Design and Architecture of the jMetalSP Framework

Antonio J. Nebro
University of Málaga
E.T.S. de Ingeniería Informática,
Campus de Teatinos
Málaga, Spain 29071
antonio@lcc.uma.es

Cristóbal Barba-González
University of Málaga
E.T.S. de Ingeniería Informática,
Campus de Teatinos
Málaga, Spain 29071
cbarba@lcc.uma.es

José García Nieto
University of Málaga
E.T.S. de Ingeniería Informática,
Campus de Teatinos
Málaga, Spain 29071
jnieto@lcc.uma.es

José A. Cordero
European Organization for Nuclear
Research (CERN)
CERN 31/3-028
Geneve, Switzerland 1211
j.cordero@cern.ch

José F. Aldana Montes
University of Málaga
E.T.S. de Ingeniería Informática,
Campus de Teatinos
Málaga, Spain 29071
jfam@lcc.uma.es

ABSTRACT

jMetalSP is a framework for dynamic multi-objective Big Data optimization. It combines the jMetal multi-objective framework with the Apache Spark cluster computing system to allow the solving of dynamic optimization problems from a number of external streaming data sources in Big Data contexts. In this paper, we describe the current status of the jMetalSP project, focusing mainly in its design and internal architecture, with the aim of offering a comprehensive view of its main features to interested researchers. Among the covered features, we describe the main components of a jMetalSP application, including dynamic problems, dynamic algorithms, streaming data sources, and data consumers. For practical purposes, we describe two test cases to illustrate how to address dynamic combinatorial and dynamic continuous optimization problems by using the proposed framework.

CCS CONCEPTS

•Software and its engineering → Software libraries and repositories; •Computing methodologies → *Heuristic function construction;*

KEYWORDS

Optimization Framework; Dynamic Multi-Objective Optimization; Multi-Objective Metaheurstics; Open Source

ACM Reference format:
Antonio J. Nebro, Cristóbal Barba-González, José García Nieto, José A. Cordero, and José F. Aldana Montes. 2017. Design and Architecture of the jMetalSP Framework. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* **??** pages.
DOI: http://dx.doi.org/10.1145/3067695.3082466

GECCO '17 Companion, Berlin, Germany

DOI: http://dx.doi.org/10.1145/3067695.3082466

1 INTRODUCTION

The optimization of problems composed of two or more conflicting objectives has received a lot of attention since the beginning of year 2000. During this time, solving these multi-objective optimization problems with metaheuristic algorithms has become a very popular approach, leading to a large amount of research in the field. As a consequence, a plethora of algorithms and techniques have been proposed. Most of these research have been focused in static problems, in the sense that they do not change during the optimization process. However, less attention has been paid to solve dynamic multi-objective optimization problems [**?**], even though it is still an up-to-date challenge.

Currently, the growing interest in Big Data applications [**?**], where many of them require to process large amounts of data coming at great speed from different streaming data sources, brings new opportunities to apply dynamic multi-objective optimization. The rationale is that both, Big Data and multi-objective optimization are found in many disciplines, such as transportation, economics, mobility, and medicine; so it is foreseeable that they converge in a near future leading to multi-objective Big Data optimization problems.

Our experience with the jMetal multi-objective optimization framework [**?**][**?**] and the Apache Spark cluster computing system [**?**] has motivated us to start the jMetalSP project from the starting idea of combining these two tools. jMetal is a widely used software in the field of multi-objective optimization and Spark is becoming one of the dominant technologies in Big Data, so using them in jMetalSP results in a framework that combines the features of the former (flexible and extensible architecture, lot of representative multi-objective metaheuristics and problems) and the latter (streaming processing, high level parallel model). Therefore, this approach allows to develop applications that can run on Hadoop [**?**], the *de facto* standard Big Data platform. jMetalSP is an open source that is hosted in GitHub[1].

In this paper, our goal is to describe the design and architecture of the current development version of jMetalSP. We focus in presenting the main components of the framework and the facilities it provides. As in jMetal, an important goal is to try to simplify as

[1] jMetalSP project site: https://github.com/jMetal/jMetalSP

much as possible the development of applications. In particular, issues such as adding streaming data sources and connecting them to algorithms that solve dynamic problems, must be done in a clean way. This is achieved by using an object-oriented architecture and an implementation based on solid software engineering principles. After describing jMetalSP, we illustrate how it can be used by means of two tests cases involving combinatorial and continuous problem representations, with dynamic versions of NSGA-II [**?**], SMPSO [**?**] and MOCell [**?**] algorithms.

The rest of the paper is structured as follows. The next section presents the architecture of jMetalSP describing software components and dynamic methods. In Section **??**, details a real-world test case for validation based on a dynamic combinatorial problem. Section **??** describes a benchmarking test case based on a dynamic continuous optimization problem. The current status of the project is commented in Section **??**. Finally, Section **??** outlines some concluding remarks and suggest the future work.

2 ARQUITECTURE OF JMETALSP

jMetalSP is designed according to an object-oriented architecture, which is depicted in Fig. **??**. It is developed on top of jMetal, so all the components of this framework (algorithms, problems, encodings, operators, quality indicators, etc.) are available. jMetalSP is implemented in the Java programming language.

The current development version of jMetalSP (1.1-SNAPSHOT) relies on the Observer pattern [**?**]. On the one hand, there are a number of *StreamingDataSource* elements (observables), each of them capable of receiving data continuously from external sources and analyze them, which can lead to updates in the *DynamicProblem* that is being optimized (observer). On the other hand, a *DynamicAlgorithm* (observable) is continuously optimizing the problem and generating results (e.g., Pareto front approximations) that, when are produced, are notified to a number of *AlgorithmDataConsumer* entities (observers). The interfaces that observers and observables have to implement are shown in Figure **??**.

The observable components produce instances of *ObservedData* subclasses, which are sent to the observers in the notification messages. They constitute key components because they determine which observers can be bound to an observable.

The *StreamingRuntime* class encapsulates the underlying streaming engine, which currently can be Spark or plain Java (based on threads).

All the jMetaSP applications share the code template depicted in Figure **??**. We can observe how different data consumers and streaming data sources can be incorporated into an application. All the classes and interfaces on jMetalSP are generic, which means they are parametrized over types, then assuring that all the classes are compatible before the execution (during compilation time).

The main architecture components are described in the next subsections.

2.1 Dynamic Problems

Dynamic multi-objective optimization problems are characterized by the fact that their objectives or their search space can vary over time, which may affect their Pareto set, their Pareto front or both of them [**?**]. In the context of jMetalSP, changes in the problems will be originated by the results of the processing and analysis of one or more streaming data sources.

As we can see in Figure **??**, the *DynamicProblem* class inherits from jMetal's *Problem* class, so it contains two basic methods: *evaluate()* and *evaluateConstraints()*. Both methods receive a *Solution*; the first method evaluates it, and the second one determines the overall constraint violation degree.

Additionally, the *DynamicProblem* class has its own methods:

- *isTheProblemModified()*. Indicates whether the data problem has been modified or not.
- *reset()*. Resets the state of the problem to unmodified.

All these methods (including *evaluate()* and *evaluateConstraints()*) must be tagged as synchronized to ensure mutual exclusion between the clients of the problem, i.e., the algorithm and the streaming data sources.

As dynamic problems implement the *Observer* interface, they must define the *update()* method and they have to register themselves into the streaming data sources they want to observe.

2.2 Dynamic Algorithms

A dynamic algorithm in jMetalSP is a conventional metaheuristic that should consider two main issues: first, the problem can change during the algorithm execution, so the state of the problem should be checked somehow and, in case of detecting a change, a re-starting procedure must be applied; second, when the stopping condition is reached, the algorithm, instead of just terminating, starts again. As can be seen in Figure **??**, a dynamic algorithm has to implement a *restart()* method.

jMetalSP is based on jMetal 5 [**?**], which includes, among other features, algorithm templates. For example, there is an *AbstractEvolutionaryAlgorithm* class that contains the *run()* method shown in Figure **??**, which mimics closely the pseudo-code of a generic evolutionary algorithm (similar templates are available for particle swarm optimization and scatter search algorithms). An advantage of using this template is that those algorithms implementing it (most of evolutionary algorithms in jMetal use it) can be easily extended by overriding only some methods. This is particularly useful to develop dynamic versions of existing algorithms. In this case, at least the *isStoppingConditionReached()* method should be redefined, because instead of stopping, the algorithm should start again.

A dynamic problem is considered as an observable entity, so when a new Pareto front approximation is produced, it is notified to the registered *AlgorithmDataConsumer* observer objects. In our version of dynamic NSGA-II, the number of produced fronts are also provided to its observers (see Section **??**).

2.3 Streaming Data Sources

The role of a streaming data source in jMetalSP is twofold: it must capture the new incoming data, which will be then analyzed. The results of this analysis may produce an instance of the *ObservedData* class to be notified to a registered observed (i.e, a dynamic problem). This is particularly interesting in the case of using Apache Spark, because its streaming features allows to make the analysis in parallel, taking advantage of Hadoop clusters.

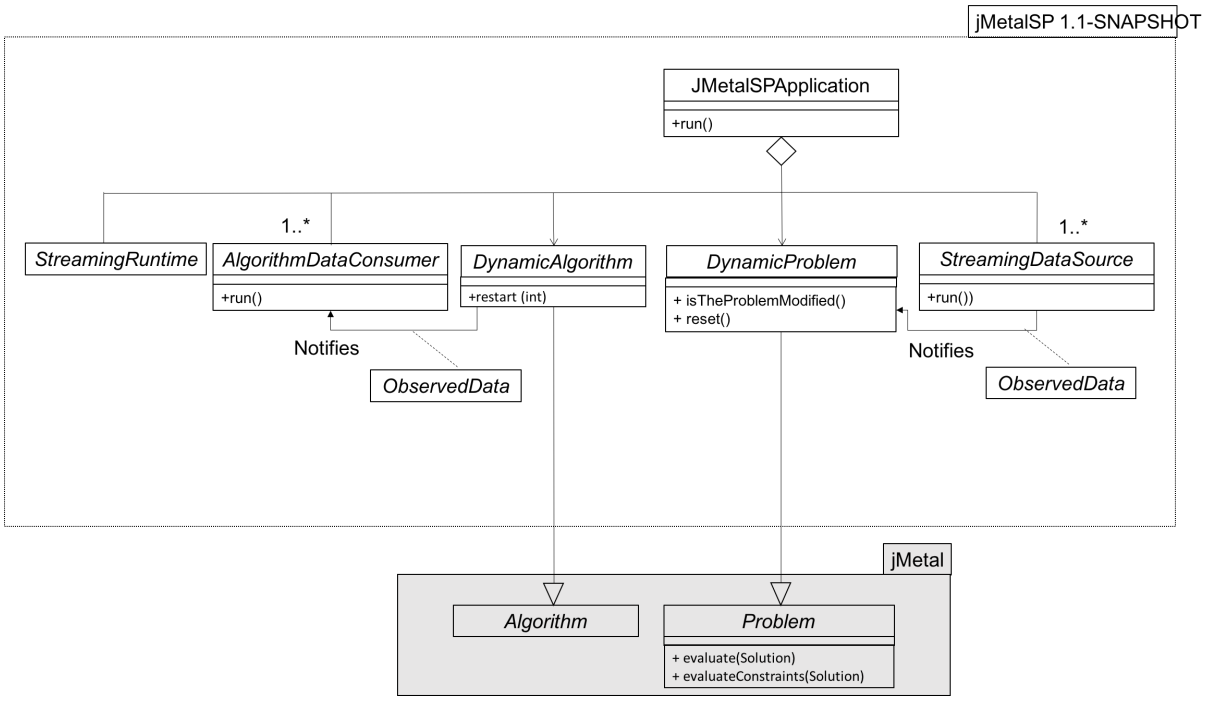

Figure 1: Overall class diagram of the jMetalSP architecture.

```
public interface Observer {
  void update(Observable<?> observable, Object data);
}

public interface Observable<Data> {
  void register(Observer observer);
  void unregister(Observer observer);

  void notifyObservers(Data data);
  int numberOfRegisteredObservers();
  void setChanged();
  boolean hasChanged();
  void clearChanged();
  String getName();
}
```

Figure 2: *Observer* and *Observable* interfaces.

The *StreamingDataSource* interface contains only a *run()* method. In the default plain Java implementation, a new thread is started and the *run()* method is invoked. An example is included in Figure **??**, which shows the code of a simple streaming data source that continuously produces the value of a counter. The observers are notified by the value of the counter after a delay (no analysis is carried out here).

In the case of using Spark, we assume that an external process is generating the counter values and writes them in files that are stored in a directory. A Spark class named *SimpleSparkStreamingCounterDataSource* that reads the files of that directory in a streaming fashion is included in Figure **??**. We can observe that the code is composed of two steps: assuming that each file contains a line with the generated value, the first sentence of the *run()* method

```
public class JMetalSPApplicationTemplate {
  public static void main(String[] args) {
    JMetalSPApplication<
      ObservedDataFromStreamingSources,
      ObservedDataFromAlgorithm,
      DynamicProblem,
      DynamicAlgorithm,
      StreamingDataSource,
      AlgorithmDataConsumer> application;

    application = new JMetalSPApplication<>();

    application
            .setStreamingRuntime(new SparkRuntime())
            .setProblem(new DynamicProblem())
            .setAlgorithm(new DynamicAlgorithm())
            .addStreamingDataSource(new StreamingDataSource1())
            .addStreamingDataSource(new StreamingDataSource2())
            .addAlgorithmDataConsumer(new DataConsumer1())
            .addAlgorithmDataConsumer(new DataConsumer2())
            .run();
  }
}
```

Figure 3: Template of a jMetalSP application.

reads all the lines in the files in the directory and transforms them into integer values; then, in the second step, the observers are notified. Compared with the former example, we can see here that there is no an implicit loop because the Spark streaming engine is executing theses sentences iteratively. The same engine takes care of reading only the new files stored in the directory since the last iteration. This two-step scheme is the same for all the streaming data sources supported by Spark (socket, directory, Kafka, etc.).

```
@Override public void run() {
  List<S> offspringPop;
  List<S> matingPop;
    population = createInitialPopulation();
    population = evaluatePopulation(population);
    initProgress();
    while (!isStoppingConditionReached()) {
        matingPop = selection(population);
        offspringPop = reproduction(matingPop);
        offspringPop = evaluatePop(offspringPop);
        population =  replacement(population, offspringPop);
        updateProgress();
  }
}
```

Figure 4: *run()* method of class *AbstractEvolutionaryAlgorithm*.

```
public class SimpleStreamingCounterDataSource {

  @Override
  public void run() {
    int counter = 0 ;
    while (true)
      Thread.sleep(DELAY);

      observedData.setChanged();
      observedData.notifyObservers(new SimpleObservedData(counter));
      counter ++ ;
    }
  }
}
```

Figure 5: Example of a simple streaming counter data source (plain Java).

```
public class SimpleSparkStreamingCounterDataSource {

  @Override
  public void run() {
    JavaDStream<Integer> values = streamingContext
      .textFileStream(directoryName)
      .map(line -> Integer.parseInt(line)) ;

    values.foreachRDD(numbers -> {
      List<Integer> numberList = numbers.collect() ;
      for (Integer value : numberList) {
        updateData.setChanged();
        updateData.notifyObservers(new SimpleObservedData(value));
      }
    }) ;
  }
}
```

Figure 6: Example of a simple streaming counter data source (Spark).

The issue to note here is that the processing of the *map()* function can be executed in parallel in a cluster, which would be advantageous if there are many lines to process and their analysis are complex procedures. In the case of complex data, their analysis would be carried out with Spark operations as this *map()* and many others, including filtering, sampling, etc.

2.4 Observed Data

This class is intended to represent the type of data the observable entities produce, so they determine which observers can register in a given observable.

The data produced by the streaming data sources can be very varied, while in the case of the algorithms we provide a concrete class called *AlgorithmObservedData*, which is used to bound algorithms and data consumers. This class contains the Pareto front approximation obtained and the value of a generated fronts counter, but it can be easily extended to include more algorithm's related data (e.g., the computing time of the last execution, the value of a quality indicator, etc.).

2.5 Algorithm Data Consumers

A dynamic algorithm is supposed to run forever to produce at least Pareto front approximations periodically, so any component interested in getting those fronts cannot wait for the completion of the algorithm, as in the case of techniques solving static problems.

Algorithm data consumers register into algorithms to be notified from the latter when new information (ie., an *AlgorithmObservedData*, as commented in the previous section) is generated. jMetalSP includes two consumer components: one that stores the fronts into a directory and another one that prints information about the fronts (number of generated fronts, number of solutions of the last front).

2.6 Streaming Runtime

The last component in the jMetalSP architecture is *StreamingRuntime*, which represents the underlying streaming system. Two classes implementing this interface are included:

- A default plain-Java based runtime (Spark is not required), which starts each streaming data source in a dedicated thread.
- An Spark-based runtime, which sets the parameters of Spark and initialize the so-called streaming context. The streaming model of Spark is based on micro-batches, so the runtime receives the batch interval (see the Spark streaming programming guide for further information[2]) as a parameter.

In former versions of jMetalSP only a Spark-based runtime were available, although we have now uncoupled it and generalized the streaming runtime into an interface (i. e., applying the dependence inversion principle). There are three main reasons to adopt this approach. First, Spark has a new experimental streaming implementation, called structured streaming, that currently is in Alpha, so we would like to easily change from the current one to the new one when available; second, some users could be interested in using jMetalSP for dynamic optimization but without Spark, so an only-Java version would be easier to them; finally, there are other streaming engines, such as Apache Flink[3], that could be incorporated to jMetalSP in a future release.

[2] http://spark.apache.org/docs/latest/streaming-programming-guide.html
[3] Apache Flink Web Site: https://flink.apache.org

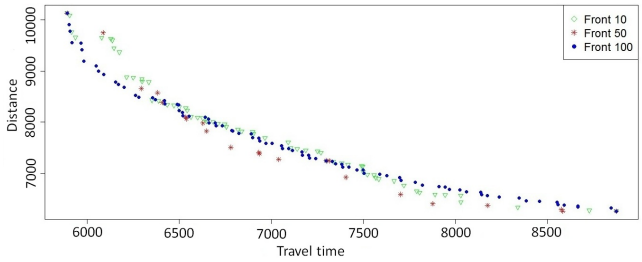

Figure 7: Pareto front approximations obtained when solving the dynamic bi-objective TSP of New York with a dynamic NSGA-II algorithm.

3 TEST CASE: DYNAMIC COMBINATORIAL PROBLEM

In this section, we describe a jMetalSP test case comprising a dynamic combinatorial optimization problem. The optimization algorithm used is NSGA-II and the target problem is a bi-objective TSP (Traveling Salesman Problem). This test case was developed in [**?**], and was motivated by the availability of real-time Traffic speed open data of the city of New York[4].

The dynamic TSP tackled here has two objectives: minimizing the total distance and minimizing the travel time, so it contains two data matrices: distance and travel time. The encoding is a permutation of integer values to represent the tours, which will be manipulated with a swap mutation operator and a partial-mapped (PMX) crossover operators. The update of data consists on a code to indicate the distance or the time matrix, the coordinates (row and column) to change, and the new value.

We have considered three different streaming data sources. The first one is a directory where an external process will be writing periodically files with data obtained from the traffic open data Web service. The traffic data is updated with a frequency of two/three times per minute, so it cannot be considered as pure real-time stream. For these reason, and to transform this problem into a Big Data optimization one, the other two data sources have been included. On the one hand, Apache Kafka [**?**] is used to simulate traffic data obtained from GPS sensors of the cars and, on the other hand, Twitter is queried to get tweets about traffic in New York.

As the purpose of this work is to validate the jMetalSP architecture, we are not particularly interested in the quality of the results. For this reason, the analyses of the Kafka and Twitter data sources are replaced by an idle loop that can be configured to consume a given amount of CPU time, and the update data are randomly produced. This way, we can adjust the analysis processing time and assess the performance of the jMetalSP application in different contexts.

To implement a dynamic version of NSGA-II, a *DynamicNSGAII* class extending the jMetal class implementing this algorithm was developed. This class has the following differences from the original algorithm:

- It is an observable entity.
- The *isStoppingConditionReached()* method pushes the current population to registered observers and carries out a restart operation after a full iteration. We have used simple restart strategy consisting in generating a new full population filled with random solutions, but more complex strategies can be applied.
- The *updateProgress()* method checks whether the problem has been changed; if so, a restart is done.

The rest of the NSGA-II code remains unchanged, which is an advantage of the object-oriented architectures of both, jMetal and jMetalSP.

The last components of the application are the two data consumers commented in Section **??**. Figure **??** shows the Pareto front approximations generated by NSGA-II throughout the optimization process after 10, 50, and 100 operations of problem data update. We can observe that the shape of the fronts vary with time, which is the expected behavior when solving a dynamic problem. The changes in the fronts are slight, because unless an important event occurs (e.g., a car accident producing a traffic jam), usually there are not drastic variations in the traffic during consecutive periods of time.

4 TEST CASE: DYNAMIC CONTINUOUS PROBLEM

This second test case is not intended to be an example of Big Data optimization application, but to show that jMetalSP can be a useful tool for solving conventional dynamic problems. Concretely, we focus here in dynamic continuous optimization, so we have selected SMPSO [**?**], a multi-objective particle swarm optimization (PSO) algorithm, to develop a dynamic version of it to solve the FDA benchmark [**?**].

As commented in Section **??**, jMetal provides a number of generic metaheuristic templates, including one for PSO algorithms, which is followed by SMPSO. The *run()* method of this template is included in Figure **??**. We can observe that this method incorporates the initialization of the PSO data (swarm, particles velocity, particles memory, leaders) and the main loop performs the steps to update those data.

As in the case of NSGA-II, the dynamic version of SMPSO is implemented by following a similar strategy: the original algorithm is an observable entity and the *isStoppingConditionReached()* and *updateProgress()* methods are redefined in a similar way.

The FDA benchmark [**?**] consists in five dynamic problems with different features depending on whether their Pareto-optimal front (POF) and/or Pareto-optimal solutions (POS) change over time:

- FDA1 has a constant convex POF and linear change in POS.
- FDA2's POF changes from convex to non-convex and its POS does not change.
- FDA3's POF changes but all convex and also linear change in POS.
- FDA4 has a constant non-convex POF and linear change in POS, which has a three dimensional space.
- FDA5's POF changes but all convex and linear change in POS, which is also a three dimensional.

[4]https://data.cityofnewyork.us/Transportation/Real-Time-Traffic-Speed-Data/xsat-x5sa

```
@Override
public void run() {
  swarm = createInitialSwarm();
  swarm = evaluateSwarm(swarm);
  initializeVelocity(swarm);
  initializeParticlesMemory(swarm);
  initializeLeader(swarm);
  initProgress();

  while (!isStoppingConditionReached()) {
    updateVelocity(swarm);
    updatePosition(swarm);
    perturbation(swarm);
    swarm = evaluateSwarm(swarm);
    updateLeaders(swarm);
    updateParticlesMemory(swarm);
    updateProgress();
  }
}
```

Figure 8: *run()* **method of class** *AbstractParticleSwarmOptimization.*

Table 1: Parameter settings and operators used for the dynamic SMPSO, dynamic NSGA-II and dynamic MOCell (L = Individual length).

Parameterization used in dynamic SMPSO [?]	
Archive Size	100 particles
C_1, C_2	1.5
w	0.9
Mutation	polynomial
Mutation probability	$1/L$
Mutation distribution index η_m	20
Selection method	Rounds
Maximum number of evaluations	50,000
Parameterization used in dynamic NSGA-II [?]	
Population Size	100 individuals
Selection of Parents	binary tournament + binary tournament
Recombination	simulated binary
Recombination probability	0.9
Mutation	polynomial,
Mutation probability	$1/L$
Maximum number of evaluations	50,000
Parameterization used in dynamic MOCell [?]	
Population Size	100 individuals
Neighborhood	1-hop neighbours (8 surrounding solutions)
Selection of Parents	binary tournament + binary tournament
Recombination	simulated binary
Recombination probability	0.9
Mutation	polynomial
Mutation probability	$1/L$
Mutation distribution index η_m	20
Archive Size	100 individuals
Maximum number of evaluations	50,000

Each FDA problem defines its own multi-objective functions, although all of them have a time dependence. They define the time in the same way, according to Equation **??**:

$$t = \frac{1}{n_t} \left\lfloor \frac{\tau}{\tau_T} \right\rfloor \tag{1}$$

where τ is the generation counter, τ_T is the number of generation for which t remains fixed, and n_t is the number of distinct steps in t. The authors of [?] recommend $\tau_T = 5$ and $n_T = 10$ values, and 31 variables with $x_I = x_1$ $|x_{II}| = |x_{III}| = 15$.

As this case study is intended to illustrate the use of jMetalSP to solve a benchmark of dynamic continuous optimization problems, we include two different approaches in the framework to manage time that follow the examples of streaming data sources commented in Figures **??** and **??**. In the case of the latter, an external process

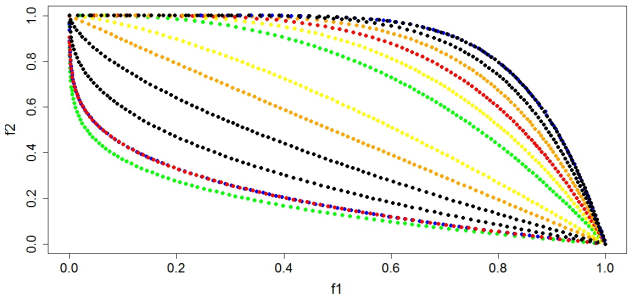

Figure 9: Pareto front for FDA2 obtained by the SMPSO algorithm.

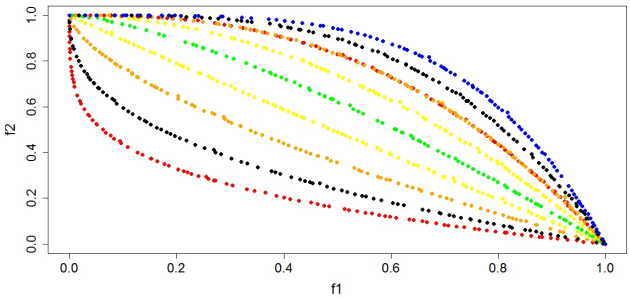

Figure 10: Pareto front for FDA2 obtained by the NSGA-II algorithm.

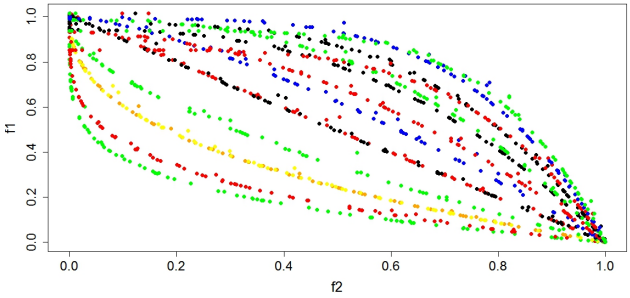

Figure 11: Pareto front for FDA2 obtained by the MOCell algorithm.

is executed in parallel to produce the files containing the data, as mentioned in Section **??**.

To illustrate the behavior of the dynamic SMPSO, we show in Figure **??** the different front approximations obtained when solving the FDA2 problem for 20 time steps. We can observe how the shapes of the Pareto front approximations change over time due to the updating counter (τ). In the FDA2 problem, the Pareto front swings from a convex to a non-convex shape.

Although the objective of this paper is not to carry out a rigorous comparative study of dynamic multi-objective metaheuristics, we have considered interesting to solve the FDA2 problem also with the dynamic NSGA-II algorithm and with another algorithm. Concretely, we have developed a dynamic version of MOCell [**?**], a multi-objective cellular evolutionary algorithm. The parameter settings of these algorithms and SMPSO are included in Table **??**.

The fronts obtained by NSGA-II and MOCell are depicted in Figures **??** and **??**, respectively. A visual comparison with Figure **??** shows that SMPSO clearly outperforms the other two algorithms concerning the diversity of the obtained Pareto front approximations; with regards to convergence, the results are similar in the three cases.

5 CURRENT STATUS AND IMPLEMENTATION DETAILS

jMetalSP is a project that is in continuous development. In its current status it is fully usable, although it is sure that new features will be added in the near feature and some changes in the architecture will be foreseeable from the experiences we gain when using it and from the feedback of interested users.

The version described in this paper (1-1.SNAPSHOT) has been developed with the following tools:

- Java JDK 1.8.0_101.
- Spark 2.0.
- Maven 3.3.9.
- jMetal 5.2.

The project is structured in nine Maven sub-modules as shown in Figure **??**. The *jmetal-core* module contains the interfaces and classes of the architecture, the *Observer* and *Observable* interfaces, and two classes providing default implementations of the runtime and observable (plain Java) interfaces. More information about the jMetalSP internals can be found in the project GitHub page in https://github.com/jMetal/jMetalSP.

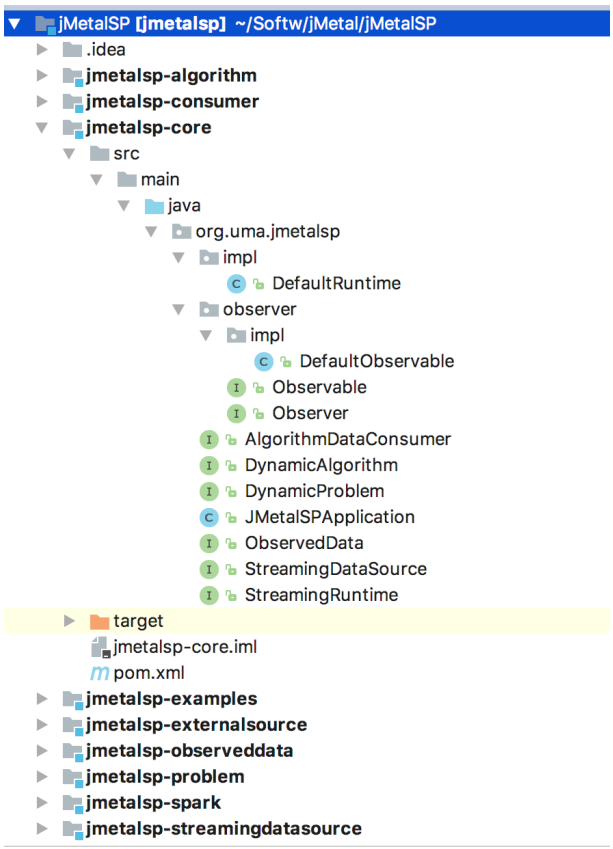

Figure 12: Structure of the jMetalSP project.

6 CONCLUSIONS

We have described the architecture of the jMetalSP framework for dynamic multi-objective optimization with metaheuristics, which is based on combining jMetal and Apache Spark. Concretely, we have presented the main components constituting jMetalSP, including the problems, algorithms, streaming data sources, data consumers, observed data, and streaming runtime.

To illustrate the features of jMetalSP we have detailed two test studies: The first one involves an NSGA-II to solve a combinatorial problem from the field of transportation consisting in a bi-objective TSP that incorporates real as well as simulated traffic data sources; The second test study is about solving dynamic continuous benchmarking problems with a multi-objective PSO (SMPSO), NSGA-II, and a multi-objective cellular evolutionary algorithm (MOCell). These two test studies are used as practical examples to show that jMetalSP is easy to use and adapt, and it incorporates reliable algorithmic implementations able to obtain successful results in different problem domains and representations.

As lines of future work, we plan to incorporate multiobjective algorithms specifically designed to solve dynamic problems and the application of jMetalSP to deal with real-world problems. We also plan to enrich the framework with advanced restarting strategies to deal with dynamic problem updates. From the development point of view, including unit and integration tests is also a pending work.

ACKNOWLEDGMENTS

This work has been partially funded by Grants TIN2014-58304-R (Spanish Ministry of Education and Science) and P11-TIC-7529 (Innovation, Science and Enterprise Ministry of the regional government of the Junta de Andalucía) and P12-TIC-1519 (Plan Andaluz de Investigación, Desarrollo e Innovación). Cristóbal Barba-González is supported by Grant BES-2015-072209 (Spanish Ministry of Economy and Competitiveness). José García-Nieto is the recipient of a Post-Doctoral fellowship of "Captación de Talento para la Investigación" Plan Propio at Universidad de Málaga.

REFERENCES

[] J.A. Cordero, A.J. Nebro, J.J. Durillo, J. García-Nieto, C. Barba-González, I. Navas, and J.F. Aldana-Montes. 2016. Dynamic Multi-Objective Optimization with jMetal and Spark: a Case Study. In *Machine Learning, Optimization, and Big Data: Second International Workshop, MOD 2016, Volterra, Italy, August 26-29, 2016, Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 10122.

[] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

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

[] M. Farina, K. Deb, and P. Amato. 2004. Dynamic multiobjective optimization problems: test cases, approximations, and applications. *IEEE Trans. on Evol. Comp.* 8, 5 (Oct 2004), 425–442.

[] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[] J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece.*

[] A.J. Nebro, J.J. Durillo, F. Luna amd B., Dorronsoro, and E. Alba. 2009. MOCell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems* 24, 7 (2009), 726–746. DOI: http://dx.doi.org/10.1002/int.20358

[] A.J. Nebro, J.J. Durillo, J. García-Nieto, C.A. Coello Coello, F. Luna, and E. Alba. 2009. SMPSO: A New PSO-based Metaheuristic for Multi-objective Optimization. In *IEEE Symposium on Computational Intelligence in Multicriteria Decision-Making (MCDM 2009).* IEEE Press, 66–73.

[] A.J. Nebro, Juan J. Durillo, and M. 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 (GECCO Companion '15).* ACM, New York, NY, USA, 1093–1100.

[] Antonio J. Nebro, Juan J. Durillo, Francisco Luna, Bernabé Dorronsoro, and Enrique Alba. 2007. *Design Issues in a Multiobjective Cellular Genetic Algorithm.* Springer Berlin Heidelberg, Berlin, Heidelberg, 126–140. DOI: http://dx.doi.org/10.1007/978-3-540-70928-2_13

[] T. White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10).* USENIX Association, Berkeley, CA, USA, 10–10.

[] Z. H. Zhou, N. V. Chawla, Y. Jin, and G. J. Williams. 2014. Big Data Opportunities and Challenges: Discussions from Data Analytics Perspectives [Discussion Forum]. *IEEE Computational Intelligence Magazine* 9, 4 (Nov 2014), 62–74.