Paradiseo: From a Modular Framework for Evolutionary Computation to the Automated Design of Metaheuristics

Johann Dreo, Arnaud Liefooghe, Sébastien Verel, Marc Schoenauer, Juan J. Merelo, Alexandre Quemy, Benjamin Bouvier, Jan Gmys

▶ **To cite this version:**

Johann Dreo, Arnaud Liefooghe, Sébastien Verel, Marc Schoenauer, Juan J. Merelo, et al.. Paradiseo: From a Modular Framework for Evolutionary Computation to the Automated Design of Metaheuristics: 22 Years of Paradiseo. GECCO 2021 - Genetic and Evolutionary Computation Conference, ACM Sigevo, Jul 2021, Lille / Virtual, France. pp.1522-1530, 10.1145/3449726.3463276 . pasteur-03220556

HAL Id: pasteur-03220556

https://hal-pasteur.archives-ouvertes.fr/pasteur-03220556

Submitted on 7 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike| 4.0 International License

Paradiseo: From a Modular Framework for Evolutionary Computation to the Automated Design of Metaheuristics

—22 Years of Paradiseo—

Johann Dreo[1], Arnaud Liefooghe[2], Sébastien Verel[3], Marc Schoenauer[4], Juan J. Merelo[5], Alexandre Quemy[6], Benjamin Bouvier[7], and Jan Gmys[8]

[1] *Systems Biology Group, Depepartment of Computational Biology, USR 3756, Institut Pasteur and CNRS, Paris, France.* johann@dreo.fr —*Corresponding author.*
[2] *Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille.* arnaud.liefooghe@univ-lille.fr
[3] *Univ. Littoral Côte d'Opale, Calais, France.* verel@univ-littoral.fr
[4] *TAU, Inria, CNRS & UPSaclay, LISN, Saclay, France.* marc.schoenauer@Inria.fr
[5] *University of Granada, Granada, Spain.* jjmerelo@gmail.com
[6] *Poznan University of Technology, Poznan, Poland.* alexandre.quemy@gmail.com
[7] public@benj.me
[8] *Inria, Lille, France.* jan.gmys@inria.fr

Abstract

The success of metaheuristic optimization methods has led to the development of a large variety of algorithm paradigms. However, *no algorithm* clearly dominates all its competitors on *all problems*. Instead, the underlying *variety of landscapes* of optimization problems calls for a variety of algorithms to solve them efficiently. It is thus of prior importance to have access to mature and flexible software frameworks which allow for an efficient exploration of the *algorithm design space*. Such frameworks should be flexible enough to accommodate any kind of metaheuristics, and open enough to connect with higher-level optimization, monitoring and evaluation softwares. This article summarizes the features of the Paradiseo framework, a comprehensive C++ free software which targets the development of modular metaheuristics. Paradiseo provides a highly *modular* architecture, a large set of *components*, *speed* of execution and *automated algorithm design* features, which are key to modern approaches to metaheuristics development.

1 Introduction

In the research domain of metaheuristics for black-box optimization, a very large variety of algorithms has been developed since the first *Evolution Strategies* appeared in 1965 [31]. Starting from nature-inspired computing methods and following recent mathematical approaches, numerous applications have shown the efficiency of those randomized search heuristics. However, following Wagner et al. [33], we observe that the metaheuristic research domain lacks mature software, while it is crippled with short-lived research prototypes on over-specific features sets. We believe this state hinders the adoption of those technologies in the industrial world and is an obstacle to breakthrough innovations. Therefore, the development of a full-featured and mature metaheuristic optimization framework is of prior importance, for both the scientific and the applied communities. In this article, we summarize our efforts towards this goal, in the guise of the Paradiseo project.

The Paradiseo framework is a 22 years old effort which aims at developing a flexible architecture for the generic design of metaheuristics for hard optimization problems. It is implemented in C++, a very mature object-oriented programming language,which is probably one of the fastest, if not *the* fastest, object-oriented programming platforms on the market [19, 30, 28]. It is also highly portable and benefits from very extensive tooling as well as an active community. Paradiseo is released as a free and open-source software, under the LGPL-v2 and CeCILL licenses (depending on the module). Its development is open and the source code is freely available on the Inria[1] and Github[2] code repositories.

[1] https://gitlab.inria.fr/paradiseo/paradiseo
[2] https://github.com/jdreo/paradiseo

1.1 History

The "Evolving Objects" (EOlib, then simply EO) framework was started in 1999 by the Geneura team at the University of Granada, headed by Juan Julián Merelo. The development team was then reinforced by Maarten Keijzer, who designed the current modular architecture, and Marc Schoenauer [21]. Later came Jeroen Eggermont, who, among other things, did a lot of work on genetic programming, Olivier König, who did a lot of useful additions and cleaning of the code, and Jochen Küpper.

The Inria *Dolphin* team, headed by El-Ghazali Talbi, did a lot of contributions starting from around 2003, on their own module collection called Paradiseo. Thomas Legrand worked on particle swarm optimization, the regretted Sébastien Cahon and Nouredine Melab worked on parallelization modules [7, 6, 4, 5]. Arnaud Liefooghe and Jérémie Humeau worked a lot on the multi-objective module [24] and on the local search one along with Sébastien Verel [18]. In the same team, C. FC.[3] and Jean-Charles Boisson made significant contributions.

The (then) EO project was taken over by Johann Dreo, who worked with the help of Caner Candan on adding the EDO module. Johann and Benjamin Bouvier have also designed a MPI parallelization module, while Alexandre Quemy also worked on parallelization code.

In 2012, the two projects (EO and Paradiseo) were merged into a single one by Johann Dreo, Sébastien Verel and Arnaud Liefooghe, who have been acting as maintainers ever since.

In 2020, automated algorithm selection design and binding toward the IOHprofiler validation tool were added by Johann Dreo.

Along the life of the project, several spin-off software have been developed, among which a port of the EO module in Java [1], another one in ActiveX[4]; GUIDE, a graphical user interface for assembling algorithms [10][5], and EASEA, a high-level declarative language for evolutionary algorithm specification [9], which later became independent [26] of the specific library.

1.2 Related Frameworks

The 1998's version of the hitch-hiker's guide to evolutionary computation (frequently asked question in the `comp.ai.genetic` Usenet newsgroup[6]) already lists 57 software packages related to the implementation of evolutionary algorithms (among which EOlib, the ancestor of Paradiseo).

Table 1: Main software frameworks for evolutionary computation and metaheuristics. Fastest languages are figured in green and slowest in red, copyleft licenses are in red. "kloc" stands for "thousands of lines of code".

Name	Language	Update	License	Contributors	kloc	Evol.	EDAs	PSO	Local Search	Cluster	Multicore	GPGPU	Multiobjective	Landscapes	States	Auto. Design
ParadisEO	C++	2021	LGPLv2	33	82	Y	Y	Y	Y	Y	Y	~	Y	Y	Y	Y
jMetal	Java	2021	MIT	29	60	Y	N	Y	N	Y	N	N	Y	N	?	N
ECF	C++	2017	MIT	19	15	Y	N	Y	N	Y	N	N	N	N	Y	N
ECJ	Java	2021	AFLv3	33	54	Y	Y	Y	N	Y	Y	Y	Y	N	Y	N
DEAP	Python	2020	LGPLv3	45	9	Y	N	N	N	Y	Y	N	Y	N	Y	N
CIlib	Scala	2021	Apachev2	17	4	Y	N	N	N	N	N	N	N	N	?	N
HeuristicLab	C#	2021	GPLv3	20	150	Y	N	Y	Y	Y	Y	N	Y	~	Y	N
Clojush	Clojure	2020	EPLv1	17	19	Y	N	N	N	N	N	N	N	N	N	N

Most of those software are now unmaintained or impossible to find. There has been, however, a constant flow of new frameworks, library or solvers every year, for decades. We were able to find at least 47 of them readily available on the web[7]. Among those projects, only 8 met all the following criteria:

1. open-source framework aiming at *designing* algorithms[8],
2. being active since 2015,
3. having more than 15 contributors.

The features of those main frameworks are compared in Table 1, where the number of lines of code was computed with the `cloc` tool[9]. Note that for HeuristicLab, the code for the GUI modules was excluded from the count. The GPGPU module of Paradiseo [27] is not counted either, as it is not maintained anymore. The number of contributors has been retrieved from the code repository's commit histories, which underestimates the number of people involved in the case of Paradiseo; the extracted number is however kept, for fairness in comparison with the other frameworks, that might face a similar bias.

Among the software close in features to Paradiseo, ECF has not been updated in 4 years. ECJ, jMetal

[3] Redacted by author's demand.

[4] http://geneura.ugr.es/~jmerelo/DegaX/

[5] Which also supported ECJ.

[6] Discussion forum which was popular before the World Wide Web and social networks. http://coast.cs.purdue.edu/pub/doc/EC/FAQ/www/Q20.htm

[7] Paradiseo, jMetal, ECF, OpenBeagle, Jenetics, ECJ, DEAP, CIlib, GP.NET, DGPF, JGAP, Watchmaker, GenPro, GAlib, HeuristicLab, PyBrain, JCLEC, GPE, JGAlib, pycma, PyEvolve, GPLAB, Clojush, µGP, pySTEP, Pyvolution, PISA, EvoJ, Galapagos, branecloud, JAGA, PMDGP, GPC++, PonyGE, Platypus, DCTG-GP, Desdeo, PonyGE2, EvoGrad, HyperSpark, Nevergrad, Pagmo2, LEAP, Operon, EMILI, pso-de-framework, MOACO.

[8] *Libraries* of solvers, like Pagmo2 or Nevergrad, do not match this criterion.

[9] version 1.82 of https://github.com/AlDanial/cloc

are close competitors, albeit programmed in Java, which is expected to run near 2.6 times slower than programs in C++[10], a key drawback for automated algorithm design (see Section 4.3). HeuristicLab suffers from the same drawback, but provides a graphical user interface for the run and analysis of solvers. Paradiseo does not provide such a GUI, but relies on dedicated third-party tools for this kind of functionality (see [10] and Section 4.3). The other frameworks do not provide the same level of features and use languages that are generally slower than C++ [28].

2 Architecture

From its inception [29], Paradiseo opted for an original architecture design, exemplified by its name, "Evolving *object*", as opposed to a procedural or functional view of the algorithm. In this section, we expose first the main concepts used in its architecture, to focus next on the design patterns that have been used in it, giving it room for evolution and improvement along the years.

2.1 Main Concepts

The core of Paradiseo is formed by the EO module, which has been designed for general evolutionary algorithms. Most of its core concepts are used across the other modules and are named after its vocabulary:

Encoding: The data structure modelling a solution to the optimization problem (which type is generally denoted `EOT`).

Evaluation: The process of associating a value to a solution, thanks to an objective function.

Fitness: The value of a solution as seen by the objective function.

Operator: A function which reads and/or alters a (set of) solutions.

Population: A set of solutions.

2.2 Main Design Patterns

Paradiseo is a *framework*, providing a large set of components that the user can assemble to implement a solver. To facilitate and enforce the design and use of components, Paradiseo is based on four main design patterns: Functor, Strategy, Generic Type and Factory. Figure 1 shows a high-level view of the global design pattern.

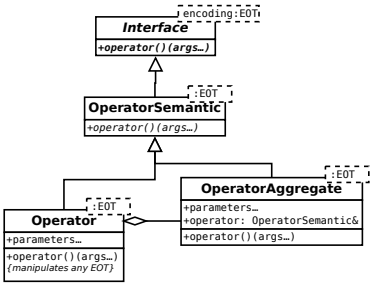

Figure 1: UML diagram of a high-level view of the main design pattern used in the Paradiseo framework.

Functor: In Paradiseo, most of the operators are *functions* (exposing the `operator()` interface) holding a *state* between calls [32, 21]. Member variables of the functors are either parameters or references to other functors, involved in the computation.

Strategy: Operators can be composed to form another one. For instance, an `eoAlgo` is essentially an operator holding a loop which calls other operators. These operators must then honor an interface, which provides the *semantic* of the underlying operation [14]. For instance, an `eoSelectOne` exposes the interface to pick a solution within a *population*: `const EOT& operator()( const eoPop<EOT>&)`.

Generic type: Almost all the operators in Paradiseo are defined over a `EOT` template holding the *encoding* of a solution to the optimization problem. This allows for two crucial features: (i) the user can provide her own data structure, without major redesign, and (ii) any operator deep in the call tree may have access to any specific interface of the encoding [32]. This was one of the earliest decisions taken, and was already presented in [21].

Factory: As many operators are abstracted through their interfaces, Paradiseo provides ways to manage them as collections or high-level aggregates, so that the user does not have to manage the details. For instance, Paradiseo provides classical stopping criterion collections or on-the-fly instantiation (see also Section 4.3).

With this approach, Paradiseo is enforcing the use of *composition* of objects, limiting the use of *inheritance*

[10] Following the "n-body" setup of the "Computer Language Benchmarks Game", which is the closest problem to our setting: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html

to interfaces (generally of abstract classes). One of the main goals of the framework is to be able to easily *compose* new algorithms, by (i) reusing existing common features (logging, parallelization, state serialization, etc.) and (ii) assembling existing algorithmic operators, for instance by hybridizing algorithms.

Figure 2 shows an example of the core concepts which a user may see: the `EO` class, from which a solution to the optimization problem will inherit the interface necessary to be used across the operators, and a set of operator interfaces and implementations, which define an algorithm.

3 Modules

Paradiseo targets modular algorithms and is thus organized in several modules: Evolving Objects (EO) for population-based algorithms such as evolutionary algorithms and particle swarm optimization, Moving Objects (MO) for local search algorithms and landscape analysis, Estimation of Distribution Objects (EDO) for estimation of distribution algorithms, and Multi-Objective Evolving Objects (MOEO) for multi-objective optimization. Each module brings its own specific concepts and features, on top of the common core features provided in the EO module. The following sections summarize this organization.

3.1 Evolutionary and Particle Swarm Algorithms — EO

The EO module defines a number of common operators which are used across all the framework:

Evaluation of populations: Operators which call the objective function for a set of solution. With several parallelization options [7, 6, 4, 5].

Initialization: Operators which generate solutions, out of the optimization loop (generally at random).

Continue: A stopping criterion which returns `false` if the iteration loop of an algorithm is to be stopped.

Checkpoints: A generic operator which is called at each iteration (for instance to collect statistics).

Updater: A generic operator which can update a parameter.

State: A serialization of a state of an algorithm.

Wrappers: Operators which transform a (set of) operators as another one.

Generic encodings: Some solution representations which are often used when solving optimization problems, like numerical vectors, binary vectors, trees, permutations.

Parameters: An abstraction of parameters, which can be used for command line interfaces, state management and dynamic algorithms.

Those features are generally used by the following modules.

The EO module holds the following necessary classes to implement evolutionary algorithms, as illustrated in Figure 3 (lower part):

Selection: Operators which pick (a set of) solutions within a population. Two levels are available: operators which pick a single solution (`eoSelectOne` interface), and operators which select more than one solution (`eoSelect`).

Variation: Operators which generate new solutions by altering existing ones. They are generally called "mutations", when they alter a single solution, and "crossover" when they alter two or more solutions at once. Both types return a boolean which is `true` if an alteration has actually been done.

Replacement: Operators which merge two populations, typically "parents" with "offsprings" (produced by the alteration of parents with variation operators).

Algorithms: High-level operators which manipulate a population generated at initialization, and iteratively apply a set of operators until a stopping condition is satisfied.

More information on the design of evolutionary algorithms within Paradiseo-EO can be found in [21].

EO also defines classes which target particle swarm optimization algorithms:

Particle: An interface on top of the `EO` class, which defines a freely moving particle.

Velocity: Operators which control the speed at which a particle is moving.

Flight: Operators which control the next position at which the particle will be.

3.2 Local Search and Landscape Analysis — MO

The MO module adds an interface which can manage *single solutions* instead of *populations*, mainly providing a fine-grained level of abstraction, following the

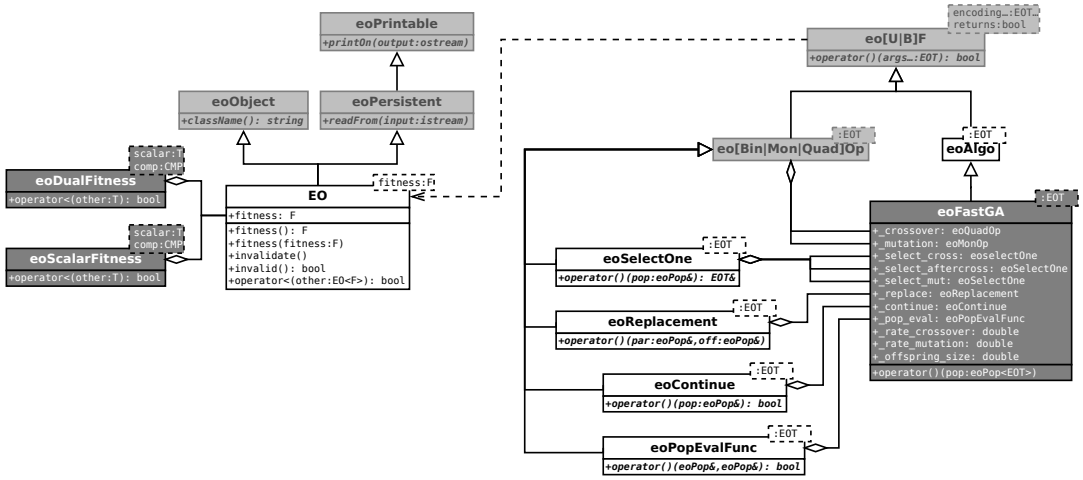

Figure 2: General overview of the main classes involved in assembling one of the Paradiseo-EO algorithms. Concrete class implementations manipulated by the user are shown in dark gray. Interfaces which define generic behavior are figured in white. Low-level convenience classes provided by the framework are figured in light gray.

same components as the EO module. It also adds the important concept of *incremental evaluation*, to allow the design of objective functions which compute the value of a solution based on the application of a *move* to an already evaluated solution. The objective function can thus take into account only the sub-parts of the solutions that have been altered, effectively improving the computation time. Those operators are tightly coupled with *neighborhoods*, which are variation operators applied to single solutions.

The MO module additionally provides components to sample the search space and estimate statistics for characterizing the *fitness landscape* of the problem in terms of features, such as the density of states, the fitness distance correlation, the autocorrelation function, the length of adaptive walks, the landscape neutrality, or the fitness cloud [17, 11]. More information about local search and landscape analysis in Paradiseo-MO can be found in [18].

3.3 Multi-Objective Optimization — MOEO

The MOEO module adds the necessary features to handle multi-objective optimization [8, 35]:

Fitness assignment: Large set of operators which convert raw objective values into ranks or fitness values used for selection and replacement. They include state-of-the-art scalarizing-, dominance- and indicator-based approaches.

Diversity preservation: Operators which maintain diversity in the population, seeking for well-spread and uniformly-distributed solutions in the objective space.

Selection: Operators which combine the ones above in order to guide the population towards Pareto-optimal solutions.

Archive: Secondary population which maintains non-dominated solutions.

Performance metrics: Quality indicators computed over populations or archives to measure solution quality in multi-objective optimization.

More information about the Paradiseo-MOEO module, and how to design multi-objective local search and evolutionary algorithms in Paradiseo are detailed in [24].

3.4 Estimation of Distribution — EDO

The EDO module encompasses the features to manage population-based algorithms which have an explicit state from which the population is derived at each iteration.

Distribution: A template, wrapping the encoding `EOT` and holding the data structure representing a probability distribution.

Estimator: Operators which compute distribution parameters from a given population.

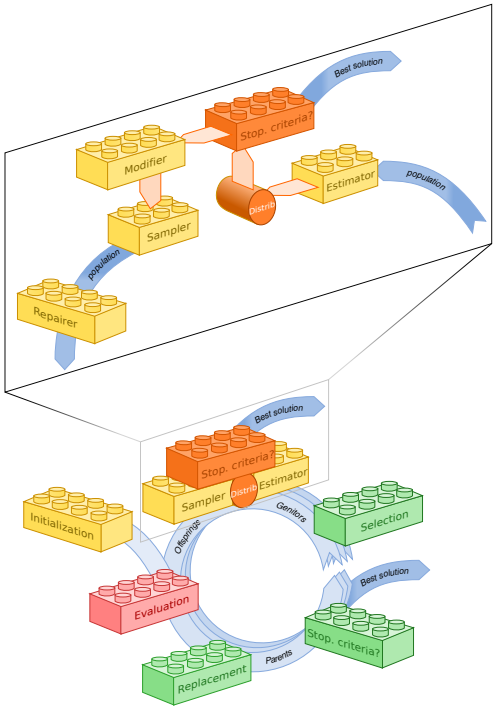

Figure 3: Modular estimation of distribution algorithm as seen from Paradiseo-EDO. The lower part of the diagram is the modular evolutionary algorithm loop. EDO adds a set of operators to replace "implicit" variation operator by "explicit" ones. The operators managing the probability distribution are shown in orange.

Sampler: Operators which compute a population from a given distribution.

Several other operators allow to manipulate and combine those objects and to plug them within EO evolutionary algorithm's variation operator, as shown on Figure 3.

4 Key Features

4.1 Modular Algorithms

The main feature of Paradiseo is to provide a large set of modular algorithms, which are assembled from a large variety of operators. This is motivated by the fact that there exists a large diversity of optimization problems, which would be more efficiently solved by specific algorithms rather than generic ones.

It is thus of prior importance to be able to easily explore the design space of algorithms, in order to find the best one for a given problem. Having a large set of reusable components is key to allow the practitioner to quickly try new algorithm variants, which may not have been tested yet. New ideas can also be experimented with minimum effort, by allowing the user to focus on a single (new) component. Figure 4 shows a simple example of a modular genetic algorithm (inspired from [12]), which allows for the instantiation of 1 630 475 different algorithm *instances*. In that case, an algorithm instance is a combination of parameterized operators, with varying functions and/or parameters[11]. Of course, considering the whole footprint of Paradiseo would allow for far larger design space.

It is also worth noting that the hybridization of two algorithms in Paradiseo is as simple as encapsulating operators with a similar interface. For instance, it is straightforward to use a local search algorithm implemented with MO as a variation operator of an evolutionary algorithm implemented with EO, then ending up with a so-called memetic algorithm.

Moreover, this modular architecture facilitates a fair comparison of algorithms in practical use cases, where *wall-clock* performance is of prior importance; e.g., for applications involving interactions with a human. In such a case, having a common code base helps conducting more unbiased studies.

4.2 Fast Computations

Paradiseo is one of the few optimization frameworks written in C++, a compiled programming language known for its runtime speed. Moreover, its design is

[11] Generally numerical or integer parameters, sometimes boolean or categorical ones.

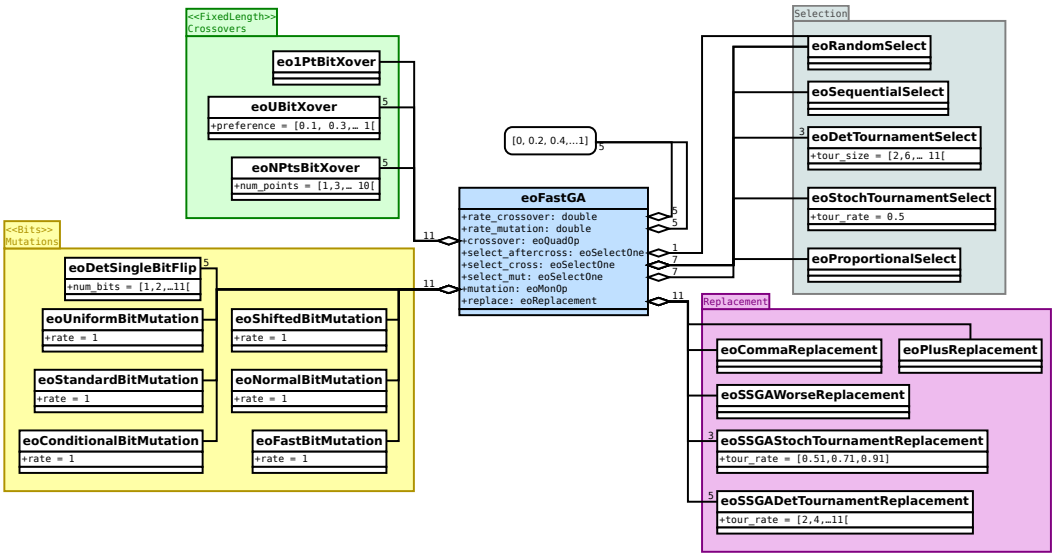

Figure 4: Example of relationships between an algorithm template (`eoFastGA`) and its related operators. Each package box groups alternative operators which may be used for the corresponding step of the algorithm.

thought to directly plug components at compile time rather than relying exclusively on dynamically-run conditional expressions.

A typical rationale in black-box optimization is to state that the efficiency of the algorithm computations is not a concern, because in real cases the objective function dominates the runtime. While this is true in essence, this argument forgets that, during the design phase of algorithms, practitioners most often do not use complex objective functions, but synthetic ones, which are very fast to compute. In that case, fast computation means fast design iterations.

For example, a CMA-ES algorithm implemented with Paradiseo is 10 times faster than its heavily optimized counterpart implemented with Python/Numpy, when solving a standard synthetic benchmark. Those measures are obtained using the reference implementation of CMA-ES available in the pycma package[12], solving the Black-Box Optimization Benchmark (BBOB [15]) of the COmparing Continuous Optimizers (COCO[13]) platform [16]. The Paradiseo implementation[14] used the independent implementation of BBOB available on the IOHexperimenter[15] platform [13]. Both benchmarks are implemented in C/C++. Running algorithms on the whole benchmark, on a single Intel Core i5-7300HQ at 2.50GHz with a Crucial P1 solid-state disk, takes approximately 10 minutes with pycma/COCO, and only 1 minute with the Paradiseo/IOH implementation.

In addition, the Symmetric Multiprocessing module (SMP) allows to wrap any operator called within a loop transparently to fully make use of CPU cores. The master-worker model has been shown to scale (near) linearly with the number of cores, while having a low communication overhead. SMP also provides a parallel island model [34] that speeds up algorithm convergence while maintaining diversity.

At last, we argue that fast computations of the algorithm and objective function are necessary features to facilitate automated algorithm design, where an algorithm is itself in charge of finding the most appropriate variant of an algorithm [22], *learn* what is the best algorithm for a given benchmark [20], or even a given problem *fitness landscape* [23, 3, 11]. In that case, running an assembled algorithm on a benchmark is the objective function of the design problem, and its computation time determines the scale at which the experiment can be conducted. This feature is discussed in more detail below.

4.3 Automated Algorithm Design

Automated algorithm design features are recent additions to Paradiseo. They target the ability to assemble algorithms at runtime without loss of performance, and easy bindings with benchmarking and algorithm selection tools. Figure 5 shows the global setting, which is detailed in this section.

[12] Version 3.0.0 of https://github.com/CMA-ES/pycma

[13] Version 2.3.2 of https://github.com/numbbo/coco

[14] Version 640fa31 of https://github.com/nojhan/paradiseo

[15] Version 2395af4 of https://github.com/nojhan/IOHexperimenter

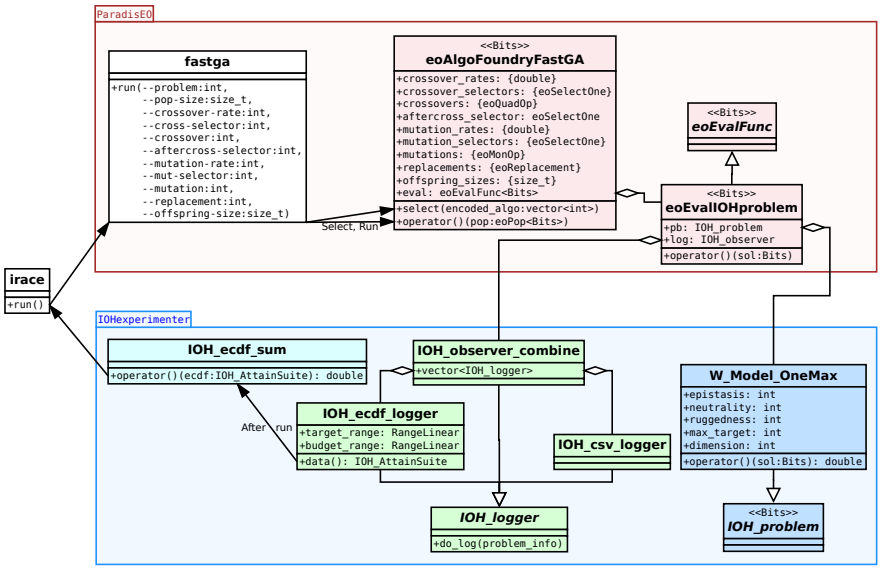

Figure 5: Flow of information involved in automated algorithm design, starting with irace calling an executable interface (white box), to instantiate and run a Paradiseo algorithm (red boxes), which will call an IOHexperimenter problem (blue boxes) while being observed by a logger (green boxes). The final performance is computed (cyan box) and returned to irace.

On-the-fly Operator Instantiation with Foundries Foundries are "Factory" classes which allow to instantiate a parameterized operator, chosen among a set of operators having the same interface. The user can indicate which classes and parameters should be managed and a foundry is responsible for instantiating when it is called with the index of the operator to be instantiated. This allows for simple numerical interfaces with algorithm selection solvers (i.e. generic hyper-parameters tuning). An algorithm foundry is thus a generic meta-algorithm, which can instantiate and call an actual algorithm class. It follows the same interface as an algorithm, but models the operators of this algorithm as *operator foundries* rather than references to operator instances. Those operator foundries are responsible for instantiating the operator when asked to do so. Figure 6 shows the classes involved.

An end-user willing to find the best algorithm variant for her needs would need to select a subset of parameterized operators of interest and add them to the foundry. Then, it is sufficient to indicate (potentially at runtime) which algorithm should be instantiated and run. Let us illustrate this with an example in Listing 1.

```
// Considering the FastGA modular algorithm,
// solving a problem with fixed initialization.
auto& foundry = store.pack<
```

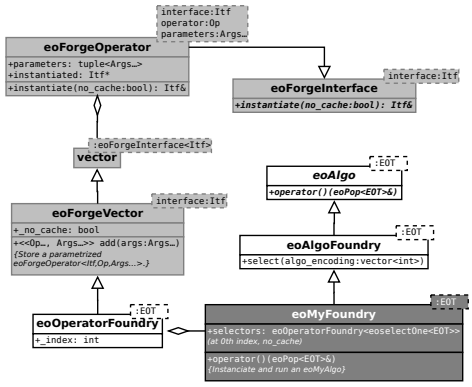

Figure 6: Classes involved in a meta-algorithm instantiation. The `eoMyFoundry` class is to be designed by the practitioner, who has to know the interfaces presented with a white background. Classes with a grey background are the underlying framework machinery.

```
    eoAlgoFoundryFastGA<Bits> >(
  init, problem, max_eval_nb, /*
    max_restarts=*/1);
// Consider different crossover operators.
for(double i=0.1; i<1.0; i+=0.2) {
  foundry.crossovers.add< eoUBitXover<Bits>
    >(i);
  foundry.crossovers.add< eoNPtsBitXover<
    Bits> >(i*10);
}
// And different variation rates.
for(double i=0.0; i<1.0; i+=0.2) {
    foundry.crossover_rates.add<double>(i);
    foundry.mutation_rates.add<double>(i);
}
// etc.
// Decide which operators to use.
Ints encoded_algo(foundry.size());
encoded_algo[foundry.crossovers      .index
    ()] = 2;
encoded_algo[foundry.crossover_rates.index
    ()] = 1;
encoded_algo[foundry.mutation_rates .index
    ()] = 3;
// etc.
// Instantiate the operators, or use cached
     objects.
foundry.select(encoded_algo);
// Run the selected algorithm.
eoPop<Bits> pop; // [...]
foundry(pop);
```

Listing 1: Excerpt of the use of an algorithm foundry.

Binding with IOH for Fast Benchmarking One of the key features when doing automated algorithm design is the ability to run the assembled algorithm against a whole benchmark, and to measure performance on this experiment. Paradiseo is not intended to host benchmarks, apart for a few examples, but provides an interface to the IOHexperimenter benchmarking platforms. IOHexperimenter provides a set of benchmarks, along with a standardized logging system, which can log both calls to the objective function and parameters status. Paradiseo provides several entry points to IOHexperimenter problems, in the form of sub-classes of the `eoEvalFunc` interface, which can be plugged into any Paradiseo algorithm. IOHexperimenter also provides a way to extract performance measures from the runs' logging outputs, with statistics computed on aggregated in-memory discrete empirical cumulative density functions. Those distributions are defined on both the computation time and the quality of solutions axes and can thus produce many different performance metrics. This allows for a fast logging and performance assessment system, which is ideal for automated algorithm design. Listing 2 shows an example of use of the Paradiseo/IOH binding when solving a particular problem.

```
// In-memory logger.
IOHprofiler_RangeLinear<size_t>
  target_range(0, max_target, buckets),
  budget_range(0, max_evals , buckets);
IOHprofiler_ecdf_logger<int,int,int>
    ecdf_logger(
  target_range, budget_range);
// Benchmark problem.
W_Model_OneMax w_model_om;
ecdf_logger.track_problem(w_model_om);
// The actual Paradiseo/IOH interface:
eoEvalIOHproblem<Bits> pb(w_model_om,
    ecdf_logger);
// [`pb` is plugged into an algorithm and
    ran...]
// The performance of the run is recovered:
IOHprofiler_ecdf_sum ecdf_sum;
long perf = ecdf_sum(ecdf_logger.data());
```

Listing 2: Excerpt of the use of the IOH binding.

It is worth noting that the same approach can be used with the IOH's file logger, which allows for a fine-grained analysis of the algorithm behavior within the IOHanalyzer graphical user interface. Although the runtime is longer because of the involved I/O accesses, this can be useful for the post-validation of the algorithm instance showing the best performance, without having to change the code.

Interface with irace for Automated Algorithm Configuration With the approach described previously, the performance of an algorithm instance can be computed on a given benchmark with the use of a single binary, without much computation time or memory overhead. Thanks to the utility features provided by Paradiseo, it is straightforward to expose the meta-algorithm and the problem interfaces as parameters to the executable (either as parameter files or as command line arguments). This allows for an easy binding with most automated algorithm configuration tools.

Several automated algorithm configuration tools have been developed in the last decade, among which one of the most used is irace [25]. Paradiseo provides a way to easily expose a foundry interface as an irace configuration file, as shown in Listing 3.

```
// Using Paradiseo parameters:
eoParser parser(argc, argv, "interface for
    irace");
auto crossover_p = parser.getORcreateParam<
    size_t>(
  /*default=*/0, "crossover",
  /*help=*/"The crossover operator", /*flag
    =*/'c',
  /*help section=*/"Operator Choice", /*
    required=*/true);
// [...] assemble a foundry [...]
// Print the irace's configuration file for
     this binary:
```

```
std::cout << "# name\t switch\t type\t range\n";
// We only need the parameter(s) and the foundry itself:
print_irace(mutation_rate_p, foundry.mutation_rates,
  std::cout);
print_irace(    crossover_p, foundry.crossovers    ,
  std::cout);
// Any other operator within the foundry [...]
/* This will output something like:
# name       switch              type range
mutationrate "--mutation-rate=" i    (0,4)
crossover    "--crossover="     c
    (0,1,2,3,4,5,6,7)
[etc.] */
```

Listing 3: Excerpt of code for exposing a Paradiseo interface to irace.

With this setting, it has been possible to conduct a large scale algorithm design study [2] , involving irace configuring a *FastGA* algorithm family (cf. Figure 4) solving a W-model problem, using a budget of approximately *1 billion* function evaluations, in approximately *3 hours* on a single core of a laptop (same setup than Sec. 4.2).

5 Conclusions

This article provides a high-level overview of the Paradiseo framework, a C++ free software which targets the development of modular metaheuristics. The main feature of Paradiseo is its ability to help practitioners to focus on creating solvers while thinking at a higher level of abstraction, thanks to:

Utility features: Paradiseo provides a large set of engineering features, which very often lack in proof-of-concept frameworks: several fine-grained parallelization options, convenient interface features (command line argument parsing, state management, useful logs, etc.). Having robust implementation of such features is often overlooked by users focusing on the algorithmic part.

Component-based architecture: The concept of *operator* is at the core of the design of Paradiseo. It allows for the composition of algorithms, without the overhead of a dynamically loading plugins or the rigidity of a monolithic structure. Solvers being assembled as a selection of components are also lightweight, as it is not necessary to build and carry all the framework's code within the executable binaries.

Modular algorithm models: Paradiseo provides several modules targeting different algorithm paradigms —probably one of the largest footprints among active frameworks. Practitioners can easily design new algorithms which differ in some operators, hybridize algorithms, or even add new algorithm templates using existing operators.

Algorithm design: Paradiseo focuses on providing a very large design space to the practitioner. Thanks to its fast computations, large-scale design experiments can be addressed. Combined with its features dedicated to automation, Paradiseo algorithm designers who want to test a new operator can easily focus on small code changes and rapidly check their efficiency and how they interact with other operators in a given paradigm. Problem solvers can sort out algorithm instances that work best.

As future works, we plan to improve the algorithm design automation features, merge more state-of-the-art modular designs, and enhance the overall user experience.

One of the main impediments to a more widespread use of the framework is that the learning curve for getting started is too steep. Solving this problem is the main objective of PyParadiseo, a module currently under development, that will expose interfaces to Paradiseo in Python, to facilitate the interoperability with external solvers, statistics program or machine learning frameworks.

Acknowledgments

During 22 years, Paradiseo has been developed by more than 50 people, with the support of the following institutions: Inria, University of Lille, University of the Littoral Opal Coast, Thales, École Polytechnique, University of Granada, Vrije Universiteit Amsterdam, Leiden University, French National Centre for Scientific Research (CNRS), French National Agency for Research (ANR), Fritz Haber Institute of the Max Planck Society, Center for Free-Electron Laser Science, University of Angers, French National Institute of Applied Sciences, Free University of Brussels, Pasteur Institute.

References

[1] Maribel García Arenas, Brad Dolin, Juan Julián Merelo Guervós, Pedro Ángel Castillo Valdivieso, Ignacio Fernández De Viana, and Marc Schoenauer. Jeo: Java evolving objects. In *GECCO*, volume 2, pages 991–994, 2002.

[2] Amine Aziz-Alaoui, Carola Doerr, and Johann Dreo. Towards Large Scale Automated Algorithm Design by Integrating Modular Bench-

marking Frameworks. In *Proceedings Companion of the Annual Conference on Genetic and Evolutionary Computation*, GECCO'21. to appear, 2021.

[3] Nacim Belkhir, Johann Dreo, Pierre Savéant, and Marc Schoenauer. Per instance algorithm configuration of cma-es with limited budget. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 681–688, 2017.

[4] Sébastien Cahon, Nordine Melab, and E.-G. Talbi. Building with paradiseo reusable parallel and distributed evolutionary algorithms. *Parallel Computing*, 30(5–6):677–697, 2004.

[5] Sébastien Cahon, Nordine Melab, and E-G Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics*, 10(3):357–380, 2004.

[6] Sébastien Cahon, Nordine Melab, E-G Talbi, and Marc Schoenauer. Paradiseo-based design of parallel and distributed evolutionary algorithms. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 216–228. Springer, 2003.

[7] Sébastien Cahon, E-G Talbi, and Nordine Melab. Paradiseo: a framework for parallel and distributed biologically inspired heuristics. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 9–pp. IEEE, 2003.

[8] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Genetic and Evolutionary Computation Series. Springer, New York, USA, second edition, 2007.

[9] Pierre Collet, Evelyne Lutton, Marc Schoenauer, and Jean Louchet. Take It EASEA. In Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN VI*, Lecture Notes in Computer Science, pages 891–901. Springer.

[10] Pierre Collet and Marc Schoenauer. GUIDE: Unifying Evolutionary Engines through a Graphical User Interface. In Pierre Liardet, Pierre Collet, Cyril Fonlupt, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution*, Lecture Notes in Computer Science, pages 203–215. Springer.

[11] Bilel Derbel and Sébastien Verel. Fitness landscape analysis to understand and predict algorithm performance for single- and multi-objective optimization. In Carlos Artemio Coello Coello, editor, *GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume, Cancún, Mexico, July 8-12, 2020*, pages 993–1042. ACM, 2020.

[12] Benjamin Doerr, Carola Doerr, and Franziska Ebel. Lessons from the black-box: Fast crossover-based genetic algorithms. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 781–788, 2013.

[13] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. arXiv: 1810.05281.

[14] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[15] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '10, page 1689–1696, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. COCO: a platform for comparing continuous optimizers in a black-box setting. 36(1):114–144.

[17] H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco, CA, USA, 2004.

[18] Jérémie Humeau, Arnaud Liefooghe, E-G Talbi, and Sébastien Verel. Paradiseo-mo: From fitness landscape analysis to efficient local search algorithms. *Journal of Heuristics*, 19(6):881–915, 2013.

[19] Robert Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*.

[20] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance Prediction and Automated Tuning of Randomized

and Parametric Algorithms. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, Lecture Notes in Computer Science, pages 213–228. Springer.

[21] Maarten Keijzer, Juan J Merelo, Gustavo Romero, and Marc Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 231–242. Springer, 2001.

[22] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. 27(1):3–45.

[23] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the Empirical Hardness of Optimization Problems: The Case of Combinatorial Auctions. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, Lecture Notes in Computer Science, pages 556–572. Springer.

[24] Arnaud Liefooghe, Laetitia Jourdan, and El-Ghazali Talbi. A software framework based on a conceptual unified model for evolutionary multiobjective optimization: Paradiseo-moeo. *European Journal of Operational Research*, 209(2):104–112, 2011.

[25] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[26] Ogier Maitre, Frédéric Krüger, Stéphane Querry, Nicolas Lachiche, and Pierre Collet. EASEA: specification and execution of evolutionary algorithms on GPGPU. 16(2):261–279.

[27] Nouredine Melab, Thé Van Luong, Karima Boufaras, and El-Ghazali Talbi. Paradiseo-mo-gpu: a framework for parallel gpu-based local search metaheuristics. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1189–1196, 2013.

[28] Juan-Julián Merelo-Guervós, Israel Blancas-Álvarez, Pedro A Castillo, Gustavo Romero, Pablo García-Sánchez, Víctor M Rivas, Mario García-Valdez, Amaury Hernández-Águila, and Mario Román. Ranking programming languages for evolutionary algorithm operations. In *European Conference on the Applications of Evolutionary Computation*, pages 689–704. Springer, 2017.

[29] Juan-Julián Merelo-Guervós, M. G. Arenas, J. Carpio, P. Castillo, V. M. Rivas, G. Romero, and M. Schoenauer. Evolving objects. In P. P. Wang, editor, *Proc. JCIS 2000 (Joint Conference on Information Sciences)*, volume I, pages 1083–1086, 2000. ISBN: 0-9643456-9-2.

[30] Sergio Nesmachnow, Francisco Luna, and Enrique Alba. An empirical time analysis of evolutionary algorithms as c programs. *Software: Practice and Experience*, 45(1):111–142, 2015.

[31] I. Rechenberg. Cybernetic Solution Path of an Experimental Problem.

[32] David Vandevoorde and Nicolai M. Josuttis. *C++ template: the complete guide*. Addison-Wesley.

[33] 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. Topics in Intelligent Engineering and Informatics, pages 197–261. Springer International Publishing.

[34] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, 7(1):33–47, 1999.

[35] E. Zitzler, M. Laumanns, and S. Bleuler. A tutorial on evolutionary multiobjective optimization. chapter 1, pages 3–38. Springer Science & Business Media.