Simplifying Problem Definitions in the HeuristicLab Optimization Environment

Andreas Scheibenpflug[1,2], Andreas Beham[1,2],Michael Kommenda[1,2],
Johannes Karder[1], Stefan Wagner[1], Michael Affenzeller[1,2]

[1]University of Applied Sciences Upper Austria
Heuristic and Evolutionary Algorithms Laboratory
Softwarepark 11, 4232 Hagenberg, Austria

[2]Johannes Kepler University Linz
Institute for Formal Models and Verification
Altenberger Strasse 69, 4040 Linz, Austria

{ascheibe, abeham, mkommend,
jkarder, swagner, maffenze}@heuristiclab.com

ABSTRACT

Software frameworks for metaheuristic optimization take the burden off researchers and practitioners to start from scratch and implement their own algorithms and problems. One such framework is HeuristicLab. While it allows using existing, already implemented algorithms and problems comfortably and provides an extensive range of tools for analyzing results, it lacks an easy to use programming interface for adding new problems. As implementing new problems is a common task, an improved and simpler problem definition interface has been created. Besides giving an overview of the implementation, we also show examples of problems built using this new interface. Additionally, we compare the new approach to three other metaheuristic frameworks. This is done by analyzing the source code of the OneMax problem implemented in each framework and comparing the resulting lines of code with previous works.

Categories and Subject Descriptors

I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

Keywords

HeuristicLab; Evolutionary Computation Frameworks; Metaheuristic Optimization Frameworks; Scripting

1. INTRODUCTION

Defining and implementing new problems is a frequent task for researchers and practitioners in the field of optimization and metaheuristics. HeuristicLab (HL)[1] [9] is a software environment primarily used for heuristic optimization and data analysis. Because of its graphical user interface (GUI) and analysis functionality it is a convenient tool for research and education. Besides that, HeuristicLab is not only used in scientific projects, but also in projects where real-world problems of business and industry are tackled. For such often time critical tasks, it would be beneficial to provide an easy way to design and develop new problems.

Implementing problems in HeuristicLab requires a deep understanding of its architecture as well as conventions and patterns used throughout the framework. HeuristicLab is focused on providing a rich user experience. For example, a user working with HeuristicLab can configure most of the data and objects that one sees in the UI, including problems. Therefore, code has to be written that reacts to such user interactions.

Problems are constructed of various different operators and components to ensure flexibility and modularity. However, this requires boiler plate code to be written that hinders quick implementation of new problems or ideas [7]. Therefore, a new and improved application programming interface (API) has been designed and integrated into HeuristicLab that allows to define problems more easily without any previous knowledge about HL.

In the following, we first show a survey of three different metaheuristic frameworks. Section 3 describes the standard application programming interface for implementing problems in HeuristicLab. We then describe how the new problem definition API works and how creating new problems is simplified. The usage of the new API is shown with two example implementations of popular benchmark problems. In the last part we compare our new approach to other metaheuristic and optimization frameworks and show its viability.

[1]http://dev.heuristiclab.com

GECCO '15, July 11 - 15, 2015, Madrid, Spain

ISBN 978-1-4503-3488-4/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2739482.2768463

2. SURVEY

This section gives an overview of how other metaheuristic software frameworks, namely ECJ, ParadisEO and DEAP, allow to add new problem implementations. For each framework, we show the code necessary to implement the OneMax problem. OneMax is a simple binary benchmark problem where the goal is to generate a bit string filled with only ones. Additionally, the steps required to execute a Genetic Algorithm with the newly implemented problem are described. Because each framework already includes the OneMax problem, the listings show these implementations.

ECJ

ECJ[2] [10] is a metaheuristic software framework developed at George Mason University's Evolutionary Computation Laboratory and written in the Java programming language. It supports a range of different vector-based solution representations as well as tree structures for genetic programming. ECJ implements mainly evolutionary metaheuristics like evolution strategies, genetic algorithms, coevolution and multi-objective algorithms like NSGA-II and SPEA2 but also includes an implementation of particle swarm optimization. Additionally, ECJ provides some already implemented problems like the SAT, lawnmower, NK landscapes, HIFF, regression and artificial ant. One of the biggest differences between ECJ and HeuristicLab is that, while configuration in HL is done mainly in the graphical user interface, algorithms and problems in ECJ are solely configured through a parameter file. This file is loaded at runtime and then used to instantiate and configure the desired components.

ECJ comes with various samples. Listing 1 shows the source code of the OneMax problem as implemented in ECJ.

```
1  public class MaxOnes extends Problem
2  implements SimpleProblemForm
3  {
4   public void evaluate(final EvolutionState
        state, final Individual ind, final int
         subpopulation, final int threadnum) {
5    if (ind.evaluated) return;
6    if (!(ind instanceof BitVectorIndividual)
        )
7     state.output.fatal("Whoa!  It's not a
         BitVectorIndividual!!!",null);
8
9    int sum=0;
10   BitVectorIndividual ind2 = (
        BitVectorIndividual)ind;
11   for(int x=0; x<ind2.genome.length; x++)
12    sum += (ind2.genome[x] ? 1 : 0);
13
14   if (!(ind2.fitness instanceof
        SimpleFitness))
15    state.output.fatal("Whoa!  It's not a
         SimpleFitness!!!",null);
16   ((SimpleFitness)ind2.fitness).setFitness(
        state, (float)(((double)sum)/ind2.
        genome.length), sum == ind2.genome.
        length);
17   ind2.evaluated = true;
18  }
19 }
```

Listing 1: Code for the OneMax problem from the ECJ tutorial section (for brevity without imports and package declaration)

Problems in ECJ have to inherit from `Problem` which states that problems have to implement a cloning and an evaluation mechanism. How evaluation is specified concretely is defined by problem forms. Therefore the OneMax problem in ECJ implements `SimpleProblemForm` that defines an evaluation method which gets a state (containing current generation, the current population, the breeder,...), the individual to evaluate, the index of the subpopulation from where the individual was taken and the id of the thread where the evaluation takes place. The reason for the last two parameters is that there may be multiple populations (for example in an island model) that can be spread over multiple processes or computers. In the evaluation function the quality is calculated and the result stored in the individual.

This problem can now be used with e.g., a genetic algorithm. Configuration is done in the parameter file, which gets loaded by ECJ. Listing 2 shows an excerpt where parts of the algorithm configuration and the selection of the problem type is displayed.

```
1  breedthreads  = 1
2  evalthreads = 1
3  state   = ec.simple.SimpleEvolutionState
4  pop   = ec.Population
5  init    = ec.simple.SimpleInitializer
6  finish    = ec.simple.SimpleFinisher
7  breed   = ec.simple.SimpleBreeder
8  eval    = ec.simple.SimpleEvaluator
9  stat    = ec.simple.SimpleStatistics
10 exch    = ec.simple.SimpleExchanger
11 generations   = 200
12 quit-on-run-complete  = true
13 checkpoint    = false
14 prefix      = ec
15 pop.subpops   = 1
16 pop.subpop.0    = ec.Subpopulation
17 pop.subpop.0.size     = 10
18 pop.subpop.0.duplicate-retries  = 0
19 pop.subpop.0.species    = ec.vector.
       VectorSpecies
20 pop.subpop.0.species.fitness  = ec.simple.
       SimpleFitness
21 pop.subpop.0.species.ind  = ec.vector.
       BitVectorIndividual
22 pop.subpop.0.species.genome-size  = 20
23 pop.subpop.0.species.crossover-type = one
24 pop.subpop.0.species.crossover-prob = 1.0
25 pop.subpop.0.species.mutation-prob  = 0.01
26 pop.subpop.0.species.pipe     = ec.vector.
       breed.VectorMutationPipeline
27 pop.subpop.0.species.pipe.source.0    = ec.
       vector.breed.VectorCrossoverPipeline
28 pop.subpop.0.species.pipe.source.0.source.0
        = ec.select.TournamentSelection
29 pop.subpop.0.species.pipe.source.0.source.1
        = ec.select.TournamentSelection
30 select.tournament.size    = 2
31 eval.problem    = ec.app.tutorial1.MaxOnes
```

Listing 2: Parameter file for the configuration of the ECJ OneMax problem

ParadisEO

ParadisEO[3] [2] is a C++ software framework based on the EO [6] framework. While EO provides population-based metaheuristics (mainly EAs and GAs), ParadisEO adds support for trajectory-based metaheuristics (e.g., Tabu Search, Simulated Annealing and tools for fitness landscape analysis), multi-objective optimization (e.g., NSGA-II, IBEA, SPEA2) as well as hybrid, parallel and distributed metaheuristics. ParadisEO also comes with a wide range of different vector-based solution representations and also tree-based representations. There are some concrete problem implementations shipped with ParadisEO, like NK landscapes, QAP, SAT or Royal Road.

While HeuristicLab is mainly configured through its user interface and ECJ through its parameter file, ParadisEO takes more of a white-box approach where it mainly provides building blocks for constructing algorithms. Problem

[2]http://cs.gmu.edu/~eclab/projects/ecj/

[3]http://paradiseo.gforge.inria.fr/

implementation and configuration is done in C++ as Listing 3 shows. First the type of solution representation (`eoBit`) is defined. Evaluation of the individual is done with a function that takes such an individual, evaluates it and returns its quality value. In the main method the initial population is constructed by generating `POP_SIZE` individuals and randomly initializing each individual. Afterwards, all parameters and operators are configured and a new genetic algorithm is initialized and run on the population and problem as specified before.

```
1
2  typedef eoBit<double> Indi;
3
4  double binary_value(const Indi& _indi) {
5    double sum = 0;
6    for (int i = 0; i < _indi.size(); i++)
7      sum += _indi[i];
8    return sum;
9  }
10
11 int main()
12 {
13   const unsigned int VEC_SIZE = 8;
14   const unsigned int POP_SIZE = 20;
15   const unsigned int MAX_GEN = 500;
16   const float CROSS_RATE = 0.8;
17   const double P_MUT_PER_BIT = 0.01;
18   const float MUT_RATE = 1.0;
19
20   eoEvalFuncPtr<Indi> eval(binary_value);
21
22   eoPop<Indi> pop;
23   for (int igeno=0;igeno<POP_SIZE;igeno++)
         {
24    Indi v;
25    for (int ivar=0;ivar<VEC_SIZE;ivar++) {
26     bool r = rng.flip();
27     v.push_back(r);
28    }
29    eval(v);
30    pop.push_back(v);
31   }
32
33   eoDetTournamentSelect<Indi> select(3);
34   eoGenContinue<Indi> continuator(MAX_GEN);
35   eoBitMutation<Indi> mutation(
         P_MUT_PER_BIT);
36   eo1PtBitXover<Indi> xover;
37
38   eoSGA<Indi> gga(select, xover, CROSS_RATE
         , mutation, MUT_RATE,
39          eval, continuator);
40   gga(pop);
41
42   pop.sort();
43   cout << "FINAL Population\n" << pop <<
         endl;
44 }
```

Listing 3: ParadisEOs' OneMax problem and algorithm configuration (taken from ParadisEOs tutorial section, without includes for brevity)

DEAP

DEAP[4] [3] is a metaheuristic software framework implemented in the Python programming language. It has a strong focus on simplicity and the ability to be used as a tool for rapid prototyping and testing of new ideas. The goal of DEAP is to provide a simple platform for creating custom algorithms and constructing EC systems. Nevertheless, DEAP comes with a range of already implemented algorithms like EA variants, CMA-ES, GP and multi-objective algorithms like NSGA-II and SPEA2. It also comes with examples of already implemented problems like the TSP, Knapsack, symbolic regression or artificial ant. Listing 4 shows an implementation for the OneMax problem [8]. The first two lines define the representation of a solution candidate. In contrast to the other frameworks, DEAP has no pre-defined implementations for solution representations. Instead, the representation is defined with the help of the `creator` class. This is possible due to the dynamic typing of the Python programming language and can be done based on any data structure such as arrays, lists, dictionaries etc. The `toolbox` is used as a configuration repository where key value pairs of operators and settings can be stored. This toolbox is then used to parameterize a new EA.

```
1  creator.create("FitnessMax", base.Fitness,
       weights=(1.0,))
2  creator.create("Individual", array.array,
       typecode='b', fitness=creator.
       FitnessMax)
3
4  toolbox = base.Toolbox()
5  toolbox.register("attr_bool", random.
       randint, 0, 1)
6  toolbox.register("individual", tools.
       initRepeat, creator.Individual, toolbox
       .attr_bool, 10)
7  toolbox.register("population", tools.
       initRepeat, list, toolbox.individual)
8
9  def evalOneMax(individual):
10     return sum(individual)
11
12 toolbox.register("evaluate", evalOneMax)
13 toolbox.register("mate", tools.cxTwoPoint)
14 toolbox.register("mutate", tools.mutFlipBit
       , indpb=0.05)
15 toolbox.register("select", tools.
       selTournament, tournsize=3)
16
17 def main():
18     pop = toolbox.population(n=300)
19     hof = tools.HallOfFame(1)
20     stats = tools.Statistics(lambda ind:
           ind.fitness.values)
21     stats.register("avg", numpy.mean)
22     stats.register("std", numpy.std)
23     stats.register("min", numpy.min)
24     stats.register("max", numpy.max)
25
26     pop, log = algorithms.eaSimple(pop,
           toolbox, cxpb=0.5, mutpb=0.2, ngen
           =40, stats=stats, halloffame=hof,
           verbose=True)
27
28     return pop, log, hof
```

Listing 4: Excerpt from DEAP's OneMax problem and algorithm configuration (without imports)

Besides different ways to configure problems and algorithms as well as running them, there is also a difference in the output the frameworks provide. HeuristicLab usually generates a chart that shows quality curves over generations and saves the best found solution in the results. ECJ on the other hand prints out the best found solution by default while in ParadisEO, no information is shown by default besides the last generation which had to be manually coded.

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

DEAP is configured in the code to show the last generation as well as the best found solution and some statistics that have also been configured explicitly.

3. STANDARD PROBLEM DEFINITION INTERFACE IN HEURISTICLAB

Metaheuristics are problem-independent optimization methods. HeuristicLab follows this principle by defining an interface between algorithms and problems which allows coupling of almost all implemented algorithms with all problems.

Problems require a representation for solution candidates. HeuristicLab offers a range of predefined representations for solutions such as binary vectors, real vectors or permutations. This is called an encoding which is the representation of the solution candidates and the operators that work on these representations. Such operators are crossover and mutation operators used by evolutionary algorithms or move operators for trajectory-based metaheuristics. Furthermore, HeuristicLab supports algorithms that need other, more specialized operators, e.g., tabu-criteria for tabu search or particle creators for particle swarm optimization. If the encoding should be used with these algorithms, it has to provide such operators.

An operator that every encoding has to implement is a solution creator. Every algorithm needs an operator for creating solution candidates and therefore offering such an operator is mandatory. Similarly, a problem has to implement an evaluator which assigns a quality value to solution candidates. As fitness functions are problem-specific, it is mandatory for a problem to provide one.

A problem in HeuristicLab is defined with a solution creator and an evaluator which couples the encoding with the problem. A problem can additionally define problem-specific operators that make use of properties of the problem when manipulating solutions. An example of such an operator would be a specific path-relinker operator for the traveling salesman problem. Figure 1 gives an overview of how problems and encodings are organized in HeuristicLab.
Besides defining the used encoding and fitness function, problems also define if their objective function should be maximized or minimized, if it is a single- or multi-objective problem and how problem data can be loaded. HL usually offers benchmark instances for problems and supports loading of problem data from files. While this is quite a useful feature, it is optional as there are problem types that do not need to load data.

A problem is also responsible for discovering and configuring operators. Algorithms usually include placeholders that can be filled with encoding- or problem-specific operators (e.g., crossovers or mutators) that the user can select in the GUI. The problem offers an operator repository where algorithms can retrieve configured operators. These operators are typically all the operators from the encoding as well as the problem-specific operators. The problem can also exclude operators, e.g. when an encoding-specific operator may not work for a certain problem.

A problem has to react to user interaction. Because users can configure problems, the developer of the problem has to check and react to possible changes in configuration and also recover from possible faulty states. The code that has to be written to support user interaction is one of the disadvantages of the standard way of creating new problems in HL because it has to be implemented for every problem. Besides requiring developers to have a deep knowledge of HL, this also leads to code duplication. The next section shows how this disadvantage is overcome with the new API.

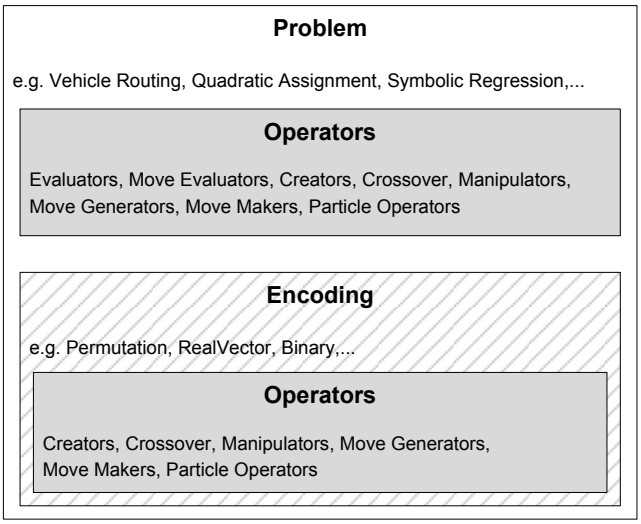

Figure 1: Architecture of problems, encodings and operators in HeuristicLab

4. NEW PROBLEM DEFINITION INTERFACE FOR HEURISTICLAB

The goal in creating a new API for problem definitions is to reduce the size of the source code required for creating a new problem. The API should be minimal and clear. Additionally, it should be easier for new developers to get started with HeuristicLab. HeuristicLab is a project with a large code base and many concepts and patterns implemented throughout the framework that one has to be familiar with to understand how HeuristicLab works. New developers are often overwhelmed by the size of the code base and the used concepts. Thus the new API should lower the burden for programmers new to HL.

In the following the changes and improvements that form the basis for the new API are described. They can be grouped into two categories:

- Removal of unnecessary code: This is achieved by removing most of the configuration code from the problems and moving them to the encodings where it can be used by multiple problems without code duplication.

- Hiding of framework concepts: HL uses various useful concepts like operators and scopes, but such concepts have the downside of increasing the learning curve of the framework. Therefore, specific concepts are concealed from the user and replaced by common known language features (e.g. interfaces that can be implemented instead of operators).

Automatically Configuring Encodings

One disadvantage of the standard problem definition is the duplication of code for configuring encoding-specific operators in every new problem.

An encoding in HeuristicLab is the implementation of a solution representation (e.g., binary vector, permutation vector, expression tree encoding,...) and operators that work on it. Solutions normally have a name which is used to store and retrieve them. Solutions also often have a length and bounds for the values that they contain. A problem is responsible to set and configure these values. It discovers all types of operators that an encoding offers and then sets these values. The problem is also responsible for reacting to changes that a user makes, e.g., if a user selects a different instance of a problem, it may require a different length of solution. The problem would react to this change and configure all operators accordingly. Code and functionality like this has to be implemented for all properties that can be configured by the user. Additionally, because this functionality is contained in the problem, it is implemented multiple times with only little variation for every problem that uses the same encoding.

Encodings can do most of the configuration by themselves and can offer an interface to problems which they can use for additional configuration. Hence, with the new problem definition API, a new interface for encodings has been created and most of this functionality was moved to the new encoding components. Problems therefore do not need to discover operators themselves and it is not necessary anymore to configure the solution representation and the operators. In the simplest case, a problem just instantiates a new encoding, defines its properties and gets the correctly configured operators resulting in nearly no configuration code.

Table 1 shows achievable code reduction by comparing lines of code (LOC) for two problems in HL which have been rewritten with the new API. The first one is the OneMax test problem and the second one is a problem which allows to use other applications for solution evaluation.

Problem	LOC Old API	LOC New API	Ratio
One Max	201	40	5.0
External Eval.	274	130	2.1

Table 1: Lines of code (LOC) compared between the old and new problem API. Measured with Visual Studio 2013 code metrics

The table shows that the new API allows to significantly reduce the amount of code that needs to be written.

Besides removing the need to configure encodings, the new API also removes the need to write operators for solution evaluation, analyzers and neighborhood operators. Writing operators is a task where the developer has to know a lot about the HeuristicLab API and its internal structure. Therefore, this has been abstracted in the new API to allow developers to more easily develop code for this functionality as described in the next section.

Wrapping operators

HeuristicLab algorithms are constructed by chaining operators together [9]. Operators are building blocks of code that can be reused. Crossovers, selectors, mutators, evaluators, and so on, are all implemented as operators that work on scopes. Scopes are hierarchical data structures that contain variables. Operators assume a certain structure of the scopes they are applied to. Because there is no direct binding between scope configuration and operators, a lot of knowledge and experience is necessary to configure and apply operators correctly.

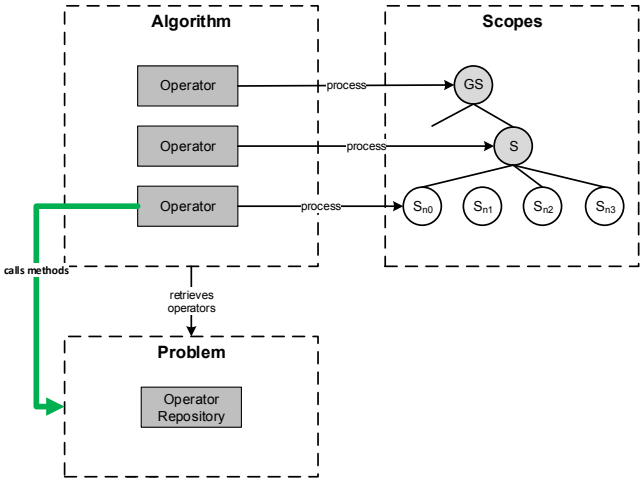

Figure 2: Operator Model

In the new problem definition API, this issue has been eliminated by replacing operators with abstract methods that can be implemented by the developer and represent a familiar setting for them. Internally, there is still a set of operators that execute these methods and prepare the data for them. Figure 2 shows how algorithms are composed of a set of operators that manipulate scopes. Operators that are specific to a problem or encoding are typically retrieved from the problem. With the new API there are now operators that wrap methods from the problem. These operators, when executed by the algorithm, retrieve data from the scope (e.g., an individual) and then call a method with the retrieved data implemented in the problem.

5. EXAMPLES

In this section two examples of problems implemented with HeuristicLab are shown. These two examples show implementations of two different problems, the first one being the OneMax problem and the second one a real-valued test function. HeuristicLab provides two different possibilities for implementing problems: The OneMax problem is implemented by inheriting from `SingleObjectiveBasicProblem`. This is the preferred way of implementing new problems when developing plugins for HeuristicLab. The other possibility, as shown with the second example, is a more prototyping-oriented way. New problems are defined in HeuristicLab using the "Programmable Problem" and does not require setting up a development environment or creating new plugins. This is based on the scripting functionality (as described in [1]) where C# code can be written, compiled and executed within HeuristicLab. In contrast to `SingleObjectiveBasicProblems`, Programmable Problems require that the problem class inherits from `CompiledProblemDefinition`.

In both cases, the problem to be defined has to imple-

ment `ISingleObjectiveProblemDefinition` or `IMultiObjectiveProblemDefinition`, depending on whether there are one or multiple objectives. This means that the following properties and methods have to be implemented in every problem:

- `Maximization`: A property that should return `true` if it is a maximization problem, otherwise `false`. With a multi-objective problem, this is an array containing a value for each objective.
- `Encoding`
  - For the `BasicProblem` the type of encoding is defined by its generic type parameter.
  - For the Programmable Problems the `Initialize` method has to be used to instantiate a new encoding.
- `Evaluate`: This is the objective function of the problem. It receives an individual and returns its quality value. In the multi-objective case it returns an array of double values which represent the quality values for each objective.
- `GetNeighbors`: This is an optional method that can be implemented if the problem should be used for trajectory-based metaheuristics. It receives an individual and returns its neighborhood, e.g., a list of individuals.
- `Analyze`: An optional method that receives a list of individuals and quality values. If implemented, it is called by the algorithm every generation/iteration and can be used to calculate statistics and store them in the result collection.

OneMax Problem

Listing 5 shows the implementation of the OneMax problem in HeuristicLab as a basic problem. Because OneMax is a problem with a binary vector as solution representation, the OneMax problem inherits from `SingleObjectiveBasicProblem <BinaryVectorEncoding>`. Having set the encoding type, the constructor defines the length of the solution candidates. The `Maximization` property is set to true as OneMax is a maximization problem. In the `Evaluate` function the binary vector from a solution candidate is retrieved, evaluated and its quality returned. The rest of the methods and constructors ensure that objects of this class can be cloned and stored in a file. The `Item` attribute on the class contains the name and description of the class which is used in the GUI when the class is displayed. If a class has a `Creatable` attribute, HeuristicLab automatically displays it in the *New Item* dialog in the *Problems* category.

Rastrigin Problem

The Rastrigin function is a real-valued problem where the goal is to minimize parameters of the function so that the result evaluates to zero:

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$$

In Listing 6 this test function is implemented as described in [4]. In contrast to the OneMax problem, `RastriginProblem` inherits from `CompiledProblemDefinition` and defines

```
1  [Item("One Max Problem",
2  "Represents a problem ...")]
3  [Creatable("Problems")]
4  [StorableClass]
5  public class OneMaxProblem :
       SingleObjectiveBasicProblem<
       BinaryVectorEncoding> {
6   public override bool Maximization {
7    get { return true; }
8   }
9
10  public OneMaxProblem()
11   : base() {
12   Encoding.Length = 10;
13  }
14
15  [StorableConstructor]
16  protected OneMaxProblem(bool deserializing
        )
17  : base(deserializing) {}
18  protected OneMaxProblem(OneMaxProblem
        original, Cloner cloner) : base(
        original, cloner){}
19   public override IDeepCloneable Clone(
         Cloner cloner) {
20   return new OneMaxProblem(this, cloner);
21  }
22
23  public override double Evaluate(Individual
        individual, IRandom random) {
24    return individual.BinaryVector().Count(b
          => b);
25  }
26 }
```

Listing 5: Code for the OneMax problem (for brevity without usings and namespace declaration)

the encoding to be used in the `Initialize` method. Like OneMax it has an evaluate method where the real vector is retrieved from the individual and its quality is computed and returned. Additionally, while leaving the `Analyze` method empty, the `GetNeighbors` method is implemented so that this problem can also be used for trajectory-based metaheuristics such as tabu search.

To solve a problem in HeuristicLab, it has to be assigned to an algorithm. In the user interface, this can be done by creating a new algorithm and using the *New problem* button or dragging an already open problem on an algorithm. Figure 3 shows a screenshot of an algorithm with a programmable problem containing the source code for the Rastrigin problem. When implementing an application or plugin, a problem can also be assigned to an algorithm as shown in Listing 7. It instantiates a new problem and a new algorithm and assigns the problem to the algorithm. Algorithms in HeuristicLab normally choose a reasonable default parameter configuration. Nevertheless, if desired, this can be changed as shown in line five. In line six, the parallel engine is chosen which automatically parallelizes solution evaluations and afterwards the algorithm is started.

6. EVALUATION

In [5] the authors compare the amount of lines of code required to implement the OneMax problem in different metaheuristic software frameworks. Table 6 shows an excerpt from their results with a new row for HeuristicLab. The columns show lines of code counts required to achieve the implementation of the OneMax problem. The *Type* col-

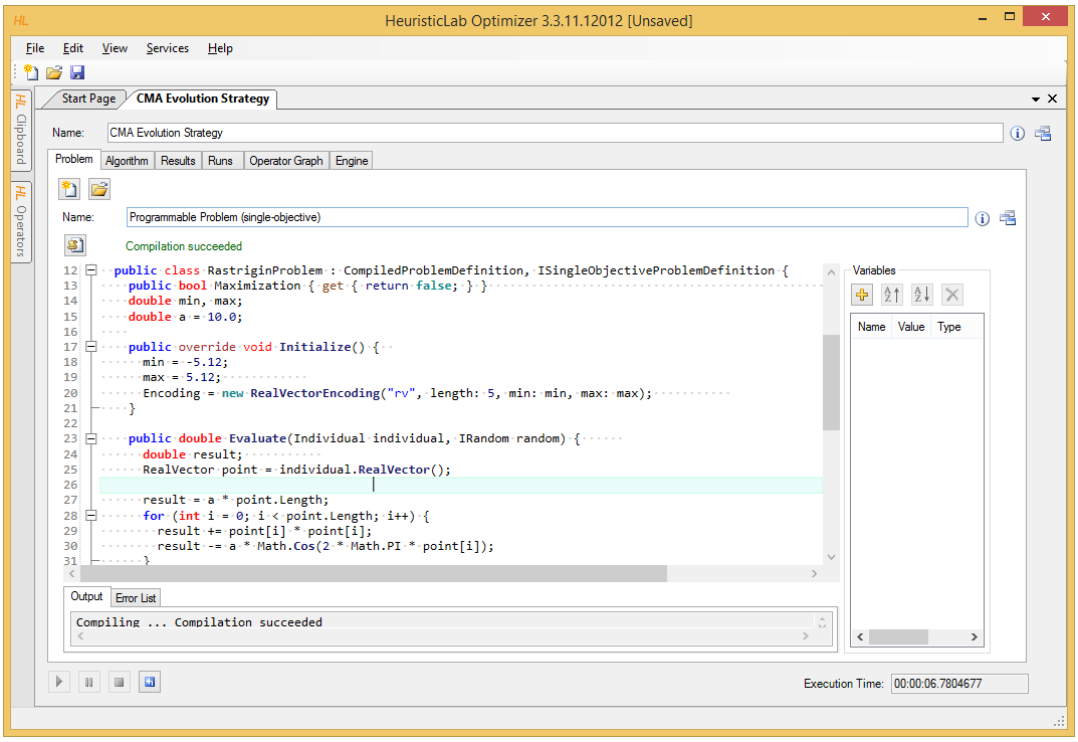

Figure 3: CMA-ES with a programmable problem

umn contains the lines of code required for implementing the solution representation, *Config* shows the configuration of the operators and algorithms and *Ex.* the number of lines needed for implementing the example.

Framework	Type	Config	Example
ECJ	202	35	26
ParadisEO	43	n/a	68
DEAP	n/a	n/a	59
HeuristicLab	40	5	31

Table 2: Comparison of number of lines of code for creating the OneMax problem between different frameworks (counted with cloc). Table (excerpt) taken from [5] with the addition of HeuristicLab.

The table shows that, thanks to the new API, HL is now very similar to the other frameworks concerning the amount of lines of code, sometimes needing significantly less code to implement a certain task.

Discussion

While lines of code is a good indicator of how much effort is required to achieve a certain task, it is not such a good measure for the complexity and difficulty. From our point of view, additional points of consideration would be:

- Knowledge of the framework: A framework can offer an high-level or a low-level API where more interaction with the framework is needed. Having a high-level API is more user-friendly as it hides much of the complexity of the framework. Such an API could hide framework-specific data types and allow the usage of data types of the host programming language and library.
- Usage of the host programming language: A framework may require different levels of knowledge of the host programming language. If a framework requires programming-language specific techniques to make use of it, the framework is more difficult to use than if it would only make use of general concepts. For example, if a framework written in C++ requires the user to know template meta-programming it has a steeper learning curve than if it would only require the user to know general concepts like classes and methods which are present in other programming languages.

A possible direction for future research would be to investigate such topics and try to find measures that can be calculated from the examples to better quantify how hard or easy a software framework is to learn.

7. CONCLUSION

In this paper we presented HeuristicLab's new API for defining single- and multi-objective problems. We showed its usage by implementing two sample problems and compared it to three other metaheuristic software frameworks. We showed that, when comparing the number of lines of code required for implementing different components, HeuristicLab sometimes requires more code to be written, though the resulting implementation may also provide more functionality for the user (e.g., better options for configuration and visualization of the implemented problem).

In the category where the code for the example is measured, HeuristicLab actually does not require much code to be written. This shows that the new API for problem definitions in HeuristicLab compares well to the other frameworks.

Additionally, having the possibility to not only use the

```
public class RastriginProblem :
    CompiledProblemDefinition,
    ISingleObjectiveProblemDefinition {
 public bool Maximization { get { return
     false; } }
 double min, max, a;

 public override void Initialize() {
  min = -5.12; max = 5.12;
  a = 10.0;
  Encoding = new RealVectorEncoding("rv",
      length: 5, min: min, max: max);
 }

 public double Evaluate(Individual
     individual, IRandom random) {
  double result;
  RealVector point = individual.RealVector
      ();
  result = a * point.Length;
  for (int i = 0; i < point.Length; i++) {
   result += point[i] * point[i];
   result -= a * Math.Cos(2 * Math.PI *
       point[i]);
  }
  return result;
 }

 public IEnumerable<Individual>
     GetNeighbors(Individual individual,
     IRandom random) {
  while (true) {
   var rand = rand.NextDouble() * (max-min)
       / 2.0;
   var neighbor = individual.Copy();
   var index = random.Next(neighbor.
       RealVector("rv").Length);
   if (random.NextDouble() < 0.5) rand=-
       rand;
   neighbor[index] = Math.Min(max, Math.Max
       (min, rand + neighbor[index]));
   yield return neighbor;
  }
 }
}
```

Listing 6: Code for the Rastrigin test function (for brevity without usings, namespace declaration and Analyze method)

new API in plugins but also in scripts that are executed from within HL allows to easily write new problems and try new ideas without having to set up a development environment. This means that HL can now also be used in a more prototyping-oriented way similar to DEAP.

8. ACKNOWLEDGMENTS

The work described in this paper was done within the Sustainable Production Steering (NPS, #843638) project and the COMET project Heuristic Optimization in Production and Logistics (HOPL, #843532), both funded by the Austrian Research Promotion Agency (FFG).

9. REFERENCES

[1] A. Beham, J. Karder, G. Kronberger, S. Wagner, M. Kommenda, and A. Scheibenpflug. Scripting and framework integration in heuristic optimization environments. In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1109–1116. ACM, 2014.

```
OneMaxProblem oneMax = new
    OneMaxProblem();

GeneticAlgorithm ga = new
    GeneticAlgorithm();
ga.Problem = oneMax;
ga.MaximumGenerations.Value = 200;
ga.Engine = new ParallelEngine();
ga.Start();
```

Listing 7: Code for creating and starting an algorithm

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

[3] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné. Deap: A python framework for evolutionary algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, pages 85–92. ACM, 2012.

[4] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computation*. Natural Computing Series. Springer-Verlag, 2003.

[5] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, and C. GagnÃl'. Deap: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 2171–2175(13), 2012.

[6] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: A general purpose evolutionary computation library. In *5th International Concerence in Evolutionary Algorithms*, pages 231–242, 2001.

[7] G. Kronberger, M. Kommenda, S. Wagner, and H. Dobler. Gpdl: A framework-independent problem definition language for grammar-guided genetic programming. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, pages 1333–1340. ACM, 2013.

[8] F.-M. D. Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. GagnÃl'. Deap – enabling nimbler evolutions. *SIGEVOlution*, 6(2):17–26, 2014.

[9] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. *Advanced Methods and Applications in Computational Intelligence*, volume 6 of *Topics in Intelligent Engineering and Informatics*, chapter Architecture and Design of the HeuristicLab Optimization Environment, pages 197–261. Springer, 2014.

[10] D. White. Software review: the ecj toolkit. *Genetic Programming and Evolvable Machines*, 13(1):65–67, 2012.