

Structuring Evolutionary Algorithms into Components using the eaLib Framework

Andreas Rummmler

Technical University of Ilmenau
Department of Electronic Circuits & Systems
PO Box 100565, 98684 Ilmenau, Germany
arummmler@acm.org

Gerd Scarbata

Technical University of Ilmenau
Department of Electronic Circuits & Systems
PO Box 100565, 98684 Ilmenau, Germany
gerd.scarbata@tu-ilmenau.de

Abstract — Using the terms of evolutionary computation (individual, chromosome, etc.), it is shown that a general-purpose representation of individuals is possible by means of genetic algorithms. The principles necessary in the software to produce genetic algorithms are presented, and the authors then introduce the concept of breaking up evolutionary algorithms into several independent operators and show how to define the interfaces for the operators. A model within which all the operators can function for flexible and fast creation of algorithms is presented and an example is given in illustration. The paper concludes with a short outlook on future work.

1 INTRODUCTION AND MOTIVATION

Algorithms based on simulated evolution have become more popular in recent years. Without question they are capable solving difficult optimization problems quite fast and producing results of very good quality. However, as there is a problem with combinatorial optimization, it is still not easy to create an algorithm with the performance to solve that problem. Although there have been a few attempts to tackle this issue on a more systematic basis (for instance the one presented in [6] and [7]) something like a cookbook is still missing. Scientists and engineers who need to create evolutionary algorithms spend a considerable amount of their time experimenting with all the operators and parameters in the field of evolutionary computation - and unfortunately this is likely to remain so.

There is however, something that can be done to help: engineers can be offered flexible and easy to use toolkits for their algorithm creation. One of these evolutionary computation frameworks is *eaLib*, which is introduced in this paper, in association with some thoughts on how genetic operators can be fitted into a common structure.

The toolkit *eaLib* is implemented in Java. This language was chosen for several reasons. Although it does have some disadvantages regarding performance when it is compared to C++ (due to its virtual machine concept [5]), Java contains several features that argue for it. It is widely distributed and has become one of the major programming languages. The development kit, including compiler and debugger, is freely available on a number of different computer platforms. The core libraries contain many functions which can be used directly and need not be adopted from external libraries, which is not the case in C++ for instance. Another important point is the built-in support for multithreading and distributed computing.

Java is a strictly object-oriented language. By taking advantage of the features of object-orientation, it is possible to combine easy handling and flexibility/extensibility, which are both necessary for toolkits such as *eaLib*.

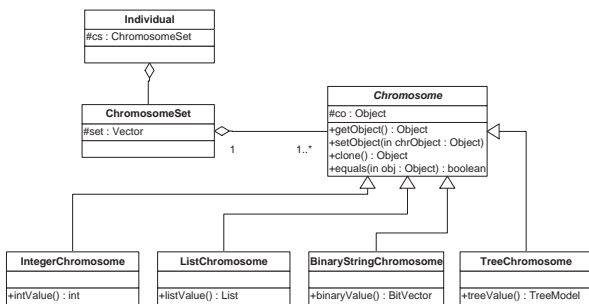
2 GENETIC REPRESENTATION

Before explaining the component-based algorithm structure developed, some thoughts on a suitable form of genetic representation must be given. Traditionally, binary strings or real numbers are applied. Although these forms are still used in several cases, they are neither particularly efficient nor very descriptive for a human user. As an example just think of a potential solution for a 50-city travelling salesman problem expressed in a binary string. . . Furthermore, almost all binary crossover and mutation operators show the problem of generating illegal bit strings in case the whole range of possible combinations can or should not be used. So in the authors opinion there is no reason to stick to these traditional forms. It seems preferable to search for genetic representations that are directly based on the sets of data to be optimized. The kind of representation used in the *eaLib* toolkit will now

be introduced.

As is well known, a potential solution to an optimization problem is represented by an individual. There is thus a class called *Individual*. The structure of this class is close reflection the structure of real creatures in the natural world. An object of the class *Individual* contains an object of the class *ChromosomeSet*. The *ChromosomeSet* itself is a collection of *Chromosomes*. Each chromosome holds a chromosome object of a particular type, which may be a type of elementary data such as an integer or a float number or a more complex type, such as lists or arrays. Even trees or matrices are possible. To enable a chromosome with the classical binary representation to be created the class *BitVector* is provided. Fig. 1 shows the UML class diagram [1] of the genetic representation. In this example four types of chromosomes are included.

Figure 1: UML class diagram of the genetic representation



For most applications of the technique the potential problem solution can be expressed using the built-in chromosome types, but in general any arbitrary data structure can be in fact used. All a user has to do is to derive a new class from the abstract base class *Chromosome* and implement the abstract methods for cloning and equality checking. Even the use of other forms of representation is possible, for instance the one used in messy genetic algorithms as demonstrated in [4].

Some Java code is given as an example below in listing 1. Imagine the task of creating an individual which contains the representation of the solution of a mathematical function in three dimensions. For that case three chromosomes of the primitive data type double are used.

Listing 1: creation of an individual

```

1 // create a chromosome set with 3 slots
2 ChromosomeSet mathSet = new ChromosomeSet( 3 );
3 // add the chromosomes with initial random double values
4 mathSet.add( new DoubleChromosome( (double) 17.734 );
5 mathSet.add( new DoubleChromosome( (double) -3.239 );
6 mathSet.add( new DoubleChromosome( (double) 189.94 );
7 // create the individual
8 Individual mathIndividual = new Individual( mathSet );
  
```

Almost every imaginable problem solution can be expressed in a similar way. Besides several informa-

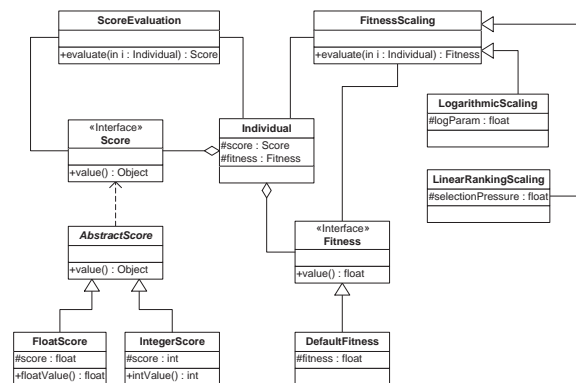
tive attributes such as identification number, name and age the class *Individual* contains two other, significantly more important, properties – the target objective value and the fitness. The target objective value (later referred to as *score*) is an absolute quality appraisal in contrast to the fitness value, which is only a relative rating [9].

The score and fitness of an individual are defined by the appropriate interfaces: *Score* and *Fitness* respectively. Default implementations of both interfaces are already provided by the toolkit.

The score of an individual can be expressed in many ways. Various different data types can be used. Although normally numbers are utilized, other data types are conceivable. The default types which are already supported by *eaLib* are integer, long, float and double. If other data structures are utilized, the user must derive a new class from *AbstractScore* and implement the method for comparing two scores.

In contrast to the various possible definitions of score, a fitness value is always a number. Therefore the fitness has been defined as a positive float number. A bigger value represents better fitness. This definition should be useful in most of the application cases, and (if not) a new class derived from *Fitness* could be introduced by the user. The appropriate methods for comparing various *Fitness* objects must be overloaded. An UML diagram showing the relations between an individual and the appropriate score and fitness is given in Fig. 2.

Figure 2: UML class diagram of score and fitness model and evaluation of both



Also included in Fig. 2 are two classes for the evaluation of the score and the fitness. Turning first to the score evaluation: the calculation of the quality of an individual is always problem specific. For that reason the user is forced to implement the evaluation himself. The class *ScoreEvaluation* is an abstract base class which defines the method *evaluate(Individual)*. The user has to derive a class from that base class and to implement this method which will then calculate a valid score for a given individual.

The fitness evaluation proceeds in a similar way, with the difference that there are several means of defining the fitness of an individual already in existence, for instance exponential scaling or linear ranking scaling. Some of these approaches are available in *eaLib*; the user is also able to introduce his own functions. Again, there is an abstract base class, this time called *FitnessScaling*, which also defines a method *evaluate(Individual)* for the calculation of a given individual's fitness value.

Several alternative means of defining a fitness value are already implemented in the toolkit and can be used directly. At the time this paper was written the user had the choice of linear, reciprocal, exponential and logarithmic scaling [9] to derive directly from the score. In addition linear [2] and nonlinear ranking scaling [3] are available for use.

3 DEFINITION OF GENETIC COMPONENTS

It is necessary at the beginning of this section to make some remarks concerning the term component. The term is heavily used in recent years without there being any universal definition as to what a component is. According to [8], a component is a runnable piece of software. It is a self-contained object with describable functionality. It has a black box character – the information contained within a component is hidden from the user. Access to methods and/or attributes of a component is provided by a standardized interface. This interface is both a communication channel and a plug to which other components can be connected. The solution of a problem is achieved by creating a net of coupled components. This net in its totality is able to solve the given problem. Furthermore, components can provide the property of persistence. That means the state of a component can be frozen until its next use.

The authors follow the definition given here, referring to a component as a module with a well-defined interface which is able to process data of a particular type (more or less) independently of other components.

In the attempt to view genetic operators as components and to define a unified interface it is necessary to think what these operators have in common. Score evaluation, selection, mutation and all the many other operators that have been introduced and used in evolutionary algorithms may at first to be quite different. But from an abstract perspective they have one important thing in common: they serve to process individuals in a particular manner. The selection operator selects good individuals for mating from a group of individuals. In contrast, the mutation operator alters the contents of individuals. A group is processed one by one sequentially.

Individuals are grouped together inside classes implementing the interface *IndividualStream*. An individual stream is a collection of individuals which provides several methods for putting individuals on the stream, removing them from the stream and iterating over all individuals contained in the stream. The underlying data structures are hidden from the user. Using this structure as a collection of individuals permits interfaces to be defined for so-called stream processors. The most simple case is a *SingleStreamProcessor* which has exactly one method: *IndividualStream process(IndividualStream is)*.

A single-stream processor takes an individual stream as an argument, processes this stream in a particular way and returns the processed stream. This can be extended to four possible cases: processors that merge streams, split up streams or work on multiple streams (Fig. 3). The UML class diagram is given in Fig. 4.

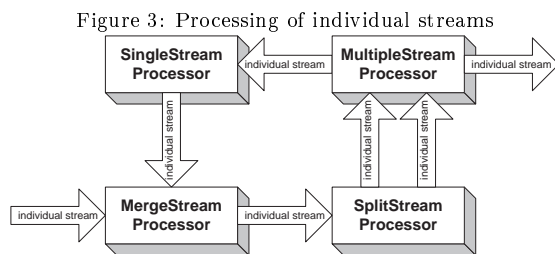
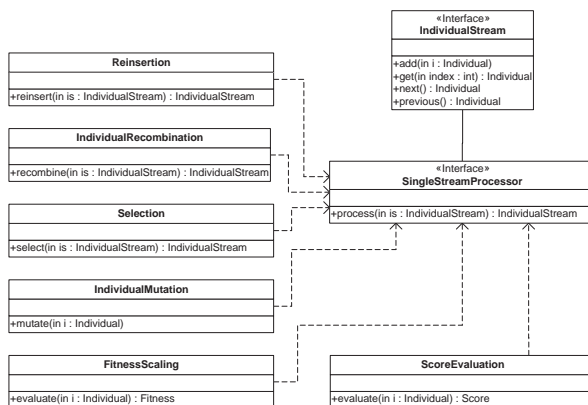


Figure 3: Processing of individual streams

In most cases it is appropriate to apply single-stream processor. The processors for splitting and merging streams can be used in algorithms which either involve parallel-running evolution cycles or implement the island population model. The interface for the multiple stream processor has only been introduced for reasons of consistency and is provided for future use.

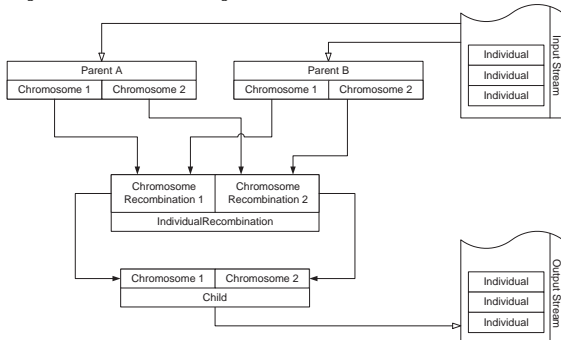
Figure 4: UML class diagram for interface definition of genetic operators



For a better illustration of how such a processor works, a closer look at the recombination mechanism

follows. The rest of this section explains how the recombination mechanism functions with regard to the genetic representation introduced in section 2. As individuals can contain any number of chromosomes with any data type, common crossover procedures cannot be employed directly on an instance of the class *Individual*. Therefore the abstract base class *Recombination* is subclassed by the two classes *IndividualRecombination* and *ChromosomeRecombination*. The former is responsible for recombining individuals, while the latter recombines chromosomes. The subclasses of *ChromosomeRecombination* are well-known operators for recombining pairs of data of types integer, float, binary string, lists and so on. In contrast, the class *IndividualRecombination* is a collection of chromosome recombination operators, one for each chromosome inside the individuals to be recombined, having the ability to handle the appropriate data type. The functionality is shown in Fig. 5. The operator takes two individuals from the input stream and constructs one (or more) child individual(s) by combining the parent chromosomes with the corresponding chromosome recombination operators. The newly created child individuals are put on the output stream which is produced as the result of this operator. The stream is passed to the following operator (for instance a mutation operator) which handles the stream in a similar way.

Figure 5: Recombining individuals contained in a stream



All components contained in the toolkit work in this vein, even the score and fitness evaluation. There are too many operators to enumerate them in this paper, an overview of available operators can be found at [11].

4 CREATING ALGORITHMS BY WIRING COMPONENTS

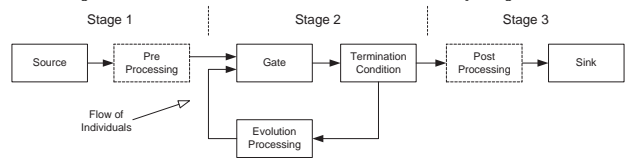
The quality of the interplay of components in a software system can be assessed on several factors. Two of the more important ones are *cohesion* and *coupling*. Cohesion is the degree of interaction within a component, while coupling is the degree of interaction between several components [12]. The cohesion

of components should be very high while the coupling should be as low as possible. This is achieved by grouping all aspects of the real-world entity of a component together inside the component. In this way future changes are localized to one specific component; the details of the changes are not visible to other components and cannot affect these.

These viewpoints were taken into account for the structuring of the genetic components. All operators work independently of one another, but share a common interface. A further step can now be taken, the creation of algorithms using these components.

One of the common cases is a genetic algorithm with only one population. A more abstract look at the structure of such an algorithm reveals that an algorithm can be separated into three stages. The first includes the initialization of a population and an optional preprocessing. The second one is the transformation stage, where individuals are evolve within an evolution cycle. In the third and last stage, the individuals can be optionally postprocessed and the result of the algorithm extracted. Genetic operators can be assigned to each of the stages of this basic scheme (Fig. 6) and the path taken by the streams of individuals inside an algorithm can be figured out.

Figure 6: Data flow inside an evolutionary algorithm

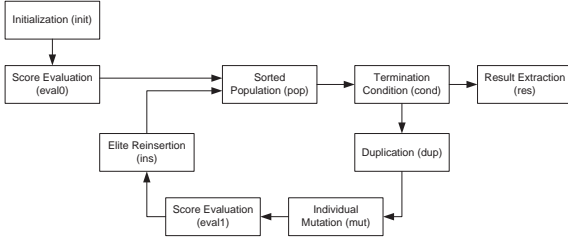


The creation of an individual stream is always performed by an initialization operator. Such an operator can be seen as an individual source. At the other end of the flow, a stream can be captured by a sink (for instance for extracting information). The first operator provides the retrieval of individuals while the second one provides storage. A combination of both is a so-called gate, where individual streams are flowing through and evolving within. A gate is nothing but a population. These three types of operators are connected by stream processors as already discussed.

To generate a valid algorithm therefore, all that is necessary is to create instances of suitable components and connect them together in a reasonable manner. To illustrate how this can be done, a simple example follows. In Fig. 7 the structure for a μ - λ -strategy according to [10] is given. Below, the corresponding piece of Java source code is shown in listing 2.

The piece of code that is shown is part of a setup-method in the class *ThreadController*, which will be explained in more detail in the next paragraphs. Reflecting the algorithm structure given in Fig. 7 all

Figure 7: Implementation example for a μ - λ -strategy



necessary components are created in lines 1 to 23 of the listing. The first one is the individual source *MyInitialization* (line 2). This is a class the user has to derive from the abstract base class *Initialization*. An object of this class creates valid chromosome sets. Because this task is problem-specific, it has to be done in a class implemented by the user. The same obtains for the score evaluation of the individuals (lines 4-7), as already explained. In line 8 a population is created which holds individuals in a sorted order. Lines 9 and 10 introduce a special operator for duplicating individual streams. This operator is used to prevent the mutation of individuals already contained in the population (as done in a μ - λ -strategy). The mutation component is set up in lines 11 to 16. Note that first some operators for chromosome mutation are created (three in this example) and then these operators are grouped together in an individual mutation operator. In this example a user-defined chromosome mutation is used, but at this point predefined operators can also come into operation. Finally the reinsertion operator and the termination condition is initialized (break after 500 generations). Both require a reference to the relevant population.

The *ThreadController* class already mentioned is the main class, monitoring the execution of an algorithm. The user has to derive a class from this controller and fill out one single method: *setup()*. The Java code given above is taken from this setup method. In this method, the components employed are initialized (as already shown) and connected. The connection structure is set up in lines 28 to 33. For this task, corresponding *connect*-methods are provided. For components with single input and output, the wiring is quite easy. For instance, the statement in line 28 connects the output of the initialization operator *init* with the input of the score evaluation operator *eval0*. Note that operators can have multiple inputs and outputs. The conditional operator is an example of this. As shown in lines 31 and 32 it passes the individual stream it gets from its input to the output slot 0 or 1, depending on the result of the evaluation of the condition(s) contained in the operator.

Something else which can be seen in the given listing is the organization of the components inside the

controller. The controller takes advantage of the built-in multithreading capabilities of Java. Every genetic component runs inside its own thread (see for instance line 2). All threads are started during the setup stage of the algorithm and are sleeping all the time. Every component knows about about its successor component(s) and is able to notify those successors when its own task is completed and the processed individual stream can be passed to the following operator(s).

Listing 2: piece of Java code for a μ - λ -strategy

```

// create instances of components
2 SourceThread init
  = new SourceThread( new MyInitialization() );
4 SingleThread eval
  = new SingleThread( new MyScoreCalculation() );
6 SingleThread eval2
  = new SingleThread( new MyScoreCalculation() );
8 GateThread pop = new GateThread( new SortedPopulation() );
9 SingleThread dup
10  = new SingleThread( new IndividualDuplication( 1 ) );
11 ChromosomeMutation[] cma = new ChromosomeMutation[3];
12 for ( int i = 0; i < 3; i++ ) {
13     cma[i] = new MyChromosomeMutation();
14 }
15 SingleThread mut
16  = new SingleThread( new IndividualMutation( cma ) );
// create reinsertion operator
18 SingleThread ins
19  = new SingleThread( new EliteReinsertion( pop.getGate() ) );
20 SinkThread res = new SinkThread( new GenericSink() );
21 Termination mg
22  = new MaxGenTermination( 500, pop.getGate() );
23 ConditionalThread cond = new ConditionalThread( mg );
24
// more optional code goes here
26
// connect all components
28 connect( init , eval0 );
29 connect( eval0 , pop );
30 connect( pop , cond );
31 connect( cond , 0 , dup );
32 connect( cond , 1 , res );
33 connect( dup , mut ); connect( mut , eval1 );
34 connect( eval1 , ins ); connect( ins , gate );
  
```

Another advantage of the use of multithreading is the implicit parallelism that the mechanism contains. With the wiring of components as described it is quite easy to create algorithms with parallel running evolution cycles or algorithms based on the island model. For that case special operators for splitting and merging individual streams are provided. This concept will, in due course, be extended to components which are able to pass streams to other components running on machines in a local network. Thus it will be possible to create real parallel-running algorithms using the same mechanisms as described above.

5 CONCLUSION AND FUTURE WORK

This paper has introduced the structuring of genetic components into independent software components. It has shown the advantages of viewing the structure of evolutionary algorithms from a more abstract and pragmatic view. Along with these considerations the toolkit *eaLib* has been introduced. *EaLib* is intended

to be easy to use, but still flexible enough for various extensions. The toolkit's general-purpose means of representing potential solutions for optimization tasks has been shown. A concept for a common interface for genetic operators has been introduced and an example has been given.

The aim of the work was to provide a framework to support experimentation with genetic operators and fast creation of algorithms. What is now needed is creative feedback from scientists. The framework could be extended in various directions, as a result of such feedback. The authors do not yet see the capabilities of parallelizing algorithms as universal enough, further development will concentrate on this issue.

REFERENCES

- [1] Object Management Group UML Resource Page. <http://www.omg.org/uml>.
- [2] BAKER, J. E. Adaptive selection methods for genetic algorithms. In *Proceedings of the International Conference on Genetic Algorithms and their Application* (1985), pp. 101–111.
- [3] BÄCK, T., AND HOFFMEISTER, F. Extended selection mechanisms in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (1991), pp. 92–99.
- [4] DEB, K., AND GOLDBERG, D. E. A messy genetic algorithm in c. Tech. Rep. 91008, University of Illinois at Urbana-Champaign, 1991.
- [5] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. Addison Wesley, Bonn, Paris, Reading, 1999.
- [6] MÜHLENBEIN, H., AND MAHNIG, T. Evolutionary algorithms: From recombination to search distributions. Research report, GMD Institute for Autonomous Intelligent Systems, Sankt Augustin, 2000.
- [7] MÜHLENBEIN, H., AND MAHNIG, T. Evolutionary computation and beyond. In *Foundations of Real World Intelligence*. CLSI Publications, Stanford, 2001.
- [8] PIEMONT, C. *Komponenten in Java*. dpunkt-Verlag, Heidelberg, 1999.
- [9] POHLHEIM, H. *Evolutionäre Algorithmen*. Springer-Verlag Berlin, Heidelberg, New York, 2000.
- [10] RECHENBERG, I. *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog, Stuttgart, 1973.
- [11] RUMMLER, A. eaLib API Documentation. <http://www.inf-technik.tu-ilmenau.de/~rummler/eng/ealib.html>, 2001.
- [12] SCHACH, S. R. *Software Engineering with Java*. Irwin/McGraw Hill, Boston, 1997.