# EasyAnalyzer: an object-oriented framework for the experimental analysis of stochastic local search algorithms*

Luca Di Gaspero[1], Andrea Roli[2], and Andrea Schaerf[1]

[1] DIEGM, University of Udine, via delle Scienze 208,
I-33100, Udine, Italy {l.digaspero | schaerf}@uniud.it

[2] DEIS, University of Bologna, via Venezia 52,
I-47023 Cesena, Italy andrea.roli@unibo.it

**Abstract**

In this paper we describe EasyAnalyzer, an object-oriented framework for the experimental analysis of SLS algorithms, developed in the C++ language. EasyAnalyzer integrates with EasyLocal++, a framework for the development of SLS algorithms, in order to provide a unified development and analysis environment. Moreover, the tool has been designed so that it can be easily interfaced also with SLS solvers developed using other languages/tools and/or with command-line executables.

## 1 Introduction

In recent years, much research effort has focused on the proposals of environments specifically designed to help the formulation and implementation of Stochastic Local Search (SLS) algorithms by means of specification languages and/or software tools. Unfortunately, as pointed out by in [10, Epilogue, pp. 533–534], the same amount of effort has not been oriented in the development of software tools for the experimental analyses of the algorithms. To this regard, [9] propose a suite of tools for visualizing the behavior of SLS algorithms, which is particularly tailored for MDF (Metaheuristics Development Framework) [11]. However, to the best of our knowledge, we can claim that at present there is no widely-accepted comprehensive environment.

In this paper we try to overcome this lack by proposing an object-oriented framework, called EasyAnalyzer, for the analysis of SLS algorithms. EasyAnalyzer is a software tool that belongs to the family of Object-Oriented (O-O) frameworks. A framework is a special kind of software library, which consists of a hierarchy of abstract classes and is characterized by the *invertion of control* mechanism for the communication with the user code (also known as the *Hollywood Principle*: "Don't call us, we'll

*This paper is an excerpt from [4].

*Proceedings of the 15th International RCRA workshop (RCRA 2008):*
*Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*
*Udine, Italy, 12–13 December 2008*

call you"). That is, the functions of the framework call the user-defined ones and not the other way round as it usually happens with software libraries. The framework thus provides the full control logic and, in order to use it, the user is required to supply the problem specific details by means of some standardized interfaces.

Our work is founded on *Design Patterns* [8], which are abstract structures of classes, commonly present in O-O applications and frameworks, that have been precisely identified and classified. The use of patterns allows us to address many design and implementation issues in a more principled way.

EasyAnalyzer provides a family of off-the-shelf analysis methods to be coupled to local search solvers developed using one of the tools mentioned above or written from scratch. For example, it performs various kinds of search space analysis in order to understand, study, and tune the behavior of SLS algorithms. The properties of the search space are a crucial factor of SLS algorithm performance [7, 10]. Such characteristics are usually studied by implementing *ad hoc* programs, tailored both to the specific algorithm and to the problem at hand. EasyAnalyzer makes it possible to abstract from algorithm implementation and problem details and to design general search space analyzers.

EasyAnalyzer is specifically designed to blend in a natural way with EasyLocal++, the local search framework developed by two of these authors [5, 6], which has recently been entirely redesigned to allow for more complex search strategies. Nevertheless, EasyAnalyzer is capable of interacting with other software environments and with stand-alone applications.

This is an ongoing work, and some modules still have to be implemented. However, the general architecture, the core modules, and the interface with EasyLocal++ and with command-line executables are completed and stable.

The paper is organized as follows. In Section 2 we show the architecture of EasyAnalyzer and its main modules. In Section 3 we go in details in the implementation of the core modules. In Section 4 we draw some conclusions and discuss future work.

## 2 The architecture of EasyAnalyzer

The conceptual architecture of EasyAnalyzer is presented in Figure 1 and it is split in three main abstraction layers. Each layer of the hierarchy relies on the services supplied by lower levels and provides a set of more abstract operations.

**Analysis system:** it comprises the *core classes* of EasyAnalyzer. It is the most abstract level and contains the control logic of the different types of analysis provided in the system. The code for the analyses is completely abstract from the problem at hand and also from the actual implementa-

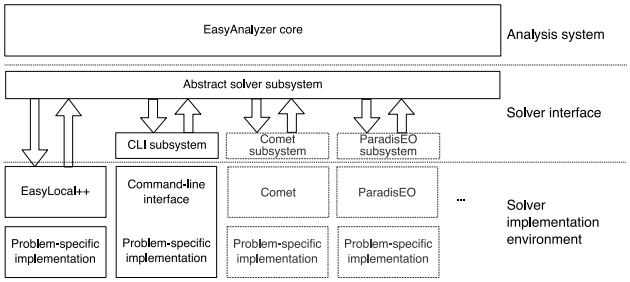


Figure 1: EasyAnalyzer layered architecture.

tion of the solver. The classes of this layer delegate implementation- and/or problem-related tasks to the set of lower level classes, which comply with a predefined service interface (described in the following).

**Solver interfaces:** this layer can be split into two components: the top one is the interface that represents an abstract solver subsystem, which simply prescribes the set of services that should be provided by a concrete solver in order to be used in the analyses. The coupling of the analysis system with the implementation is dealt with by this component.

The lower component is the concrete implementation of the interface for a set of SLS software development environments. Notice that in the case of EasyLocal++, this component is not present since EasyAnalyzer directly integrates within the development framework classes. The reason is that in the design of the solver interface we reuse many choices already made for EasyLocal++ thus allowing immediate integration.

For other software environments, instead, the solver subsystem component must be explicitly provided. Depending on the capabilities of the software environment, these interfaces can be implemented in a problem-independent manner (so that they can be directly reused across all applications) or it might require to be customized for the specific problem. Although in the second case the user could be required to write some additional code, our design limits this effort since our interfaces requires just a minimal set of functionalities.

**Solver environment:** it consists of the (possibly generic) SLS software development environment plus the problem-specific implementation. In some cases these two components coincide, as for solvers that do not make use of any software environment. In this case the interaction with the solver can make use of a simple command-line interface.

At present, we have implemented the direct integration with EasyLo-

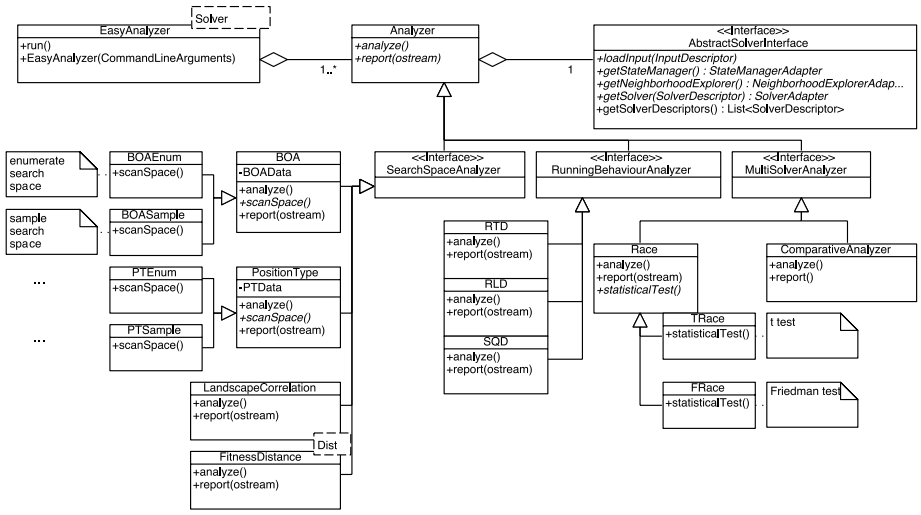


Figure 2: UML class diagram of the analysis system.

CAL++ and to the command-line interface[1] by means of a set of generic classes (i.e., C++ classes that make use of templates that should be instantiated with the concrete command-line options). We plan to implement also the interfaces to other freely available software environments like, e.g., ParadisEO [3] and Comet [13].

In the following subsections we present more in detail the problem-independent layers of the EASYANALYZER architecture and we give some examples of code.

## 2.1 The analysis system

The main classes of the analysis system are shown in Figure 2 using the UML 2.0 notation [12]. As in Figure 1 we report in solid lines the fully implemented components (dotted lines for the forthcoming ones).

Let us start our presentation with the `EasyAnalyzer` class. This class relies on the *Factory method* pattern to set up the analysis system on the basis of a given solver interface. Notice that the interface is specified as a template parameter, so that we are able to write the generic code for instantiating the analysis system regardless which of the concrete implementations is provided. Furthermore, the `EasyAnalyzer` class provides a standardized command-line interface for the interaction with the analysis system. This task is accomplished by managing a command-line interpreter object that is directly configured by the analysis techniques. That is, each analysis

[1] In Figure 1 the implemented components are denoted by solid lines while dotted lines denote components only designed.

technique "posts" the syntax of the command-line arguments needed by the interpreter object that is in charge of parsing the command line and dispatching the actual parameters to the right component.

The main component of the analysis system is the `Analyzer` class, which relies on the *Strategy* pattern. This component represents the interface of an analysis technique, whose actual "strategy" is going to be implemented in the concrete `analyze()` method defined in the subclasses. The `report(ostream)` method is used to provide, on an output stream, a human- and/or machine-readable report of the analysis, depending on the parameters issued on the command line.

The `Analyzer` class is then specialized on the basis of the SLS features that are subject of the analysis into the following three families:

`SearchSpaceAnalyzer:` these analyzers deal with features that are related to the search space. Several crucial properties of the search space can be analyzed with these modules, such as landscape characteristics and states reachability.

`RunTimeBehaviorAnalyzer:` their aim is to analyze the run-time behavior of the solvers. Analyses belonging to this family are, e.g., *run-time distribution* (`RTD`), *run-length distribution* (`RLD`) and *solution quality distribution* (`SQD`).

`MultiSolverAnalyzer:` they handle and evaluate groups of solvers. For example the `Race` analyzer tries to find-out the statistically best configuration of a solver among a set of candidate configurations by applying a racing procedure [2].

The interface with the services provided by the analysis system is established with the `AbstractSolverInterface` abstract class, which relies on the *Façade* pattern whose aim is to provides a simple interface to a complex subsystem. This class and the underlying classes and objects responsibilities are going to be detailed in the following subsection.

## 2.2 The solver interface

The architecture of the solver interface is shown in the top part of Figure 3. The derived classes on the bottom are the implementation of this interface in the EasyLocal++ framework.

The `SolverInterface` class acts as a unified entry point (the *Façade*) and as the coordinator of a set of underlying classes (*Abstract Factory* and *Factory method* patterns). Indeed, according to the EasyLocal++ design, we identify a set of software components that take care of different responsibilities in a SLS algorithm and we define a set of *adapter* classes for them. These

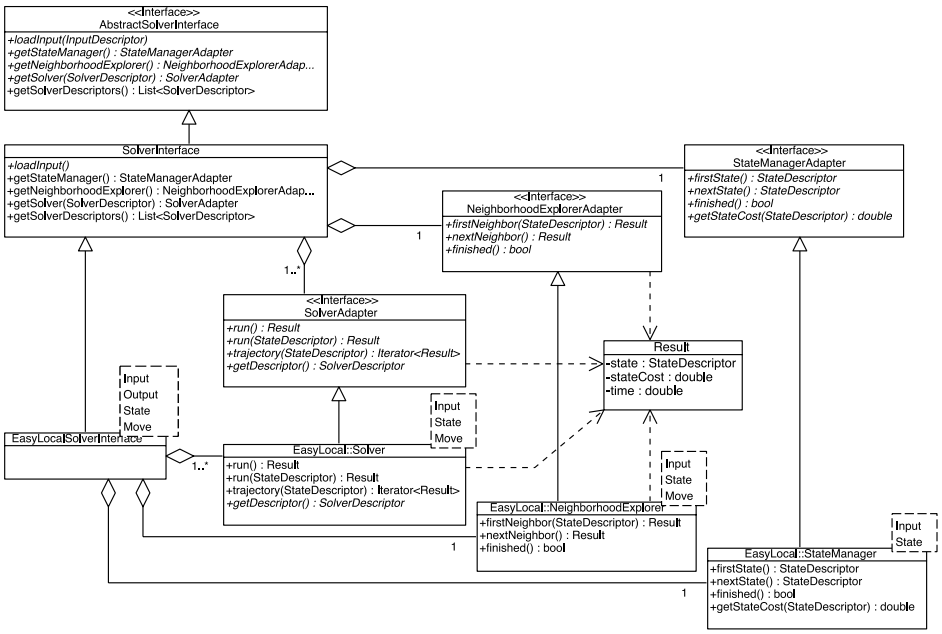


Figure 3: UML class diagram of the solver interface.

adapters have a straight implementation in EasyLocal++ (Figure 3, bottom part), and are those components that instead must be implemented for interfacing with different software environments. The components we consider are the following: `StateManagerAdapter:` it is responsible for all operations on the states of the search space that are independent of the definition of the neighborhood. In particular, it provides methods to enumerate and to sample the search space, and it allows us to evaluate the cost function value on a given state. The component relies on `StateDescriptor`s for the exchange of information with the analysis system (in order to avoid the overhead of sending a complex space representation).

`NeighborhoodExplorerAdapter:` it handles all the features concerning the exploration of the neighborhood. It allows to enumerate and to sample the neighbors of a given state, and to evaluate the difference in the cost function.

`SolverAdapter:` it encapsulates a single SLS algorithm or a complex solution strategy that involves more than one single SLS technique. Its methods allow us to perform a full solution run (either starting from a random initial state or from a state given as input), possibly storing all the trajectory from the initial state to the final one. This component returns also information on the running time and on the state costs.

In order to use EasyAnalyzer the user needs to instantiate the `Solver` template of the `EasyAnalyzer` class with the proper implementation of the `AbstractSolverInterface`. As for the EasyLocal++ solver, this interface is already provided with the framework, whilst for the command-line interaction the functionalities must be implemented by the user in the stand-alone executable.

# 3 Implementation of EasyAnalyzer

In this section we describe two representative examples of the analyzers currently implemented, with emphasis on the design process that relies on the abstractions provided by the Solver interfaces.

## 3.1 `SearchSpaceAnalyzer`

In this section we illustrate the design and implementation of an analyzer for *Basins of attraction* (BOA), useful for studying the reachability of solutions. Given a deterministic algorithm, the basin of attraction $\mathcal{B}(\overline{s})$ of a search space state $\overline{s}$ (usually a minimum), is defined as the set of states that, taken as initial states, give origin to trajectories that end at point $\overline{s}$. The quantity $rBOA(\overline{s})$, defined as the ratio between the cardinality of $\mathcal{B}(\overline{s})$ and the search space size (assumed finite), is an estimation of the reachability of state $\overline{s}$. If the initial solution is chosen at random, the probability of finding a global optimum $s^*$ is exactly equal to $rBOA(s^*)$. Therefore, the higher is this ratio, the higher is the probability of success of the algorithm. The estimation of basins of attraction characteristics can help in the *a posteriori* analysis of local search performance, to provide explanations for the observed behavior. Moreover, it can also be useful for the *a priori* study of the most suitable models of a problem, for instance for comparing advantages and disadvantages of models that incorporate symmetry-breaking or implied

The development of a specific analyzer starts from the implementation of the interface `SearchSpaceAnalyzer` that declares the basic methods `analyze()`, for the actual analysis to be performed, and `report()`, defining the output of the analysis. The main goal of a BOA analyzer is to find the size of all, or a sample of, the local and global minima basins of attraction, corresponding to the execution of a given (deterministic) algorithm $\mathcal{A}$. Therefore, a BOA analyzer must be fed with problem instance and search algorithm and its task is to scan the search space for finding attractors and their basins. The procedure of search space scanning can be implemented in several ways, and it could primarily be either an exhaustive enumeration or a sampling. Attractors and their basins can be then computed by running algorithm $\mathcal{A}$ from every possible initial state $s$, returned by the scan method, till the corresponding attractor.The main parts of the `analyze()` method for the `BOA` class are as detailed in Listing 1.

Listing 1: The `analyze()` method for the `BOA` class.

```
void BOA::analyze(){
  BOAData data;
  initializeAnalysis(); // loads instance and solver
  StateDescriptor state = scanSpace();
  while (state.isValid()) // while there are feasible states
  { const Result& result = solver.run(state);
    updateBOAInfo(result.getStateDescriptor());
    state = scanSpace();}}
```

The BOA analyzer is designed through the *Template Method* pattern, that enables the designer to define a class that delegates the implementation of some methods to the subclasses. In this case, the implementation of the method `scanSpace()` is left to the subclasses, so as to make it possible to implement a variety of different search space scanning procedures, such as enumeration and uniform sampling. These methods rely on `StateManagerAdapter` for enumeration and random sampling of the search space, respectively.

In an analogous way, EasyAnalyzer implements classes for performing *position type* analysis [10], i.e., classify each state as (strict) local minimum/-maximum, plateau, slope or ledge as a function of the cost of its neighbors.

### 3.2 `MultiSolverAnalyzer`

EasyAnalyzer includes also a set of analyzers that manage a set of SLS solvers and whose aim is to perform a comparative analysis among different solvers.

We have developed the set of classes that implement the Race approach by [2]. This procedure aims at selecting the parameters of a SLS algorithm by testing each candidate configuration on a set of trials. The configurations that perform poorly are discarded and not tested anymore as soon as sufficient statistical evidence against them is collected.

In order to perform the analysis, the user must specify a set of solvers that are going to be compared in the Race and a set of instances on which the solvers will be run.

We present here the method `analyze()` of the class `Race` (Listing 2). The method works in a loop that evaluates the behavior of the configurations on an instance and collects statistical evidence about them. We would like to remark that our implementation follows the lines of the R package [1].

Listing 2: The `analyze()` method for the class `Race`.

```
void Race::analyze()
{ initializeAnalysis(); // loads instances, solvers and
                        // sets up the set of aliveSolvers
  replicate = 0;
  do
  { performReplicate(instances[replicate % instances.size()],
      replicate);
```

```
    if (replicate >= min_replicates)
    // the test is performed only after a minimum number of
        replicates
    { TestResult res = statisticalTest(seq(0, replicate),
        aliveSolvers, conf_level);
      updateAliveSolvers(res.survived);
      statistics[replicate] = res.statistic;
      p_values[replicate] = res.p_value;
    }
    replicate++;
  }
  while (aliveSolvers.size()>1 && replicate<max_replicates);
}
```

The class `Race` makes use of the *Template Method* pattern: the selection algorithm relies on the implementation of the abstract `statisticalTest()` method, which is implemented in two different sub-classes for the Student's $t$-test (`TRace`) and the Friedman's test (`FRace`).

# 4 Conclusions

We have presented EasyAnalyzer, a software tool for the principled experimental analysis of SLS algorithms. The tool is very general and can be used across a variety of problems with a very limited human effort. In its final version, it will be able to interface natively with a number of development environment, whereas in its current form it is interfaced with EasyLocal++, but also with any solver at the price of configuring a command-line interface. The design of EasyAnalyzer deliberately separates the problem-/implementation-specific aspects from the analysis procedures. This allows us, for example, to (re)use directly new analyses classes —developed at the framework level— by applying them to all the solvers for which a Solver interface already exists. We believe that our attempt to define such an environment can be regarded as an initial step toward engineering the experimental analysis of SLS algorithms.

# References


[1] M. Birattari. The race package for R. racing methods for the selection of the best. Technical Report TR/IRIDIA/2003-37, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2003.

[2] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 11–18, New York (NY), USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[3] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[4] L. Di Gaspero, A. Roli, and A. Schaerf. EasyAnalyzer: An object-oriented framework for the experimental analysis of stochastic local search algorithms. In *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics*, volume 4638 of *Lecture Notes in Computer Science*. Springer–Verlag, 2007.

[5] L. Di Gaspero and A. Schaerf. Writing local search algorithms using EasyLocal++. In *Optimization Software Class Libraries*. Kluwer Academic Publishers, 2002.

[6] L. Di Gaspero and A. Schaerf. EasyLocal++: An object-oriented framework for flexible design of local search algorithms. *Software—Practice and Experience*, 33(8):733–765, 2003.

[7] C. Fonlupt, D. Robilliard, P. Preux, and E.-G. Talbi. Fitness landscapes and performance of metaheuristic. In S. Voß, S. Martello, I. Osman, and C. Roucairol, editors, *Metaheuristics – Advances and Trends in Local Search Paradigms for Optimization,*, chapter 18, pages 255–266. Kluwer Academic Publishers, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing, Reading (MA), USA, 1995.

[9] S. Halim, R. Yap, and H.C. Lau. Viz: a visual analysis suite for explaining local search behavior. In *Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06)*, pages 57–66, New York (NY), USA, 2006. ACM Press.

[10] H.H. Hoos and T. Stützle. *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann Publishers, San Francisco (CA), USA, 2005.

[11] H.C. Lau, W.C. Wan, M.K. Lim, and S. Halim. A development framework for rapid meta-heuristics hybridization. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 362–367, 2004.

[12] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc., Sebastopol (CA), USA, 2005.

[13] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, Cambridge (MA), USA, 2005.