Component-Based Design of Multi-Objective Evolutionary Algorithms Using the Tigon Optimization Library

João A. Duro
j.a.duro@sheffield.ac.uk
University of Sheffield
Sheffield, UK

Daniel C. Oara
dcoara1@sheffield.ac.uk
University of Sheffield
Sheffield, UK

Ambuj K. Sriwastava
a.k.sriwastava@sheffield.ac.uk
University of Sheffield
Sheffield, UK

Yiming Yan
yiming.yan@outlook.com
University of Sheffield
Sheffield, UK

Shaul Salomon
shaulsal@braude.ac.il
ORT Braude College of Engineering
Karmiel, Israel

Robin C. Purshouse
r.purshouse@sheffield.ac.uk
University of Sheffield
Sheffield, UK

ABSTRACT

Multi-objective optimization problems involve several conflicting objectives that have to be optimized simultaneously. Generating a complete Pareto-optimal front (POF) can be computationally expensive or even infeasible, and for that reason there has been an enormous interest in using multi-objective evolutionary algorithms (MOEAs), which are known to generate a good approximation of the POF. MOEAs can be difficult to implement, and even for experienced optimization experts it can be a very time consuming task. For this reason several optimization libraries exist in the literature, providing off-the-shelf access to the most popular MOEAs. Some optimization libraries also provide a framework to design MOEAs. However, existing frameworks can be too stringent and do not provide sufficient flexibility for the design of more sophisticated MOEAs. To address this, a recently proposed optimization library, known as Tigon, features a component-based framework for the design of MOEAs with a focus on flexibility and re-usability. This paper demonstrates the generality of this new framework by showing how to implement different types of MOEAs, covering several paradigms in evolutionary computation. The work in this paper serves as a guide for researchers, and others alike, to build their own MOEAs by using the Tigon optimization library.

CCS CONCEPTS

• **Computing methodologies** → **Search methodologies**; • **Applied computing** → **Multi-criterion optimization and decision-making**; • **Software and its engineering** → **Search-based software engineering**.

KEYWORDS

Software engineering, evolutionary algorithms, multi-objective optimization, algorithm design

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463194

ACM Reference Format:
João A. Duro, Daniel C. Oara, Ambuj K. Sriwastava, Yiming Yan, Shaul Salomon, and Robin C. Purshouse. 2021. Component-Based Design of Multi-Objective Evolutionary Algorithms Using the Tigon Optimization Library. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463194

1 INTRODUCTION

Many multi-objective evolutionary algorithms (MOEAs) have been proposed over the years for solving multi-objective optimization problems [22]. The general principle of these algorithms is to mimic nature's evolutionary principles to guide a population of solutions towards the Pareto-optimal front (POF). Relying on a population of solutions, as opposed to using a single solution at any time (e.g. trajectory methods [2]), can be seen as advantageous for solving multi-objective optimization problems since multiple trade-off solutions can be found in one single run. Although there is no guarantee to identify optimal trade-offs, a good approximation to the POF can provide valuable information to a decision-maker about the potential synergies that exist between the criteria in the optimization problem.

Due to the their popularity, MOEAs can be found in several software libraries, making it easier for researchers, and optimization practitioners alike, to use them without having to code an MOEA from scratch. Some open-source software libraries also provide a framework for the design of MOEAs; examples found in the literature are PISA [4], ParadisEO-MOEO [15], jMetal [18], and MOEADr [5]. However, many of these libraries provide a static framework for the design of MOEAs where it is only possible to swap certain traditional evolutionary computation components, such as selection and variation operators. MOEADr in particular, provides a component-based framework for the design of MOEAs, but focusses only on decomposition-based MOEAs. To design more sophisticated MOEAs there is a need for more flexible and intuitive frameworks.

More recently, a new release of the Liger optimization workflow software[1] has introduced the Tigon optimization library in [9]. One of the major contributions of Tigon in [9] has been the proposal of a new component-based framework for the design of MOEAs

[1]Latest releases of the Liger software can found in the following GitHub repository https://github.com/ligerdev/liger.

with a focus on flexibility and re-usability. An MOEA is made up of several simple interacting components, and many of the existing components can be re-used for the implementation of different types of MOEAs. Another interesting feature is the ability to customise the components or even to change the algorithm dynamically at run-time. This opens up the possibility for the design of more advanced MOEAs, such as those found in the automatic algorithm design literature [3, 16].

It has been demonstrated in [9] how to use the Tigon component-based framework to implement an elitist Pareto-dominance-based MOEA, known as NSGA-II [7], and this paper focuses on demonstrating its generality to implement different types of MOEAs covering several paradigms in evolutionary computation. Besides NSGA-II, other algorithms covered in this paper are: MOGA [11], SMS-EMOA [10], MOEA/D [21] and ParEGO [14].

This paper is organised as follows. An introduction to the Tigon optimization library, including the details of the component-based framework for the design of MOEAs is provided in Section 2. The implementation of the different MOEAs by using the Tigon component-based framework is described in Section 3. This paper concludes with Section 4.

2 TIGON OPTIMIZATION LIBRARY

Tigon is an object-oriented C++ optimization library first introduced in [9] that forms part of the Liger software. One of the key features of Tigon is that it uses the *Decorator* design pattern [12] to implement a component-based framework to design MOEAs. By 'component-based' we mean that an MOEA is broken down into simple reusable components (or operators), and each one performs a simple action (e.g. selection, crossover and mutation) on a population of solutions. In the remainder of this paper we use the term 'set' to refer to a population of solutions. In order to implement the sophisticated behaviour of an MOEA, many operators rearrange the existing solutions into different sets according to some criterion (e.g. based on their distance in objective space), and several of these sets can co-exist concurrently during the run-time of an algorithm. Knowing that sets only store *pointers* to the solutions, any operation involving the creation, manipulation and deletion of sets is expected to have a very low run-time complexity. A solution is only deleted once it is no longer referenced by any of the existing sets, which is facilitated by the use of C++ smart pointers.

In the following sections we provide more details about the decorator design pattern in Section 2.1, and the approach used to help the operators to identify which sets they need to operate on in Section 2.2.

2.1 The Tigon decorator design pattern

The *Decorator* design pattern is used in object-oriented programming languages for adding behaviour to (or decorate) an object dynamically, without affecting other objects from the same class. This is reportedly [12] more efficient than relying on the approach of subclassing, where the behaviour of a base class can be extended by other subclasses. One drawback of subclassing is that extensions are fixed during compile-time and cannot be changed during run-time. Another advantage of the decorator design pattern is that it enforces the Single Responsibility Principle [20], meaning that each

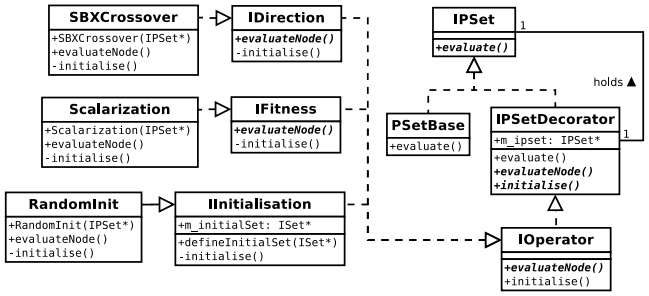

Figure 1: UML diagram showing architecture of the decorator design pattern. Taken from [9] and edited.

class is responsible for a single part of the algorithm's functionality. Creating classes that encapsulate more than one responsibility, each corresponding to a different functionality, are more difficult to maintain. Also, given that multiple functionalities are encapsulated in the same class, a single change to one functionality could have side-effects, such as breaking other functionalities.

A UML diagram showing the architecture of the decorator design pattern is depicted in Figure 1. The three classes in the top right (i.e. *IPSet*, *PSetBase* and *IPSetDecorator*) provide the implementation of the design pattern[2], while the other classes on the left are the decorators of *IPSet* (i.e. *SBX Crossover*, *Scalarization* and *RandomInit*). The decorator classes are now referred to as operators in the remainder of this paper. All operators are under the interface *IOperator*, and each operator is responsible for implementing a single functionality of the algorithm. There are other operators in the library besides the three operators shown in Figure 1, and for a complete list the reader is referred to https://github.com/ligerdev/liger/tree/master/src/libs/tigon/Operators. Depending on their functionality, the operators are organized under several interfaces. For instance, all operators under *IInitialisation*, such as *RandomInit*, are responsible for initialising solutions (i.e. setting the values of their decision vectors by using some sampling technique).

The intention of the design pattern is to decorate *IPSet* objects by stacking up all operators on top of each other, and adding a new functionality each time an operator calls an overridden method. This behaviour is achieved by the following steps:

(1) *PSetBase* and *IPSetDecorator* both become a subclass of *IPSet*, and the latter holds a pointer to *IPSet*, namely *m_ipset*;
(2) all operators need to be subclasses of *IPSetDecorator* and override *evaluateNode()*. This is also where all the functionality of the operators is placed in their respective classes;
(3) add *evaluate()* to both *IPSet* and *IPSetDecorator*. Notice that *evaluate()* becomes available to all operators via class inheritance.

The C++ implementation of evaluate() in *IPSetDecorator* is as follows:

```
void IPSetDecorator::evaluate() {
  m_ipset->evaluate();
  evaluateNode();
}
```

[2]The implementation of the decorator design pattern is in https://github.com/ligerdev/liger/tree/master/src/libs/tigon/Representation/PSets.

In the above code there is a recursive call to *evaluate()* which has the effect of stacking up all operators on top of each other, and following this, *evaluateNode()* executes the operators code in the same order that they have been stacked up. An example showing how to call the decorator pattern in C++ is as follows:

```
PSetBase*         base      = new PSetBase();
ProblemGenerator* problem   = new ProblemGenerator(base);
RandomInit*       initialiser = new RandomInit(problem);
Evaluator*        evaluator   = new Evaluator(initialiser);
evaluator->evaluate(); // Invoke evaluate() method
```

In this particular example the intention is to initialise a population of solutions, and then to evaluate them (i.e. determine their performance or fitness) with respect to a given optimization problem. The first operator is *PSetBase*, which does not add any functionality and the only purpose is to serve as an anchor to the execution starting point. The operators are then initialised in the order that we wish to execute them, and during initialisation each one takes, as input, the previous operator object. Once all operators have been initialised a call to *evaluate()* creates a sequence of events that are shown in Figure 2. Notably, initially all operators call *evaluate()* (steps 1-4), and once the sequence of calls reaches the *PSetBase* class, *evaluateNode()* is called by each operator in turn following the provided sequence (steps 5-11).

We have shown a very simple example where a population has been initialised, and then evaluated for a given optimisation problem. Using this component-based approach it is possible to implement the behaviour of more complex algorithms, as will be shown in Section 3. In the example shown above only one set of solutions has been created, in this case by *RandomInit*. However, it is common for other operators to create new sets by rearranging the existing solutions according to some criterion, and these multiple sets can co-exist concurrently during the execution of an algorithm. The approach used to help the operators to identify which sets they need to operate on is described in the following section.

2.2 A tag-based approach to help coordinate the information flow between operators

To help the operators to identify which sets they need to operate on, we use the concept of tags. A tag is an identifier that takes the format of a *string*, and can be added to sets and operators. More than one tag can be added to a single set, each one giving an indication of what might happen to the set. For instance, considering the example from Figure 2:

(1) the operator *RandomInit* initialises a set randomly and adds the tag *For Evaluation*. This gives an indication that there is a need to evaluate the solutions in the set, that is, determine their fitness values. Following the operation of *RandomInit*, other operators are able to find this particular set with the tag *For Evaluation*.
(2) The operator *Evaluator* looks for sets with the tag *For Evaluation*, and once a set has been found, it evaluates all its solutions.

An operator can have multiple tags. For an operator to operate on a set, the set needs to contain all the tags of the operator, but the set can also contain other tags that are not in the operator. For instance, consider the following three sets and their tags:

(1) *Set-1* has *Tag-1*, *Tag-2*, and *Tag-3*;
(2) *Set-2* has *Tag-1* and *Tag-2*, and;
(3) *Set-3* has *Tag-2* and *Tag-3*.

If an operator has *Tag-1* and *Tag-2*, then it operates on *Set-1* and *Set-2* since both sets contain all the tags defined for the operator, but *Set-3* is ignored since the set does not contain *Tag-1*.

In addition, the operators categorise the tags as either input tags or outputs tags. An input tag means that the operator will only read the solutions without modifying them. On the other hand, an output tag means that the operator will modify the solutions in the set (e.g. change the values of their decision vectors).

The tags have an influence on the way information flows between operators, and we use an *operators-tags* diagram, as shown in Figure 3, to represent the interaction between operators and sets. In Figure 3 there are two diagrams, each with four operators. The circle below an operator indicates that the operator is a set creator. The filled circle (●) below *Operator-A*, signifies that the set(s) contain new solutions. The non-filled circle (○) below *Operator-B* in the left diagram, signifies that all solutions in the set are pointers to existing solutions, meaning that the same solutions could exist in other sets. This implies that a change to any of these solutions will be automatically reflected in other solutions found in other sets. Moreover, when an operator has an input tag, the solutions in the corresponding sets are read-only, and this is represented by a line with an arrow, where the arrow points to a side of the operator. When an operator has an output tag, the solutions in the corresponding sets have read–write permissions, and this is represented by a line without an arrow, connected to the bottom of the operator. The diagram in the left is now described in point a) while the one in the right in point b) as follows:

a) The sets created by *Operator-A* are tagged with *Tag-1*. *Tag-1* is an input tag for *Operator-B* (represented by the red line with an arrow), and an output tag for *Operator-C* and *Operator-D* (represented by the red line without an arrow). The sets created by *Operator-B* are tagged with *Tag-2*. *Tag-2* is an input tag for *Operator-D* (represented by the blue line with an arrow). Given this representation the sequence of events is as follows: 1) *Operator-A* creates sets and these are tagged with *Tag-1*; 2) *Operator-B* reads the sets from *Operator-A* and creates new sets by simply re-arranging the solutions into the new sets, and these are tagged with *Tag-2*; 3) *Operator-C* modifies the sets from *Operator-A*; and 4) *Operator-D* reads the sets from *Operator-B* and uses this information to further modify the sets from *Operator-A*.
b) The sets created by *Operator-A* are tagged with *Tag-1*. *Tag-1* is an input tag for *Operator-B* (represented by the red line with an arrow), and an output tag for *Operator-D* (represented by the red line without an arrow). The sets created by *Operator-B* are tagged with *Tag-2*. *Tag-2* is an input tag for *Operator-C* (represented by the blue line with an arrow) and also an input tag for *Operator-D* (represented by the green line with an arrow). The sets created by *Operator-C* are tagged with *Tag-2* and *Tag-3*. Given that *Operator-D* has both input tags *Tag-2* and *Tag-3*, it means that *Operator-D*

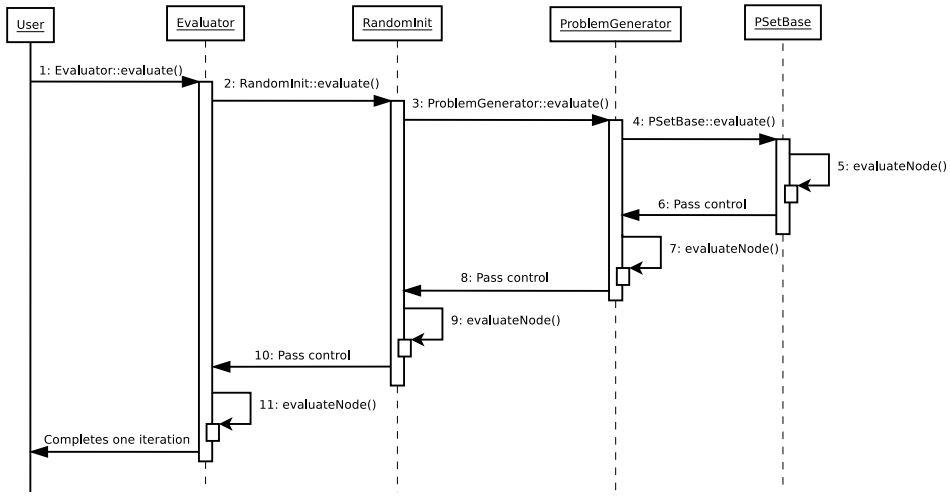

Figure 2: Sequence diagram showing an example of the decorator design pattern. Taken from [9] and edited.

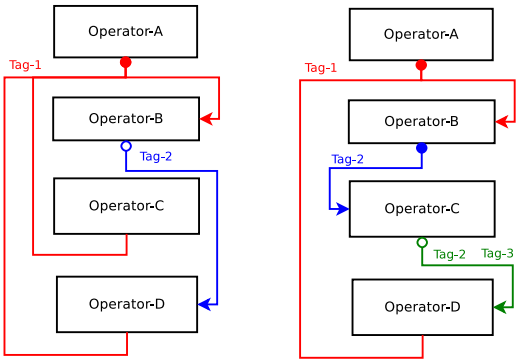

Figure 3: Operators-tags diagram showing the effect of tags in the information flow between four operators.

only reads the sets created by *Operator-C*, and avoids the ones from *Operator-B*. Given this representation, the sequence of events is as follows: 1) *Operator-A* creates sets and these are tagged with *Tag-1*; 2) *Operator-B* reads the sets from *Operator-A* and creates new sets with new solutions, and these are tagged with *Tag-2*; 3) *Operator-C* reads sets from *Operator-B* and creates new sets by re-arranging existing solutions into the new sets, and these are tagged with *Tag-2* and *Tag-3*; and 4) *Operator-D* reads the sets from *Operator-C* and uses this information to modify the sets from *Operator-A*.

3 COMPOSING MULTI-OBJECTIVE EVOLUTIONARY ALGORITHMS

In this section we describe how existing MOEAs in the Tigon optimization libraries have been implemented by the use of the component-based framework, involving operators and tags. For each case, an operators-tags diagram will include the operators *Initialisation* and *Evaluator*. The first creates a population of solutions by using some sampling technique, and the set that represents this population is referred to as the *Main Optimization Set*. The second, as the name suggests, evaluates all the solutions from sets that have been tagged *For Evaluation*. Moreover, the implementation of the algorithms described in this section can be found in https://github.com/ligerdev/liger/tree/master/src/libs/tigon/Algorithms.

3.1 MOGA

MOGA [11] is the first non-elitist Pareto-dominance-based MOEA to make use of the non-dominated classification as a way to separate the population into ranks, and also introduced the concept of niching as a mechanism to maintain diversity amongst the non-dominated solutions. The operators-tags diagram for this MOEA is shown in Figure 4, and the details are as follows.

The solution fitness assigned procedure is conducted by the operators *NonDominance Ranking*, *Average Fitness* and *Shared Fitness*. *NonDominance Ranking* first assigns a rank to each solution, where the rank of a solution i is equal to one plus the number of solutions that dominate solution i. In this way, non-dominated solutions are assigned rank 1, and the other solutions have rank higher than 1 up-to a maximum of N, where N is the population size. One set is created for each rank, and all solutions with the same rank are placed in the same set.

Average Fitness determines an average fitness for each solution based on their ranks. This involves first sorting in ascending order all the solutions with respect to their ranks, and a raw fitness is determined by a linear mapping function that preserves the ranks between solutions. A solution's fitness is then computed as the average of the raw fitness with respect to all solutions in the same rank. Given that *Average Fitness* needs to access all solutions in the entire population in order to determine the average fitness, the output tag of the operator is set to *Main Optimization Set*. *Shared Fitness* then calculates a niche count for each solution, and its value is used to modify the solution's fitness. This procedure operates on each rank separately, and for this we set the output tag of *Shared Fitness* to *Fitness*. This means that *Shared Fitness* is able to operate

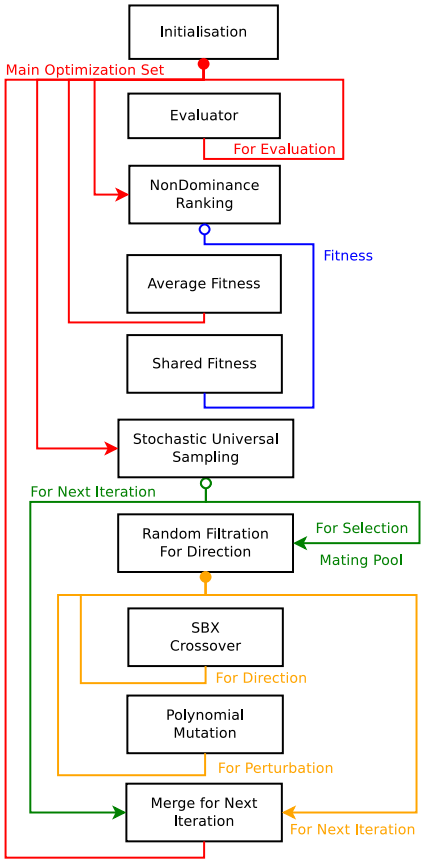

Figure 4: MOGA operators-tags diagram.

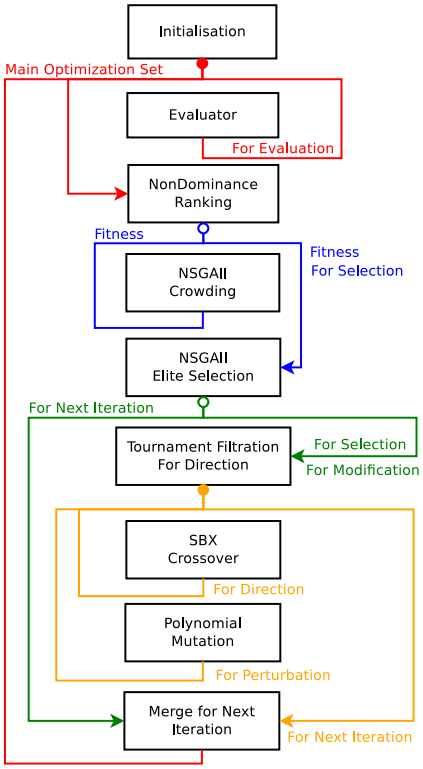

Figure 5: NSGA-II operators-tags diagram. Taken from [9] and edited.

on the sets created by *NonDominance Ranking* one by one, where each set corresponds to a different rank.

The *Stochastic Universal Sampling* implements a selection process from the classical genetic algorithm literature [1]. The solutions are selected from the *Main Optimization Set* based on their fitness, and new sets are created with these solutions (corresponding to the green lines). Following the selection process, two variation operators (crossover and mutation) operate on the selected solutions. However, the set created by *Stochastic Universal Sampling* contains multiple solutions (often more than two), but the crossover operator only operates on sets with two solutions. To address this, the operator *Random Filtration For Direction* creates multiple sets with two solutions each from the *Stochastic Universal Sampling* set (corresponding to the yellow lines). The variation operators *SBX Crossover* and *Polynomial Mutation* then operate on each set individually. The operator *Merge for Next Iteration* merges the sets from *Stochastic Universal Sampling* and *Random Filtration For Direction*. The solutions in the merged set replace the old population in the *Main Optimization Set*. This completes one iteration of the MOGA algorithm.

Remarks: At the end of each iteration several different sets have been created by the operators. *NonDominance Ranking* and *Stochastic Universal Sampling* have created sets with solutions from the *Main Optimization Set*, and the sets created by *Random Filtration For Direction* contain new solutions. These new solutions are only evaluated at the beginning of the next iteration by the operator *Evaluator*. In case it is desirable to check the performance of these solutions at the end of each iteration, then the operator *Evaluator* needs to be added at the end, after *Merge for Next Iteration*. Moreover, the procedure in *Shared Fitness* operates separately on the solutions of each rank, and if the complete population were to be used instead, then it would require changes to the procedure to ensure that solutions have been separated into ranks, thereby wasting computing resources. This demonstrates the effectiveness of rearranging the solutions into new sets and the importance of tags.

3.2 NSGA-II

NSGA-II [7] is an elitist Pareto-dominance-based MOEA that shares many similarities with MOGA, such as the determination of ranks between solutions and the use of a diversity-preserving mechanism. A major difference is that an elite population is preserved between successive iterations. The operators-tags diagram is shown in Figure 5, and the details are as follows.

For NSGA-II the initial population size is set to $2N$ where N is the number of non-dominated solutions in the final population. Like MOGA, NSGA-II also generates ranks between solutions by using *NonDominance Ranking*, but the ranks are determined in a different way. The first rank corresponds to the set of non-dominated solutions, and the next rank is determined by excluding the first

rank solutions from the population. New ranks are determined in this way until all solutions have been excluded. The diversity-preserving mechanism is implemented by *NSGA-II Crowding*, which determines how crowded each solution is in the objective space. The least crowded solutions are attributed a better fitness than other more crowded ones. This procedure operates on each rank separately, and for this we set the output tag of *NSGA-II Crowding* to *Fitness*.

The operator *NSGA-II Elite Selection* constructs an elite population by selecting one rank at a time until the population size is higher than or equal to N. In case the population size exceeds N, then the solutions from the last rank are sorted with respect to their crowding distance values, and the more crowded solutions are removed from the population until the population size is equal to N. The set created by *NSGA-II Elite Selection* is read by the operator *Tournament Filtration For Direction*. This operator is similar to *Random Filtration For Direction* since it rearranges the solutions from the population provided as input into several sets, each containing only two solutions. The only difference is that *Tournament Filtration For Direction* also conducts tournament selection, meaning that all solutions from the population are first paired with each other, and only those that have better fitness survive. Those solutions that survive this process are then placed into sets to undergo variation. Following this, the variation operators (crossover and mutation) and *Merge for Next Iteration* operate in the same way as mentioned for MOGA.

Two variants of NSGA-II that exist in the Tigon optimization library are NSGA-II-PSA [19] and NSGA-III [6]. In the former, *NSGA-II Crowding* and *NSGA-II Elite Selection* are replaced by two equivalent operators that use a clustering partition-based selection algorithm as opposed to the crowding distance. In the latter, *NSGA-II Crowding* is replaced by an operator that uses a decomposition-based niching mechanism to select solutions.

Remarks: NSGA-II shares many similarities with MOGA, notably the use of ranks and a diversity-preserving mechanism. *Tournament Filtration For Direction* likewise *Random Filtration For Direction* from MOGA creates sets with new solutions. These are then merged with the elite population created by *NSGA-II Elite Selection* into the *Main Optimization Set*.

3.3 SMS-EMOA

SMS-EMOA [10] is a steady-state indicator-based MOEA. It relies on Pareto-dominance for convergence and an indicator (in this case the hypervolume metric [23]) ensures a good spread of solutions across the POF. However, once Pareto-dominance becomes ineffective, convergence is ensured by the hypervolume metric. The steady-state selection scheme used by this MOEA means that only one solution is created and tested to be inserted in the population at each iteration of the algorithm, which is different from the generational scheme used by the previous two MOEAs, where at each iteration a new entire population is created and tested to replace the old population. The operators-tags diagram is shown in Figure 6, and the details are as follows.

The operator *NonDominance Ranking* is used to separate the solutions into non-dominance ranks by using the same procedure as NSGA-II. *SMS-EMOA Reduce* operates on the last rank and the

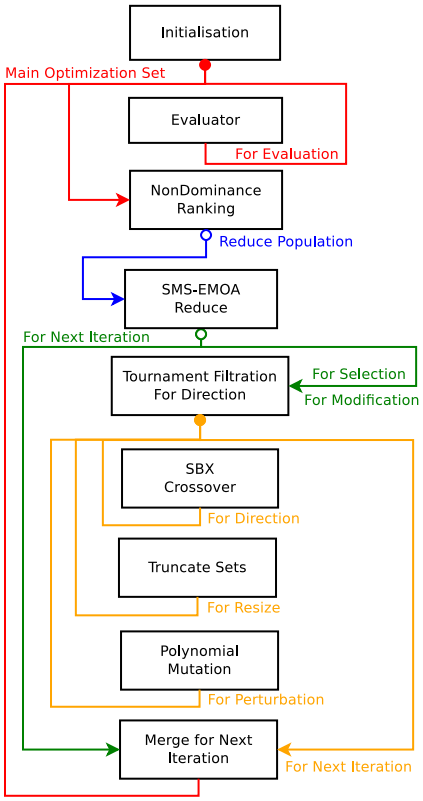

Figure 6: SMS-EMOA operators-tags diagram.

solution with the worst contribution towards the hypervolume metric is not added to the set created by the operator (corresponding to the green lines). To implement a steady-state selection scheme: first *Tournament Filtration For Direction* creates only one set with two solutions that subsequently undergo crossover; and second, the operator *Truncate Sets* removes one of the solutions from the set. At this stage the set contains only one solution, which undergoes mutation, and finally it is merged with the solutions from set created by *SMS-EMOA Reduce* in order to replace the old population in the *Main Optimisation Set*.

Remarks: Many operators have parameters that can be tuned by the user. For instance, it is possible to define the number of sets and their number of solutions in many filtration operators, such as *Tournament Filtration For Direction*. This has shown to be useful in order to implement the steady-state selection scheme in SMS-EMOA. We acknowledge that the *SMS-EMOA Reduce* could perhaps be partitioned into several smaller operators, given that currently this operator performs a very specific task that might only be useful for SMS-EMOA.

3.4 MOEA/D

MOEA/D [21] is a decomposition-based MOEA that decomposes a multi-objective optimization problem into several concurrent single-objective subproblems, each approaching the POF from a different direction. Each solution in the population is attributed a direction (reference) vector that defines a target direction in objective space

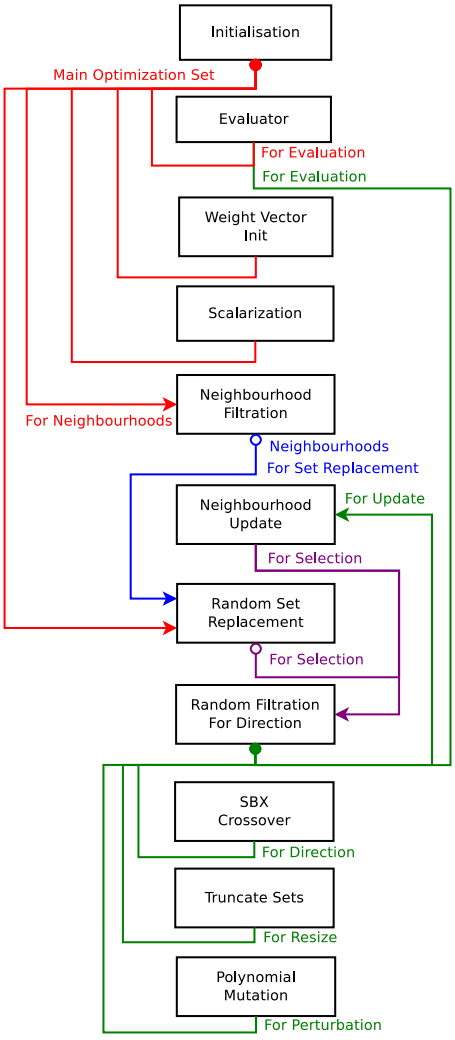

Figure 7: MOEA/D operators-tags diagram.

towards the POF. The operators-tags diagram is shown in Figure 7, and the details are as follows.

Weight Vector Init attributes a different direction vector to each solution in the population by using some design of experiments technique (e.g. the Simplex Lattice method). *Scalarization* determines a scalarized fitness value per solution by using the direction vector, a corresponding weighting vector, and a given scalarization function (e.g. weighted Chebyshev). *Neighbourhood Filtration* divides the population into N neighbourhoods based on the distance between the direction vectors, where N is the population size, and each neighbourhood contains a total of n solutions (by default n is set to 5). Notice that each neighbourhood is represented by a single set.

In the next step, the goal of the *Neighbourhood Update* is to replace existing solutions in the neighbourhoods (or the entire population) by N child solutions. However, the child solutions are only available after the *Random Filtration for Direction* operation, and another requirement are the sets coming from *Random Set Replacement*

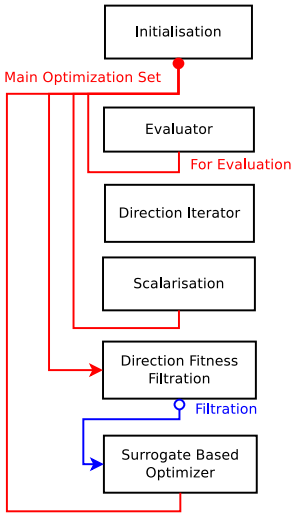

Figure 8: ParEGO operators-tags diagram.

which are also not yet available. This means that *Neighbourhood Update* does not operate during the first iteration. The sets created by *Random Set Replacement* are copies of the neighbourhoods, but for a given probability some sets are replaced by the *Main Optimisation Set*. This implementation follows a particular functionality of the MOEA/D algorithm, which states that when promoting a child solution into a neighbourhood, this neighbourhood is sometimes replaced by the entire population. *Random Set Replacement* implements this behaviour by creating new copies of the neighbourhoods but replacing some of the neighbourhoods by the entire population. Following this, *Random Filtration for Direction* creates N sets with two solutions each from the neighbourhoods created by *Random Set Replacement*. *SBX Crossover*, *Truncate Sets* and *Polynomial Mutation* operate in the same way as described for the SMS-EMOA algorithm.

Remarks: The neighbourhood structure proved to be one of the most challenging implementations involving the concept of sets and tags—in part due to the probabilistic procedure used by MOEA/D to update the neighbourhoods. The current design for this algorithm meant that *Neighbourhood Update* does not operate during the first iteration since the required sets are not yet available.

3.5 ParEGO

ParEGO [14] is a hybrid surrogate-based multi-objective extension of the single-objective efficient global optimization (EGO) algorithm [13], known to have a better performance than the majority of existing MOEAs when applied to optimization problems where function evaluations are expensive or otherwise restricted in number. At each iteration, a surrogate model is learnt based on the scalarized fitness values of the solutions, and a search over the surrogate model reveals a new single solution by balancing local exploitation with global exploration. The operators-tags diagram is shown in Figure 8, and the details are as follows.

Direction Iterator does not interact with any sets and therefore there are no connections in the diagram. This operator iterates over a set of direction vectors, and each time it is called, a new direction

vector is chosen randomly from the set of available direction vectors. The chosen direction vector is saved in the *IPSet* class (see Figure 1), and therefore it becomes visible to the other operators via class inheritance. The *Scalarization* operates in the same way as described for MOEA/D, but the difference is that all solutions are scalarized with respect to the direction vector chosen by *Direction Iterator*, as opposed to each solution having its own direction vector. Learning a surrogate model can be computationally expensive if the number of solution is too high. To address this, *Direction Fitness Filtration* selects a subset of solutions from the *Main Optimization Set* and the chosen solutions are added to a new set (corresponding to the blue line). This new set serves as input to the *Surrogate Based Optimizer* and a surrogate model is learnt based on the fitness values of the solutions. A single-objective search procedure is conducted over the surrogate model to find a new candidate solution that maximises the expected improvement function. We employ a single-objective genetic algorithm known as ACROMUSE [17] to conduct the search. After each iteration a single solution is found by the *Surrogate Based Optimizer* for a different direction vector.

A variant of ParEGO found in the Tigon optimization library is known as sParEGO [8]. The new variant extends ParEGO for dealing with stochastic optimization problems where the solution's performance following each evaluation can be uncertain. For this, two operators are added to the operators-tags diagram just after *Scalarization*. The first implements an uncertainty quantification approach which assigns a random variate to each solution based on the fitness of nearby solutions, and the second uses a given robustness criterion applied to each random variate to determine a new robust fitness value for each solution.

Remarks: ParEGO is different from the other MOEAs in the sense it does not use any selection or variation operators. A large part of the algorithm functionality is embedded in the operator *Surrogate Based Optimizer*, where a single solution is found by solving a single-objective problem. The other operators extend the algorithm to the multi-objective domain. Moreover, some operators do not interact with sets, but their functionality is crucial for the algorithm. One example is *Direction Iterator*: this operator's purpose is to iterate over a set of direction vectors, and this is independent from the existing solutions.

4 CONCLUSION

This paper builds on a new component-based framework for the design of MOEAs, part of the Tigon optimization library. The framework enables the user to design an MOEA by assembling a set of components, and the way the components interact with several populations of solutions is controlled by a user-led visually intuitive tag-based approach. We have shown that the new framework is general and flexible enough for the implementation of a range of MOEAs covering several paradigms in evolutionary computation. The algorithms considered in this work include: MOGA (non-elitist Pareto-dominance-based); NSGA-II (elitist Pareto-dominance-based); SMS-EMOA (steady-state indicator-based); MOEA/D (decomposition-based); and ParEGO (surrogate-based). For future work, the endeavour of the authors will be to extend the current framework to find novel algorithm designs by exploiting automatic algorithm configuration methods [3, 16]. For this, we may need to diversify and increase the number of operators in the Tigon library. Existing operators are mostly inspired by strategies found in the MOEAs described in this paper; other metaheuristics, not just evolutionary algorithms, could be also considered (e.g. particle swarm optimization and simulated annealing). It is also our interest to conduct a comparative analysis between Tigon and other optimization frameworks.

5 ACKNOWLEDGEMENTS

The authors would like to acknowledge financial support from Innovate UK and Ford Motor Company as part of the Advanced Propulsion Centre UK project DYNAMO (grant 113130), and also EPSRC and Jaguar Land Rover as part of the jointly funded Programme for Simulation Innovation (PSi) (EP/L025760/1). Daniel Oara acknowledges EPSRC studentship support (EP/M508135/1 and EP/M506618/1). The authors would also like to thank University of Sheffield undergraduate project student Christopher J Gaskell for his implementation of NSGA-III.

REFERENCES

[1] James Baker. 1987. Reducing Bias and Inefficiency in the Selection Algorithm. In *Second International Conference on Genetic Algorithms and Their Application.*

[2] Zahra Beheshti and Siti Mariyam Shamsuddin. 2013. A Review of Population-based Meta-Heuristic Algorithm. *International Journal of Advances in Soft Computing and its Applications* 5 (2013), 1–35.

[3] Leonardo C. T. Bezerra, Manuel López-Ibáñez, and Thomas Stützle. 2016. Automatic Component-Wise Design of Multi-Objective Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 20, 3 (2016), 403–417. https://doi.org/10.1109/TEVC.2015.2474158

[4] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. 2003. PISA — A Platform and Programming Language Independent Interface for Search Algorithms. In *Evolutionary Multi-Criterion Optimization (EMO 2003) (Lecture Notes in Computer Science)*, Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele (Eds.). Springer, Berlin, 494 – 508.

[5] Felipe Campelo, Lucas S. Batista, and Claus Aranha. 2020. The MOEADr Package: A Component-Based Framework for Multiobjective Evolutionary Algorithms Based on Decomposition. *Journal of Statistical Software, Articles* 92, 6 (2020), 1–39. https://doi.org/10.18637/jss.v092.i06

[6] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (2014), 577–601. https://doi.org/10.1109/TEVC.2013.2281535

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[8] João A. Duro, Robin C. Purshouse, Shaul Salomon, Daniel C. Oara, Visakan Kadirkamanathan, and Peter J. Fleming. 2019. sParEGO – A Hybrid Optimization Algorithm for Expensive Uncertain Multi-objective Optimization Problems. In *Evolutionary Multi-Criterion Optimization*, Kalyanmoy Deb, Erik Goodman, Carlos A. Coello Coello, Kathrin Klamroth, Kaisa Miettinen, Sanaz Mostaghim, and Patrick Reed (Eds.). Springer International Publishing, 424–438.

[9] João A. Duro, Yiming Yan, Ioannis Giagkiozis, Stefanos Giagkiozis, Shaul Salomon, Daniel C. Oara, Ambuj K. Sriwastava, Jacqui Morison, Claire M. Freeman, Robert J. Lygoe, Robin C. Purshouse, and Peter J. Fleming. 2021. Liger: A cross-platform open-source integrated optimization and decision-making environment. *Applied Soft Computing* 98 (2021), 106851. https://doi.org/10.1016/j.asoc.2020.106851

[10] Michael Emmerich, Nicola Beume, and Boris Naujoks. 2005. An EMO Algorithm Using the Hypervolume Measure as Selection Criterion. In *Evolutionary Multi-Criterion Optimization*, Carlos Coello Coello, Arturo Hernández Aguirre, and Eckart Zitzler (Eds.). Lecture Notes in Computer Science, Vol. 3410. Springer Berlin / Heidelberg, 62–76. https://doi.org/10.1007/978-3-540-31880-4_5

[11] C. M. Fonseca and P. J. Fleming. 1998. Multiobjective Optimization and Multiple Constraint Handling with Evolutionary Algorithms—Part I: A Unified Formulation. *IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans* 28, 1 (Jan 1998), 26–37. https://doi.org/10.1109/3468.650319

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[13] Donald R. Jones, Matthias Schonlau, and William J. Welch. 1998. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization* 13 (1998), 455–492. https://doi.org/10.1023/A:1008306431147

[14] Joshua Knowles. 2006. ParEGO: A Hybrid Algorithm With On-Line Landscape Approximation for Expensive Multiobjective Optimization Problems. *IEEE Transactions on Evolutionary Computation* 10, 1 (Feb 2006), 50–66. https://doi.org/10.1109/TEVC.2005.851274

[15] Arnaud Liefooghe, Laetitia Jourdan, and El-Ghazali Talbi. 2011. A software framework based on a conceptual unified model for evolutionary multiobjective optimization: ParadisEO-MOEO. *European Journal of Operational Research* 209, 2 (2011), 104–112. https://doi.org/10.1016/j.ejor.2010.07.023

[16] Manuel López-Ibáñez and Thomas Stützle. 2012. The Automatic Design of Multi-objective Ant Colony Optimization Algorithms. *IEEE Transactions on Evolutionary Computation* 16, 6 (2012), 861–875. https://doi.org/10.1109/TEVC.2011.2182651

[17] Brian McGinley, John Maher, Colm O'Riordan, and Fearghal Morgan. 2011. Maintaining Healthy Population Diversity Using Adaptive Crossover, Mutation, and Selection. *IEEE Transactions on Evolutionary Computation* 15, 5 (2011), 692–714. https://doi.org/10.1109/TEVC.2010.2046173

[18] Antonio J. Nebro, Juan J. Durillo, and Matthieu Vergne. 2015. Redesigning the jMetal Multi-Objective Optimization Framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (Madrid, Spain) *(GECCO '15)*. ACM, New York, NY, USA, 1093–1100. https://doi.org/10.1145/2739482.2768462

[19] Shaul Salomon, Christian Domínguez-Medina, Gideon Avigad, Alan Freitas, Alex Goldvard, Oliver Schütze, and Heike Trautmann. 2014. PSA Based Multi Objective Evolutionary Algorithms. In *EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation III*, Oliver Schuetze, Carlos A. Coello Coello, Alexandru-Adrian Tantar, Emilia Tantar, Pascal Bouvry, Pierre Del Moral, and Pierrick Legrand (Eds.). Springer International Publishing, Heidelberg, 233–259. https://doi.org/10.1007/978-3-319-01460-9_11

[20] Robert J Winter. 2014. *Agile Software Development: Principles, Patterns, and Practices*. Wiley Online Library.

[21] Qingfu Zhang and Hui Li. 2007. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation* 11, 6 (December 2007), 712–731. https://doi.org/10.1109/TEVC.2007.892759

[22] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. 2011. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation* 1, 1 (2011), 32–49. https://doi.org/10.1016/j.swevo.2011.03.001

[23] Eckart Zitzler and Lothar Thiele. 1998. Multiobjective optimization using evolutionary algorithms — A comparative case study. In *Parallel Problem Solving from Nature — PPSN V*, Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 292–301. https://doi.org/10.1007/BFb0056872