Chapter 7

The EvA2 Optimization Framework

The following chapter gives a rough sketch of the EvA2 framework, which was extended and used throughout this work. EvA2 (an Evolutionary Algorithms workbench, revised version 2) is a comprehensive heuristic optimization framework with emphasis on evolutionary algorithms implemented in Java (88; 90). It is a revised version of the JavaEvA (168) optimization toolbox, which has been developed as a resumption of the former EvA software package (181).

EvA2 integrates several derivative free optimization methods while concentrating on population-based sampling methods, such as evolution strategies, genetic algorithms, differential evolution, particle swarm optimization, as well as classical techniques such as multistart hill climbing or simulated annealing.

Through a modular structure, sampling operators are easily exchangeable between the methods, and the combination and hybridization of strategies is simplified. This allows the simple combination of various methods, for example assembling different local search methods with population-based sampling approaches as in memetic algorithms (see Sec. 2.4.1).

Furthermore, the uniformity of the framework allows for the simple comparison of various optimization strategies, which can resort to a large set of common benchmark functions and real-world applications and a statistics module for evaluation.

7.1 The General Structure

Figure 7.1 displays a sketch of the basic structure of the EvA2 framework, which splits functionality by two aspects: a logical classification in a Java package tree and a distinction between the central functionality and less frequently used add-on modules. All modules comply to the Java package tree. The EvA2 core is contained in the central EvA2Base module with all basic classes and functionality. It is divided into a client and a server part.

The EvA2 client implements a rich GUI interface for the manual configuration of optimization runs. The optimization core lies in the EvA2 server package, which contains classes representing generic target functions (problems), solution representations

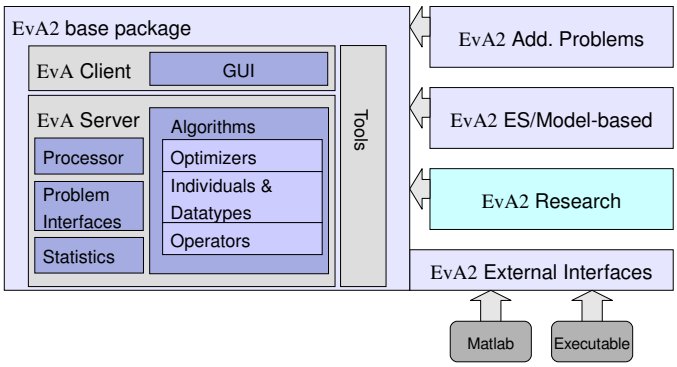

Figure 7.1: Sketch of the EvA2 architecture within the base package and with extending packages.

(individuals), sample sets (populations), and heuristic operators and search strategies (optimizers) which implement direct search on generic problem instances.

There are three additional modules, namely the problems pack EvA2Probs, which contains a large number of additional benchmark and application problems, e.g., the ones treated in Sec. 6.6. The EvA2ESModel pack contains additional functionality for model-based optimization and an alternative ES implementation. Any additional module can be loaded by simply adding it to the Java class path. EvA2 then automatically detects the additional classes and presents them in the GUI. Classes from the add-on modules have the same status as the core classes and may access the full functionality.

While the three modules base, problems, and ESModel are publicly available both in source and binary versions under the LGPL[1], the EvA2Research module contains non-disclosed implementations, such as current ongoing research and classes developed in industrial co-operations. Besides that, the research branch follows the same logical structure and builds upon the base functionality provided to the public.

Through external interfaces of the base package, additional platforms such as Matlab or arbitrary executables are supported as well, see Sec. 7.2.8.

Figure 7.2 illustrates the information flow within an EvA2 optimization run, which is steered by the main Processor class. For an optimization to run, three main components must be defined: the optimization strategy, the problem instance and a termination criterion. These components are accessible from the GUI and available for the Processor class. A set of post-processing parameters may be additionally defined. Post-processing can be useful to remove redundancy from a larger solution set through clustering, or to refine solutions using local search.

Algorithm 10 delineates the main loop of a Processor instance, which contains the key elements of Alg. 2. Because the representation of a solution is strongly connected to the problem at hand, the individual representation and the initialization method depend on

[1] http://www.gnu.org/licenses/lgpl.html

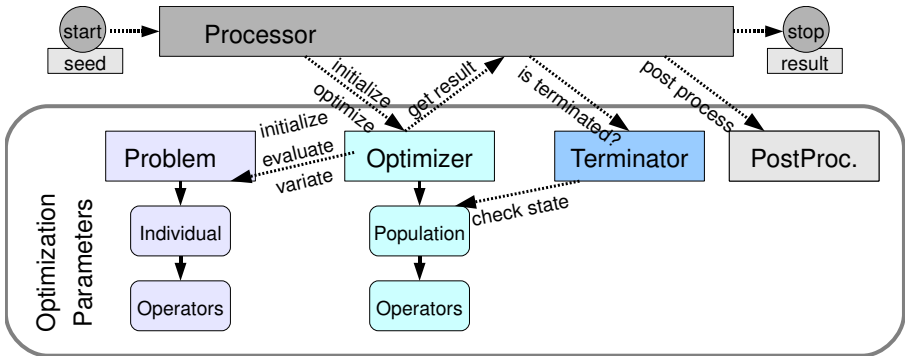

Figure 7.2: Illustration of the optimization components and control flow in EvA2.

the problem (see Fig. 7.2). Within the main loop, the optimizer is called repeatedly and does its work by resampling and reevaluating new candidate solutions. After any iteration, the termination criterion is checked, which may depend on the current population, an archived population of identified solutions, or both.

After each iteration, the statistics instance is updated and may read, process, and log information collected from the the current population, or even from the optimizer or the problem instance if they provide the appropriate interface (see Sec. 7.2.4). The next section sheds more light on the key features and functionality of EvA2.

7.2 Features of EvA2

Basic to any optimization application is the choice of the solution representation. Many problems have a natural representation within a binary, real-valued or integer-valued space. EvA2 thus features a range of standard data types and operators applicable to these basic representations, namely binary individuals, real-valued individuals, and integer-valued individuals. Typical operators are variation mechanisms, e.g., inspired by genetics, such as binary mutation and crossover. The exemplary evolutionary operators in Sec. 2.3.5 are implemented as well. An overview is presented in Fig. 7.3.

7.2.1 Generic Problem Classes

Using the individual classes that encapsulate data types and represent candidate solutions, generic problem classes are provided with EvA2. This makes it easy to implement a target function if the basic representation is a typical binary, real-valued or integer data type. Basically only one function needs to be implemented, which performs the evaluation of a single data array delivered from EvA2. All additional functionality, such as random initialization, variation and statistics are provided by the framework.

Algorithm 11 presents a minimal function implementation extending the basic real-valued problem class `AbstractProblemDouble` and realizing a modulated hyper-parabola

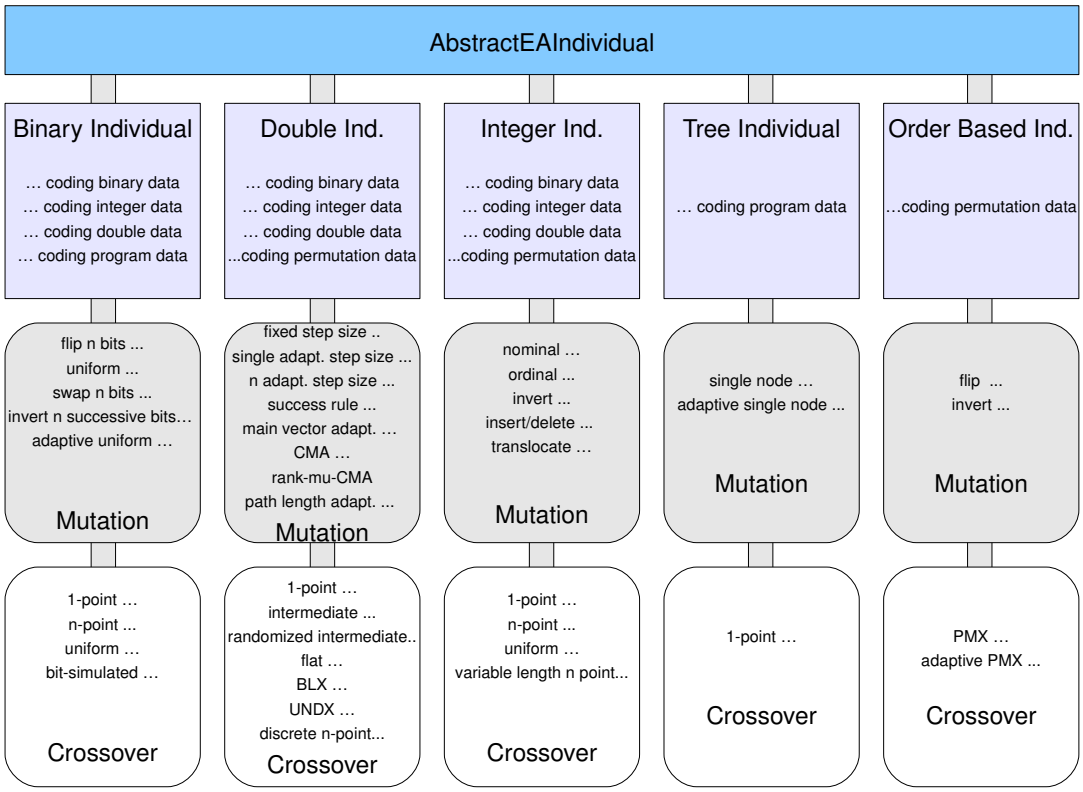

Figure 7.3: Overview over EvA2 data types and operators (not exhaustive).

Algorithm 10 Pseudo code of the main processing loop in EvA2.

```
problem.initialize
begin optimizer.initialize
  initialize params
  problem.initialize(P)
  problem.evaluate(P)
end optimizer.initialize
terminator.initialize(P)
do
  begin optimizer.optimize
    // details are strategy-dependent
    P' := selectSeeds(P)
    P'' := sampleNeighborhood(P', params) // apply variation
    problem.evaluate(P'')
    adapt optimizer params
    P := P''
  end optimizer.optimize
  statistics.log(P, optimizer, problem)
while (not terminator.isTerminated(optimizer.getArchive,
                                   optimizer.getCurrentPopulation))
if (post-processing is active)
  postProcess(P, statistics as listener)
end if
```

function f_{Rs} (Rastrigin's, see Tab. 2.1 on page 16). This works analogously for binary-valued and integer-valued problems. A large number of benchmark functions is already provided with EvA2, which counts 38 basic classes in the base pack and more than one hundred if the problems package is included.

7.2.2 Optimization Strategies

Because of the generic structure of EvA2, a large set of optimization strategies have been implemented. The most popular methods are listed in Tab. 7.1. Because some optimization strategies are defined explicitly for specific data types, they are applicable only for compatible individual classes.

Genetic programming is one example, as it requires individuals representing programs. Program representations may be based on binary arrays or tree structures (see Fig. 7.3), where the binary data is decoded using a technique called "grammatical evolution" (145).

Differential evolution and particle swarm optimization, on the other hand, rely on real-valued data types, which are naturally represented in a real-valued form. However, EvA2 allows to use binary or integer representations with a real-valued encoding step as well.

All optimization strategies are easily accessible from both the GUI and direct API usage. An API example is given in Sec. 7.3, while the next section outlines some of the GUI features provided by EvA2.

Algorithm 11 Exemplary code implementing f_{Rs} in EvA2 (see Tab. 2.1).

```
1  public class RastriginsProblem extends AbstractProblemDouble {
2    private double a=10.;
3    private double w=2*Math.PI;
4    // useful constructors
5    public RastriginsProblem() {};
6    public RastriginsProblem(int dim) { super(dim); };
7    public RastriginsProblem(RastriginsProblem o) {
8      super.cloneObjects(o);
9      this.a=o.a;
10     this.w=o.w;
11   }
12   public Object clone() {
13     return (Object) new RastriginsProblem(this);
14   }
15   // calculating the target function for a given vector x
16   public double[] eval(double[] x) {
17     double[] result = new double[1]; // output dimension is 1
18     result[0]=a*x.length;
19     for (int i = 0; i < x.length; i++)
20       result[0]  += x[i]*x[i] - a*Math.cos(w*x[i]);
21     return result;
22   }
23   // methods for the GUI
24   public void setValA(double newA) { this.a=newA; }
25   public double getValA() { return a; }
26   public void setValW(double newW) { this.w=newW; }
27   public double getValW() { return w; }
28   public static String globalInfo() { return "The␣Rastrigin's␣
       Problem␣in␣EvA2"; }
29 }
```

Table 7.1: List of some popular optimization strategies implemented in EvA2.

- Evolution Strategies, various mutation operators, CMA-ES, IPOP-ES, rank-μ-CM-ES
- Genetic Algorithms with various codings, Order-based GA
- Particle Swarm Optimization (constricted, dynamic, various topologies, Tribes, CGPSO)
- Genetic Programming, Evolutionary Programming
- Differential Evolution
- Scatter Search
- CHC Adaptive Search
- Cluster-based niching EA, CBN-GA, CBN-ES, CBN-DE
- CBN-PSO, periodic CBN-PSO
- NichePSO, ANPSO, Island-model EA
- Multiobjective EA (NSGA II,PESA II,SPEA II)
- Memetic Algorithms
- Gradient Descent
- Simulated Annealing, Nelder-Mead-Simplex
- Population-based Incremental Learning

7.2.3 GUI and Reflection

The graphical interface of EvA2 is built from standard Java-Swing components and makes heavy use of Java Reflection to produce generic GUI objects for any accessible object instance. Specifically, this means that the programmer does not need to bother with Java GUI programming to make his own classes accessible from the GUI. EvA2 automatically detects getter and setter methods of new classes and creates graphical elements according to their types, which dramatically reduces development times.

An example GUI element for the Rastrigin's implementation in Alg. 11 is shown in Fig. 7.4. Note that the elements "valA" and "valW" have been automatically added because the programmer defined the methods `getValA`, `setValA`, `getValW`, and `setValW` in Alg. 11. Thereby, the problem parameters can be directly altered during runtime, which simplifies experiments. The info text has been taken from the `globalInfo` method by default. All other GUI fields are inherited from `AbstractProblemDouble`, and their meaning is explained by tool tips.

Note that the automatic GUI representation is not restricted to simple data types such as numbers or booleans (e.g., the fields "doRotation" or "withConstraints" in Fig. 7.4). The EvA2 GUI provides specific GUI elements for enumerations, strings, and arrays. Complex data types are handled recursively by the generic editor. The template individual which defines the solution data type is represented by its own GUI element. That GUI element is constructed as soon as the entry "EAIndividual" is clicked on in Fig. 7.4.

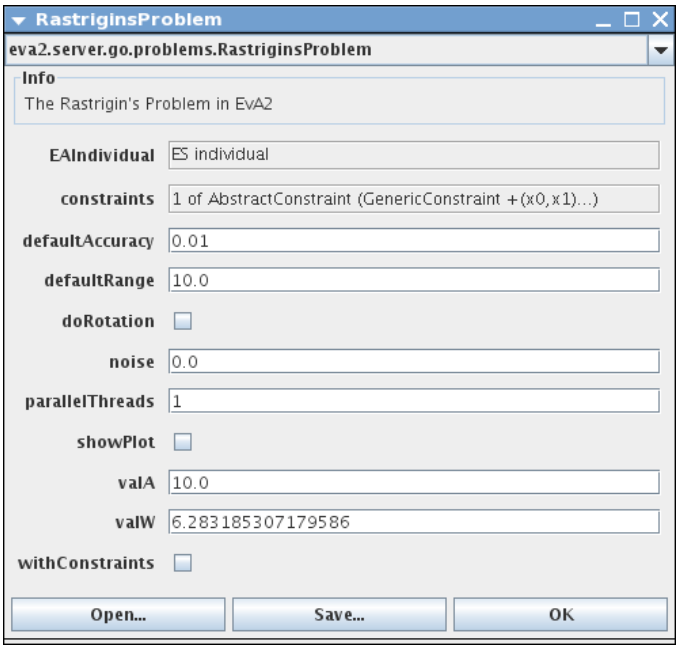

Figure 7.4: Example GUI element produced automatically by EvA2 for the problem implementation in Alg. 11.

The same holds for the "constraints" array in the example screenshot.

Thus, by following some simple design rules, the use of Java Reflection enables very quick GUI access. As in the given example in Alg. 11 and Fig. 7.4, only the getter and setter methods for a parameter must be defined to make it accessible in a generic way, such as pairs of `getValW`, `setValW` for a parameter *w*. This provides for a simple way to experiment and test new implementations under various conditions within the EvA2 framework.

7.2.4 Reflective Statistics

To ease the evaluation of optimization results, the statistics part of EvA2 makes use of reflection as well. This allows to visualize data produced internally during an optimization run and observe distribution and averaged values. By default, information on the best, worst and mean fitness in the population is displayed over time. Further built-in data fields include the information on feasibility of individual (in case of constrained optimization) as well as average, minimal, and maximal distances within the population.

When a problem or optimizer class implements a generic interface tagging it as a statistics informer, it may report any online data, which is collected by the statistics module of EvA2 and treated in the same way as the built-in data fields.

Algorithm 12 presents an example of this mechanism which allows the mean particle

Algorithm 12 Additions to allow the statistics class to keep track of additional data, e.g., the speed of PSO particles.

```
1  //... within the PSO class
2  public String[] getAdditionalDataHeader() { // name of the data field
3    return new String[]{"meanCurSpeed"};
4  }
5  public Object[] getAdditionalDataValue(PopulationInterface pop) {
6    // calculate and return current value of the data field
7    return new Object[]{getPopAvgNormedVelocity((Population) pop)};
8  }
9  public String[] getAdditionalDataInfo() { // optional meta information
10   return new String[]{"The␣mean␣current␣speed␣of␣all␣particles␣
       relative␣to␣the␣problem␣range"};
11 }
```

velocities in PSO to be tracked. The additional data value is polled within EvA2 after each iteration of the optimizer. The statistics module adds a data graph for the mean velocity (annotated with "meanCurSpeed") and automatically tracks the averaged and final values across multiple runs. This mechanism again speeds up development times due to the simple visualization of arbitrary data gathered during optimization.

The user may also activate and deactivate the data visualization for any built-in or additional data field from within the GUI, where the String returned by `getAdditionalDataInfo` is used as a tool tip information for the user.

7.2.5 Adaptive Parameters

An additional strength of the EvA2 framework building on Java Reflection is the simple availability of dynamic and adaptive parameter control methods. As soon as a getter and setter method for a specific property *valW* is implemented, a time-dependent control strategy can automatically read and write the parameter value by calling `get-` and `setValW` as in the last section.

An exponential variation of any parameter, for example, can therefore be implemented by a single class, `ExponentialDecayAdaption`, which takes as arguments the initial value v_{init} and a halving time $T_{1/2}$ as well as the name of the member variable, in this case, "valW". The EvA2 framework then assures that the adaptive parameter instance is notified at the start of an optimization run and after each iteration, and the current number of evaluations is forwarded. By triggering the setter-function "setValW" of the reference object, it will then produce the desired, dynamic behavior.

In a similar way, a sinusoidal adaption or linear adaption is possible. In the latter case, the overall runtime needs to be known beforehand (see Sec. 7.2.6). The dynamic parameter settings for the CBN-PSO variants in Chap. 6 have been realized in this way.

Furthermore, adaptive parameters may depend on the state of the optimizer itself or

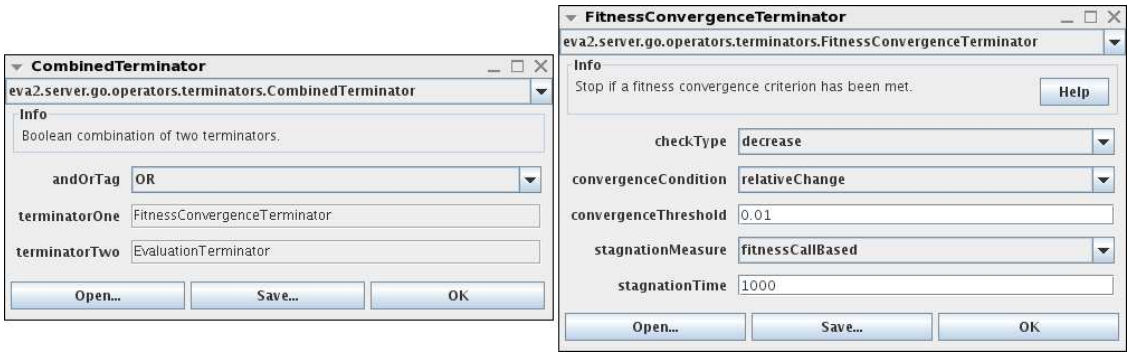

Figure 7.5: Screenshot of the `CombinationTerminator` (left) which realizes logical combinations of other terminators such as an `EvaluationTerminator` and the `FitnessConvergenceTerminator` (right).

of the population. As an example, this is the case for a diversity-driven adaption of the inertness parameter in PSO (188), which has been implemented in EvA2 within the class `PSOActivityFeedbackControl`. It is based on the same mechanism as the generic sinusoidal or linear adaption mechanisms and controls PSO through a swarm activity measure.

7.2.6 Terminators

Besides the problem instance and the optimizing strategy, a third component remains to be defined before an optimization run can be started with EvA2, namely the termination criterion. A terminator is an object which defines the stopping criterion for the optimization, which may depend on the number of function evaluations, the number of iterations, or a population measure.

In realistic applications, a maximal number of allowed function evaluations is often defined. An instance of the `EvaluationTerminator` class, which takes an integer as an upper limit for function calls, is appropriate in such a case. However, in realistic applications, it may be preferable to run the optimization only as long as there is actually improvement, so as to avoid wasting time on a converged or stagnated optimizer state.

To this end, EvA2 provides several flexible types of terminators as well as the possibility to combine them logically. In a typical use case, the optimization run should be stopped either if the population has not improved for a certain time period or if a maximal number of function calls have been performed.

Using two specific terminators and combining them in a logical OR using the `CombinedTerminator`, this can be achieved easily. Figure 7.5 shows a screenshot of the example, where the `FitnessConvergenceTerminator` is set up to stop as soon as the best fitness has not improved by more than 1% for at least 1000 fitness evaluations. This principle can also be used in a nested way, so that quite complex termination criteria can be

Table 7.2: List of terminators available in EvA2.

- `CombinedTerminator`
- `EvaluationTerminator`
- `DiversityTerminator`
- `EvaluationTerminator`
- `FitnessConvergenceTerminator`
- `FitnessValueTerminator`
- `GenerationTerminator`
- `KnownOptimaFoundTerminator`
- `ParetoFrontMetricTerminator`
- `PhenotypeConvergenceTerminator`

defined. Tab. 7.2 gives a short list of implemented terminator classes.

7.2.7 Metrics

Several of the listed terminators are based on a property of the population, e.g., the best fitness, the best solution representation, or the average distance among individuals. These are based on a common abstract class terminating if a certain population measure has stagnated.

Measuring population properties is one application of abstract metrics implemented in EvA2. Distances between candidate solutions are an important concept for development and evaluation in general, and for niching methods specifically.

In the typical case, the individual distance is measured based on the data representation, e.g., using the euclidean distance for real-valued individuals or the Hamming-distance for binary individuals. This principle is implemented in the `PhenotypeDistanceMetric` used by default in EvA2.

Besides, the distance measure may be based on the genotypic representation if a coding step is involved. Moreover, a distance in objective space can be considered using the `ObjectiveSpaceMetric`. It measures the distance between target function values $f(x)$ and $f(y)$ for two candidate solutions x and y.

The special case of `IndividualDataMetric` calculates the distance between candidate solutions based on an arbitrary annotation accessible by a key identifier. This is useful for clustering PSO particles based on their historically best positions, for example, but is applicable analogously for any similar purpose, e.g., to compare individuals based on the degree of local optimality within a memetic algorithm.

As described in Sec. 6.6.2, the `DoubleIntegralMetric` implements the integral distance between irrigation schedules and is used as a problem specific distance measure. All of these distance metric implementations can be used exchangeably within algorithms for clustering or for evaluation purposes.

7.2.8 The Matlab Interface

During the work on EvA2 and accompanying projects, it has become apparent that the Matlab mathematical programming environment is frequently used in several problem fields, e.g., in bioinformatics or in industrial engineering. Due to the lack of both powerful and easily available optimization tools within Matlab, an interface for EvA2 has been implemented.

Although Java calls are supported directly from Matlab, this functionality is widely undocumented and required extensive testing. Finally, a Matlab object definition has been created which encapsulates the functionality to set up an EvA2 optimization run from Matlab through a Matlab function handle. This allows to optimize an arbitrary Matlab m-function which follows a simple template, returning a single real valued result array for any given candidate solution x and an optional parameter p.

Algorithm 13 Example m-function for optimization from Matlab

```
function z = exampleFun(x, p)
switch p
  case 1  % hyper parabola
    z(1)=sum(x.*x);
  case 2  % Himmelblau function
    z(1) = ((x(1)^2 + x(2) - 11)^2 + (x(1) + x(2)^2 - 7)^2);
  case 3  % simple binary function
    z(1)=abs(bitxor(x(1)+x(2), bin2dec('11001100')));
end
```

Algorithm 14 Optimization from the Matlab console

```
javaaddpath '/home/username/workspace/EvA2Base.jar'
R=[[-5 -5]; [5 5]];
JI=JEInterface(@exampleFun, R, 2);
JI=optimize(JI,4);
[sol, solFit]=getResult(JI);
```

Algorithm 13 lists an exemplary Matlab m-File that could be used as a target function with the EvA2 Matlab interface. Besides the candidate solution vector x, an arbitrary parameter field p is handed over. In this case, p is used as a switch between benchmark functions, however it could be used in any other way or not used at all. EvA2 will not introspect p but just forward it to the target function.

To optimize the Himmelblau function in $R = [-5,5]^2$ using Particle Swarm Optimization, it suffices to execute the lines of code listed in Alg. 14. Besides defining the search range and constructing the EvA2-Interface object 'JI', the function argument 2 addresses

the Himmelblau function (see Alg. 13) and the `optimize` call in line 4 takes an identifier of the optimization strategy (in the example 4 for PSO). All available optimization identifiers can be printed by a call to `showOptimizers(JI)`.

Besides additional options to configure the termination criterion, specific properties of the optimizers can be set. This is implemented using Java Reflection again: Any property which is accessible through a generic setter method can be addressed by adding an option pair ('identifier', value) to an optimization call. To exemplary start a (5,15)-ES, the call in line 4 would look as follows:

```
JE=optimizeWith(JE, 1, 'mu', 5, 'lambda', 15)
```

Therein, 1 is the ID of the evolution strategy technique within the Matlab interface. Any property that is accessible from the GUI can thereby be specified directly from the Matlab command line.

7.3 API Usage

To illustrate how the EvA2 API can be applied, a simple example is presented in Alg. 15. To optimize the 15-D real-valued function `RastriginsProblem` given in Alg. 11 (f_{Rs}, see Tab. 2.1 on page 16), a `Processor` instance is set up by the `OptimizerFactory` class. Setting the terminator to stop after 10,000 evaluations and indicating a standard PSO as the optimizing strategy, the run is started and directly returns the real-valued solution vector. The solution may be processed further or directly printed as in the example in Alg. 15.

Algorithm 15 Simple `OptimizerFactory` usage example optimizing f_{HP}.

```
1  import eva2.OptimizerFactory;
2  import eva2.server.go.problems.RastriginsProblem;
3
4  public class TestingFRsPso {
5    public static void main(String[] args) {
6      RastriginsProblem fRs = new RastriginsProblem(15);
7      OptimizerFactory.setTerminator(new
           EvaluationTerminator(10000));
8      double[] sol =
           OptimizerFactory.optimizeToDouble(OptimizerFactory.PSO,
           fRs, "eva2log.dat");
9      System.out.println("Found solution: " +
           BeanInspector.toString(sol));
10   };
11 }
```

Depending on the application, the user is able to configure any setting of the optimization run, be it by using different terminators, or by retrieving the PSO object from the factory and altering PSO parameters using the default getter and setter methods.

7.4 Summary

For a short introduction on the EvA2 optimization framework, this chapter outlined the basic architecture, several key features, and usage examples. The core functionality of EvA2 is contained in a single Java module defining a client-server-based package structure. Any extending modules maintain the same logical structure and can be loaded simply by adding them to the Java class path.

The object-oriented architecture allows to switch search operators between strategies in a straight-forward manner, however some care must be taken with optimization strategies which are specifically defined for certain data types.

Generic problem classes are predefined and allow to add own problem implementations, which can then be treated by the various data types and optimization strategies already implemented in EvA2. By making use of the powerful Java Reflection API, EvA2 generates GUI elements for newly implemented classes automatically.

The applicant does not have to care about GUI elements and can concentrate on the developmental process, which is supported by the ability to directly test and modify new algorithms through the generic GUI. A similar automation is available for the logging and visualization of additional statistical data fields, which can be defined through a simple interface. The additional data are tracked by EvA2 in a transparent manner and treated equally to the data collected internally, which is useful for both evaluation and development.

Besides a mechanism to realize generic self-adaptive parameters using reflection, the strengths of various predefined termination criteria and distance metrics have been sketched. Using the EvA2 Matlab interface, the direct application on optimization problems given in Matlab is possible, besides the typical use with the GUI or Java API.

The core functionality of EvA2 is freely available under the LGPL license and can therefore be used in both research and industrial co-operations. Publications in diverse fields such as robotics (69; 91; 92), bioinformatics (47; 45; 187; 46; 163), geo- and hydroinformatics (37; 10; 36; 8) have recently relied on EvA2 as a developmental or applicative platform. And beyond student teaching and research projects, several industrial co-operations were carried out with EvA2, such as the improvement of excavator design with multiple criteria, the optimization of automatic transition control in trucks for fuel efficiency, and the optimization of cylinder firing patterns in combustion engines.