance Rank	33.33%
	Diversity Strategy	K-th Nearest Neighbour	28.57%	Adaptive Grid	50%
		Hypervolume Contribution	28.57%	-	
		None	28.57%	-	

instance, and used the Kruskal-Wallis statistical test at 95% of confidence (Derrac et al., 2011) to address statistical differences on the hypervolume values. Moreover, for each instance, we calculate the rank of the algorithms based on their mean hypervolume. In this ranking process, algorithms without statistical difference are considered tied. In the end, we calculate the mean hypervolume and mean rank for each algorithm across all instances. We also compute the Friedman statistical test at 95% of confidence over the seven mean hypervolume values of each algorithm to determine if there is any overall statistical difference between their results. Finally, we compute the Vargha-Delaney's $\hat{A}_{12}$ effect size (Vargha and Delaney, 2000).

We use multiple group p-value analysis (Kruskal-Wallis and Friedman) based on multiple groups (algorithms) due to the number of algorithms used in the experimentation. Using Mann-Whitney U instead would result in a large amount of pair p-values. While Kruskal-Wallis is used to compute differences in a given experimental subject, Friedman is used on the means of the algorithms to give an overall statistical analysis across multiple systems. Furthermore, for the post-hoc test we use the suggestion of Siegel and Castellan (1988). It is also the default post-hoc technique used for such statistical tests in R.

For the sake of succinctness, we select and report only some of the generated algorithms. In the next paragraphs we present the results for the best, median and worst algorithms of each method according to their mean rank in terms of hypervolume. The best, median, and worst algorithms obtained by the GE algorithm are named GE_Best, GE_Median, and GE_Worst respectively. Similarly, the best, median, and worst algorithms obtained by `irace` are I_Best, I_Median, and I_Worst respectively.

Table 5 presents the mean hypervolume for each instance and algorithm, and its mean rank in parentheses[4]. The last row presents the overall mean hypervolume and mean rank. The last column presents the p-value obtained using the Kruskal-Wallis test, except in the last row where the Friedman test result is shown. The best values (greatest for hypervolume and lowest for ranks) and the statistically equivalent values to the best ones are highlighted in bold.

In general, the results obtained by the best and median generated algorithms are better than the ones obtained by the conventional ones. GE_Best and I_Best are able to obtain the best or equivalent to the best results according to the Kruskal-Wallis test in almost every instance of the problem. Even the worst generated algorithms (GE_Worst and I_Worst) perform better than NSGA-II and obtain competitive results to SPEA2 in overall.

I_Best performs better in general, which results in the best mean rank (3.14). GE_Best obtains the second best re-

[4]Data as used for the test can be downloaded at `https://github.com/GiovaniGuizzo/jMetalGrammaticalEvolution/blob/master/results.7z`.

Table 5. Hypervolume means and ranks

System	GE Best	GE Median	GE Worst	IRACE Best	IRACE Median	IRACE Worst	NSGA-II	SPEA2	p-value
AJHotDraw	**0.88 (2.00)**	**0.87 (2.00)**	0.55 (5.50)	**0.79 (2.50)**	0.64 (5.00)	0.56 (5.50)	0.27 (7.50)	0.51 (6.00)	2.2E-16
HealthWatcher	**1.00 (4.00)**	**1.00 (4.00)**	**0.98 (4.00)**	**0.99 (4.00)**	**0.99 (4.00)**	**0.94 (4.50)**	0.92 (7.50)	**0.99 (4.00)**	1.1E-09
TollSystems	0.90 (5.00)	**1.00 (2.50)**	**0.95 (4.00)**	**0.93 (4.00)**	0.90 (5.00)	0.88 (5.00)	0.77 (6.50)	**0.94 (4.00)**	1.4E-08
BCEL	**0.76 (2.00)**	0.67 (6.00)	0.46 (7.50)	**0.74 (2.50)**	**0.74 (2.50)**	0.63 (6.50)	0.72 (3.50)	0.69 (5.50)	2.2E-16
JBoss	**0.96 (4.50)**	**1.00 (4.50)**	**0.97 (4.50)**	**0.96 (4.50)**	**0.98 (4.50)**	**0.91 (4.50)**	**0.90 (4.50)**	**0.96 (4.50)**	0.008
JHotDraw	**0.69 (3.00)**	**0.73 (2.00)**	0.52 (6.00)	**0.72 (2.50)**	0.55 (5.50)	0.51 (6.00)	0.45 (6.50)	0.60 (4.50)	1.2E-14
MyBatis	**0.70 (2.00)**	0.61 (5.00)	0.45 (7.50)	**0.75 (2.00)**	**0.69 (2.00)**	0.49 (7.00)	0.61 (5.00)	0.59 (5.50)	2.2E-16
Mean	**0.84 (3.21)**	**0.84 (3.71)**	**0.70 (5.57)**	**0.84 (3.14)**	**0.78 (4.07)**	**0.71 (5.57)**	**0.66 (5.86)**	**0.75 (4.86)**	3.606E-4

sults with a mean rank difference of only 0.07 to I_Best and with no statistical difference according to the Friedman and Kruskal-Wallis tests. The only system (TollSystem) in which GE_Best is statistically worse than the other algorithms is in fact the smallest one. The best known solution for this system is always found by GE_Median. Furthermore, the median and worst algorithms obtained by the GE algorithm perform better than the median and worst algorithms of `irace` respectively.

The Friedman test presents differences only between GE_Best and NSGA-II, GE_Median and NSGA-II, and GE_Best and I_Worst. Even though I_Best obtains the best results (equal hypervolume to GE_Best and GE_Median, but better rank), the Friedman test result shows no statistical differences between I_Best and any other algorithm. Perhaps with more systems in the experiment, the statistical power would be higher and we could see more differences between the algorithms. With only seven systems, the more conservative statistical assessment of the non-parametric Friedman test takes place, preventing us from assuming and report other differences.

Figures 4-6 show the fronts obtained by the algorithms. We present only the fronts of the three more complex systems based on the number of dependencies presented on Table 1. The other instances are omitted because the algorithms found similar solutions for them.

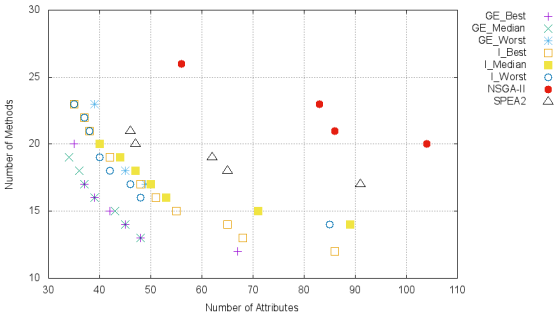

Figure 4. Fronts found by the algorithms for *AJHotDraw*

Based on the presented fronts, for the AJHotDraw instance, the solutions found by NSGA-II and SPEA2 are dominated by the ones found using the other algorithms. Moreover, the GE algorithms obtain better solutions. For the MyBatis instance, the best solutions are obtained by I_Best and GE_Best. For the JHotDraw instance, I_Best found two non-dominated solutions, and one of them is also found by GE_Best and GE_Worst. In general, with exception of JHotDraw, the algorithms are able to find diversified solutions in the fronts.

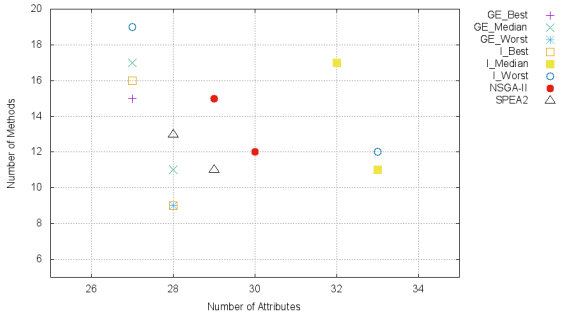

Figure 5. Fronts found by the algorithms for *JHotDraw*

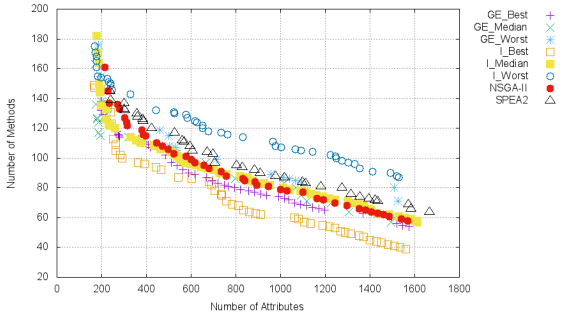

Figure 6. Fronts found by the algorithms for *MyBatis*

In order to provide another way to analyse the differences between the algorithms, we present the Vargha-Delaney's $\hat{A}_{12}$ effect size (Vargha and Delaney, 2000; Arcuri and Briand, 2014). The $\hat{A}_{12}$ test measures the probability of an effect magnitude regarding the difference between a group X and another group Y, where an $\hat{A}_{12}$ value of 1.0 means that X always outperforms Y, 0.0 means that Y always outperforms X, and 0.5 means that X and Y are equally good. According to Vargha and Delaney (2000), the $\hat{A}_{12}$ values can be read as follows:

$$\hat{A}_{12}\text{ magnitude} = \begin{cases} Large & \text{if } 0.71 < \hat{A}_{12} \leq 1.0 \\ Medium & \text{if } 0.64 < \hat{A}_{12} \leq 0.71 \\ Small & \text{if } 0.56 < \hat{A}_{12} \leq 0.64 \\ Negligible & \text{if } 0.44 \leq \hat{A}_{12} \leq 0.56 \\ Small & \text{if } 0.36 \leq \hat{A}_{12} < 0.44 \\ Medium & \text{if } 0.29 \leq \hat{A}_{12} < 0.36 \\ Large & \text{if } \hat{A}_{12} < 0.29 \end{cases}$$

This statistical test is computed over the 30 hypervolume values of each algorithm and compared to the other ones in each instance. Because this is a binary comparison, there

would be 196 effect size values to report: 28 values for each of the seven instances. In this sense, for a cleaner view of such data, we summarize these results in Figures 7 and 8, and in Table 6. The figures present respectively the box plot for the $\hat{A}_{12}$ effect size values obtained in the comparisons between I_Best and the other algorithms, and between GE_Best and the other algorithms in all instances. We chose these two algorithms because they were the best obtained algorithms by each generation algorithm. Any value above 0.5 is a better $\hat{A}_{12}$ in favour of the main algorithm.

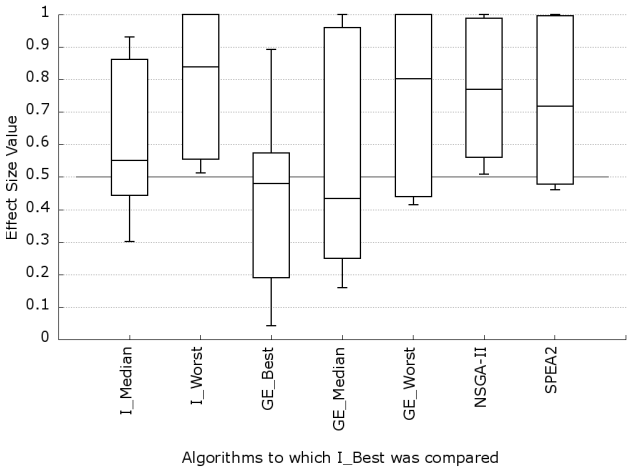

Figure 7. Box plot of the effect size values for I_Best

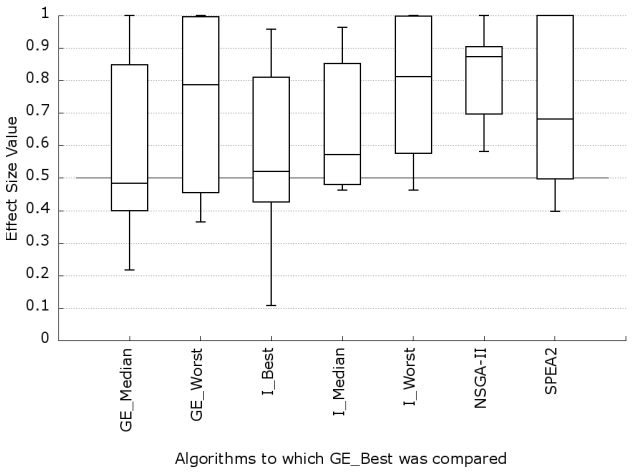

Figure 8. Box plot of the effect size values for GE_Best

I_Best performs similarly to GE_Best and, as expected, both perform better than NSGA-II and SPEA2. Furthermore, the differences between the best, median and worst algorithms become clearer. For example, in Figure 7 we can see that GE_Best obtained closer results to I_Best, then GE_Median and then GE_Worst. This "ladder" is also visible in Figure 8 regarding I_Best, I_Median and I_Worst.

Table 6 presents the number of non-negligible favourable effect size values minus the number of non-negligible unfavourable effect size values for each algorithm in a given row when compared to another algorithm in the respective column. For example, GE_Best (first row) obtained 5 favourable, 1 negligible, 1 unfavourable effect size values when compared to GE_W (fourth column), hence the difference is 4 (5 favourable minus 1 unfavourable). For each cell, the maximum value is 7 and the minimum is -7, representing, respectively, that the row algorithm was significantly better or worse than the column algorithm in all systems. The last column shows the sum of the differences.

Table 6. Non-negligible effect size counts per algorithm.

Alg.	GE_B	GE_M	GE_W	I_B	I_M	I_W	NSGA-II	SPEA2	Total
GE_B	—	0	4	0	5	6	7	4	26
GE_M	0	—	6	2	3	7	4	5	27
GE_W	-4	-6	—	-3	-3	1	3	-2	-14
I_B	0	-2	2	—	2	5	6	4	17
I_M	-5	-3	3	-2	—	6	7	1	7
I_W	-6	-7	-1	-5	-6	—	2	-4	-27
NSGA-II	-7	-4	-3	-6	-7	-2	—	-3	-32
SPEA2	-4	-5	2	-4	-1	4	3	—	-5

We could observe some ties in the number of effect size counts between some of the algorithms. GE_Best tied with GE_Median and I_Best, whereas I_Best only tied with GE_Best and not with GE_Median. Furthermore, GE_Median also obtained the best total sum of count differences, even though its hypervolume average is the same and the ranking is lower than GE_Best and I_Best. This indicates that these algorithms performed very similarly in the testing set used in this work, which can also be observed in the proximity between their Pareto fronts shown in Figures 4-6. NSGA-II shows the worst results in the effect size comparisons, whereas SPEA2 is more competitive to the other algorithms by overcoming the worst generated algorithms.

Comparing the generated algorithms and the traditional ones, we observe that all generated MOEAs are able to outperform NSGA-II and obtain competitive results to SPEA2. Additionally, some algorithms performed significantly better than the traditional ones, specially I_Best, GE_Best and GE_Median. In this sense, we can answer RQ1 by asserting that the proposed approach is capable of generating MOEAs that are better than conventional ones in the literature.

Regarding RQ2, it is not possible to conclude which design algorithm is better for this problem. `irace` obtains the best overall MOEA, but GE generates a better set of algorithms considering the median and worst generated algorithms. In other words, both best and median algorithms of GE are able to statistically outperform NSGA-II and sometimes SPEA2, which occurs less often for the MOEAs generated by `irace`. However, `irace` generated the best algorithm considering hypervolume values, rankings, and statistical differences across all systems.

4.3 Further Discussion

An advantage of GE is that its grammar provides a flexible way of design algorithms. By using such a grammar, the tester can create complex structures by changing the arrangement of the grammar rules. This enables not only the generation of MOEAs with a common template, but also MOEAs using different procedures (e.g. multiple crossover and mutation). On the other hand, GE needs more configuration (number of parameters) than `irace`.

Considering the structure of the generated MOEAs, probably a tester would not think about designing such uncommon

algorithms, since they are quite different from well-known MOEAs presented in the literature. Table 7, presents the components and parameters used by the best and worst algorithms generated by GE and IRACE. For instance, GE_Best uses a mutation operator with 100% of probability and no crossover operator. Of course, these characteristics are dependent on the training instance we used, but they are contradictory to the "common wisdom" of low mutation and great crossover probabilities. Different training scenarios could provide a more human-like designed algorithm, but yet the manual design of MOEAs may not be so powerful when compared to automated mechanisms such as the approach herein proposed. In fact, as stated by Eiben and Smit (2011), the design and configuration of an EA is an optimization problem itself, hence optimization algorithms may be successfully applied in this context.

Beyond the promising results of the proposed approach, it also requires reduced effort to design and configure a multi-objective algorithm. In a conventional scenario, the tester might avoid the boring tasks of configuring or tuning the MOEA, and may simply use a common MOEA with arbitrary parameters and components, which can yield not so good results. The proposed approach can solve this problem by automatically generating MOEAs that present the best results.

The downside of the approach is the great amount of computational resources used in the training phase. In this paper we executed the design algorithms for 10,000 evaluations, and assigned 2,000 training evaluations for each MOEA, resulting in 20 million total fitness evaluations each training run. Each training run took 113 hours on average to execute due to the great number of fitness evaluations during the training. A more cheaper strategy could be adopted by reducing from 10,000 to 5,000 fitness evaluations, and from 2,000 to 1,000 training budget for each MOEA, however, we would still be using 5 million fitness evaluations for the training (around 28 hours for training time). This elevated cost is not something exclusive of our approach, but rather something that is inevitably inherent to offline design algorithms (Bezerra et al., 2014; Guizzo et al., 2015; Burke et al., 2012).

However, one should bear in mind that the 20 million fitness evaluations were also assigned to the tuning of NSGA-II and SPEA2. Even when we assign the same amount of resources, automatically tuning an existing algorithm results in suboptimal results. If we consider the manual effort of designing a MOEA or tuning an existing one, then this effort must be added to the computational resources spent in the tuning process. In such case, the tester would have to perform experiments and compare the algorithms manually, which is already done automatically by our approach. All in all, letting our approach design and configure a new algorithm seems like the best option, despite its cost.

Although a great number of MOEAs evaluations were used for the training, we discovered that the best algorithm is found, on average for the 30 training executions, around the $7{,}120^{th}$ evaluation out of 10,000. In some executions the best algorithm was found as early as the first 3,000 evaluations, but sometimes it was not found until the last 100 evaluations. This data will be important for future work, where we are going to investigate a better stopping criterion for the training in order to reduce the cost of this phase.

4.4 Threats to Validity

The experimentation test was only performed using seven systems. In order to minimise this external threat, we extracted systems of different sizes and characteristics from related work (Assunção et al., 2014; Mariani et al., 2016; Guizzo et al., 2015). It is worth mentioning that the systems are implemented with two different paradigms: object and aspect oriented. All of these details help mitigate the threat and improve the generalisation of our results.

We only used AJHSQLDB as the training instance on both GE and `irace`. Even though in an earlier and preliminary experimentation we observed similar results using MyBatis as training instance, this may affect the final output. We expect a degradation on the results when using smaller training instances, thus this is something we intend to test in future work.

The results are susceptible to the components and parameters used in the training phase. In this sense, a greater number of components and parameters would result in different MOEAs and, probably, better algorithms. In order to minimise this threat and improve generalisation, we used a comprehensive set of components from multiple different algorithms in the literature.

Another possible threat is that due to the execution costs, in the training phase we execute each design algorithm (GE and `irace`) 10 times. This number was chosen based on other works in the literature of automatic design algorithms (**??**). We think it is enough to show the behaviour of our approach and offer a preliminary evaluation.

During the testing phase we used only the hypervolume indicator due to its compatibility to our scenario and because it is considered the only Pareto-dominance compliant indicator (Zitzler et al., 2007). We tried to mitigate this threat by also presenting the Pareto fronts found during the evaluation and making our experimental results available.

5 Related Work

This section reviews studies addressing the ITO problem and after related work on automatic design of algorithms.

5.1 Studies addressing the ITO problem

In the literature the ITO problem has been addressed by many studies and different contexts. The work of Hewett and Kijsanayothin (2009) is focused on the integration order of components that can be a subsystem, module, or object-oriented class. The authors use a graph-based algorithm to solve the problem and reduce the number of required stubs. The heuristics were applied in test dependency graphs and the results showed an improvement of the results when compared to other approaches. The authors also stated that their algorithm was faster to be executed than these approaches. Other works such as Abdurazik and Offutt (2009) and the ones reviewed by Briand et al. (2003) also use graph-based algorithms and other similar heuristics to solve this problem in the Object Oriented (OO) context. The approach of Jiang et al. (2019) introduces a strategy to consider the control coupling complexity estimated by using the concept of transitive relationship

Table 7. Components and parameters of the best and worst algorithms generated by GE and IRACE

Component/Parameter	GE_Best	GE_Worst	IRACE_Best	IRACE_Worst
Population size	50	100	50	50
Initialization	Random	Random	Parallel Diversified	Parallel Diversified
Selection	Ranking	Ranking	Ranking	Ranking
→ Converg. Strategy	Dominance Strength	Raw Fitness	Dominance Strength	Dominance Strength
→ Divers. Strategy	K-th Nearest Neighbour	–	Adaptive Grid	K-th Nearest Neighbor
→ Source	Archive and Population	Population	Population	Archive and Population
Mating Strategy	Steady State	Steady State	Steady State	Steady State
Crossover Operator	-	PMX Crossover (80%)	Single Point Crossover (80%)	Single Point Crossover (90%)
Mutation Operator	Swap Mutation (100%)	Swap Mutation (100%)	Swap Mutation (100%)	Swap Mutation (100%)
Replacement	Generational	Generational	Ranking	Generational
→ Elitism Size	5	10	0	1
→ Converg. Strategy	Raw Fitness	Dominance Strength	Raw Fitness	Dominance Strength
→ Divers. Strategy	K-th Nearest Neighbour	Crowding Distance	K-th Nearest Neighbour	–
Archive	Ranking	–	–	Ranking
→ Converg. Strategy	Raw Fitness	–	–	Dominance Strength
→ Divers. Strategy	Hypervolume Contribution	–	–	Adaptive Grid
→ Archive Size	75	–	–	75

between classes. Other works such as Bansal et al. (2009) propose hybrid algorithms combining graph-based and genetic algorithms also in OO context. The work of Ré et al. (2007) proposed an extension to encompass Aspect-Oriented (AO) programs.

Even though the mentioned works can solve the integration and testing order problem, sometimes they may only find sub-optimal solutions (Assunção et al., 2014; Briand et al., 2002b). To overcome this situation, some works such as Assunção et al. (2013); Briand et al. (2002b); Vergilio et al. (2012b) use search-based approaches. In Briand et al. (2002b) the author proposed a way to optimally solve this problem by using GAs. However, the authors used an aggregation of coupling measures as fitness function, in contrast to Assunção et al. (2013) and Vergilio et al. (2012b), where multi-objective approaches were used and good results were obtained. Cabral et al. (2010) also used a multi-objective search-based algorithm, more specifically Ant Colony Optimization (ACO), and obtained better results than an approach with aggregation of objectives.

We can observe that MOEAs obtained the best results to solve the ITO problem, and this means that it a suitable problem for applying our MOEA design approach. From all existing works addressing the ITO problem, the work of Assunção et al. (2014) introduced an approach that can be applied in different contexts of the problem, such as AO and OO programs, because of this we used such a formulation in our work.

5.2 Works on automatic design of algorithms

In the literature we find works on automatic design and configuration of meta-heuristics that use different techniques (Bezerra et al., 2014, 2015; López-Ibáñez and Stützle, 2012; Smit et al., 2010; Dréo, 2009). Some of them are focused only on the automatic configuration of parameters of mono-objective algorithms (Smit et al., 2010; Dréo, 2009). Related work are the ones that use GE and `irace`.

We can find in the literature some works that use GE for the automatic design and configuration of heuristics (Lourenço et al., 2012, 2013, 2015; Burke et al., 2012; Mascia et al., 2014; Marshall et al., 2014b,a; Hutter et al., 2007). They focus on simple heuristics (Marshall et al., 2014b,a) and meta-heuristics, such as local search algorithms (Burke et al., 2012; Mascia et al., 2014; Hutter et al., 2007) and EAs (Lourenço et al., 2012, 2013, 2015). We would like to mention the works of Lourenço et al. (2012, 2013, 2015) that propose a GE approach to automatically design and configure mono-objectives EAs. Their grammar encompasses the main components and parameters of EAs, such as population size, crossover and mutation operators. The EAs returned by the approach obtained better results when compared to traditional ones on the Royal Roads problem. However, we did not find works applying GE in the context of MOEAs, or even for other types of multi-objective algorithms.

Regarding the use of `irace`, relate work (Bezerra et al., 2014, 2015; López-Ibáñez and Stützle, 2012) also consider EAs and a set of components and parameters. The studies of Bezerra et al. (2014, 2015) are the only ones addressing the context of MOEAs. They use many MOEAs components and parameters, such as replacement, archiving and fitness assignment. As result, the automatically generated MOEAs could outperform traditional ones in instances of combinatorial and continuous problems.

Our work is mainly inspired by Bezerra et al. (2014, 2015) and Lourenço et al. (2012), because the idea of using GE and `irace` for the automatic design of EAs seems very promising. Furthermore, in the automatic design of MOEAs, there are many parameters and components that can be explored. However, our approach differs from them by using a different set of components and parameters, such as initialization procedures, source of parents selection, replacements procedures and the possibility of elitism. The set herein proposed is broader and more suitable for the MOEAs context.

Another difference is the application of automatic design of MOEAs in the SBSE area. None of the works mentioned above generate algorithms specialized for solving software engineering problems, such as ITO.

Regarding the ITO problem, a great number of approaches found in the literature use search-based algorithms (Wang

et al., 2011; Briand et al., 2002a). The most promising are based on MOEAs (Assunção et al., 2014; Vergilio et al., 2012a; Guizzo et al., 2015). However, to implement and configure a MOEA solution can be very difficult for the tester, who is not usually an expert on the optimization field. To ease such a task, the work of Mariani et al. (2016) introduces a hyper-heuristic based on GE to derive MOEAs for solving the ITO problem. Such a hyperheuristic was also used to select products for the Sofware Product Line (SPL) testing (Jakubovski Filho et al., 2017). These works found good results and serve as motivation to propose our approach, that now encompasses a set of elements and components that allow the use of GE and `irace` and comparison of both design algorithms. As far as we know, there is no work in the literature that applies and compares both.

6 Concluding Remarks

This paper presented an approach for the automatic design of MOEAs applied to the ITO problem. For this purpose, the main components and parameters of MOEAs were identified and a set of alternatives were assigned for them. They were formally mapped in a grammar (GE) and in a parameter space (`irace`) to be used by the design algorithms.

An empirical evaluation was conducted in two phases. The first was the training phase, where the GE and `irace` algorithms were executed using one instance of ITO, generating ten MOEAs each. In the testing phase, the twenty obtained MOEAs were executed using all seven instances of the problem. Furthermore, the traditional MOEAs NSGA-II and SPEA2 were also executed using such instances. Their parameters were tuned using the proposed approach.

The MOEAs generated by the approach with both design algorithms were compared using the hypervolume quality indicator and three statistical tests: Vargha-Delaney's $\hat{A}_{12}$ Effect Size, Kruskal-Wallis and Friedman. They obtained similar results and also some similarities regarding the most selected components. In addition, the generated MOEAs are able to outperform the traditional ones with statistical difference.

In short, the introduced approach can successfully generate MOEAs to better solve the ITO problem, improving the performance of the traditional algorithms and reducing effort, freeing the tester from tasks like: choice among existing MOEAs, implementation and configuration.

We intend to apply the approach for other ITO instances, testing contexts and problems. Some mentioned limitations should be addressed, for example, to use other training instances and quality indicators in the fitness evaluation.

In addition to this, the approach serves as a basis for future works that can extend the proposed set of elements to cover other desired characteristics of the MOEAs and to be used in other domains, contributing to the SBSE practitioners to implement solutions for other SE problems. Ideally, the generated MOEAs could be generalized to other SE problems without the need of retraining.

Acknowledgement

This work is supported by the Brazilian funding agencies Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under the grant: 305968/2018-1.

We would like to thank Gian M. Fritsche for his help on the setup of `irace`.

References

Abdurazik, A. and Offutt, J. (2009). Using coupling-based weights for the class integration and test order problem. *Computer Journal*, 52(5):557–570.

Arcuri, A. and Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250.

Assunção, W. K. G., Colanzi, T. E., Vergilio, S. R., and Pozo, A. T. R. (2013). Generating integration test orders for aspect oriented software with multi-objective algorithms. *Revista de Informática Teórica e Aplicada*, 20(2):301–327.

Assunção, W. K. G., Colanzi, T. E., Vergilio, S. R., and Pozo, A. (2014). A multi-objective optimization approach for the integration and test order problem. *Information Sciences*, 267(0):119–139.

Bansal, P., Sabharwal, S., and Sidhu, P. (2009). An investigation of strategies for finding test order during integration testing of object oriented applications. In *Proceedings of the ICM2CS*, pages 1–8.

Barros, R. C., Basgalupp, M. P., Cerri, R., da Silva, T. S., and de Carvalho, A. C. (2013). A Grammatical Evolution Approach for Software Effort Estimation. In *Genetic and Evolutionary Computation Conference*, pages 1413–1420.

Bezerra, L., López-Ibáñez, M., and Stützle, T. (2015). Automatic component-wise design of multi-objective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 20:1–1.

Bezerra, L. C. T., López-Ibáñez, M., and Stützle, T. (2014). Automatic Design of Evolutionary Algorithms for Multi-Objective Combinatorial Optimization. In *Parallel Problem Solving from Nature*, volume 8672 of *Lecture Notes in Computer Science*, pages 508–517. Springer International Publishing.

Briand, L., Labiche, Y., and Wang, Y. (2003). An investigation of graph-based class integration test order strategies. *Software Engineering, IEEE Transactions on*, 29(7):594–607.

Briand, L. C., Feng, J., and Labiche, Y. (2002a). Using genetic algorithms and coupling measures to devise optimal integration test orders. In *International Conference on Software Engineering and Knowledge Engineering*, pages 43–50.

Briand, L. C., Feng, J., and Labiche, Y. (2002b). Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th SEKE*, pages 43–50. ACM.

Burke, E. K., Hyde, M. R., and Kendall, G. (2012). Grammatical Evolution of Local Search Heuristics. *IEEE Transactions on Evolutionary Computation*, 16(3):406–417.

Cabral, R. d. V., Pozo, A., and Vergilio, S. R. (2010). A pareto ant colony algorithm applied to the class integration and test order problem. In Petrenko, A., SimÃ£o, A., and Maldonado, J. C., editors, *Testing Software and Systems*, volume 6435 of *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg.

Coello, C. A. C., Lamont, G. B., and Veldhuizen, D. A. V. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition*. Springer Science, 2nd edition.

Colanzi, T., Assunção, W., Farah, P., Vergilio, S., and Guizzo, G. (2019). A review of ten years of the symposium on search-based software engineering. In Nejati, S. and Gay, G., editors, *Search-Based Software Engineering*, pages 42–57, Cham. Springer International Publishing.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

Derrac, J., García, S., Molina, D., and Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18.

Dréo, J. (2009). Using Performance Fronts for Parameter Setting of Stochastic Metaheuristics. In *Genetic and Evolutionary Computation Conference*, pages 2197–2200.

Durillo, J. J. and Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771.

Eiben, A. and Smit, S. (2011). Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31.

Eiben, A. and Smit, S. (2012). Evolutionary Algorithm Parameters and Methods to Tune Them. In *Autonomous Search*, pages 15–36. Springer Berlin Heidelberg.

Eiben, A. E. and Smith, J. E. (2003). *Introduction to evolutionary computing*. Springer Science & Business Media.

Fonseca, C. M. and Fleming, P. J. (1993). Genetic Algorithms for Multiobjective Optimization: Formulation Discussion and Generalization. In *International Conference on Genetic Algorithms*, pages 416–423.

Guizzo, G., Fritsche, G. M., Vergilio, S. R., and Pozo, A. T. R. (2015). A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem. In *Genetic and Evolutionary Computation Conference*, pages 1343–1350.

Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys*, 45(1):1–61.

Hewett, R. and Kijsanayothin, P. (2009). Automated test order generation for software component integration testing. In *Proceedings of the 24th ASE*, pages 211–220.

Hutter, F., Hoos, H. H., and Stützle, T. (2007). Automatic Algorithm Configuration Based on Local Search. In *AAAI Conference on Artificial Intelligence*, pages 1152–1157.

Jakubovski Filho, H. L., Lima, J. A. P., and Vergilio, S. R. (2017). Automatic generation of search-based algorithms applied to the feature testing of software product lines. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 114–123, New York, NY, USA. ACM.

Jiang, S., Zhang, M., Zhang, Y., Wang, R., Yu, Q., and Keung, J. W. (2019). An integration test order strategy to consider control coupling. *IEEE Transactions on Software Engineering*, pages 1–1.

Knowles, J. D. and Corne, D. W. (2000). Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation*, 8(2):149–172.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., and Birattari, M. (2011). The IRACE package, Iterated Race for Automatic Algorithm Configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium.

López-Ibáñez, M. and Stützle, T. (2012). The Automatic Design of Multiobjective Ant Colony Optimization Algorithms. *IEEE Transactions on Evolutionary Computation*, 16(6):861–875.

Lourenço, N., Pereira, F. B., and Costa, E. (2012). Evolving evolutionary algorithms. In *Genetic and Evolutionary Computation Conference*, pages 51–58.

Lourenço, N., Pereira, F. B., and Costa, E. (2013). The Importance of the Learning Conditions in Hyper-heuristics. In *Genetic and Evolutionary Computation Conference*, pages 1525–1532.

Lourenço, N., Pereira, F. B., and Costa, E. (2015). The Optimization Ability of Evolved Strategies. In *Progress in Artificial Intelligence*, volume 9273 of *Lecture Notes in Computer Science*, pages 226–237. Springer International Publishing.

Mariani, T., Guizzo, G., Vergilio, S., and Pozo, A. (2016). Grammatical evolution for the multi-objective integration and test order problem. In *Genetic and Evolutionary Computation Conference*, pages 1069–1076.

Marshall, R. J., Johnston, M., and Zhang, M. (2014a). Developing a Hyper-Heuristic Using Grammatical Evolution and the Capacitated Vehicle Routing Problem. In *Simulated Evolution and Learning*, volume 8886 of *Lecture Notes in Computer Science*, pages 668–679. Springer International Publishing.

Marshall, R. J., Johnston, M., and Zhang, M. (2014b). Hyper-heuristics, Grammatical Evolution and the Capacitated Vehicle Routing Problem. In *Genetic and Evolutionary Computation Conference*, GECCO Comp '14, pages 71–72.

Mascia, F., nez, M. L.-I., Dubois-Lacoste, J., and Stützle, T. (2014). Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, 51:190–199.

O'Neill, M. and Nicolau, M. (2017). Distilling the salient features of natural systems: Commentary on "On the mapping of genotype to phenotype in evolutionary algorithms" by Whigham, Dick and Maclaurin. *Genetic Programming*

and Evolvable Machines, 18(3):379–383.

Ré, R., Lemos, O. A. L., and Masiero, P. C. (2007). Minimizing stub creation during integration test of aspect-oriented programs. In *Proceedings of the 3rd Workshop on Testing Aspect-oriented Programs*, WTAOP '07, pages 1–6, New York, NY, USA. ACM.

Ryan, C. (2017). A rebuttal to Whigham, Dick, and Maclaurin by one of the inventors of Grammatical Evolution: Commentary on "On the Mapping of Genotype to Phenotype in Evolutionary Algorithms" by Peter A. Whigham, Grant Dick, and James Maclaurin. *Genetic Programming and Evolvable Machines*, 18(3):385–389.

Ryan, C., Collins, J. J., and Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96. Springer Berlin Heidelberg.

Siegel, S. and Castellan, N. J. (1988). *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill international editions. Statistics series. McGraw-Hill.

Smit, S. K., Eiben, A. E., and Szlávik, Z. (2010). An MOEA-based Method to Tune EA Parameters on Multiple Objective Functions. In *International Joint Conference on Computational Intelligence*, pages 261–268.

Talbi, E.-G. (2009). *Metaheuristics: From Design to Implementation*. Wiley Publishing.

Vargha, A. and Delaney, H. D. (2000). A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

Vergilio, S., Pozo, A., Árias, J., Cabral, R., and Nobre, T. (2012a). Multi-objective Optimization Algorithms Applied to the Class Integration and Test Order Problem. *International Journal on Software Tools for Technology Transfer*, 14(4):461–475.

Vergilio, S. R., Pozo, A., Árias, J. a. C. G., da Veiga Cabral, R., and Nobre, T. (2012b). Multi-objective optimization algorithms applied to the class integration and test order problem. *International Journal on Software Tools for Technology Transfer*, 14(4):461–475.

Wang, Z., Li, B., Wang, L., and Li, Q. (2011). A Brief Survey on Automatic Integration Test Order Generation. In *International Conference on Software Engineering and Knowledge Engineering*, pages 254–257.

Whigham, P. A., Dick, G., and Maclaurin, J. (2017a). Just because it works: a response to comments on "On the Mapping of Genotype to Phenotype in Evolutionary Algorithms". *Genetic Programming and Evolvable Machines*, 18(3):399–405.

Whigham, P. A., Dick, G., and Maclaurin, J. (2017b). On the mapping of genotype to phenotype in evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 18(3):353–361.

Zitzler, E., Brockhoff, D., and Thiele, L. (2007). The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In Obayashi, S., Deb, K., Poloni, C., Hiroyasu, T., and Murata, T., editors, *Evolutionary Multi-Criterion Optimization*, pages 862–876, Berlin, Heidelberg. Springer Berlin Heidelberg.

Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: improving the strength Pareto evolutionary algorithm. Technical report, Department of Electrical Engineering, Swiss Federal Institute of Technology.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.