Experiences using Julia for Implementing Multi-Objective Evolutionary Algorithms

Antonio J. Nebro[1,2] [0000−0001−5580−0484]
Xavier Gandibleux[3] [0000−0002−5055−4680]

[1] Departamento de Lenguajes y Ciencias de la Computación. University of Málaga, 29071 Málaga, Spain
[2] ITIS Software, University of Málaga, 29071, Málaga, Spain
ajnebro@uma.es

[3] Faculté des Sciences et Techniques — Département Informatique
Laboratoire des Sciences du Numérique de Nantes (UMR CNRS 6004)
Nantes Université, France
Xavier.Gandibleux@univ-nantes.fr

Abstract. Julia is a programming language suitable for data analysis and scientific computing that combines simplicity of productivity languages with characteristics of performance-oriented languages. In this paper, we are interested in studying the use of Julia to implement Multi-Objective MetaHeuristics. Concretely, we use the Java-based `jMetal` framework as a reference support and investigate how Julia could be used to design and develop the component-based architecture for multi-objective evolutionary algorithms that `jMetal` provides. By using the NSGA-II algorithm as an example, we analyze the advantages and shortcomings of using Julia in this context, including aspects related to reusing `jMetal` code and a performance comparison.

Keywords: Multi-Objective MetaHeuristics · Evolutionary Algorithms · Julia Programming Language · Open-Source Solver

1 Introduction

Scientific programming has traditionally adopted one of two programming language families: productivity languages (Python, MATLAB, R) for easy development, and performance languages (C, C++, Fortran) for speed and a predictable mapping to hardware [2]. In recent years, the Julia programming language has emerged with the aim of decreasing the gap between productivity and performance languages [3]. To achieve it, Julia is optimized for talking to LLVM (a middleman between the source code and the compiled native code), while looking as similar as possible to high-level languages.

In this paper, we are interested on the use of Julia for Multi-Objective MetaHeuristics [9, 13] based on Evolutionary Algorithms [5, 10], a field where its capabilities in handling complex computational tasks efficiently can be advantageous. Our starting point is our 18 years of experience designing and developing `jMetal`,

a Java-based framework for multi-objective optimization with metaheuristics [8, 14]. During this time, one of our main concerns has been to make the software easy to understand and use, paying attention to how to design an architecture for multi-objective metaheuristics fostering code reuse and flexibility. In the last release, `jMetal` includes a component-based template that allow to implement evolutionary algorithms in a very flexible way, by combining components of a provided catalog. This architecture is the basis of the automatic algorithm design module of `jMetal` [15, 16], and well-known multi-objective optimizers such as NSGA-II [6], MOEA/D [17], SMS-EMOA [1] are implemented using them.

By using the NSGA-II algorithm as a case study, this paper tries to answer the following research questions:

- Research question 1 (RQ1):
  Can the component-based architecture of `jMetal` be easily translated into a Julia project, taking into account that `jMetal` relies heavily on the object-oriented features of Java and Julia is not an object-oriented language (like C++ and Java)?
- Research question 2 (RQ2):
  Does the NSGA-II algorithm implemented in Julia outperform the NSGA-II version in Java in computation time?
- Research question 3 (RQ3):
  Can evolutionary algorithm operators (selection, variation, replacement) and optimization problems be easily ported from `jMetal` to Julia?

The goal of this work is not to develop a port to Julia of `jMetal`; in fact, Julia users have at their disposal the `Metaheuristics.jl` package [7], which includes NSGA-II among other multi-objective metaheuristics. We are aimed at getting an insight of the difficulties encountered when using Julia when addressing the defined research questions as well as discussing those Julia features not available in Java that can facilitate the implementation of metaheuristics and related utilities (statistical tests, graphics, notebooks, etc). Note that, while the NSGA-II implementation in `Metaheuristics.jl` has five parameters to be configured, the component-based NSGA-II in `jMetal` has more than twenty, which is facilitated by the use of such architecture.

The rest of the paper is organized as follows. Section 2 describes the main features of the jMetal framework, focusing on the component-based architecture for multi-objective evolutionary algorithms. How this scheme is implemented in Julia is presented in Section 3. Section 4 is devoted to a performance comparison, and it is followed by a discussion section. Finally, Section 7 draws conclusions and outlines open research lines.

2 The `jMetal` Framework

`jMetal` is a software package for multi-objective optimization with metaheuristics that we started to develop in 2006 as an internal tool for our own research on

multi-objective optimization. Since then, jMetal has been continuously evolved, being last current stable version (jMetal 6.2) released in 2023. The current release version is hosted in GitHub[4].

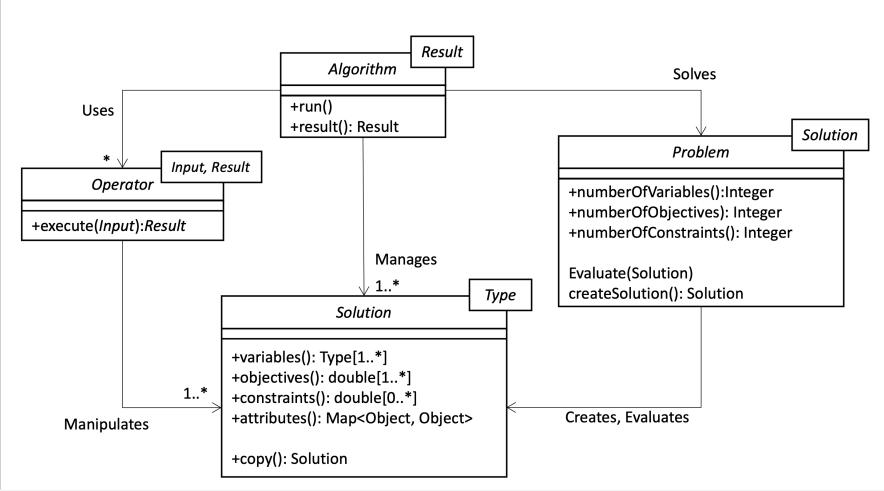

Fig. 1. UML class diagram of jMetal core classes.

The core of jMetal is based on an object-oriented design comprising four core entities: algorithms, problems, solutions, and operators, as depicted in Fig. 1. They are related among them following the idea that an algorithm solves a problem by using operators that manipulate solutions. Each entity is defined as an interface that is extended to include sub-entities and specific implementations. For example, the operator interface is extended with three interfaces for crossover, mutation, and selection operators.

Algorithm 1 Pseudo-code of an evolutionary algorithm

```
1: P(0) ← GenerateInitialSolutions()
2: t ← 0
3: Evaluate(P(0))
4: while not StoppingCriterion() do
5:     M(t) ← Select(P(t))
6:     Q(t) ← Variate(M(t))
7:     Evaluate(Q(t))
8:     P(t + 1) ← Update(P(t), Q(t))
9:     t ← t + 1
10: end while
```

[4] jMetal: https://github.com/jMetal/jMetal

Focusing on multi-objective evolutionary algorithms, we take as a reference a general description given by the pseudo-code included in Algorithm 1. Depending on how this pseudo-code is implemented, the result will be an algorithmic architecture that will have a high impact in terms of code readability, code reusability and performance of the algorithms implemented with it. In `jMetal` 6.2, we took the approach of designing a component-based architecture, composed of a template where all the steps of Algorithm 1 are incorporated as objects or components (see the code snippet of the template is shown in Listing 1.1).

```
public class EvolutionaryAlgorithm<S extends Solution<?>>
    implements Algorithm<List<S>>{

  private List<S> population;
  private Evaluation<S> evaluation;
  private SolutionsCreation<S> createInitialPopulation;
  private Termination termination;
  private Selection<S> selection;
  private Variation<S> variation;
  private Replacement<S> replacement;
  ...
  public void run() {
    population = createInitialPopulation.create();
    population = evaluation.evaluate(population);
    initProgress();
    while (!termination.isMet(attributes)) {
      List<S> matingPop = selection.select(population);
      List<S> offspringPop = variation.variate(population, matingPop);
      offspringPop = evaluation.evaluate(offspringPop);
      population = replacement.replace(population, offspringPop);
      updateProgress();
    }
    ...
  }
```

Listing 1.1. Template for component-based evolutionary algorithms in jMetal.

An interesting point of adopting this scheme is that `jMetal` provides a component catalog, so the implementation of a particular evolutionary algorithms is obtained by selecting the proper components from the catalog. Table 1 shows a subset of the available components.

An implementation of the NSGA-II algorithm can be obtained by selecting the following components:

- Solutions creation (for the initial population): random
- Evaluation: sequential
- Selection: binary tournament
- Variation: crossover and mutation
- Replacement: ranking (dominance ranking) and density estimator (crowding distance).

If NSGA-II is used to solve continuous problems, the variation operators are simulated binary crossover and polynomial mutation. The flexibility of the component-based architecture allows to easily obtain NSGA-II variants. For example, if we are interested on a differential evolution-based implementation of NSGA-II, we just need to use the differential selection selection and variation components, and extending NSGA-II to adopt an external archive to store the

Table 1. Component catalog.

Component	**Implementations**
Solutions creation	Latin hypercube sampling
	Random
	Scatter Search
Evaluation	Sequential
	Multithreaded
	Sequential with external archive
Termination	Computing time
	Evaluations
	Keyboard
	Quality indicator
Selection	N-ary tournament
	Random
	Differential evolution
	Neighborhood
	Population and neighborhood
Variation	Crossover and mutation
	Differential evolution
Replacement	(μ, λ)
	$(\mu + \lambda)$
	Ranking and density estimator
	Pairwise
	MOEA/D replacement strategy
	SMS-EMOA replacement strategy

found non-dominated solutions only requires to replace the sequential evaluation component by the "sequential with external archive" included in Table 1.

3 Component-Based Evolutionary Algorithms in Julia

In this section, we describe the approach adopted to implement the component-based scheme for multi-objective evolutionary algorithms in Julia. We call `MetaJul`[5] the Julia project we have developed to conduct our study.

```
abstract type Solution end
abstract type Problem{T} end
abstract type Algorithm end

abstract type Operator end
abstract type MutationOperator <: Operator end
abstract type CrossoverOperator <: Operator end
abstract type SelectionOperator <: Operator end

abstract type Component end
abstract type SolutionsCreation <: Component end
abstract type Evaluation <: Component end
abstract type Termination <: Component end
abstract type Selection <: Component end
abstract type Variation <: Component end
abstract type Replacement <: Component end
```

Listing 1.2. Core abstract types in MetaJul.

At a first glance, the main difficulty relies in the fact that `jMetal` is developed on the oriented-oriented features of Java while Julia is not an object-oriented language (as C++ and Java), so a direct translation is not possible. The general adopted approach to implement `jMetal` entities (i.e., operators, components, operators, algorithms, etc.) is to define a Julia struct to store the entity parameters and a set of functions to manipulate it, making use of dynamic dispatching to allow the Julia compiler to choose the right function in case of identical functions applied to different structs.

We start by defining a set of abstract types to represent the core classes of Fig. 1 (*Solution*, *Problem*, *Algorithm*, and *Operator*), which are shown in Listing 1.2. We can observe that the components of a generic evolutionary algorithm are represented also by abstract types.

To define the template for evolutionary algorithms, we define a Julia struct that stores the control parameters and the components with a function that implements the pseudo-code shown in Algorithm 1. A code snippet of the struct is included in Listing 1.3, while the function is shown in Listing 1.4. By using this scheme, the NSGA-II algorithm can be constructed as shown in Listing 1.5, where we can see how the problem and the values of the population and offspring population sizes are indicated as well as the NSGA-II components are assigned. In the current implementation of `MetaJul`, the stopping condition can be alternatively set to run the algorithm for a maximum amount of time, other crossover and mutation operators for continuous problems are uniform mutation

[5] `MetaJul` project: https://github.com/jMetal/MetaJul

and BLX-α crossover, and there is an evaluation component can use a bounded external archive using the crowding distance as density estimator to replace solutions when the archive is full.

```
mutable struct EvolutionaryAlgorithm <: Algorithm
  name::String
  problem::Problem
  populationSize::Int
  offspringPopulationSize::Int

  foundSolutions::Vector

  solutionsCreation::SolutionsCreation
  evaluation::Evaluation
  termination::Termination
  selection::Selection
  variation::Variation
  replacement::Replacement
end
```

Listing 1.3. Struct containing the parameters and components of a generic evolutionary algorithm.

```
function evolutionaryAlgorithm(ea::EvolutionaryAlgorithm)
  population = ea.solutionsCreation.create(ea.solutionsCreation.parameters)
  population = ea.evaluation.evaluate(population, ea.evaluation.parameters)

  while !ea.termination.isMet(ea.termination.parameters)
    matingPool = ea.selection.select(population, ea.selection.parameters)

    offspringPopulation = ea.variation.variate(population, matingPool, ea.
        variation.parameters)
    offspringPopulation = ea.evaluation.evaluate(offspringPopulation, ea.
        evaluation.parameters)

    population = ea.replacement.replace(population, offspringPopulation, ea.
        replacement.parameters)
  end

  return population
end
```

Listing 1.4. Function that performs the steps of an evolutionary algorithm.

```
problem = zdt1Problem()

solver::EvolutionaryAlgorithm = EvolutionaryAlgorithm()
solver.name = "NSGA-II"

solver.problem = problem
solver.populationSize = 100
solver.offspringPopulationSize = 100

solver.solutionsCreation = DefaultSolutionsCreation((problem = solver.problem,
    numberOfSolutionsToCreate = solver.populationSize))

solver.evaluation = SequentialEvaluation((problem = solver.problem, ))

solver.termination = TerminationByEvaluations((numberOfEvaluationsToStop = 25000,
    ))

mutation = PolynomialMutation((probability=1.0/numberOfVariables(problem),
    distributionIndex=20.0, bounds=problem.bounds))

```

```
crossover = SBXCrossover((probability=1.0, distributionIndex=20.0, bounds=problem.
    bounds))

solver.variation = CrossoverAndMutationVariation((offspringPopulationSize = solver
    .offspringPopulationSize, crossover = crossover, mutation = mutation))

solver.selection = BinaryTournamentSelection((matingPoolSize = solver.variation.
    matingPoolSize, comparator = compareRankingAndCrowdingDistance))

solver.replacement = RankingAndDensityEstimatorReplacement((dominanceComparator =
    compareForDominance, ))

foundSolutions = evolutionaryAlgorithm(solver)
```

Listing 1.5. Configuring and running the NSGA-II algorithm in `MetaJul`.

4 Performance Evaluation

One of the potential advantages of Julia over other programming languages such as Java is its computing performance. The aim of this section is to determine if this holds when comparing the NSGA-II versions of `jMetal` and `MetaJul`.

Before conducting the comparative, it is worth mentioning that, although reducing the computing time of the algorithms is an important aspect in `jMetal`, we have not applied aggressive optimization techniques to try to speed up the execution times as much as possible. The same holds for the NSGA-II implementation of `MetaJul`, so this matter must be taken into account when evaluating the two versions of NSGA-II. The target computer is a MacBookPro laptop with Apple M1 Max processor, 35 GB of RAM, and macOS Sonoma 14.1.2. The versions of the Java JDK and Julia are, respectively, Open JDK 19.0.2 and Julia 1.10.0.

For the comparison, we have configured NSGA-II with the following settings:

- Population and offspring population sizes: 100
- Crossover: SBX crossover (probability: 0.9, distribution index: 20.0)
- Mutation: polynomial mutation (probability: 1/L, where L is the number of problem variables, distribution index: 20.0).
- Stopping condition: 100000 function evaluations.

As benchmark problem, we have selected the ZDT1 problem because it is scalable in terms of the number of variables. In the experimentation, we have configured ZDT1 with 30 (default value), 100, 500, 1000, and 2000 variables. The obtained computing times (the median of five independent runs) are reported in Fig. 2.

We observe in the figure that the implementation of NSGA-II in `jMetal` requires less time when ZDT1 has 30 and 100 variables, what indicates that in those scenarios, where the number of function calls and memory management operations are high compared to the scientific computing code, Java is more efficient than Julia. However, from 500 to 2000 variables, the superiority of `MetaJul` over `jMetal` increases as the number of variables does, confirming the clear advantages of Julia when the application becomes computationally intensive.

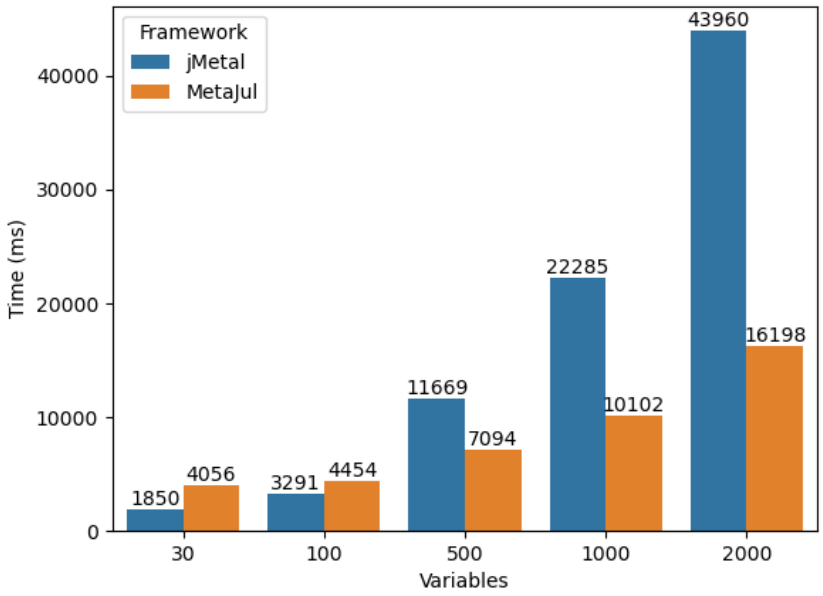

Fig. 2. Comparison of `jMetal` vs `MetaJul` NSGA-II. The problem is ZDT1 with 30, 100, 500, 1000, and 2000 decision variables.

Based on these results, we can assert that the NSGA-II version included in `MetaJul` can clearly outperforms the one included in `jMetal` when solving large-scale optimization problems (i.e., problems having more than 100 decision variables) but, for smaller problems, the `jMetal` version is likely to be faster.

5 Porting `jMetal` Resources to `MetaJul`

After implementing the NSGA-II algorithm to validate the component-based architecture, the next step would be to enrich MetaJul with additional material, which would include not only high-level components, but also operators, benchmark problems and utilities (e.g. quality indicators). We faced these kind of situations during the development of the `jMetal` project, leading us to adapt code from different sources (e.g., implementations in C and MatLab of problems defined in international competitions). The drawbacks of this approach are that it is very time-consuming, prone to errors, and difficult to carry out when we are not familiar with the programming language of the original code.

Although the `MetaJul` code has been written using the mentioned approach, by translating manually `jMetal` code, we explore in this section whether the use of LLMs (Large Language Models [4]) could be of assistance in this context. We

focus first on porting continuous multi-objective problems included in `jMetal` to `MetaJul`. As LLM, we use ChatGPT 4.0.

The `MetaJul` struct for continuous problems is included in Listing 1.6. We can observe that both the objectives and constraints are vector of functions, which are invoked by an evaluate function whenever solution has to be evaluated. A problem is constructed by incorporating the variable bounds, objectives and constraints by calling the *addVariable()*, *addObjective()* and *addConstraint()* functions. To illustrate the definition of a problem, we include in Listing 1.7 a function with the code of the constrained problem Srinivas [6].

```
abstract type AbstractContinuousProblem{T<:Number} <: Problem{T} end

mutable struct ContinuousProblem{T} <: AbstractContinuousProblem{T}
  bounds::Vector{Bounds{T}}
  objectives::Vector{Function}
  constraints::Vector{Function}
  name::String
end

function addObjective(problem::ContinuousProblem{T}, objective::Function) where {T
    <:Number}
  push!(problem.objectives, objective)

  return Nothing
end

function addConstraint(problem::ContinuousProblem{T}, constraint::Function) where
    {T<:Number}
  push!(problem.constraints, constraint)

  return Nothing
end

function addVariable(problem::ContinuousProblem{T}, bounds::Bounds{T}) where {T<:
    Number}
  push!(problem.bounds, bounds)
  return Nothing
end

function evaluate(solution::ContinuousSolution{T}, problem::ContinuousProblem{T})
    ::ContinuousSolution{T} where {T<:Number}
  for i in 1:length(problem.objectives)
    solution.objectives[i] = problem.objectives[i](solution.variables)
  end

  for i in 1:length(problem.constraints)
    solution.constraints[i] = problem.constraints[i](solution.variables)
  end

  return solution
end
```

Listing 1.6. Struct for continuous problems in MetaJul and associated functions.

```
function srinivasProblem()
  problem = ContinuousProblem{Float64}("Srinivas")

  addVariable(problem, Bounds{Float64}(-20.0, 20.0))
  addVariable(problem, Bounds{Float64}(-20.0, 20.0))

  f1 = x ->  2.0 + (x[1] - 2.0) * (x[1] - 2.0) + (x[2] - 1.0) * (x[2] - 1.0)
  f2 = x -> 9.0 * x[1] - (x[2] - 1.0) * (x[2] - 1.0)

```

```
  addObjective(problem, f1)
  addObjective(problem, f2)

  c1 = x -> 1.0 - (x[1] * x[1] + x[2] * x[2]) / 225.0
  c2 = x -> (3.0 * x[2] - x[1]) / 10.0 - 1.0

  addConstraint(problem, c1)
  addConstraint(problem, c2)

  return problem
end
```

Listing 1.7. Function defining problem Srinivas.

Our goal now is to port from `jMetal` to `MetaJul` the ConstrEx and Tanaka constrained continuous problems included in [6]. So we asked the LLM with a question that includes, first, the `jMetal` class that defines the Srinivas problem and how this code is implemented in `MetaJul` (by providing the struct in Listing 1.6 and the *srinivasProblem()* function defined in Listing 1.7); second, we ask for the implementation of a Julia function from the Java code of the ConstrEx problem, which is included next. As a result, we obtain a *constrExProblem()* function (see Listing 1.8) that can be incorporated to `MetaJul` as returned, requiring no modifications. We repeat the same steps the Tanaka problem, with the same result.

```
function constrExProblem()
    problem = ContinuousProblem{Float64}("ConstrEx")

    # Define variable bounds
    addVariable(problem, Bounds{Float64}(0.1, 1.0))
    addVariable(problem, Bounds{Float64}(0.0, 5.0))

    # Objective functions
    f1 = x -> x[1]
    f2 = x -> (1.0 + x[2]) / x[1]

    addObjective(problem, f1)
    addObjective(problem, f2)

    # Constraints
    c1 = x -> x[2] + 9 * x[1] - 6.0
    c2 = x -> -x[2] + 9 * x[1] - 1.0

    addConstraint(problem, c1)
    addConstraint(problem, c2)

    return problem
end
```

Listing 1.8. Function defining problem ConstrEx returned by ChatGPT 4.0.

We can argue the considered problems have a simple formulation and that the original `jMetal` code is very well structured, so the LLM has no difficulties in make the translation. If we turn to complex benchmarks such as WFG [11] and LZ09 [12], which contain a significant amount of auxiliary code, the presented approach cannot be directly adopted.

We have explored applying this approach to port `jMetal` evolutionary operators to `MetaJul`. However, in general, the operators are implemented by using auxiliary classes, so there is no a simple way to indicate to the LLM how a

jMetal operator is implemented in Julia and use it as an example to generate automatically the code of other operators.

6 Discussion

From the analysis carried out in previous sections, we can proceed to answer the three research questions defined in the introduction.

6.1 Research Questions

RQ1 - component-based architecture in Julia: The combination of a Julia struct to store the evolutionary algorithm parameters and components with a function that computes the code of a generic evolutionary algorithm is a simple approach that provides a flexibility which is similar to the equivalent jMetal code. In this sense, MetaJul include examples of using the template to instantiate single-objective evolutionary algorithms for solving continuous and binary optimization problems. The current component catalog of MetaJul is reduced compared to jMetal, but we have not observed any limitation and it would only remain to add more components.

RQ2 - Performance evaluation and comparison: The component-based template for evolutionary algorithms has an impact on the performance of the NSGA-II implementation in MetaJul, which is penalized in comparison with jMetal when it is not intensive in the execution of scientific code. However, when the problem to optimize is large-scale, MetaJul clearly outperforms jMetal.

RQ3 - Reusing jMetal stuff: As an alternative to do manual translations, the porting of jMetal entities to MetaJul assisted by an LLM appears as a feasible choice when most of the original jMetal code is composed of mathematical operations and the equivalent code in Julia can be provided as an example, as it happens with some continuous problems.

6.2 Further Remarks

The goal of including the component-based architecture of multi-objective evolutionary algorithms included in jMetal to MetaJul has been covered successfully, but there is a performance penalty associated to the adopted approach that makes the NSGA-II algorithm in MetaJul slower that the jMetal version in scenarios where the numerical computing code is not dominant. This is a consequence of trying to translate to a Julia project the structure of a Java application, which probably does not takes into account Julia features that could have a positive impact in reducing computing times. In this sense, it should be

tested if the algorithms included in `MetaJul` are easily understandable to be used by Julia users.

Although we have explored the use of LLM to translate entities of `jMetal` to `MetaJul` and some limitations have been identified, two scenarios where LLMs can be helpful are the direct translation of pieces of plain Java code to Julia, particularly those containing mathematical operations, and to optimize Julia code.

As this paper is about experiences using Julia to implement multi-objective evolutionary algorithms coming from the Java-based framework `jMetal`, it is worth mentioning a number of Julia features not provided by Java that have helpful in this context:

- Parametric and non-parametric Statistical tests (library HiphotesisTests.jl). To use the Wilcoxon rank-sum test (Mann-Whitney U test) to compare algorithms, in `jMetal` we had to generate a R script to use it.
- Jupyter notebooks. The availability of notebooks in Julia bring many benefits, including examples of how to configure and run algorithms and visualize the results they produce, what would enhance the documentation the framework.

7 Conclusions

In this paper, we have investigated the issue of implementing multi-objective evolutionary algorithms in Julia from the perspective of the experiences accumulated in the development of the Java-based jMetal framework. We have developed a project called MetaJul for this purpose. Our focus have been the translation to MetaJul of the component-based architecture for multi-objective evolutionary algorithms included in the last release of jMetal.

Our study reveals that the implementation of such an architecture in Julia is feasible and does not entail many complexities. We have validated it by implementing the NSGA-II algorithm and we have conducted a comparative study with the NSGA-II version of jMetal; the comparison has shown, as expected, that Julia excels when the algorithm is intensive in numerical computing.

We have explored also the issue of using LLMs to generate code from jMetal to MetaJul in an automatic way, showing cases (constrained multi-objective problems) where this has been feasible and pointing out the difficulties of translating other kinds of stuff.

On the basis of the experiences gained in this paper, we can define several lines that are worth addressing. Analyzing the MetaJul project to optimize code performance and the incorporation of new component-based metaheuristics, such as MOEA/D, are ongoing lines of work. We have centered our study on continuous optimization; it would be of interest to extend it to combinatorial problems (e.g., multi-objective formulations of knapsack, facility location, or packing).

Acknowledgments

This work has been partially funded by AETHER-UMA (A smart data holistic approach for context-aware data analytics: semantics and context exploitation) project grant, CIN/AEI/10.13039/501100011033/PID2020-112540RB-C41, and by the Junta de Andalucia, Spain, under contract QUAL21 010UMA.

References

1. Beume, N., Naujoks, B., Emmerich, M.: SMS-EMOA: Multiobjective selection based on dominated hypervolume. European Journal of Operational Research **181**(3), 1653–1669 (2007)
2. Bezanson, J., Chen, J., Chung, B., Karpinski, S., Shah, V.B., Vitek, J., Zoubritzky, L.: Julia: dynamism and performance reconciled by design. Proc. ACM Program. Lang. **2**(OOPSLA) (oct 2018). https://doi.org/10.1145/3276490, https://doi.org/10.1145/3276490
3. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017)
4. Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., Ye, W., Zhang, Y., Chang, Y., Yu, P.S., Yang, Q., Xie, X.: A survey on evaluation of large language models. ACM Trans. Intell. Syst. Technol. (jan 2024), just Accepted
5. Coello, C., Lamont, G., van Veldhuizen, D.: Evolutionary Algorithms for Solving Multi-Objective Problems. Genetic and Evolutionary Computation, Springer US (2007)
6. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002)
7. de Dios, J.A.M., Mezura-Montes, E.: Metaheuristics: A julia package for single- and multi-objective optimization. Journal of Open Source Software **7**(78), 4723 (2022)
8. Durillo, J.J., Nebro, A.J.: jMetal: A java framework for multi-objective optimization. Advances in Engineering Software **42**(10), 760–771 (2011)
9. Ehrgott, M., Gandibleux, X.: Approximative solution methods for multiobjective combinatorial optimization. TOP (Spanish journal of operations research) **12**(1), 1–63 (2004)
10. Emmerich, M., Deutz, A.: A tutorial on multiobjective optimization: fundamentals and evolutionary methods. Natural Computing **17**, 585–609 (2018), https://doi.org/10.1007/s11047-018-9685-y
11. Huband, S., Hingston, P., Barone, L., While, L.: A review of multi-objective test problems and a scalable test problem toolkit. IEEE Transactions on Evolutionary Computation **10**(5), 477–506 (October 2006)
12. Li, H., Zhang, Q.: Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii. Trans. Evol. Comp **13**(2), 284–302 (apr 2009)
13. Liu, Q., Li, X., Liu, H., Guo, Z.: Multi-objective metaheuristics for discrete optimization problems: A review of the state-of-the-art. Applied Soft Computing **93**, 106382 (2020), https://doi.org/10.1016/j.asoc.2020.106382
14. Nebro, A.J., Durillo, J.J., Vergne, M.: Redesigning the jMetal multi-objective optimization framework. Genetic and Evolutionary Computation Conference pp. 1093–1100 (7 2015)

15. Nebro, A.J., López-Ibáñez, M., Barba-González, C., García-Nieto, J.: Automatic configuration of NSGA-II with jMetal and irace. Genetic and Evolutionary Computation Conference pp. 1374–1381 (7 2019)
16. Nebro, A.J., López-Ibáñez, M., García-Nieto, J., Coello, C.A.C.: On the automatic design of multi-objective particle swarm optimizers: experimentation and analysis. Swarm Intelligence (2023)
17. Zhang, Q., Li, H.: MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. IEEE Transactions on Evolutionary Computation **11**(6), 712–731 (2007)