Hindawi Publishing Corporation
Applied Computational Intelligence and Soft Computing
Volume 2015, Article ID 302172, 12 pages
http://dx.doi.org/10.1155/2015/302172

Research Article

A Software Tool for Assisting Experimentation in Dynamic Environments

Pavel Novoa-Hernández,[1] Carlos Cruz Corona,[2] and David A. Pelta[2]

[1]*Department of Mathematics, University of Holguín, Avenue XX Aniversario S/N, 80100 Holguin, Cuba*
[2]*Department of Computer Science and Artificial Intelligence, Center for Research in Information and Communication Technologies (CITIC-UGR), University of Granada, Periodista Daniel Saucedo Aranda S/N, 18071 Granada, Spain*

Correspondence should be addressed to Pavel Novoa-Hernández; pavel@decsai.ugr.es

Received 13 January 2015; Revised 26 March 2015; Accepted 3 April 2015

Academic Editor: T. Warren Liao

Copyright © 2015 Pavel Novoa-Hernández et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In real world, many optimization problems are dynamic, which means that their model elements vary with time. These problems have received increasing attention over time, especially from the viewpoint of metaheuristics methods. In this context, experimentation is a crucial task because of the stochastic nature of both algorithms and problems. Currently, there are several technologies whose methods, problems, and performance measures can be implemented. However, in most of them, certain features that make the experimentation process easy are not present. Examples of such features are the statistical analysis of the results and a graphical user interface (GUI) that allows an easy management of the experimentation process. Bearing in mind these limitations, in the present work, we present DynOptLab, a software tool for experimental analysis in dynamic environments. DynOptLab has two main components: (1) an object-oriented framework to facilitate the implementation of new proposals and (2) a graphical user interface for the experiment management and the statistical analysis of the results. With the aim of verifying the benefits of DynOptLab's main features, a typical case study on experimentation in dynamic environments was carried out.

1. Introduction

Several decision-making scenarios can be modeled as optimization problems. Among them, a special class is the known dynamic optimization problems (DOPs), which are characterized by the presence of certain time-varying elements of the mathematical model (e.g., objective function and search space). Because of the complexity involved in these problems, the application of metaheuristics methods has gained an increasing interest in the last decade [1–3].

In this context, as in other similar fields, experimentation plays an important role if one takes into account the stochastic nature of metaheuristics and DOPs. In fact, most of the existing results reported about experimentation in dynamic environments rely on metaheuristics solving artificial DOPs [4–8]. However, experimentation in dynamic environments is a hard task. Generally, it requires not only a solid knowledge of computer programming but also an effective control of the experiment factors. Thus, such work can be tedious and error prone due to the large number of parameters to control by the researcher.

Although there are several technologies that help in the implementation of algorithms, problems, and performance measures, generally they require a great effort by the researcher in regard to simulation of the experiments and the processing of the results. This latter aspect is frequently carried out through descriptive and inference statistics. In the context of dynamic environments, there is no evidence of technology that fulfills those requirements at the same time.

Bearing in mind these limitations, in this work, we propose DynOptLab, a free noncommercial tool for experimental analysis in dynamic environments. DynOptLab is composed of two main elements: (1) an object-oriented framework for implementing algorithms, problems, and performance measures and (2) a graphical user interface (GUI) for the management of the computational experiments. In particular, DynOptLab's GUI also includes a module to perform statistical analysis of the results.

In order to better describe our proposal, the rest of the paper is organized as follows. Section 2 gives the necessary background on experimentation in dynamic environments. Section 3 describes the proposed tool through its main components and features. Further, with the aim of verifying the benefits of our proposal, a typical case study on experimentation in dynamic environments is presented in Section 4. Finally, Section 5 outlines the conclusion and future works.

2. Experimental Analysis in Dynamic Environments

A dynamic optimization problem is formally defined as follows. Being $\Omega^{(t)} \subseteq \mathbb{R}^n$ the search space, the goal is

$$\min f^{(t)}(x), \tag{1}$$

where $x \in \Omega$ and t represents the time. In experimentation, it is usual to assume t as a discrete magnitude ($t \in \mathbb{N}$), being associated to the objective function evaluations. As the cautious reader can observe, each element of the model is marked by t, meaning that in every time t a different objective function or search space could appear. So, the goal of an algorithm in solving DOPs is to find the best solution as fast as possible, before the arrival of a new change.

There are different criteria to measure the algorithm performance in dynamic environments. Most of the existing measures are based on the absolute error, in terms of the function values, of the best solution attained by the algorithm and the current optimum of the problem. For example, [9] proposed the offline error, which is the average of the absolute error during the run. Alternatively, [10] employed the same measure but considered only the previous time instant before the change. Other studies, such as [11, 12], have proposed measures based on different aspects of the algorithm (e.g., adaptability, distance to the optimum, and stability).

Similar to performance measures, in literature, there are several artificial DOPs, which are crucial in the study and comparison of the algorithms in dynamic environments. In this context, two popular problem generators are the problem generators Moving Peaks Benchmark (MPB) [9] and the Generalized Dynamic Benchmark Generator (GDBG) [10, 13]. Those generators allow for obtaining multiple problem instances according to the selected parameter setting (e.g., by varying the objective function, the change frequency, and the change type).

Once a computational experiment ends, the results, in terms of the performance measures, are used to analyze the algorithm at hand. Often, to statistically process the results is recommended. Using descriptive statistics is the common way to do so [5, 14–16]. However, if the study involves more complex analysis (e.g., algorithms comparison), then statistical tests are necessary [17–22]. In that sense, [23] suggested using nonparametric tests for conducting such analysis. The main reason for this suggestion is that the normality assumption of data is frequently violated or simply hard to verify because the available data is not enough.

Table 1: Comparison among some available technologies for experimentation in dynamic environments.

Technology	Visualization of the results?	Statistical tests?
EvolvingObjects	No	No
EASEA	Yes	No
GUIDE	Yes	No
CILib	No	No
GAUL	No	No
Apache Commons Math	No	Yes
MATLAB Optimization Toolbox	Yes	No

The interested reader in the topic of experimentation in dynamic environments is referred to the works of [1–3], which contain more details on algorithms, problems, performance measures, and experimentation in dynamic environments. In addition, the website *Intelligent Strategies in Uncertain and Dynamic Environments* (http://www.dynamic-optimization.org/) contains useful references on this topic.

2.1. Related Technologies. Currently, there are many technologies that can be employed by researchers for experimentation in dynamic environments. Most of them are software libraries or application frameworks, which have been conceived for stationary optimization. In what follows we review some of these available technologies, which can at least allow for

(1) implementing problems, algorithms, and performance measures in continuous domain,

(2) executing computational experiments,

(3) displaying the experimental results through a graphical user interface (GUI),

(4) performing statistical tests for algorithm comparison.

Table 1 shows some technologies that include the first two requirements above. Regarding the other requirements, one sees that EvolvingObjects [24] do not fulfill them. However, this C++ framework is extremely efficient and easy to extend in the implementation of problems, methods, and measures [25]. Additionally, there are projects such as EASEA (http://easea.unistra.fr/easea/index.php/EASEA_platform) or GUIDE (https://gforge.inria.fr/projects/guide) that propose graphical user interfaces to better interact with EvolvingObjects. Despite this, none of them process statistically the results.

Similarly, CILib (http://www.cilib.net/) is a software library that does not fulfill requirements (3) and (4). However, it is very efficient in experiment execution and currently includes an implementation of the Moving Peaks Benchmark. It is important to highlight that CILib is implemented in Java.

On the other hand, the GAUL (http://gaul.sourceforge.net/index.php) library, implemented over C and C++ languages, is devoted to the optimization by evolutionary algorithms (e.g., genetic algorithms). GAUL contains several examples already implemented, supporting the execution of experiments over different processors through the MPI

technologies. However, as far as we know, no software exists which includes a GUI for this library.

In the last years, an important project has been developed by the Apache Software Foundation related to mathematics. This project, named *Commons Math*, is a software library implemented in Java. Commons Math provides multiple mathematical features, including the optimization by metaheuristics and statistical tests. Despite this, it does not have GUI and some important statistical tests, that is, those suggested in [23] to properly analyze the experiment results.

Another relevant technology in this context is the *Optimization Toolbox* from MATLAB (http://www.mathworks.com/products/matlab/). This toolbox offers a GUI, several commands, and in-built functions, which makes experimentation with optimization problems easy. MATLAB also provides statistical tests which are included in the *Statistics Toolbox*. Some of the suggested nonparametric tests are included in this toolbox (e.g., Friedman and Wilcoxon tests). Nevertheless, it is worth noting that both, MATLAB and the mentioned toolboxes, are nonfree software.

Summarizing this section, one can see that there are many alternatives in the selection of technologies for experimentation in dynamic environments. However, most of these technologies do not fulfill all the requirements stated at the beginning of this section. Of course, a solution to this issue could be to properly extend some of those frameworks (e.g., EvolvingObjects or CILib). Unfortunately, it generally requires (1) an in-depth knowledge on the framework at hand in order to extend it and (2) agreeing with the employed software license. For these reasons, we have developed DynOptLab from scratch, with aim of adding some extra features, as we shall explain in the next section.

3. Technical Aspects of DynOptLab

The proposed tool, DynOptLab (Dynamic Optimization Laboratory), was programmed on Java technology, which is an efficient, high-level, and multiplatform language developed by Sun Microsystems (Oracle Corporation) (http://www.oracle.com/us/sun/). As was mentioned before, DynOptLab consists of (1) a framework for the creation of problems, algorithms, and performance measures and (2) a GUI to manage execution of the experiments. Thus, the final user (researcher) only needs to focus on programming the proposals, and if a good generalization is achieved, then it is possible to set them through the GUI. Both components are explained in what follows.

3.1. Framework. An object-oriented framework (OOF) is a reusable design of a system that describes how this system should be decomposed in a set of interacting objects. Different from software architectures, an OOF is expressed by a programming language, and it is based on a specific problem domain [26]. Basically, an OFF is composed of two main elements: *hot spots* and *frozen spots*. The hot spots represent extensible code through abstract classes or interfaces. Frozen spots are features that the final user cannot change. These features define the logic of the problem, in our case, experimentation in dynamic environments. In that sense, our aim with the proposed framework is to provide the researchers with the facilities of adding new problems, algorithms, and performance measures. The class diagram in Figure 1 depicts the proposed framework. Note that we highlighted the hot spots of our framework in bold. In what follows, we will explain the main details of each of these entities.

Algorithm Interface. This interface represents an algorithm to be executed in the given experiment, and it allows for the inclusion of new algorithms to the framework, that is, by its implementation. It contains the `init()` and `iterate()` methods, which are designed to set the initial state of the algorithm and to perform a single iteration, respectively. In turn, these methods are called by the `Experiment` class for controlling the algorithm execution. Besides, it also includes a `ProblemDefinition` instance as formal description of the optimization problem to solve. This instance is provided by the `Experiment` class through the method `setProblem()`. Finally, methods `setSeed()` and `generationSize()` allow for setting the random seed by the algorithm during the run and obtaining its number of function evaluations per iteration. The latter is important information required by the `Experiment` class in order to properly call the performance measures based on the algorithm iteration.

ProblemDefinition Interface. This interface represents a basic definition of an optimization problem. Consequently, it provides the necessary information to the `Algorithm` interface: a method for evaluating a solution in the objective function, the search space dimension, the maximum and minimum coordinates of the search space, and whether it is a minimization problem or not. In this way it hides sensible information and methods related to the dynamic problem (represented by the `DynamicOptimizationProblem` interface), for instance, the optimum value and location methods for governing the dynamic of the problem, among others.

DynamicOptimizationProblem Interface. It represents a dynamic optimization problem and includes not only the corresponding problem definition but also specific methods for controlling the environment dynamics. By implementing it, the researcher can add new DOPs to the framework. It contains the important methods `init()` and `change()`, which are both called by the `Experiment` class. The first one aims to initialize (reset), at the beginning of every run, the state of the problem, while the second one allows for changing the problem (e.g., by making the transition from $f^{(t)}$ to $f^{(t+1)}$).

Measure Interface. This interface represents a specific performance measure to assess the algorithm in a given problem. It also allows the researcher to add new measures to the framework. Concrete classes that implement this interface can record the algorithm performance on several runs. This can be done since the `Experiment` class notifies them through the `notifyMe()` method, by taking

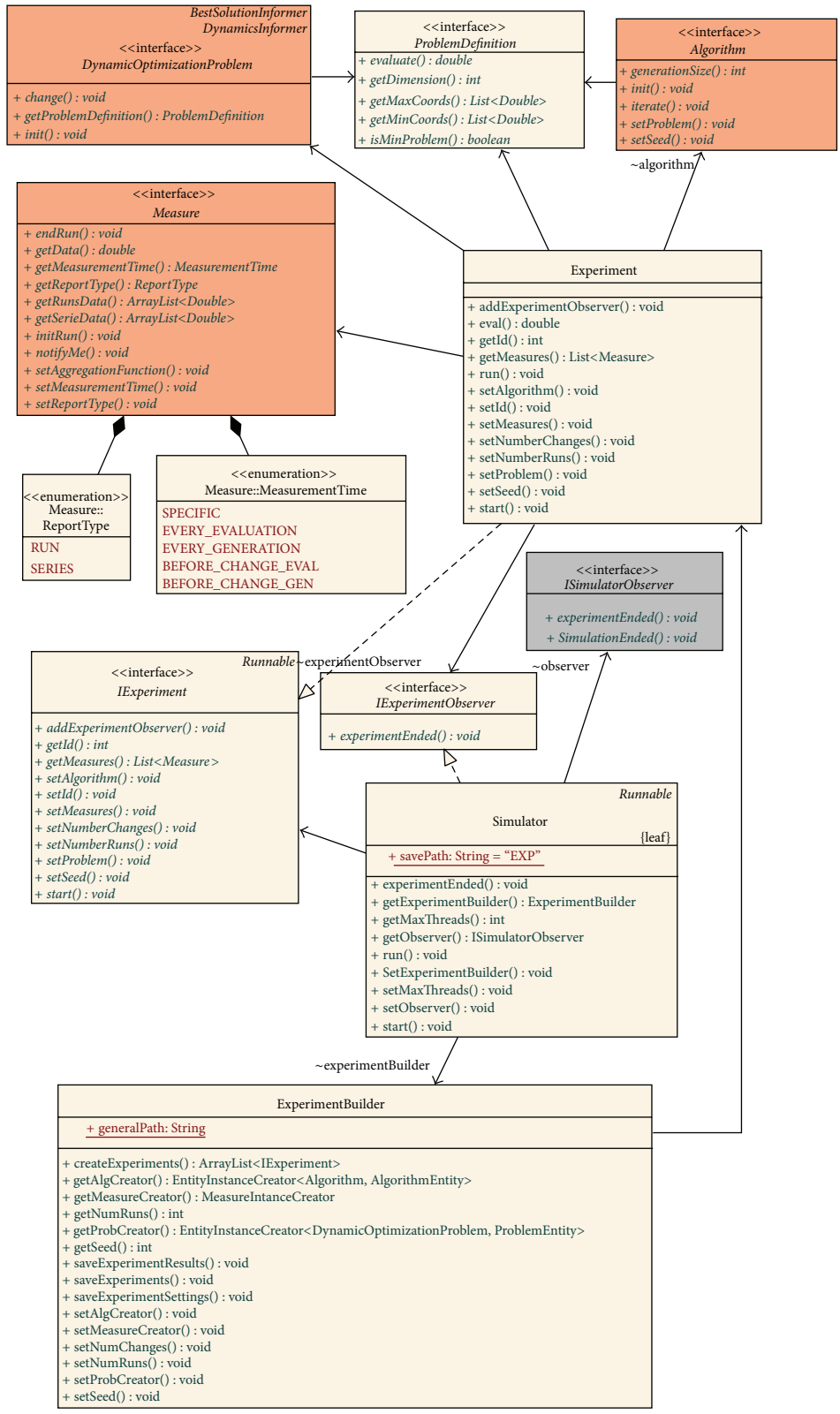


Figure 1: Class diagram corresponding to the framework of DynOptLab. Classes in bold are the framework hot spots. Some methods and attributes are excluded for a better understanding.

into account their measurement time and report type. These two options can be defined by the enumeration structures `MeasurementTime` and `ReportType`, respectively. Note that there are two possibilities regarding the measurement time, `RUN` or `SERIES`. In the first one, the measure records the required values during the run and aggregates according to the function stated through the `setAggregationFunction()`, while, in the second case, the values are collected as time series. On the other hand, the `MeasurementTime` enumeration can be one of five types: `SPECIFIC` (defined by the user), `EVERY_EVALUATION` (every single evaluation), `EVERY_GENERATION` (every single generation), `BEFORE_CHANGE_EVAL` (in the last function evaluation before the change), and `BEFORE_CHANGE_GEN` (before the last algorithm iteration before the change). In addition, this interface defines several methods for interacting with the `Experiment` and `ExperimentBuilder` classes. For instance, `initRun()` and `endRun()` are used by the `Experiment` class to inform the measure of the starting and ending of a run. As a consequence, the `Measure` instance can perform specific tasks related to the computation of the measure itself (i.e., to initialize and/or to aggregate certain variables). On the other hand, `get` methods provide useful information for the `ExperimentBuilder` for saving the generated data of the measure in files.

Experiment Class. This class implements the `IExperiment` interface, and its major goal is to control the execution of single instances of problem and algorithm. It receives through the `sets` methods the following inputs: an `Algorithm` instance, a `DynamicOptimizationProblem` instance, the number of runs to perform, an initial random seed, the number of changes for the environment, and a set of `Measures` instances. Note that since the `IExperiment` interface extends the `Runnable` interface, the `Experiment` class must implement the method `run()`. As a result, it becomes an independent execution unit, which can be exploited by the `Simulator` class, with the aim of parallelizing the execution of multiple experiments.

Simulator Class. The main objective of this class is the execution of multiple experiment units, that is, pairs of problem-algorithm. It can be done either sequentially or in parallel. The latter is an effective strategy to cope with the computational complexity involved by exploiting the technology of modern computer processors. Specifically, the `Experiment` class is related to this class through the design pattern *Observer* [27], which is a typical scheme for event-based models. Accordingly, when an experiment ends, it informs the `Simulator` class. In this case, the *Simulator* class removes the experiment from its queue and creates a new thread for executing a new experiment. It is also worth pointing out that the `Simulator` class communicates with DynOptLab's GUI in two different forms: one through the *ExperimentBuilder* class, which provides the experiment set to be executed, and a second one through the design pattern *Observer*, which is presented by the implementation of the interface *ISimulationObserver* at a certain class of the GUI.

ExperimentBuilder Class. This class has two major goals, to build multiple `Experiment` instances and to save the related results in files. The first task is done through the `createExperiments()` method, while the second one is done through the `save` methods. Note that the experiment results are saved together with the corresponding settings. Finally, through the `gets` and `sets` methods, this class interacts with the `Simulator` class and DynOptLab's GUI.

In spite of the above technical aspects, the researcher only needs to interact with interfaces related to algorithms, problems, and measures. The framework also allows the parameter setting of algorithms, problems, and measures, during the run of the application. This online assignation of parameters is possible thanks to the library *SimpleXML* (http://www.simplexml.sourceforge.net/). More details on this feature are given in what follows.

3.2. Graphical User Interface. The GUI of DynOptLab is very simple and intuitive. It was developed on the library SWT (Standard Widget Toolkit) from the Eclipse (http://www.eclipse.org/) project. SWT provides a set of visual components (widgets) for building GUI in Java. The objective of selecting SWT is to achieve a similar native aspect in different platforms (e.g., Microsoft Windows and Linux).

DynOptLab's main window is composed of five tabs, as is shown in Figure 2. Specifically, these tabs correspond to the modules *Problems*, *Algorithms*, *Measures*, *Experiments*, and *Results*. The first three tabs are devoted to the management of problems, algorithms, and measures. As was mentioned before, this process is carried out thanks to the library SimpleXML. Specifically, each class representing a problem, algorithm, or measure is associated with an XML file that contains the parameters subject to variation. These parameters represent class attributes, which are declared as annotated attributes, according to the SimpleXML technology. In this way, the researcher can externally interact with the compiled code, by setting several values for each attribute (parameter) to study. Hence, different instances of the same problem or algorithm can be obtained.

All the XML files have simple and similar structure; that is, they contain (1) a short name, (2) a detailed description, (3) the full name of the implement class, and (4) the parameter settings, represented as lists of values. In the particular case of performance measures, the XML file also includes the measurement time, the aggregation function, and the report type. The next sections explain these and other specific features of DynOptLab.

3.2.1. Management of Problems, Algorithms, and Measures. One of the main features of DynOptLab is the management of factors and response variables of the experiments. In our case, the experiment factors are the parameters that define problems and algorithms, while the response variables are the performance measures [28]. So, the management of problems, algorithms, and measures can be made through the corresponding tabs: *Problems*, *Algorithms*, and *Measures*, which have a very similar structure. For instance, Figure 2 shows the tab corresponding to problems. Note that the

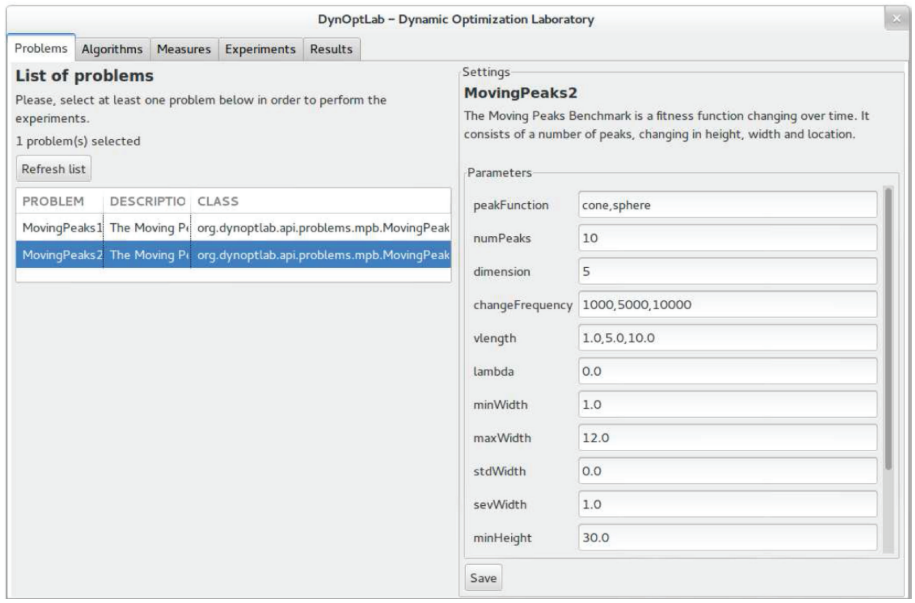


Figure 2: *Problems* tab, for the selection and configuration of the problems in DynOptLab.

interface is divided into two main zones. On the left zone, there is a list of the available classes (e.g., those corresponding to XML files). This list allows the user to select of a particular class. Once the class is selected, the right zone shows the parameters that can be subject to variation of the class. It is important to remark that the list of classes is filled from the XML files present in the folders *algorithms*, *problems*, and *measures*. In turn, such folders are on the same route of the main application. On the other hand, parameters of the right zone can be set as a single value or as a list of values separated by comma (see parameters *peakFunction*, *changeFrequency,* and *vlength* in Figure 2).

It is worth observing that this feature of setting several values for each parameter allows for obtaining multiple instances of problems and algorithms. So, the development of multifactorial experiments is possible. For instance, if a given problem has two parameters that are set with n and m values, respectively, then the number of instances derived from the combination of these values is $n \cdot m$.

3.2.2. Management of the Experiments Execution. Once the problems, algorithms, and measures are selected, the next step is to manage the experiments execution. To this end, the user can use the *Experiments* tab of DynOptLab's main window (Figure 3). This interface offers a summary of the number of problems, algorithms, and measures selected. Observe that the field named *Problem-algorithm pairs* shows the number of single experiments to be executed.

Additionally, the *Experiments* tab allows setting (1) the initial random seeds, (2) the number of environment changes, (3) the number of runs (executions) for every pair problem-algorithm, and (4) the number of threads to parallel execute the pairs. As was explained in Section 3.1, such a setting is sent to `Simulator` class, which is responsible for the experiments execution.

To start the simulation, the user has to click the button *Run experiments*. After that, the bottom panel shows the current state of the simulation, including the final message *Experiments completed!*. Each finished experiment (pair of problem-algorithm) is automatically saved in terms of the used performance measures.

3.2.3. Visualization and Statistical Analysis of the Results. The results of the experiments (from the *Experiments* tab) are automatically summarized in the *Results* tab (Figure 4). Besides, in this tab, it is possible to load results from previous executions, that is, by selecting the option *Load other results*.

With the aim of organizing the display and analysis better, the *Results* tab is divided into four subtabs: *Experiment results*, *Comparison*, *Statistical analysis*, and *Post-hoc analysis*. The first one, shown in Figure 4, is devoted to displaying the results of a given problem-algorithm pair. If the user selects an element on the list located in the left zone, then the *Experiment results* tab will show the results of the performed runs. From Figure 4, it is possible to observe that these results are shown for a particular measure. Additionally, at the bottom zone of the tab, there is a descriptive summary of the results.

Despite the benefits of this descriptive summary, it is usually interesting to compare several algorithms in certain problems. To this end, DynOptLab allows for multiple comparisons based on the loaded experiments from the left zone. At the bottom of this left zone, the button *Make a comparison* is responsible for conducting the comparison. This button is enabled only in the case of selecting two or more experiments. This feature is depicted in Figure 5.

Similar to the *Experiment results* tab, in the *Comparison* tab, the user can select the performance measure and the descriptive statistics to show the results. These results are

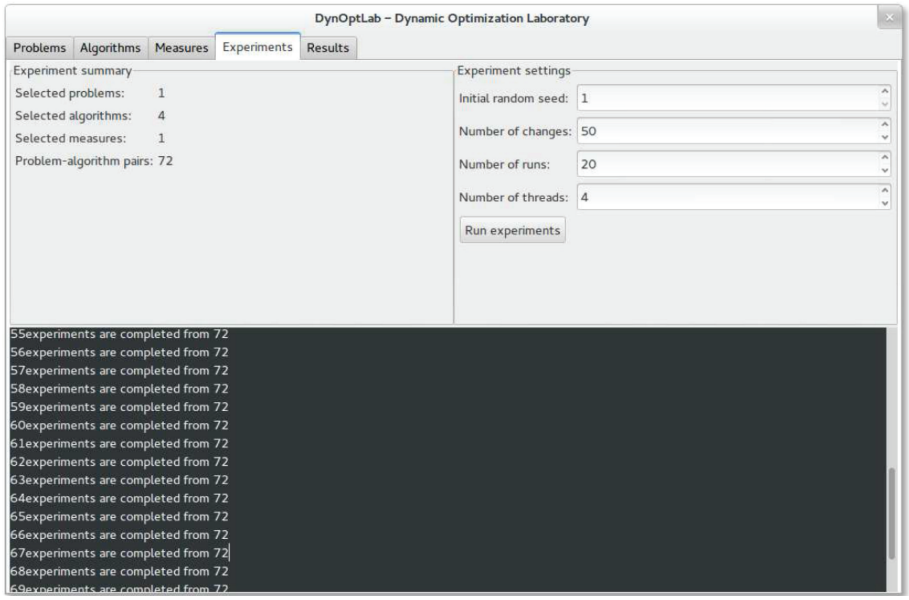


Figure 3: *Experiments* tab, devoted to the management of the experiments.

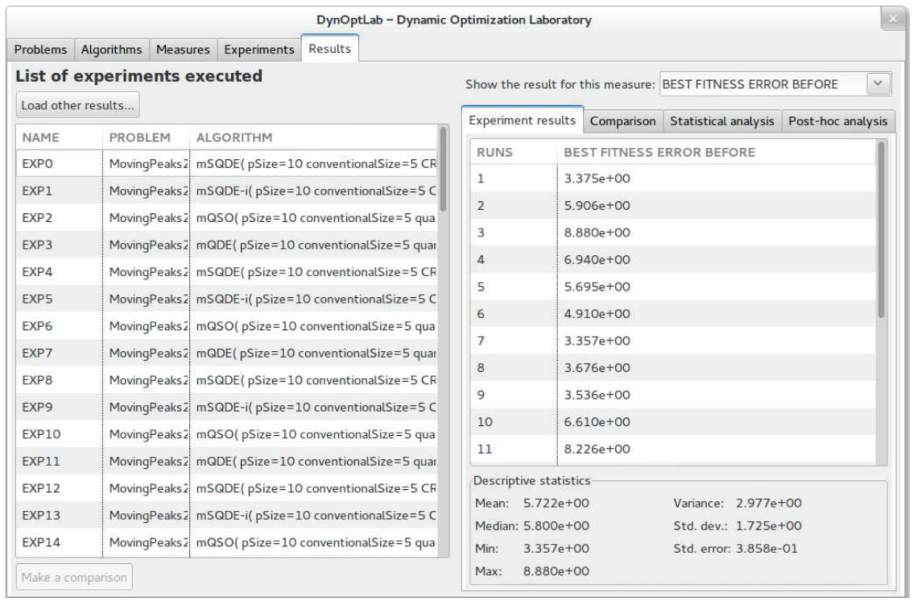


Figure 4: *Results* tab, which visualizes and allows for statistically processing the results of the experiments.

visualized in a table in the central zone of the tab. Furthermore, at the bottom of this tab, the user has two options: to proceed with a statistical analysis and to export the table data. In the latter, the data can be saved in three popular formats: as a Latex table, as a CSV file, and as a simple text file.

Regarding the statistical analysis, DynOptLab provides two nonparametric, statistical tests using as input the data in the comparison table. Friedman and Iman-Davenport tests, which are devoted to detecting general differences among all algorithms, have been specifically included. The results of the tests are visualized by the *Statistical analysis* tab (see Figure 6). This interface is divided into three zones. In the top zone, a table shows the average ranks of the algorithms from the Friedman test. These ranks are also visualized through a bar chart in the middle zone. Finally, additional results of the tests are listed in the bottom zone. In that sense, we have included specific information of the tests (i.e., statistics, degree of freedom, and p value).

Inside the *Statistical analysis* tab, the user can proceed with a post hoc analysis, in the case of obtaining p values lower than 0.05 (significance level) from Friedman and Iman-Davenport tests. Specifically, the results of the post hoc tests are handled by the *Post-hoc analysis* tab (Figure 7). Following the suggestions of [23], DynOptLab incorporates

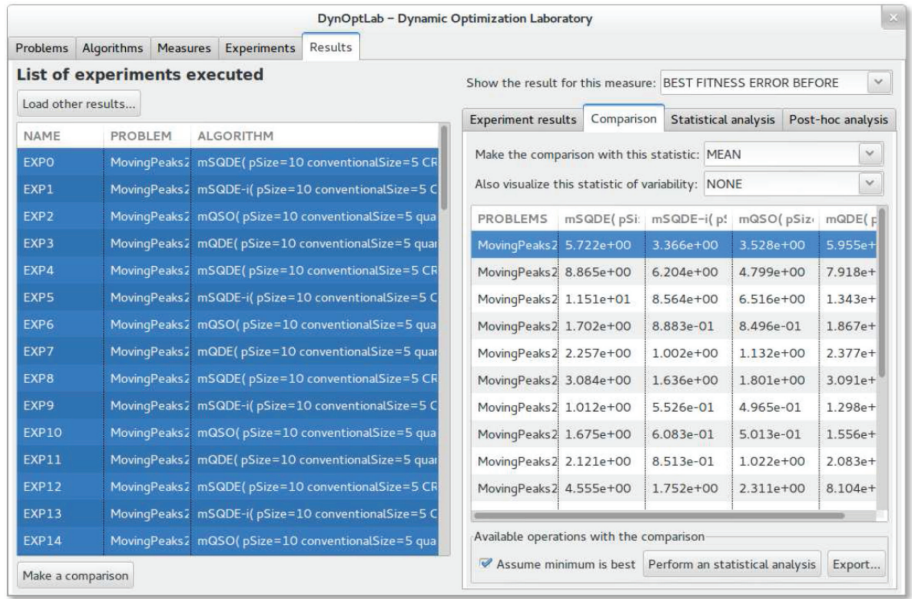


Figure 5: *Comparison* tab, which shows the comparison among several algorithms.

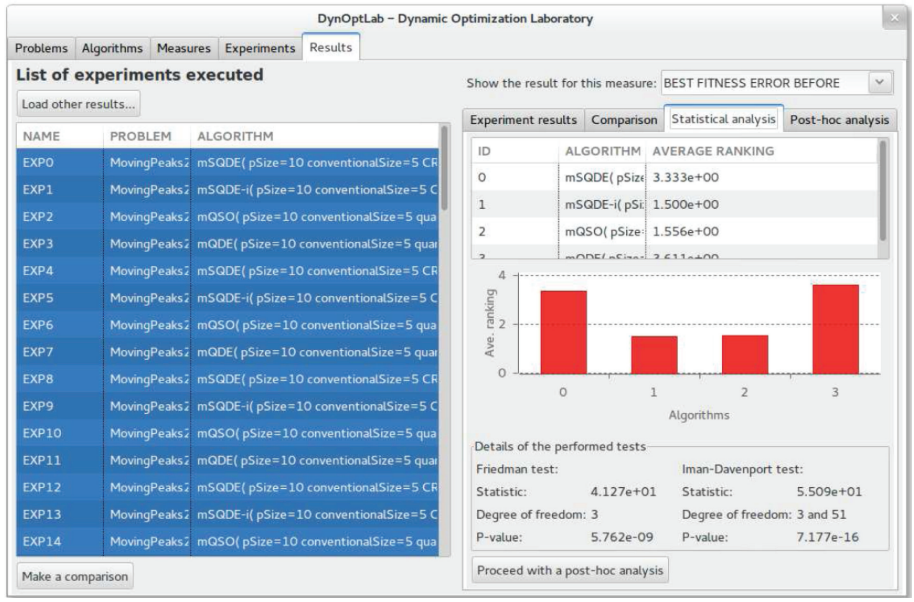


Figure 6: *Statistical analysis* tab, devoted to the Friedman and Iman-Davenport statistical tests.

nonparametric tests, such as Holm, Bonferroni, Hocheberg, and Nemenyi. So, two different analyses are possible from DynOptLab, one for comparing the best algorithm against the rest and a second one for performing pairwise comparisons among the algorithms. In both cases, the related p values are listed through tables, and it is also possible to show the adjusted p values. These adjusted p values are more reliable than their unadjusted counterparts. For more details, the user is referred to [23].

4. A Case Study

With the aim of seeing DynOptLab in action, in this section, we will use it for handling a typical experimental study in dynamic environments. This case study has been illustrated by the figures we used. So, in what follows we only comment on the specific details of the figures that are related to our case study. Essentially, we want to analyze the performance of four algorithms: mQSO [29], mQDE [8], mSQDE [8], and

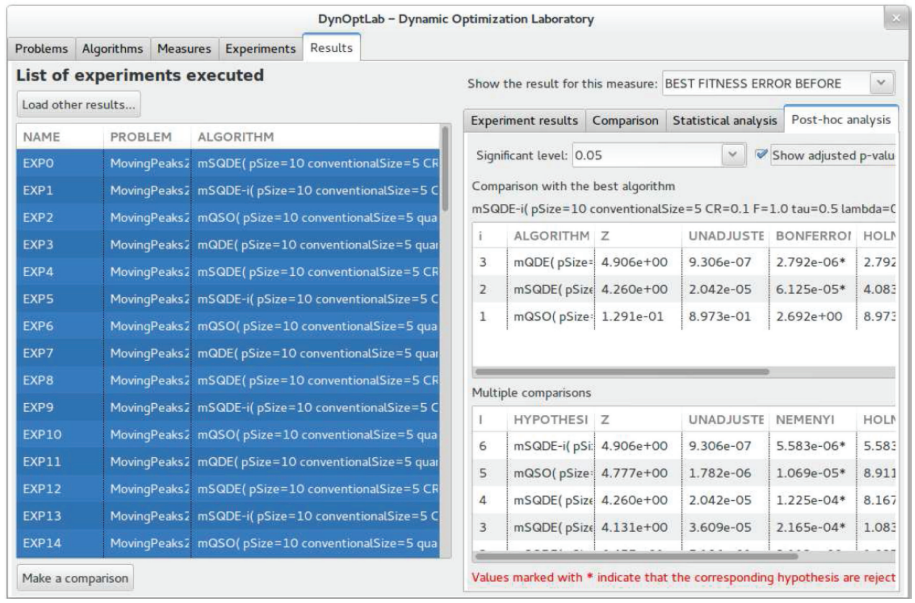


Figure 7: *Post-hoc analysis* tab, devoted to performing post hoc tests.

mSQDE-i [8]. As a test bed, we selected the Moving Peaks Benchmark (MPB) [9], which is a popular test bed in dynamic environments. Specially, we considered several instances of MPB's scenario 2 by varying three factors: the peak function (*peakFunction*), the shift severity (*vlength*), and the change frequency (*changeFrequency*). The considered values are the following:

(i) peakFunction = {*cone*, *sphere*};

(ii) vlength = {1.0, 5.0, 10.0};

(iii) changeFrequency = {1000, 5000, 10000}.

As was mentioned before, the presence of several values in a problem (or algorithm) is interpreted by DynOptLab as combination of factors. Hence, the above parameter setting leads to 18 different problem instances, which together with the 4 algorithms we considered give 72 experiments (i.e., pairs of problem-algorithm to be executed).

For assessing the algorithm performance, we rely on the *best error before the change* measure [5, 8, 10], where the lower it is, the better the algorithm is. In general, we planned 20 runs for every pair of problem-algorithm, and we assumed that all the problem instances change 50 times.

To see how DynOptLab can handle this design of experiments, consider first the class diagram of Figure 8. This diagram shows how to extend the DynOptLab framework, in order to include the considered problems, algorithms, and performance measures. For the sake of simplicity, we only show the mQSO algorithm in the diagram. Regarding the performance measure, the diagram shows the `BestFitnessError` class, because this measure is the core of the best error before the change, as we will show further on.

DynOptLab's interfaces related to the configuration of the experiment are shown in Figures 3, 9, and 10. In this regard, one sees how the parameter values are set to the problem and how multiple algorithms (the four we considered) can be selected at the same time. Furthermore, from Figure 10, we can see how the class `BestFitnessError` can be transformed into the best error before the change, that is, by selecting `BEFORE_A_CHANGE_EVAL` as measurement time. Similarly, Figure 3 illustrates the configuration and execution experiments. Once the execution is finished, the results are loaded by DynOptLab as is shown in Figure 4. By selecting all the experiments from the left zone of the Results tab, it is possible to perform the algorithms comparison depicted in Figure 5. In turn, such a comparison can be statistically analyzed through the nonparametric test from the *Statistical analysis* tab. See, for example, that the best rank is obtained by algorithm mSQDE-i, followed by mQSO. This can be easily seen from the corresponding bar chart. It is important to note that the presence of p values lower than 0.05 from Friedman and Iman-Davenport tests indicates that significant difference exists at group level. In order to detect which pair of algorithms are really different, the user can rely on the information given by the *Post-hoc analysis* tab. In this case, see, for instance, that the best algorithm (mSQDE-i) is significantly better than mSQDE and mQDE, while it is not different with respect to mQSO. These conclusions are observed with the help of symbol $*$, which indicates that the null hypothesis is rejected. Of course, this null hypothesis states that the behaviors of the algorithms involved are the same. Similar information can be obtained from the multiple comparison carried out in the bottom zone of the *Post-hoc analysis* tab (Figure 7).

5. Conclusions and Future Works

In this work, we proposed DynOptLab, a free and noncommercial tool for experimental analysis in dynamic environments. This tool provides not only a framework to easily

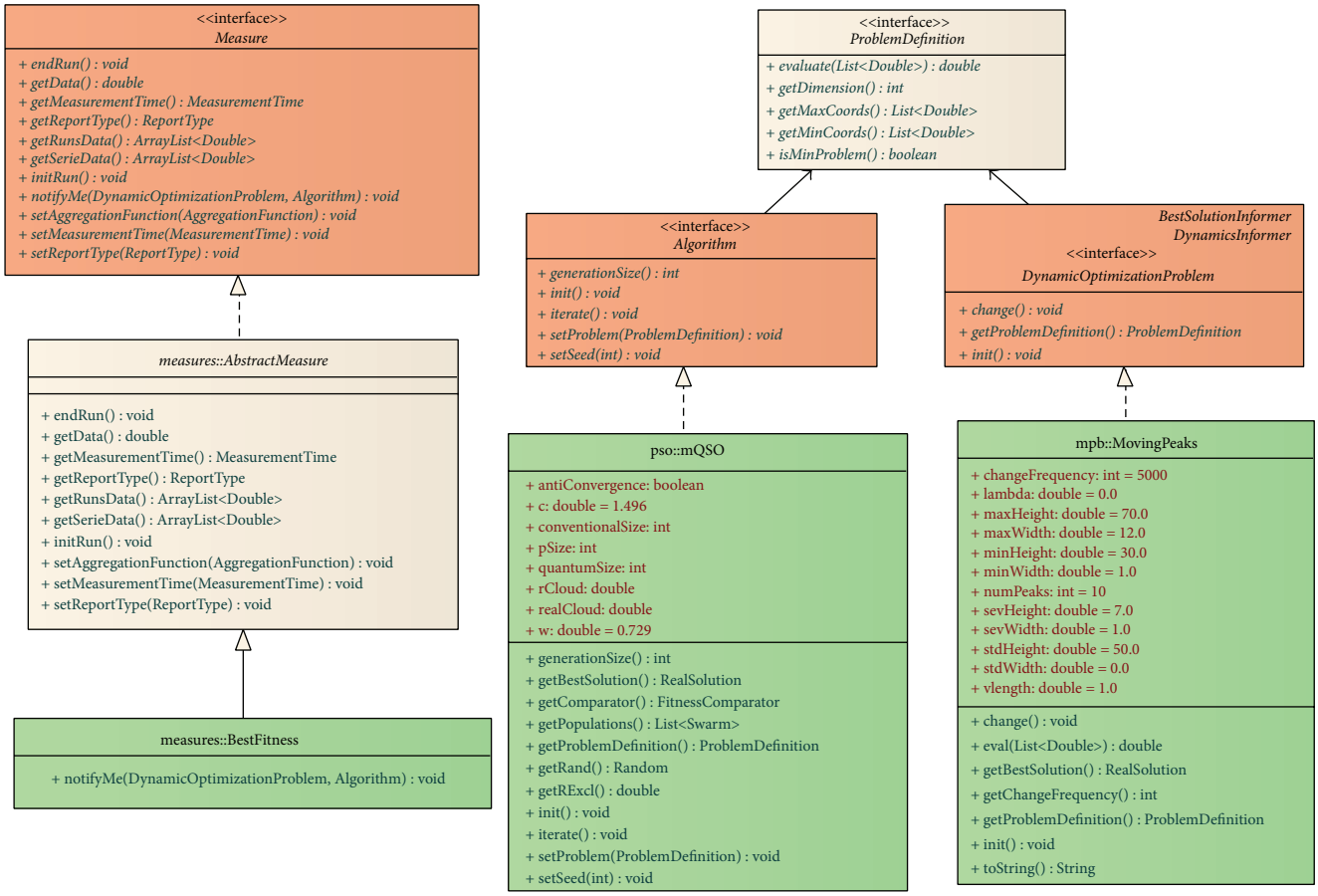


Figure 8: Example of how to extend DynOptLab's framework.

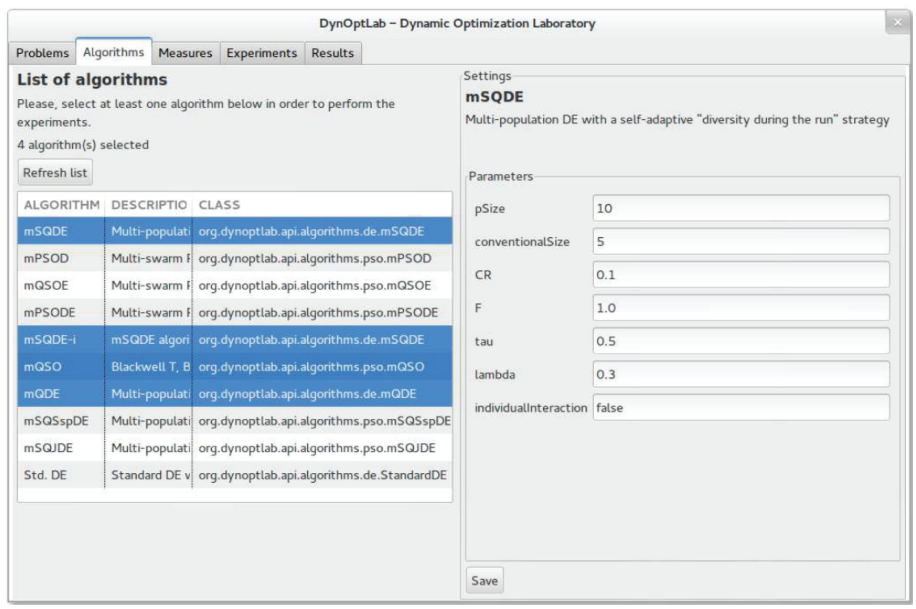


Figure 9: Selection and configuration of the algorithms in DynOptLab.

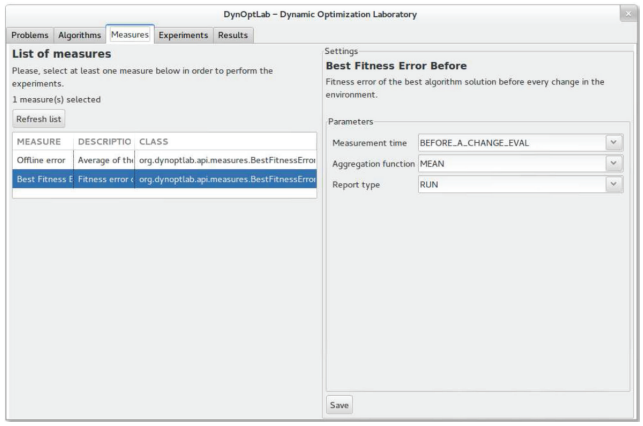


Figure 10: Selection and configuration of performance measures in DynOptLab.

include new problems, algorithms, and performance measures, but also a graphic user interface to efficiently manage experiments and to statistically analyze the results.

The main features of DynOptLab were observed through the study of a typical case in the context of experimentation in dynamic environments. In that sense, DynOptLab can efficiently handle the considered design of the experiment.

Despite this progress, we believe that this is a first step to obtain a better tool. Our future work will be devoted to the inclusion of other problems and algorithms, with the aim of obtaining a framework with the state-of-the-art exponents on the subject.

DynOptLab is currently available at the website of the Models of Decision and Optimization (MODO) Research Group, specifically at the following URL: http://modo.ugr.es/DynOptLab/.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

P. Novoa-Hernández has the support of a postdoctoral scholarship from the Eureka SD project (Erasmus Mundus Action 2) coordinated by the University of Oldenburg, Germany. C. Cruz Corona and D. A. Pelta acknowledge support from Projects TIN2011-27696-C02-01, Spanish Ministry of Economy and Competitiveness, P11-TIC-8001 from the Andalusian Government (including FEDER funds from the European Union), and GENIL-PYR-2014-9 Project from University of Granada.

References

[1] Y. Jin and J. Branke, "Evolutionary optimization in uncertain environments—a survey," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 3, pp. 303–317, 2005.

[2] C. Cruz, J. R. González, and D. A. Pelta, "Optimization in dynamic environments: a survey on problems, methods and measures," *Soft Computing*, vol. 15, no. 7, pp. 1427–1448, 2011.

[3] T. T. Nguyen, S. Yang, and J. Branke, "Evolutionary dynamic optimization: a survey of the state of the art," *Swarm and Evolutionary Computation*, vol. 6, pp. 1–24, 2012.

[4] D. Pelta, C. Cruz, and J. L. Verdegay, "Simple control rules in a cooperative system for dynamic optimisation problems," *International Journal of General Systems*, vol. 38, no. 7, pp. 701–717, 2009.

[5] J. Brest, "Constrained real-parameter optimization with e-self-adaptive differential evolution," in *Constraint-Handling in Evolutionary Optimization*, E. Mezura-Montes, Ed., vol. 198 of *Studies in Computational Intelligence*, pp. 73–93, Springer, Berlin, Germany, 2009.

[6] M. C. du Plessis and A. P. Engelbrecht, "Using competitive population evaluation in a differential evolution algorithm for dynamic environments," *European Journal of Operational Research*, vol. 218, no. 1, pp. 7–20, 2012.

[7] P. Novoa-Hernández, C. C. Corona, and D. A. Pelta, "Efficient multi-swarm PSO algorithms for dynamic environments," *Memetic Computing*, vol. 3, no. 3, pp. 163–174, 2011.

[8] P. Novoa-Hernández, C. C. Corona, and D. A. Pelta, "Self-adaptive, multipopulation differential evolution in dynamic environments," *Soft Computing*, vol. 17, no. 10, pp. 1861–1881, 2013.

[9] J. Branke, "Memory enhanced evolutionary algorithms for changing optimization problems," in *Proceedings of the Congress on Evolutionary Computation*, P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, Eds., vol. 3, pp. 1875–1882, IEEE Press, Washington, DC, USA, 1999.

[10] C. Li and S. Yang, "Fast multi-swarm optimization for dynamic optimization problems," in *Proceedings of the 4th International Conference on Natural Computation (ICNC '08)*, pp. 624–628, October 2008.

[11] K. Weicker, "Performance measures for dynamic environments," in *Parallel Problem Solving from Nature—PPSN VII*, J. J. M. Guervós, P. Adamidis, H.-G. Beyer, H.-P. Schwefel, and J.-L. Fernández-Villacañas, Eds., vol. 2439 of *Lecture Notes in Computer Science*, pp. 64–73, Springer, Berlin, Germany, 2002.

[12] E. Alba and B. Sarasola, "Measuring fitness degradation in dynamic optimization problems," in *Applications of Evolutionary Computation*, C. di Chio, S. Cagnoni, C. Cotta et al., Eds., vol. 6024 of *Lecture Notes in Computer Science*, pp. 572–581, Springer, Berlin, Germany, 2010.

[13] C. Li, S. Yang, T. T. Nguyen et al., "Benchmark generator for cec'2009 competition on dynamic optimization," Tech. Rep., Department of Computer Science, University of Leicester, Leicester, UK, 2008.

[14] C. Li and S. Yang, "A generalized approach to construct benchmark problems for dynamic optimization," in *Simulated Evolution and Learning*, vol. 5361 of *Lecture Notes in Computer Science*, pp. 391–400, Springer, Berlin, Germany, 2008.

[15] C. Li and S. Yang, "A clustering particle swarm optimizer for dynamic optimization," in *Proceedings of the 11th IEEE Congress on Evolutionary Computation (CEC '09)*, pp. 439–446, IEEE Press, Piscataway, NJ, USA, May 2009.

[16] C.-K. Au and H.-F. Leung, "An empirical comparison of CMA-ES in dynamic environments," in *Parallel Problem Solving from Nature—PPSN XII*, C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, Eds., vol. 7491 of *Lecture Notes in Computer Science*, pp. 529–538, Springer, Berlin, Germany, 2012.

[17] S. Yang, "Associative memory scheme for genetic algorithms in dynamic environments," in *Applications of Evolutionary Computing*, F. Rothlauf, J. Branke, S. Cagnoni et al., Eds., vol. 3907 of *Lecture Notes in Computer Science*, pp. 788–799, Springer, Berlin, Germany, 2006.

[18] C. M. Fernandes, C. F. Lima, and A. C. Rosa, "UMDAs for dynamic optimization problems," in *Proceedings of the 10th Annual Genetic and Evolutionary Computation Conference (GECCO '08)*, pp. 399–406, July 2008.

[19] W. Du and B. Li, "Multi-strategy ensemble particle swarm optimization for dynamic optimization," *Information Sciences*, vol. 178, no. 15, pp. 3096–3109, 2008.

[20] S. Yang, H. Cheng, and F. Wang, "Genetic algorithms with immigrants and memory schemes for dynamic shortest path routing problems in mobile ad hoc networks," *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, vol. 40, no. 1, pp. 52–63, 2010.

[21] P. Novoa-Hernández, D. Pelta, and C. Corona, "Improvement strategies for multi-swarm PSO in dynamic environments," in *Proceedings of the 2nd International Workshop on Nature Inspired Cooperative Strategies for Optimization (NICSO '10), November 2010*, J. González, D. Pelta, C. Cruz, G. Terrazas, and N. Krasnogor, Eds., pp. 371–383, Springer, Berlin, Germany, 2010.

[22] P. Novoa-Hernández, C. C. Corona, and D. A. Pelta, "Self-adaptation in dynamic environments—a survey and open issues," *International Journal of Bio-Inspired Computation*. In press.

[23] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization," *Journal of Heuristics*, vol. 15, no. 6, pp. 617–644, 2009.

[24] M. Keijzer, J. Merelo, G. Romero, and M. Schoenauer, "Evolving objects: a general purpose evolutionary computation library," *Artificial Evolution*, vol. 23, no. 10, pp. 829–888, 2002.

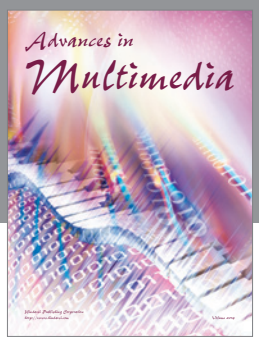
[25] E. G. Talbi, *Metaheuristics: From Design to Implementation*, John Wiley & Sons, 2009.

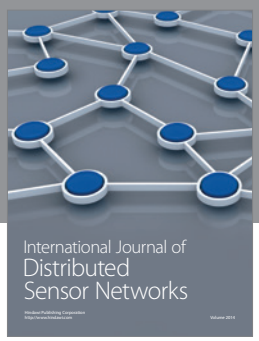
[26] M. Fayad, D. Schmidt, and R. Johnson, "Application frameworks," in *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, p. 638, John Wiley & Sons, 1st edition, 1999.

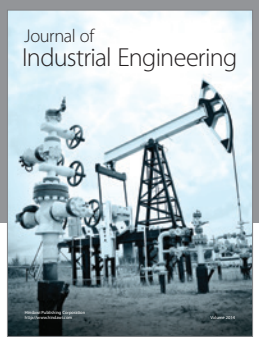
[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1995.

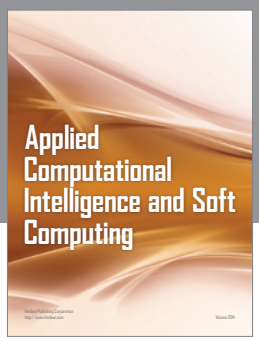
[28] T. Bartz-Beielstein, *Experimental Research in Evolutionary Computation: The New Experimentalism*, Springer, Berlin, Germany, 2006.

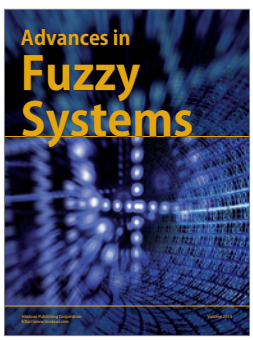
[29] T. Blackwell and J. Branke, "Multiswarms, exclusion, and anti-convergence in dynamic environments," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 4, pp. 459–472, 2006.











Hindawi

Submit your manuscripts at
http://www.hindawi.com