An Extensible JCLEC-based Solution for the Implementation of Multi-Objective Evolutionary Algorithms

Aurora Ramírez, José Raúl Romero and Sebastián Ventura
Dept. of Computer Sciences and Numerical Analysis, University of Córdoba
Campus of Rabanales, 14071 Córdoba, Spain
{aramirez, jrromero, sventura}@uco.es

ABSTRACT

The ongoing advances in multi-objective optimisation (MOO) are improving the way that complex real-world optimisation problems, mostly characterised by the definition of many conflicting objectives, are currently addressed. To put it into practice, developers require flexible implementations of these algorithms so that they can be adapted to the problem-specific needs. Here, metaheuristic optimisation frameworks (MOFs) are essential tools to provide end-user oriented development solutions. Even though consolidated MOFs are continuously evolving, they seem to have paid little attention to the new trends in MOO. Recently, new frameworks have emerged with the aim of providing support to these approaches, but they often offer less variety of basic functionalities like diversity of encodings and operators than other general-purpose solutions. In this paper we identify a number of relevant features serving to satisfy the requirements demanded by MOO nowadays, and propose a solution, called JCLEC-MOEA, on the basis of the JCLEC framework. As a key contribution, its architecture has been designed with a twofold purpose: reusing all the features already given by a mature framework like JCLEC, and extending it to enable new developments more flexibly than current alternatives.

Categories and Subject Descriptors

[**Computing methodologies**]: Artificial intelligence -*Search methodologies*; [**Software and its engineering**]: Software creation and management -*Designing software*

Keywords

Multi-objective evolutionary algorithms, many-objective evolutionary algorithms, metaheuristic optimisation framework, JCLEC

1. INTRODUCTION

Multi-objective optimisation problems (MOPs) are likely to be the most frequently happening problems in real-world applications. In the last decades, multi-objective evolutionary algorithms (MOEAs) [1] have been proposed to effectively deal with a wide range of MOPs like those appearing in economics [2] or engineering [3] domains. Recently, the growing interest in the resolution of problems having a large number of conflicting objectives has caused the appearance of a new kind of specialised algorithms, the so-called many-objective approaches. In this context, authors have adapted some previous ideas and proposed new techniques to provide solutions that are capable of covering significantly more complex objective spaces [4].

Thus, in a short space of time, practitioners are being provided with a wide range of alternatives to support their specific MOO processes. In this scenario, metaheuristic optimisation frameworks (MOFs) [5] play a key role since, even when they were originally designed for experimental purposes, they provide mechanisms for the existing algorithms to be more easily adaptable to the complexity of each specific problem domain. An essential aspect in achieving this is to provide flexible algorithm implementations as a way to facilitate the application of recent trends within the evolutionary computation (EC). Improving the modularity of algorithms and their reusability by making them publicly available could encourage developers to include these algorithms in their code while decreasing the development effort, and researchers to perform further empirical studies using these new techniques while increasing the testing effectiveness. The MOF extensibility allows taking advantage of the most modern techniques to address MOPs on different application areas, even though such a solution would require the development of new domain-specific operators or defining constraints. Additionally, MOFs provide complete and configurable workbenches, where researchers can easily check diverse configurations on a given MOP, collect statistical outcomes or analyse the returned solutions.

Because of the large number of MOFs currently supporting EC, it can be usual that they all provide some common facilities, differences among them mostly lying on the presence of advanced features of interest for specific developments like genetic programming (GP) or grammatical evolution (GE). Focusing on multi-objective optimisation (MOO), many mature frameworks like ECJ [6] have not yet integrated multi-objective approaches beyond the implementation of some consolidated proposals like NSGA-II. This has led to the appearance of specific frameworks like

GECCO '15 Companion, July 11–15, 2015, Madrid, Spain

ISBN 978-1-4503-3488-4/15/07. . . $15.00
DOI: http://dx.doi.org/10.1145/2739482.2768461

jMetal [7] or MOEA Framework [8], mostly focused on recent advances within MOO. As a counterpart, they lack of support in terms of the variety of genetic operators or other advanced EC techniques usually provided by the aforementioned frameworks.

This paper compiles the set of key features required by a MOF to properly accomplish the expectations of both developers and researchers regarding MOO, considering both the implementation of new multi-objective end-user-oriented approaches and their experimental support. More specifically, we present JCLEC-MOEA, an adapted architecture founded on the basis of JCLEC (Java Class Library for Evolutionary Computation) [9], with the aim of filling the gap between this framework and the features of interest for MOO practitioners. The proposed architecture serves as a platform to develop new MOO-specific techniques, where special emphasis has been put on modularity and extensibility, as well as on the reusability of the currently available utilities in JCLEC.

Regarding the extensibility mechanisms, JCLEC-MOEA does not only make a clear distinction between the different steps of an evolutionary algorithm, not being considered as a single monolithic piece of code, but it also considers what stages of the evolution are revisited by the multi-objective approach and consequently should be clearly differentiated from the rest of the process. As a result, steps of the search process can be customised to create specific variants of an evolutionary approach. Additionally, algorithms also become independent of the type of EC model to be applied, i.e. genetic algorithm (GA), evolution strategy (ES), genetic programming or evolutionary programming (EP), allowing to take advantage of all the facilities of JCLEC in this regard. Some recently proposed algorithms within both categories, multi- and many-objective algorithms, like HypE, MOEA/D or NSGA-III, have been implemented in line with this philosophy. Aspects like providing highly-configurable implementations or the ease of extension and reuse of a wide range of well-established EC paradigms, operators and encodings make JCLEC-MOEA an interesting alternative in comparison to other solutions.

The rest of the paper is organised as follows. First, Section 2.2 introduces JCLEC, following a brief summary other currently available MOFs. The architecture of JCLEC-MOEA is introduced and explained in Section 3, and Section 4 discusses the different ways in which JCLEC-MOEA can be extended. Finally, Section 5 outlines some conclusions and future work.

2. BACKGROUND

2.1 The JCLEC framework

JCLEC is a Java library focused on the development of evolutionary algorithms, including genetic programming. Its core provides a variety of encodings and genetic operators that can be combined to solve a specific optimisation problem. Experiments are created using a configuration file in XML format, whose tags and elements can be customised by researchers in order to include its own parameters[1]. As a result, customised operators or new optimisation problems can be added easily without requiring the code to be recompiled. The set of available evolutionary techniques is comprised of generational schemes, steady-state evolution, niching methods and evolutionary programming. As for genetic programming, Koza's style, strongly-based and grammar-based approaches can be applied. Despite being mostly focused on single-objective optimisation, JCLEC implements two well-known multi-objective approaches, SPEA2 and NSGA-II.

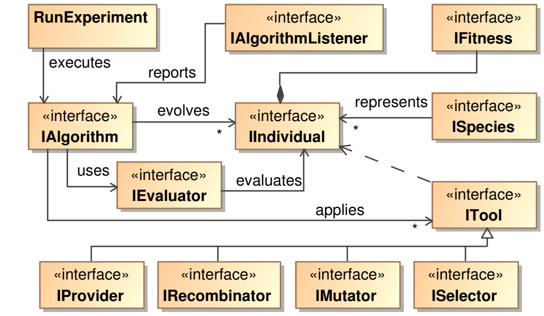

Figure 1: Core classes and interfaces in JCLEC

Classes and interfaces comprising the core of JCLEC are shown in Figure 1. `RunExperiment` is the class in charge of starting the execution of an algorithm, which is represented by the `IAlgorithm` interface. Every algorithm evolves a set of individuals (`IIndividual`), i.e. the population, and makes use of a number of tools that serve to conduct the different steps of the optimisation process: the initialisation (`IProvider`), the application of genetic operators (`IRecombinator` and `IMutator`), and the selection of individuals (`ISelector`).

An algorithm also requires the assignment of an evaluation mechanism to calculate the fitness of individuals according to the optimisation problem being solved. In JCLEC, the interface `IEvaluator` declares the evaluation method, **`evaluate`**, and two other additional abstract classes, `AbstractEvaluator` and `AbstractParallelEvaluator`, define how this evaluation task has to be performed, sequentially or in parallel, respectively. In any case, the *evaluator* is in charge of assigning a fitness value to each individual using a subclass of `AbstractFitness`, an implementation of `IFitness`.

In JCLEC, `PopulationAlgorithm` is an abstract class that implements the `IAlgorithm` interface, being responsible for defining the iterative process of an EC algorithm. Figure 2 depicts the control flow performed by this class, where italics indicate abstract operations. Subclasses of `PopulationAlgorithm`, such as `SG` (Simple Generational GA), `SS` (Steady-State GA) or the current versions of SPEA2 and NSGA-II, implement the overall search process, comprising the following key methods: `doSelection`, where a *selector* creates the mating pool; `doGeneration`, where genetic operators like *recombinators* and *mutators* are executed; `doReplacement`, where survivors are selected among the current population and offspring; and `doUpdate`, responsible for creating the next population. Several different sets of individuals, referring to the current population, parents, offspring, and survivors, are managed by this class along the evolution. Evolution can be stopped in JCLEC when a maximum number of generations or evaluations is reached. Additionally, the

[1] Configuration files are managed using the Apache Commons Configuration software library, http://commons.apache.org

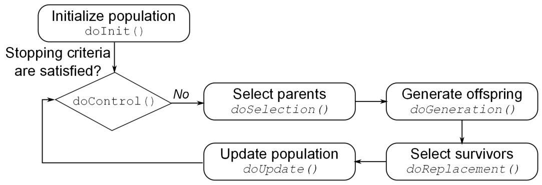

Figure 2: Iterative process of PopulationAlgorithm

execution could be interrupted if the best individual in the population achieves an acceptable value for the problem being optimised.

Finally, IAlgorithmListener specifies the set of methods used to report the outcomes and get information about the algorithm execution during the search process.

2.2 MOFs for multi-objective optimisation

A complete survey of multi-purpose metaheuristic frameworks, developed in 2011 [5], provided a review of the supported optimisation techniques by each MOF, as well as an evaluation of the facilities enabled to adapt the algorithms to different domains. The authors also revised aspects related to design issues, licensing and documentation. Because of the broad scope of this study, multi-objective metaheuristics were grouped together within a single characteristic mainly focused on the presence of certain MOEAs. Besides, they established some assumptions that resulted in the exclusion of interesting frameworks from the MOO perspective, such as jMetal, which was considered too specific, or PISA [10], because of its non-object-oriented nature.

Nevertheless, MOFs have significantly evolved since then, contributing to the appearance and improvement of software libraries. Here, a listing of the most representative frameworks supporting MOO is compiled with the aim of providing the big picture of the currently available proposals.

ECJ (2014) [6]. Developed in Java, it is likely to be the most frequently used framework because of its maturity and successive updates. ECJ has a valuable set of advanced EC models, including coevolution, GP, islands or GE, as well as other metaheuristics like particle swarm optimisation (PSO). However, ECJ only provides two MOEAs, SPEA2 and NSGA-II.

EvA2 (2014) [11]. This Java library implements metaheuristics like EC and its variants, PSO or Hill Climbing, and provides a greater number of MOEAs than ECJ. In fact, EvA2 can execute MOGA, NSGA, NSGA-II, PESA, PESA-II, Random Weight GA, SPEA, SPEA2 and VEGA.

HeuristicLab (2014) [12]. Developed for the Microsoft .NET environment using C#, this framework looks for the implementation of arbitrary heuristic optimisation algorithms, leading to an architecture based on plugins developed by a community of contributors. NSGA-II is the only MOEA publicly available for developers.

jMetal (2014) [7]. This Java framework provides a wide range of implemented algorithms, from the well-known IBEA, MOEA/D, NSGA-II, PAES or SPEA2, to other proposals like cellular algorithms and PSO. Moreover, jMetal implements a parallel version of most of them and other additional experimental utilities like quality indicators, benchmarks and statistical analysis. On the other hand, this framework does not support the variety of EC models, operators and stopping criteria provided by other frameworks.

MOEA Framework (2015) [8]. A Java framework that combines native implementations of MOEAs, such as MOEA/D or NSGA-III, with the execution of other non-native algorithms invoked from the external packages jMetal and PISA. Like jMetal, MOEA Framework also provides a variety of benchmarks and quality indicators.

Opt4J (2015) [13]. With this Java framework, optimisation problems can be solved using EC, PSO and local search methods. The multi-objective approaches currently available are MOPSO, NSGA-II, SMSMOEA and SPEA2.

ParadisEO-MOEO (2012) [14]. ParadisEO is an object-oriented framework developed in C++. It is composed of four interconnected modules, MOEO being the module specifically devoted to multi-objective optimisation. Algorithms currently available are IBEA, MOGA, NSGA, NSGA-II, SEEA and SPEA2. ParadisEO-MOEO provides a smaller collection of indicators than jMetal and MOEA Framework. Benchmarks are included as external contributions.

PISA [10] (2003). It was conceived as a language-independent platform based on the interchangeability of files, where a set of *optimizers* can be applied to a variety of optimisation problems. The set of MOEAs is comprised of ϵ-MOEA, FEMO, HypE, IBEA, MSOPS, NSGA-II, SEMO2, SHV, SPAM and SPEA2. PISA also provides implementations for most of the quality indicators defined in the literature, as well as test problems like the DTLZ and the ZDT families of functions.

PyGMO [15] (2014). A scientific library coded in Python that is mainly focused on providing the necessary support to build parallel global optimisation algorithms using the island model paradigm. The applicable techniques include PSO, genetic algorithms and differential evolution, among others. The multi-objective algorithms currently available are NSGA-II, NPSO, SMS-EMOEA, SPEA2 and VEGA.

On the one hand, frameworks like ECJ, EvA2 or HeuristicLab were conceived to support a great variety of metaheuristic techniques. Therefore, they primarily promote modularity and extensibility, providing a wide range of encodings and operators. On the other hand, more recent frameworks like jMetal or MOEA Framework are usually more focused on satisfying MOO-specific requirements, offering a wide range of algorithms, quality indicators and benchmarks. With regard to their implementation, it should be noted that some frameworks, e.g. jMetal or HeuristicLab, restrict their multi-objective approaches to genetic algorithms. Even though most of the aforementioned MOFs have launched their respective updates in the recent years, it is notorious that only jMetal, MOEA Framework and PISA have included the latest trends in multi-objective algorithms.

3. DESIGN OF JCLEC-MOEA

3.1 JCLEC-MOEA foundations

Compiling the key characteristics provided by other existing software solutions, as well as the most relevant aspects required to the resolution of MOPs, has served to identify the features that will guide the design of JCLEC-MOEA:

Built on top of a multi-purpose MOF. Having a mature MOF as the starting point is important to make the most of other basic functionalities like encodings, genetic operators or experimental support.

Ease of extension. Providing a number of extension points that properly allow integrating new algorithms, indicators and benchmarks is a key aspect to promote scalability, extensibility and integrability. It makes necessary a precise specification of the architecture, including its public interfaces and modules. Decomposition of the algorithms in different steps may also benefit extensibility, providing a greater flexibility to change their behaviour and ensuring a controlled growth.

Availability of generic algorithms. Algorithms should be abstracted in such a way that it would be possible to combine each of them with any available, current or future, EC paradigm looking for the independence of the evolution steps that are proposed by the MOO approach.

Flexibility of the MOP definition. Algorithms should work with the minimal set of restrictions according to the number of objectives, the nature of the optimisation problem, e.g. all objectives to be minimised or maximised or even combining both types, the solution encoding and the presence of constraints.

Regular updating of algorithms. The rapid advance of MOO techniques makes necessary to count on the most recent algorithm variants in order to provide developers and researchers with competitive tools and benchmarks.

Experimental support. Utilities for MOO research include the availability of well-established benchmarks and quality indicators in order to validate the performance of algorithms, as well as the generation of reports containing experimental outcomes.

JCLEC-MOEA has been conceived to offer an extensive catalogue of encodings and operators, including a tree-based encoding specifically well-suited for GP, making use of the broad range of assets available in JCLEC. The architecture proposed below allows creating highly modular algorithms that can be implemented independently of the underlying EC model. Furthermore, JCLEC-MOEA is able to deal with both minimisation and maximisation MOPs, as well as with constrained problems. It offers most of the latest algorithms in the field of MOO, including many-objective approaches, and provides other utilities to researchers like well-known benchmarks and quality indicators.

3.2 Overview of the architecture

Figure 3 shows an overview of the main classes and interfaces that compose JCLEC-MOEA, including those required to extend JCLEC in order to adapt its functionality to the specific requirements of MOO. Auxiliary classes and their relationships have been omitted to save space.

Since JCLEC-MOEA is built to the basis of JCLEC, it has required an in-depth analysis of the previous structure and future requirements in order to create a smooth but consistent adaptation between both layers. This process has led to a number of classes that enable the extensibility and interoperability between both approaches. Those that can be extended by the external developer to continue expanding the JCLEC-MOEA functionalities are identified in Figure 3 as "extension points". Firstly, `MOAlgorithm` represents a general multi-objective evolutionary algorithm, which inherits from `PopulationAlgorithm`. `MOAlgorithm` is also declared as an abstract class that is currently refined into four concrete classes, each one representing a different EC paradigm: GA, ES, GP and EP. `MOAlgorithm` also declares a property named `strategy` that is used to delegate some steps of the evolution to the specific multi-objective approach.

In JCLEC-MOEA, the evaluation of solutions can be performed sequentially or in parallel using `MOEvaluator` or `MOParallelEvaluator`, respectively. These classes inherit their properties and methods from their respective classes in the JCLEC layer and handle the set of objectives used to evaluate solutions. More specifically, the *evaluator* iteratively receives the objective value from each objective function extending the class `Objective`. The set of objectives can be easily altered to test different variants of the same MOP. Additionally, the class `Objective` serves as a basis for defining the necessary information about each objective of the MOP, i.e. its bounds and a flag indicating whether the objective should be maximised or minimised. The *evaluator* makes use of the objective values in order to generate a fitness object, represented by an instance of `MOFitness` for each individual within the population. The `IMOEvaluator` interface declares the methods that should be implemented by any *evaluator* in JCLEC-MOEA. As for the reporting functionalities, the abstract class `MOReporter` provides the facilities to extract information from the algorithm outcomes during and after the execution of any `MOAlgorithm`. Three specific *reporters* are initially provided, though it could be easily extended. A further description will be presented later in Section 3.4.

3.3 Algorithms and strategies

An important feature of JCLEC-MOEA lies on the independence between the different stages of the evolutionary process. Frequently, MOEAs contain procedures for selecting and replacing individuals, known as *mating selection* and *environmental selection*, respectively. These methods can be reinforced by some kind of evaluation method and the use of an external archive of solutions. Hence, implementations of MOEAs should not include either the initialisation process or the generation of offspring, which are actually defined by the optimisation problem and the EC model. JCLEC-MOEA adopts this schema, and delegates the control of these specific steps to a class named `MOStrategy`, an adapted implementation of the *Strategy* design pattern. Afterwards, each MOEA is implemented in terms of a different strategy that can be interchanged, i.e. each is declared as a subclass of `MOStrategy`. More precisely, `MOAlgorithm` is responsible for invoking the multi-objective strategy at certain stages of the evolution, hiding the specific procedures to be applied by the algorithm. Developers can configure the strategy by simply adapting the configuration file given by JCLEC.

The iterative process shown in Figure 2 is done in JCLEC-MOEA by combining the two aforementioned classes, `MOAlgorithm` and `MOStrategy`. In this way, a given MOO approach can be applied to any EC model, not only GAs. More in detail, `MOAlgorithm` inherits its properties and methods from `PopulationAlgorithm`, what gives the possibility of reusing generic functionalities like the creation of the initial

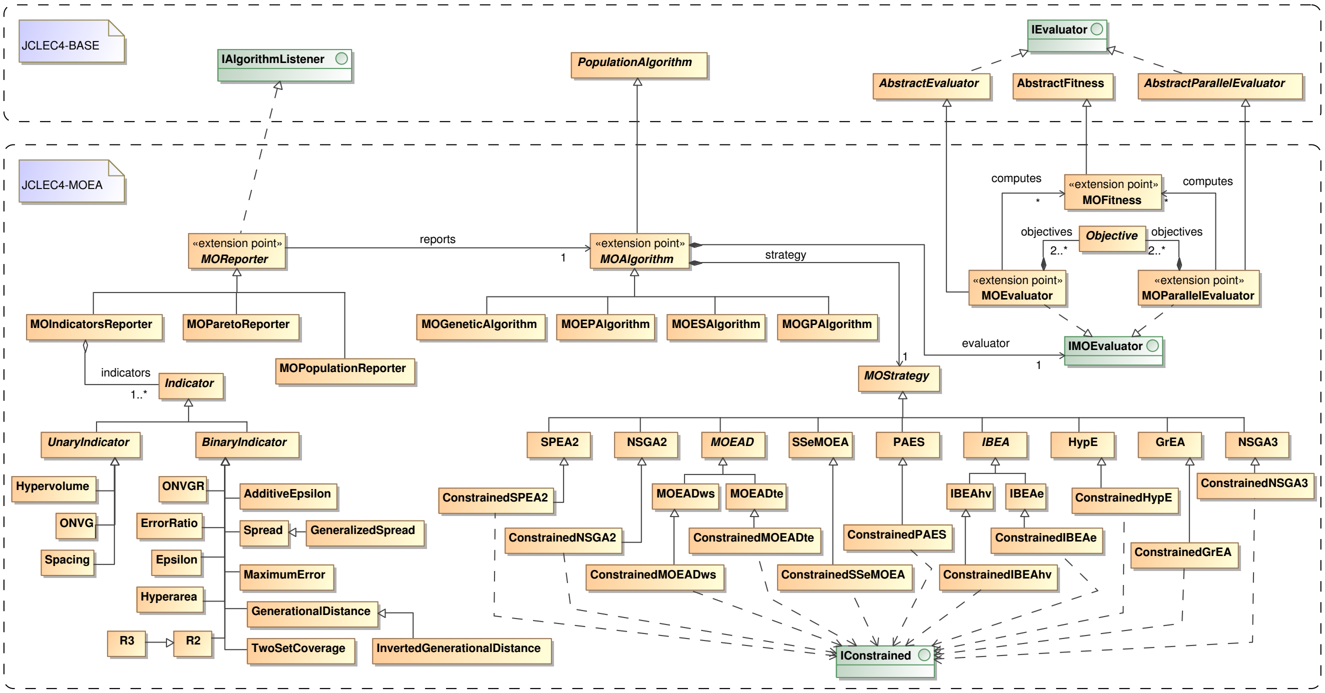

Figure 3: Class diagram of JCLEC-MOEA

population (method `doInit`) and the checking of the stopping criteria (method `doControl`). The abstract methods of `PopulationAlgorithm` are properly implemented by `MOAlgorithm`, with the exception of the `doGeneration` method. As a result, specifying how the genetic operators will be used to generate offspring is the only point that its subclasses need to care about. For example, while `MOGeneticAlgorithm` requires the configuration of a *recombinator* and a *mutator*, `EPAlgorithm` just triggers a *mutator*. In any case, these elements can be selected from the set of operators available in JCLEC. Regarding the methods for selection, replacement and update, Figure 4 shows how `MOAlgorithm` delegates these steps to `MOStrategy`. Having that `MOAlgorithm` already manages the entire set of individuals required along the search process, `MOStrategy` will request the corresponding sets required by each procedure. Next, it will return output sets to `MOAlgorithm`, including parents, survivors and the archive.

On the other hand, `MOStrategy` specifies four abstract methods: `matingSelection` for selecting parents from the current population and from the archive, `environmentalSelection` for the replacement, `updateArchive` for managing individuals within the external population, and `fitnessAssignment` for the execution of a specific evaluation process. Notice that the latter is not really invoked by `MOAlgorithm`, but this step is internally run by the strategy under certain circumstances. For example, SPEA2 evaluates individuals in terms of the strength and density values before updating the archive, whereas MOEA/D uses the evaluation method to decide whether an offspring will replace some individuals in the next generation. Two additional methods, `initialize` and `update`, can be used to implement auxiliary procedures at the beginning of the search process or after every generation, respectively. Finally, `MOStrategy` implements the interface `IConfigure` from JCLEC, which declares the necessary methods to set and retrieve the parameters from the configuration file. Splitting and modularising the different elements of a MOEA makes drastically quicker the development of variants if only the fewest number of methods need to be overridden.

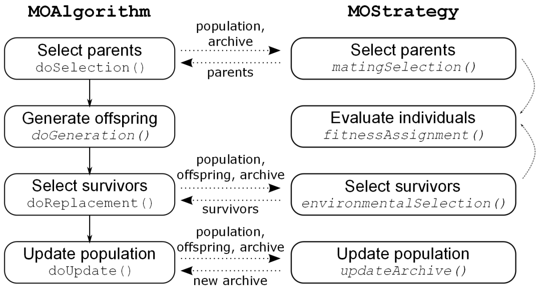

Figure 4: `MOAlgorithm` and `MOStrategy` cooperation

As a starting point to promote flexibility, JCLEC-MOEA offers a number of highly-configurable strategies for different types of multi-objective evolutionary approaches [1, 4, 16]:

- *MOEAs based on the Pareto dominance.* The most representative algorithms within this category, SPEA2 and NSGA-II, which were implemented as GAs extending directly from `PopulationAlgorithm` in JCLEC, have been adapted to their respective strategies in JCLEC-MOEA.
- *Decomposition approaches.* MOEA/D is probably the

Table 1: List of parameters of an experiment in JCLEC-MOEA

Multi-objective evolutionary algorithm	
algorithm-type	The evolutionary algorithm
mo-strategy	The multi-objective strategy and its parameters
Encoding and creation of individuals	
species	The selected encoding and the genotype length
provider	The process to create the initial population
Evaluation of individuals	
evaluator	The type of evaluator
objective	Every objective of the MOP and its parameters
Generation of offspring	
recombinator	The crossover operator and its probability (if required)
mutator	The mutation operator and its probability (if required)
Listeners	
type	The type of reporter
pareto-front	The file containing the true Pareto front (only for `MOIndicatorsReporter`)
indicator	Every quality indicator to be calculated (only for `MOIndicatorsReporter`)

best-known algorithm whithin this category. It was proposed as a general approach, where different evaluation mechanisms and specialised operators could be applied. Therefore, JCLEC-MOEA defines the strategy `MOEAD` as an abstract class where only the general behaviour is implemented. `MOEADws` and `MOEADte` represent two concrete implementations of `MOEAD` using the weighted sum (ws) and the Tchebycheff (te) approach, respectively.

- *Algorithms based on the landscape partition.* The steady-state algorithm ϵ-MOEA (`SSeMOEA`), and the grid-based evolutionary algorithm for many-objective optimisation, `GrEA`, are provided by JCLEC-MOEA. They use common functionalities to compare individuals according to the hypercubes they belong to.

- *Evolution strategies.* `PAES` proposes a multi-objective approach for evolution strategies. JCLEC-MOEA includes the (1+1), (1+λ) and (μ+λ) variants.

- *Indicator-based algorithms.* The inclusion of a quality indicator to guide the search is the key aspect of these algorithms. JCLEC-MOEA includes two well-known approaches: `HypE`, which is based on the estimation of the hypervolume, and `IBEA`, which was presented as a generic framework where any binary indicator can be applied. For this reason, `IBEA` has been defined as an abstract strategy, and its two subclasses, `IBEAhv` and `IBEAe` make use the hypervolume and the ϵ-indicator in the fitness evaluation, respectively. Other concrete subclasses might be incorporated as other indicators are developed.

- *Reference-point based approaches.* This type of algorithms emphasise the search of those non-dominated solutions that are close to one or more reference points. JCLEC-MOEA provides an implementation of the new NSGA-III.

JCLEC has never included any specific constraint-handling technique assuming that domain knowledge would be usually required to deal with constraints. Nevertheless, for multi-objective approaches, it is a common practice to define a specific constraint-handling mechanism. With this aim, JCLEC-MOEA provides a variant of each strategy, where some steps of the evolution like the evaluation of solutions or the comparison between them have been properly adapted to deal with constrained problems.

More specifically, when the strategy defines some kind of fitness function, it is overridden in order to assign poor fitness values to infeasible individuals. On the contrary, other strategies are modified according to the way in which comparisons between individuals are made, promoting the selection of feasible solutions. Other constraint-handling techniques can be implemented and combined with the existing strategies in order to build additional variants, allowing the flexible adaptation of these algorithms to real-world problems. JCLEC-MOEA defines the interface `IConstrained`, which declares two methods, `isFeasible` and `degreeOfInfeasibility`, serving to provide information about the sort of solution encoded. This interface should be only implemented by those individuals representing solutions of a constrained MOP.

3.4 Utilities in JCLEC-MOEA

A set of supplementary features have been incorporated into JCLEC-MOEA to provide support to both researchers and developers. Firstly, a number of quality indicators comprised of both unary and binary measures is provided. Additionally, some classes to visualise the outcomes of the experiments, originally named *reporters* in JCLEC, have been implemented to show information about different aspects of the occurring evolutionary process. More specifically, three different *reporters* have been designed: `MOPopulationReporter`, which reports the population state; `MOParetoReporter`, which collects the set of non-dominated solutions; and `MOIndicatorsReporter`, which calculates the quality of the Pareto front approximation by using one or more indicators. Finally, JCLEC-MOEA includes well-known test problems, considering all the types of encodings previously supported by JCLEC, including the Knapsack problem, the TSP, the DTLZ and ZDT families of continuous optimisation functions, and a symbolic regression problem.

4. EXTENDING JCLEC-MOEA

This section describes the extension mechanisms provided by JCLEC-MOEA to those developers interested in the creation of new strategies and in their adaptation, or in the resolution of some MOP. In all these cases, it is necessary to code some additional classes and prepare the corresponding XML configuration file in order to define the experiment. The set of elements to be determined in the configuration consists of the algorithm (EC model), the multi-objective strategy, the selected encoding, the genetic operators, the reporters and other general parameters like the population size or the random seed. Due to space limitations, Table 1 is focused on the most relevant parameters required by JCLEC-MOEA, in addition to those items that are part of the regular set-up of JCLEC.

4.1 Adding new strategies

To add a new multi-objective approach, the class `MOStrategy` has to be specialised to implement the mating selection and the environmental selection procedures. Additionally, it should be decided whether a solution archive is required. Developers can specify new additional parameters, represented by their own labels. Their respective values should be retrieved by the `configure` method of the proposed strategy, which is automatically invoked during the experiment building process. Specific values can be obtained or set using the `Configuration` declared by the Apache Commons Configuration library.

```
public class myMOStrategy extends MOStrategy {
   // Parameters of the strategy
   private double param1;
   private int param2; // default value is 10
   public void configure(Configuration settings) {
         param1 = settings.getDouble("myParameter1");
         param2 = settings.getInt("myParameter2", 10);
   }
}
```

Then the strategy can be executed by invoking a new experiment with the proper configuration, that is, by setting the path as the value of its mo-strategy parameter. Compiling again is not required to add new functionalities.

4.2 Changing the currently existing strategies

Developers usually need to make changes in some steps of an existing algorithm for a variety of reasons. A case in point are the several improvements made to NSGA-II in the search of alternatives that can deal with complex Pareto fronts. Under these circumstances, current strategies can be simply extended and overridden. In the example below, a new variant of NSGA-II would consider a new diversity preservation technique in order to replace the original crowding distance. Only the *environmentalSelection* method need to be reimplemented:

```
public class myNSGA2Variant extends NSGA2 {
   public List<IIndividual> environmentalSelection(List<
        IIndividual> population, List<IIndividual>
        offspring, List<IIndividual> archive) {
      // Fast non-dominated sorting
      survivors = new ArrayList<IIndividual>();
      survivors.addAll(population);
      survivors.addAll(offspring);
      fronts = super.fastNonDominatedSort(survivors);
      // New selection of survivors
      ...
      // Return survivors
      return survivors;
   }
}
```

When the source strategy is defined as a general approach, the development of new variants becomes simpler. This is the case of MOEA/D and IBEA in JCLEC-MOEA, where the corresponding strategies include the definition of the abstract methods to specify those parts of the approach that can vary from one implementation to another. On the one hand, MOEA/D can define different evaluation mechanisms, all of them being characterised by the use of the objective values of the individual (*ind*) and the set of weight vectors (*lambda*). Thus, the fitness function should be implemented as follows:

```
public class myMOEADVariant extends MOEAD {
   protected double fitnessFunction(IIndividual ind,
        double [] lambda){
      // New evaluation mechanism
      ...
      return fitnessValue;
   }
}
```

On the other hand, IBEA sets the fitness value to every individual (*ind0*) depending on the comparison against the rest of population members (*ind1*) according to the policies determined by a binary indicator, so that each new variant should only implement the indicator as follows:

```
public class myIBEAVariant extends IBEA {
   protected double computeIndicator(IIndividual ind0,
        IIndividual ind1) {
      // New binary indicator
      ...
      return indicatorValue;
   }
}
```

A further possibility is that a solution requires combining procedures of different algorithms with other proposals or even a self-implemented approach. This is the case of GrEA and NSGA-III, since both of them make use of the fast non-dominated sorting approach defined by NSGA-II. In JCLEC-MOEA, a strategy is allowed to call the selection mechanisms of other strategies, promoting the creation of hybridised approaches.

4.3 Addressing real-world problems

Irrespective of whether the chosen strategy is already provided or a new one, at least the set of objectives to be optimised needs to be implemented to solve any MOP. Firstly, each objective function is determined by a class that inherits from `Objective`. Here, the evaluation process for this particular objective should be coded as follows:

```
public class myObjectiveFunction extends Objective {
   public IFitness evaluate(IIndividual individual) {
      // Evaluate this objective in the given individual
      double objectiveValue = ...
      return (new SimpleValueFitness(objectiveValue));
   }
}
```

The resulting class has to be added to the list of objectives specified inside the configuration file. Other aspects, such as the objective bound or if they are expected to be minimised or maximised, are set in this file too. Then, the evaluator parameter is determined by `MOEvaluator` or `MOParallelEvaluator`.

In case of a constrained MOP, consideration must be given to other additional domain-specific aspects related to the problem. For instance, ConstrainedNSGA2 could be set as the multi-objective strategy to be used. Notice that it would not affect the selection of the EC paradigm, which is determined by the parameter algorithm-type. As for the definition of constraints, the mechanisms to distinguish between feasible and infeasible solutions have to be clearly specified. With this purpose, a new encoding should be created by implementing the interfaces IIndividual and IConstrained. The selected encoding will be internally used by the *species* module, which is in charge of defining and creating individuals.

5. CONCLUDING REMARKS

This paper presents JCLEC-MOEA, an adapted extension of JCLEC for the resolution of MOPs. The proposed architecture captures and maintains the capabilities of JCLEC, providing different EC models and a wide range of valuable encodings and operators, at the same time that it expands its functionality in order to incorporate new specific tools for MOO practitioners. The overall idea is founded on promoting important features like scalability, adaptability and ease of extension, among others. In fact, as a result, a highly configurable system has been developed, putting special emphasis on the flexibility to make use of different multi-objective approaches or just some particular parts of them.

JCLEC-MOEA implements a great variety of algorithms, all under the guidance of the *Strategy* design pattern recommendations, leading to highly modular and reusable implementations that make complex solutions like hybridisation easier to produce. JCLEC-MOEA also provides further utilities, including benchmarks, configuration facilities and quality indicators that can assist the developer to perform an accurate validation and comparison of new algorithms.

In the future we plan to keep JCLEC-MOEA up to date with the most recent MOO proposals and include other different multi-objective metaheuristics, as well as integrate this library with VisualJCLEC [17], a visual experimental environment for JCLEC.

6. ACKNOWLEDGMENTS

Work supported by the Spanish Ministry of Economy and Competitiveness, project TIN2014-55252-P, and the Spanish Ministry of Education under the FPU program (FPU13/01466).

7. REFERENCES

[1] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multiobjective evolutionary algorithms: A survey of the state of the art," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 32 – 49, 2011.

[2] M. Castillo Tapia and C. A. Coello Coello, "Applications of multi-objective evolutionary algorithms in economics and finance: A survey," in *IEEE Congress on Evolutionary Computation (CEC 2007)*, pp. 532–539, Sept 2007.

[3] G. R. Zavala, A. J. Nebro, F. Luna, and C. A. Coello Coello, "A survey of multi-objective metaheuristics applied to structural optimization," *Structural and Multidisciplinary Optimization*, pp. 1–22, 2013.

[4] T. Wagner, N. Beume, and B. Naujoks, "Pareto-, Aggregation-, and Indicator-Based Methods in Many-Objective Optimization," in *Evolutionary Multi-Criterion Optimization*, vol. 4403 of *LNCS*, pp. 742–756, Springer, 2007.

[5] J. A. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez, "Metaheuristic optimization frameworks: a survey and benchmarking," *Soft Computing*, vol. 16, no. 3, pp. 527–561, 2012.

[6] S. Luke, "ECJ: A Java-based Evolutionary Research System." Version 22, 2014. http://cs.gmu.edu/ eclab/projects/ecj/.

[7] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Adv. in Eng. Softw.*, vol. 42, no. 10, pp. 760–771, 2011.

[8] D. Hadka, "MOEA Framework User Manual." Version 2.3., October 2014. http://www.moeaframework.org.

[9] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a Java framework for evolutionary computation," *Soft Computing*, vol. 12, no. 4, pp. 381–392, 2008.

[10] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA – A Platform and Programming Language Independent Interface for Search Algorithms," in *Evolutionary Multi-Criterion Optimization (EMO 2003)*, LNCS, pp. 494–508, Springer, 2003.

[11] M. Kronfeld, H. Planatscher, and A. Zell, "The EvA2 Optimization Framework," in *Learning and Intelligent Optimization Conference (LION)*, pp. 247–250, 2010.

[12] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller, "Architecture and Design of the HeuristicLab Optimization Environment," in *Advanced Methods and Applications in Computational Intelligence*, vol. 6 of *Topics in Intelligent Engineering and Informatics*, pp. 197–261, Springer, 2014.

[13] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J: A Modular Framework for Meta-heuristic Optimization," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO'11)*, pp. 1723–1730, 2011.

[14] A. Liefooghe, L. Jourdan, and E.-G. Talbi, "A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO," *European Journal of Operational Research*, vol. 209, no. 2, pp. 104–112, 2011.

[15] D. Izzo, "PyGMO and PyKEP: Open Source Tools for Massively Parallel Optimization in Astrodynamics (The Case of Interplanetary Trajectory Optimization)," in *5th International Conference on Astrodynamics Tools and Techniques (ICATT)*, 2012.

[16] C. von Lücken, B. Barán, and C. Brizuela, "A survey on multi-objective evolutionary algorithms for many-objective problems," *Computational Optimization and Applications*, vol. 58, no. 3, pp. 707–756, 2014.

[17] J. I. Jaén, J. R. Romero, and S. Ventura, "VisualJCLEC: A visual framework for evolutionary computation," in *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)*, pp. 119–125, 2012.