GENERICITY IN EVOLUTIONARY COMPUTATION SOFTWARE TOOLS: PRINCIPLES AND CASE-STUDY

CHRISTIAN GAGNÉ and MARC PARIZEAU.

Laboratoire de Vision et Systèmes Numériques (LVSN),
Département de Génie Électrique et de Génie Informatique,
Université Laval, Québec (QC), Canada, G1K 7P4.
{cgagne,parizeau}@gel.ulaval.ca

This paper deals with the need for generic software development tools in evolutionary computations (EC). These tools will be essential for the next generation of evolutionary algorithms where application designers and researchers will need to mix different combinations of traditional EC (e.g. genetic algorithms, genetic programming, evolutionary strategies, etc.), or to create new variations of these EC, in order to solve complex real world problems. Six basic principles are proposed to guide the development of such tools. These principles are then used to evaluate six freely available, widely used EC software tools. Finally, the design of Open BEAGLE, the framework developed by the authors, is presented in more detail.

Keywords: Evolutionary computation; genetic algorithms; genetic programming; software engineering; object oriented programming.

1. Introduction

In the last fifteen years, Object Oriented (OO) methodologies for software development have gained significantly in popularity in the computer world. These approaches promote code reuse and development by abstraction, thus flexibility and genericity. In the same years, different nature-inspired optimization techniques have been unified under a common denomination, Evolutionary Computation (EC). Using an OO terminology, EC can be seen as an abstract class where the different specific algorithms (genetic algorithms, evolution strategies, etc.) are the concrete implementations. The highly diverse and adaptable nature of evolutionary algorithms make EC software tools good candidates for generic OO architecture. But designing such generic software tools is quite difficult given that most of the software components must be replaceable or modifiable: representations, fitness measures, variation and selection operations, evolutionary models, etc. This paper is a study on genericity in EC software tools, with principles on the development of such tools and a case-study of a generic EC framework.

The paper is structured as follows. First, a review on the previous works is

presented. Then, the advantages of using a generic EC software tool in the form of an OO programmed framework are put forward. Then the different stages in developing a generic EC framework are discussed, and the six proposed quality criteria to characterize the genericity of an EC software tool are introduced. An analysis of some existing EC software tools genericity is presented. The latter part of the paper is a case-study of Open BEAGLE, the generic EC framework we are developing. It includes a presentation of the framework architecture, with a special focus on the genericity mechanisms. Finally, a code example with Open BEAGLE is shown, illustrating some architectural points of the framework that make it highly flexible and, at the same time, quite easy to use.

2. Generic EC Software Tools Principles

Traditionally, EC is divided into three categories[3]: Genetic Algorithms (GA)[18], first developed in the United States in the 1960's and 1970's by Holland and his students[a], Evolution Strategies (ES)[1,36], developed at about the same time in Berlin by Rechenberg and Schwefel, and finally Evolutionary Programming (EP)[1,11], created in the United States in the 1960's by Fogel. This taxonomy is essentially due to historical and geographical factors, the different scientific communities having progressed separately until the beginning of the 1990's.

Given the obvious similarities between these three approaches, it has been proposed to combine them under the unique name "evolutionary computation". Many EC conferences and scientific journals have been created during the 1990's, thus reducing historical distinctions and facilitating exchange of ideas. This can be illustrated by the appearance of agnostic techniques related to the different EC flavors, for example multiobjective optimization[8] or co-evolution[2,17]. Another predictable[b] effect is that in the near future, the GECCO[c] conference should be structured around the different field aspects (representations, algorithms, applications, etc.), not the current historically based EC flavors (GA, GP, etc.).

Despite this important trend toward the unification of EC, most of the specialists are associated with a particular EC flavor and their scientific and technical approaches are often influenced by this choice. An example of this is the widespread use of specialized software implementing one particular EC flavor. But the recent unification of the domain will probably lead to the common use of generic software tools not dedicated to any particular EC flavor, allowing the quick development of new approaches. We think that the use of this kind of software tools is desirable and should be more widespread in the community in the recent future.

From a software engineering point of view, the development of a really generic

---

[a]Despite its distinctive characteristics and its importance, Genetic Programming (GP)[4,21] is generally considered as a GA component according to EC historical taxonomy.
[b]This idea was discussed by some researchers present at the GECCO 2003.
[c]Genetic and Evolutionary Computation COnference, one of the most important scientific conferences on EC.

EC software tool is complex. In this section, we will analyze this idea and discuss the problems inherent to the development of a generic EC software tool.

2.1. *Previous works*

In the middle of the 1990's, Ribeiro Filho et al.[37] presented different existing EC software tools, with particular emphasis on GA tools. The paper, centered on technical characteristics, presents an interesting classification of EC tools in three categories: 1) application-oriented tools, essentially used in particular application contexts, 2) algorithm-oriented tools, typically libraries implementing a particular EA, and 3) toolkits, relatively generic software suites. At about the same time, two journals published articles[10,42] on particular GP systems implementations in C++. Keith and Martin[20] have also made a good analysis on different ways of implementing representation of genetic programs.

In more recent years, several papers on EC software tools have been published. Papers by Tan et al.[43] and MacCallum[27] concern specialized EC tools and graphical user interfaces. Adopting a novice point of view, Wilson et al.[49] have made a somewhat superficial analysis of three GP software tools: lil-gp, ECJ, and Grammatical Evolution. Others[13,19,23,28,39,45,46] have presented generic EC software tools architectures, but they did not discuss the general concepts related to the development of such tools. The paper by Lenaerts and Manderick[25] is of special interest here since it discusses generic tool development from a global standpoint. The paper presents the advantages of using and developing an EC framework according to an OO methodology. The authors expose the problems encountered when using most of the available libraries, and the advantages that would benefit EC researchers, particularly in GP field, if they were to use generic and extensible frameworks. They also make a very interesting presentation of different design patterns[15] that can be applied when designing a GP framework.

2.2. *EC Framework*

EC can be seen as an abstract class of algorithms, and the different flavors like GP, GA, and ES can be seen as concrete instantiations of them. An EC instantiation is achieved by specifying two principal characteristics: the population representation (data structure), and genetic and natural selection operations (algorithms) used. This modeling is complex for EC software tool developers because the different aspects of an EC instantiation must be uncoupled as much as possible, while allowing a certain code re-use. Certain types of operations like natural selection operations can be applied to all representations while others like crossover and mutation operations are specific to the representation used. Furthermore, the EC field is constantly progressing and it is not possible to predict new approaches or variations that will be interesting to implement in the software tool in the future. Thus it is necessary to include flexible mechanisms in order to avoid important modifications in the tool's basic structures. So generic EC software tool development includes many issues.

From the software engineering point of view the best way of modeling it is in the form of a framework.

Within the scope of OO programming, a framework is defined by Gamma et al.[15] as: *A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.*

A common characteristic of all EC flavors is the need to evaluate the fitness of solutions for a specific problem. Thus it is essential that a part of the architecture be left empty in order for the user to at least implement the fitness evaluation operation. On the other hand, depending on the problem to solve, the user can be interested in different possibilities: to use a standard configuration of a particular EC flavor, to define his own EA using different standard algorithm components, to define his own genetic and natural selection operations, or to define a new individual representation. This is why a good EC software tool must be flexible. But it must also implement some default standard operations and specify relations between the different entities of the system.

2.3. *Progression of an EC Framework*

Roberts and Johnson[38] present patterns capturing the essence of frameworks in different development stages, from a white box framework, where the user has to define a set of components that inherit (in OO terms) from basic components, to a black box framework, where most specialized components already exist and where the user is only required to arrange some components to solve his problem. A framework progresses generally from the white box to the black box model. A typical progression starts from the development of a relatively important component set forming a library capturing the application field. Application components are defined by deriving new classes of basic components. During the framework progression, architecture elements that are changed regularly need to be modeled by simpler, loosely coupled components. These components must be easy to combine together without having to define new objects. Also, the number of components in the framework is generally increasing. The framework reaches the black box model when it is possible to implement new applications only by connecting existing components. Script languages and/or graphical interfaces are then often developed in order to allow a development environment that does not require thorough knowledge of the framework internal mechanisms for simple applications.

Development stages of an EC framework are consistent with the progression model from a white box framework to a black box framework. This progression often comes with encapsulation of genetic and selection operations in the form of components that can be connected to form the desired Evolutionary Algorithm (EA). However, a "pure" black box EC framework is not possible in practice since

the user must define the fitness evaluation function for his particular problem. This can be done by defining a new component or using a high-level language script. It is also possible to provide some fitness evaluation components with the framework, for a set of classical problems. But in general this approach is not applicable because fitness evaluation functions are too specific and thus impossible to state in advance.

A black box EC framework combined with a graphical interface allows EC evolutions to be performed without having to master the internal mechanics. But the expert user can still take advantage of the architecture flexibility to define new components. Designing such an EC framework requires an important effort to developers. This is why, to the best of our knowledge, only three really generic, flexible and user friendly EC frameworks have reached the stage of a black box[d].

2.4. *Genericity Criteria*

Six criteria are proposed here to qualify the genericity of a framework: generic representation, generic fitness, generic operations, generic evolutionary model, parameter management, and configurable output.

(1) **Generic representation**: For a personalized EA, it must be possible to define new individual representations without limitation on the data structures used. These new representations can be defined using existing representations as a basis. For example, a standard GA representation can be used as a basis to define a vector of graph indexes representation in a combinatorial optimization problem[30]. The user can also define unusual representations, for example graph-based genetic programming[44], which is significantly different from classical tree-based GP representation. It should be possible to reuse some existing genetic and selection operations with new representations, depending on their singularity.

(2) **Generic fitness**: Individual fitness measures should be as independent as possible from representations and selection operations. It should be possible to define and use fitness measures that are particular to a given application. For example, the user may want to change a fitness measure that assigns high values to good individuals (maximization of fitness) to one that assigns low values to good individuals (minimization of fitness). This should be possible without having to recode representations or selection operations. The framework should also support multiobjective evolution fitness measures in a transparent manner. Such specifications can be filled by the use of polymorphism. This can be implemented through an abstract representation of the fitness value, that would include mechanisms to compare two values without knowledge of the concrete fitness type. Then, it should be possible to define a generic selection operation, for example tournament selection, based on the abstract comparison mecha-

---

[d]i.e. ECJ, EO, and Open BEAGLE. They will be presented hereinafter.

6   *Christian Gagné and Marc Parizeau*

nism. Such a selection operator will work on any kind of fitness value, including custom ones, as far as the comparison mechanism is defined.

(3) **Generic operations**: Limitations on the type of genetic and selection operations that can be implemented in the software tool should be minimized. Operations can be relatively classic, like two-parent crossover, or completely unusual. Operations should also be relatively independent and have minimal side effects, in order to use them in conjunction with other operations. When possible, operations should be independent from representations. This approach allows development of new operations without altering existing ones. This favors the creation of a library of components and the development of new EC flavors. For example, one can imagine a library of components having many variation operators for a bit string representation (bits inversion mutation, one-point or two-points crossover, uniform crossover, etc.). For an application based on such a representation, the user can use one or many of these operators with or without other operators, and he is not constrained with respect to arrangement or compatibility. Developing good generic operations requires a good design sense in order to choose the right granularity for our building blocks. A generic operator must be neither too coarse nor too fine. Indeed, a too coarse operator may limit the flexibility required by the user for rapidly developing new evolutionary models, while a too fine operator may put too much emphasis on the interactions between operators.

(4) **Generic evolutionary model**: Genetic and selection operations should be applied to the population with flexible and configurable algorithms. Thus the evolutionary model must be as flexible as possible, without a rigid structure. Ideally, it should be possible to define the model only by connecting operators together in a given order. For example, in the generational GA case, the EA can be seen as a successive application of natural selection, crossover and mutation operations on each individual of the population. On the other hand, a steady-state GA is characterized by natural selection, crossover, and mutation operations randomly applied to individuals. Each iteration corresponds to the creation of a new individual in the population, replacing an existing one. Finally the ES (μ, λ) and $(\mu + \lambda)$ models are an even more complex arrangement of selection, crossover, and mutation operations. It should be possible to introduce unusual operators in an existing model without having to rewrite it.

The need for a generic evolutionary model thus implies the development of some-kind of a procedural programming environment for EC. As with most programming environment, branching and looping statements are necessary to allow declaration of generic EC models. In the actual case, the branching decisions are generally taken from a set of parameter values, while loops are often applied on all of the individuals composing a population.

(5) **Parameters management**: An EC framework often includes a mechanism that allows the dynamic modification of parameter values (population size, mutation probabilities, etc.) from a configuration file or other user interface. It is

then desirable that the parameter management mechanism includes only parameters relevant to the EA used and allows the easy addition of new parameters for customized EA. It is also interesting to enable algorithm configuration directly from a file, without having to recompile a new application.

(6) **Configurable output**: Output to a file or another user interface must be configurable. For outputs of the evolution state, each representation, fitness measure or operation have their own specific information that can be provided to the user. For example, outputs concerning evolution progression statistics differ depending on the use of a single or a multi-objective fitness measure. In GP applications it is also interesting to have statistics about tree sizes and depths. In a generic tool it should be possible to add new outputs like these statistics, and the new data in user information outputs or result file outputs should be harmoniously integrated to the current outputs.

All of these characteristics force the user to understand some mechanisms that are essential for tool flexibility. Some users are not willing to make this effort so they prefer monolithic EC library tools that are specialized for a given flavor, these tools being easier to learn in the short term. This choice becomes expensive when the users discover the limitations of the library. Then they have to change the library code in order to modify existing functionalities or to support new ones. Initial advantages of the library are then lost. Furthermore, addition of components such as new genetic operations or new representations are practically impossible because they generally imply that each modification is permanent and irreversible. Thus, modifications and new functionalities are in competition with those that already exist. For example, the extension of a GA monolithic library in order to support multiobjective evolution fitness measures may possibly alter its compatibility with old applications using single objective fitness measures. Learning to use a generic EC framework is rewarding in the medium term if the user plans to experiment with different variations of the EA used.

2.5. *Genericity Analysis of Software Tools*

Despite the large number of EC software tools, only a few of them are widely used in the community, and fewer are generic. In order to illustrate the ideas presented above, six tools will be analyzed in more detail from the genericity point of view: ECJ[26], EO[19,29], GAlib[48], lil-gp[34], GPLAB[40,41], and Open BEAGLE[14]. We have chosen these tools based on the following criteria: 1) they implement GP, which is an important and complex EC flavor; 2) they are flexible; 3) they are relatively popular in the community; and 4) they are interesting for this study, from a programming language or architectural point of view. Table 1 presents a genericity evaluation of these tools according to the six criteria of Section 2.4.

ECJ[e] is a generic EC framework coded in Java. It is probably the most popular

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

8   *Christian Gagné and Marc Parizeau*

Genericity criteria	ECJ 13	EO 0.9.3a	GAlib 2.4.6	lil-gp 1.1	GPLAB 2	Open BEAGLE 2.2.0
Generic representation	2	2	2	0	0	2
Generic fitness	2	2	0	0	0	2
Generic operations	2	2	1	2	2	2
Generic evolutionary model	2	2	1	1	1	2
Parameter management	2	2	2	1	2	2
Configurable output	2	1	0	1	0	2

(2 = complete, 1 = partial, 0 = missing)

public EC system coded in Java. It respects all genericity criteria described above. It has a Java executable that requires only a configuration file and a Java component with the fitness evaluation function. The configuration file states the ECJ elements to use in order to form the desired EA as well as the EC parameters. ECJ is coded in Java, a high-level OO language. This facilitates the programming of new modules but also requires large resources both in terms of memory and execution time[33], compared to the performance of tools coded in other languages such as C or C++. ECJ operations can be put together according to a generic evolutionary model, without having to code any class. ECJ is thus a full black box EC framework.

EO[f,19] is a generic EC framework coded in C++. The objective of its developers was to make possible an evolutionary process with any type of representation, as long as an objective quality measure can be defined. EO includes different operators to initialize and modify the individual representations and the evolutionary process-ing, as well as integration operators. It also has many utility classes for parameter management and, to a certain extent, for output configuration. If the user wants a more complex evolutionary model, he must use integration operators to build his own evolutionary model. The use of these specialized operators requires a good un-derstanding of the framework and its components. On the whole, EO is a generic black box EC framework, but is somewhat difficult to use and master, because of its complex underlying mechanisms.

Related to EO, EASEA[g,9] greatly simplifies the use of a given EC software tool. It allows the integration of EC specifications in a high-level programming language and the transformation of these specifications in C++ code that is compilable with

[f]Evolving Objects, http://eodev.sourceforge.net
[g]EAsy Specification of Evolutionary Algorithms, http://sourceforge.net/projects/easea

EO. A graphical interface named GUIDE allows rapid prototyping. EASEA/GUIDE forms a coherent and interesting software suite to develop a simple application without having to master internal EO mechanisms. Thus EASEA/GUIDE helps to hide some complexity related to EO. But from a genericity point of view it is evident that the EASEA/GUIDE suite induces limitations that make it less interesting to EO experts.

GAlib[h] is an EC library coded in C++ that allows the use of generic representations. Unfortunately, despite the fact that representations are generic, the library is relatively rigid. First, the fitness measure is fixed to be a scalar. Second, there are only six precise types of operators: population initialization, fitness evaluation, individuals selection, termination criterion, two-parents crossover, and mutation. It is not possible to define operators outside this scope with existing evolutionary models. One (and only one) operator of each type must be provided to a particular evolutionary model for a given evolution. The GAlib evolutionary model is not generic since it is coded directly in a class. Parameters can be dynamically added to the system but the output is not modifiable. However the evolutionary model is simplified to some specific algorithms and well defined operations. This makes GAlib a library that is relatively easy to use and to master for beginners.

The following two software tools are specialized for GP. We will evaluate them according to the above genericity criteria. lil-gp[i] is a C language re-implementation of the GP system little-lisp[21]. It is widely used in the community and is recognized as one of the fastest GP systems. Of course the representation of individuals is fixed to GP trees, but the fitness measure is also limited to the Koza's GP fitness measure[21]. The evolutionary model consists of successive applications of operators to the population. It is quite generic because there is no restriction on the type of operators, even though the user is limited to generational algorithms. Parameters can be added, but the user must provide a routine to analyze the configuration file and to extract data from these parameters. lil-gp is a typical specialized monolithic library easy to use if the application stays in the initial scope. But it is difficult to modify or to extend.

On the other hand, GPLAB[j,41] is a MATLAB toolbox for GP applications. Like lil-gp, GPLAB supports only one representation and the fitness measure is limited to a single real value. The evolutionary model is fixed but the operators composing this model can be of any type. GPLAB may not satisfy half of the genericity criteria, but it presents the significant advantage of being built in an environment that can be considered as a generic framework for scientific programming. Despite the fact that it is a slow, interpreted language, MATLAB is a high-level development environment offering a set of incomparable mathematical and graphical functionalities. This is why the present evaluation of GPLAB's genericity does not give justice to the tool's

[h]http://lancet.mit.edu/ga
[i]http://garage.cse.msu.edu/software/lil-gp
[j]http://gplab.sourceforge.net

10 *Christian Gagné and Marc Parizeau*

qualities.

Finally, Open BEAGLE[k] is a black box EC framework coded in C++. Except for the programming language used, Open BEAGLE is similar to ECJ from the point of view of flexibility and ease of use. It is undoubtedly more generic than GAlib, lil-gp, and GPLAB. On the other hand, EO includes a variety of mechanisms allowing for very generic evolutionary models, even though we think that this makes its use quite complicated. Like ECJ, the evolutionary model of an Open BEAGLE application can be dynamically modified in the configuration file, without having to recompile. There are plans to make the generation of configuration files possible from an application with a graphical interface, much like EASEA/GUIDE.

3. Case-Study: Open BEAGLE

Open BEAGLE is a black box EC framework coded in C++. The recursive acronym BEAGLE means *the Beagle Engine is an Advanced Genetic Learning Environment*[l]. Beagle is also the name of the English vessel on which the naturalist Charles Darwin did his famous world tour. The name Beagle has been used in the 1980's for a pattern recognition software developed by Forsyth and based on evolutionary principles[12]. The adjective Open has been added to the name of the framework to distinguish it from Forsyth's software, and also to insist on the open source aspect of the project. The project started in 1998 in the form of a GP library coded in C++. This first prototype has been completely re-written in 1999 in order to resolve some fundamental problems in the architecture. In the years 2001-2002 the software has been again re-written from scratch, in order to make it a generic EC framework. In 2002 Open BEAGLE was publicly launched on the Web[14]. At the end of year 2003 the development of the framework was moved on SourceForge.net[m], which is a collaborative development Web site offering different services. The framework developments follow an open source methodology[35]. Future contributions from external users will be evaluated and integrated to the framework if they are interesting.

3.1. *Open BEAGLE Architecture*

The framework architecture follows the OO programming principles, where abstractions are represented by loosely coupled objects and where it is common and easy to reuse the code. Open BEAGLE architecture is divided into three different levels as presented in Figure 1. OO foundations forms the basis as an OO extension of the C++ and the Standard Template Library (STL). The generic framework is built on these foundations. It is composed of elements characterizing all types of EC. Finally, different modules specialize the generic framework by implementing specific EC flavors.

---

[k]http://beagle.gel.ulaval.ca
[l]In French, the acronym means *Beagle est un Environnement d'Apprentissage Génétique Logiciel Évolué.*
[m]http://sourceforge.net/projects/beagle

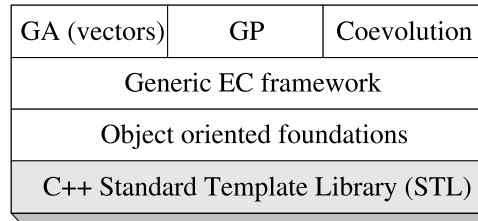| GA (vectors)	GP	Coevolution
Generic EC framework		
Object oriented foundations		
C++ Standard Template Library (STL)		

Fig. 1.   Architecture of Open BEAGLE framework.

3.2. *Object Oriented Foundations*

The OO foundations form the basis of the Open BEAGLE architecture. They are inspired from design patterns[15] and other environments such as the STL[32], the Java library[6], and CORBA[16].

Open BEAGLE C++ classes are all derived from the same abstract class `Object`. This class contains a set of functionalities like a reference counter that, in conjunction with intelligent pointers, allows for automated management of memory deallocation such as in high-level OO languages. Open BEAGLE relies heavily on inheritance by polymorphism. This means that objects must be dynamically instantiated. It is difficult to copy or clone a given object when its exact type is unknown. This is why allocators have been integrated to the framework. These allocators behave like object factories that can allocate, clone, and copy a specific type of object. A generic object container is also integrated to Open BEAGLE OO foundations. The container is a dynamic array of Open BEAGLE object pointers and is compatible with the generic container interface of STL. It uses an allocator to instantiate the contained objects.

Open BEAGLE XML files contain the population representation, parameter values and evolution results. The way they are read and written is an important characteristic of an EC framework. The XML (eXtensible Markup Language)[5] is perfect for modeling data since it is flexible, standard, understandable by humans, and easy to edit. Any XML file format can be transformed into another XML file format using XSLT (eXtensible Stylesheet Language Transformations)[7], as long as the necessary information is present and correctly tagged. This is important since it allows backward compatibility for file format changes, the interaction with other systems[n], the use of XML files[o], and the transformation of XML files to XHTML files for data visualization in a Web browser. Open BEAGLE includes classes to read and to write XML that are compatible with standard C++ I/O streams. Open BEAGLE classes know how to read and write themselves in XML. All this enables a complete integration of the XML language within the system.

---

[n]One can imagine that it would be easy to develop a tool that convert both EAML files[47] and Open BEAGLE files.
[o]And even some file text formats.

3.3. *Generic EC Framework*

The generic EC framework is an extension of the OO foundations. It offers a solid basis to implement different EC. It is composed of a generic structure of populations, an evolution system, and a set of operators included in an evolver. All of the generic EC framework components are integrated together as modules and can be replaced or specialized independently. This modular design provides a lot of flexibility and simplifies the implementation of any EC flavor.

In Open BEAGLE populations are structured into four hierarchical levels: vivarium, demes, individuals, and genotypes (see Figure 2). The *vivarium* includes
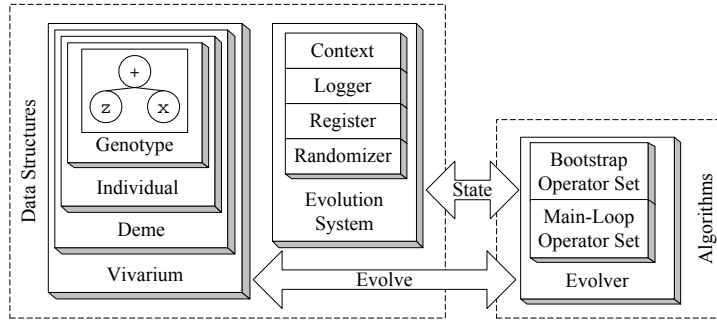

Fig. 2.   Open BEAGLE generic EC framework.

statistics on the last generation, the hall-of-fame containing the best-of-run individuals, and all individuals present in the evolutionary system. These individuals are divided into demes[24]. A *deme* is a closed environment where a group of individuals evolves independently. A deme also includes statistics on the last generation and a hall-of-fame with its best-of-run individuals. At each generation, individuals can migrate between the demes of a vivarium.

Individuals represent potential solutions to a problem. An individual can be defined by two types of data: its fitness measure (in a given environment) and one or more genotypes. The *genotype* contains the genetic description of an individual. In the generic EC framework the genotype is an interface that must be specialized in a specialized framework. For example, the genotype in GP is defined as a tree. The organization of individuals, genotypes, and fitness measures conforms to the genericity criteria of generic representation and fitness, as presented in Section 2.4.

The framework also includes an *evolution system* which is composed of four components: the context allocator, the register, the logger, and the randomizer (see Figure 2). The framework *context* is the present state of the evolving process. It includes essential data such as the presently processed deme, individual, and genotype, as well as the present generation number. For certain EA, a more specialized context can be defined. For example, a stack associated with the GP tree presently

processed (a genotype) is added to the context in the specialized GP framework. The concept of context in Open BEAGLE is similar to the execution context in computers, which involves register values, counters, and pointers.

Since Open BEAGLE parameters are distributed in many different elements, an agent named the *register* is used to centralize information. Parameters are stored in the register as an association between an identifier and the value held in an Open BEAGLE object. Elements such as operators can dynamically access these parameters. The register is also responsible for reading the XML configuration files. Thus the register implements mechanisms that conform to the genericity criterion of parameter management.

The *logger* acts as a user interface, processing all messages generated by the framework. These messages are associated with a type (architectural entity), a class (the C++ type of the object at the origin of the message), and a verbosity level. The logger can also be configured to output only the messages with a verbose level less or equal to a given value. It is possible to specialize the logger to transmit the output messages to different entities, for example a graphical interface. The default logger transmits output messages in XML format to a file and/or the console. Moreover, the logger can be very practical for application debugging, by using a high verbosity level. Thus the logger conforms to the specifications of the genericity criterion of a configurable output, as defined in Section 2.4.

The *randomizer* can generate random integers or floating point numbers according to uniform or Gaussian distributions. The generator's seed can be set to an arbitrary value. This value is recorded in the register, allowing for evolution replication.

Operators and evolvers are central concepts of the framework. The evolving process as implemented in Open BEAGLE consists in a sequence of operations that are iteratively applied to the demes of the vivarium. Each genetic operation is defined as an operator. The *evolver* has two operator sets: the bootstrap operator set and the main-loop operator set. The *bootstrap operator* set is the list of operations applied to each deme to construct an initial population. The *main-loop operator* set is the list of operations to iteratively apply to each deme at each generation, starting from generation 1. The operators and evolvers model is based on the Strategy design pattern, for the particular case of EC. Figure 3 presents the Open BEAGLE XML configuration of a generational GA evolver.

In the bootstrap operator set, the `GA-InitBitStrOp` operator generates the initial population, evaluates the fitness with the `MyEvalOp` operator, and computes statistics on this population with the `StatsCalcFitnessSimpleOp` operator. In the main-loop operator set, a tournament selection operation is applied by the `SelectTournamentOp` operator, then a one-point crossover operator (`GA-CrossoverOnePointBitStrOp`) and bit inversion mutation operator (`GA-MutationFlipBitStrOp`) are applied. The different operators retrieve their parameters from the register (e.g. crossover and mutation probabilities). There-

14   *Christian Gagné and Marc Parizeau*

```
1    <?xml version="1.0"?>
2    <Beagle>
3      <Evolver>
4        <BootStrapSet>
5          <GA-InitBitStrOp/>
6          <MyEvalOp/>
7          <StatsCalcFitnessSimpleOp/>
8        </BootStrapSet>
9        <MainLoopSet>
10         <SelectTournamentOp/>
11         <GA-CrossoverOnePointBitStrOp/>
12         <GA-MutationFlipBitStrOp/>
13         <MyEvalOp/>
14         <StatsCalcFitnessSimpleOp/>
15         <TermMaxGenOp/>
16         <MilestoneWriteOp/>
17       </MainLoopSet>
18     </Evolver>
19   </Beagle>
```

Fig. 3.   Open BEAGLE XML configuration of a generational bit string GA evolver.

after the fitness evaluation operator (`MyEvalOp`) and statistics computation operator (`StatsCalcFitnessSimpleOp`) are executed. Operator `TermMaxGenOp` is then used to check whether the maximum number of generations has been reached, in which case it sets a flag in the context that will force the evolver to stop at the end of the main loop. Finally, operator `MilestoneWriteOp` is used to write at regular intervals an XML file with the actual evolution state (parameters, evolver, statistics, population). This file can be used to analyze results and even to restart the evolutionary process.

This evolver and operators model works well in the case of generational EC, since only one evolutionary process mechanism is necessary at the population level. However, for other types of EC such as ES or steady-state GA, an evolutionary mechanism at the individual level is necessary. For this purpose, the breeder model has been developed. It consists of an extension to the evolver and operators model. It has two principal architectural elements: the replacement strategies, which are standard operators present in the bootstrap and main-loop operator sets, and the breeders operators, which can be connected together as well as to the replacement strategies to perform evolution at the individual level.

A breeder processing pipeline is a tree structure with a replacement strategy at the root and breeder operators associated with other nodes. A replacement strategy calls the breeder operator sub-trees to generate new individuals with its characteristic algorithm. These calls are generally parametrized by the breeding probabilities of each sub-tree. New individuals are inserted into the population according to the specific algorithm of the replacement strategy, hence the name. Each call to a

breeder operator sub-tree results in the generation of a new bred individual. A non-terminal node performs an operation on the individuals received from its children, and then returns the result to its parent. A terminal node consists in either the selection of an individual in the present population that is returned to its parent, or the initialization of a new individual.

For example, in the steady-state replacement strategy, a new generation of individuals is produced by calling its sub-trees as often as there are individuals in the population (one call = one individual bred). The probability of calling each sub-tree is given by their breeding probability. Assume that there are three sub-trees, representing respectively a crossover, a mutation and a reproduction operation. The breeding probability of each sub-tree is respectively the crossover, mutation, and reproduction probability. Figure 4 presents the configuration in XML of such a steady-state GA evolver.

```
1    <?xml version="1.0"?>
2    <Beagle>
3      <Evolver>
4        <BootStrapSet>
5          <GA-InitBitStrOp/>
6          <MyEvalOp/>
7          <StatsCalcFitnessSimpleOp/>
8        </BootStrapSet>
9        <MainLoopSet>
10         <SteadyStateOp>
11           <MyEvalOp>
12             <GA-CrossoverOnePointBitStrOp matingpb="ga.cx1p.prob">
13               <SelectTournamentOp/>
14               <SelectTournamentOp/>
15             </GA-CrossoverOnePointBitStrOp>
16           </MyEvalOp>
17           <MyEvalOp>
18             <GA-MutationFlipBitStrOp mutationpb="ga.mutflip.indpb">
19               <SelectTournamentOp/>
20             </GA-MutationFlipBitStrOp>
21           </MyEvalOp>
22           <SelectTournamentOp repropb="ec.repro.prob"/>
23         </SteadyStateOp>
24         <StatsCalcFitnessSimpleOp/>
25         <TermMaxGenOp/>
26         <MilestoneWriteOp/>
27       </MainLoopSet>
28     </Evolver>
29   </Beagle>
```

Fig. 4.   Open BEAGLE XML configuration of a steady-state bit string GA evolver.

The evaluation operator is located at the root of the crossover and mutation

sub-trees as new individuals generated in a steady-state algorithm must have a valid fitness before being integrated in the population. This is not necessary for the reproduction sub-tree, which is composed of only the selection operator of line 22 in Figure 4. The selection operator generates individuals that are copies of existing individuals for which the fitness is already valid. In general, the operators used by generational and steady-state evolvers are the same. However, they function in different modes of operation: they process either a single individual in breeder mode, or a complete deme in generational mode. These two modes of operation are controlled through calls to distinct methods.

The concepts of evolver, operators, and breeder conform to the generic operators and the generic evolutionary model criteria presented in Section 2.4. Also, these are generic mechanisms for component arrangement that are mostly responsible for the Open BEAGLE's black-box development stage.

3.4. *Specialized Frameworks*

Specialized frameworks are built upon the generic framework. Three specific frameworks are actually implemented: linear representation framework with a support for bit string representations, real-value vectors and ES (x_i, σ_i) pairs vectors; a tree-based GP framework; and a co-evolution framework for the simultaneous evolution of several species. The user can implement its own EC flavor, from an existing specialized framework or directly from the generic EC framework.

The linear representation EC framework, also called the GA framework, is quite simple. For each of the three representations implemented, it defines a specific genotype and includes initialization operators, three generic crossover operators (one-point, two-points, and uniform crossover) as well as specific mutation operators (bit inversion, real-value mutation, and ES adaptative mutation). The specialized framework also includes functionalities allowing the transformation of a bit string into a real number vector.

The GP specialized framework is more complex. New mechanisms specific to the paradigm have been defined. To genetically program a computer, two specific points relative to the problem field must be established. The first point is the datum type, that is the data (variables) type that will be managed by genetic programs. Once the datum is defined, the user needs to specify the primitives that will be used to build GP individuals. A primitive is an operation associated with the nodes of the GP tree. It is specific to the application. It corresponds to terminals and to functions used in an application, as described by Koza[21]. Primitives process and return variables of the datum type used.

In Open BEAGLE the datum type must be derived from the root class `Object`. This can be done by using an Open BEAGLE predefined type or by adapting a foreign type to the class interface. To create a primitive that can be used in GP trees, the user must define a concrete class derived from the abstract class of primitives, where a pure virtual function must be overdefined to implement the
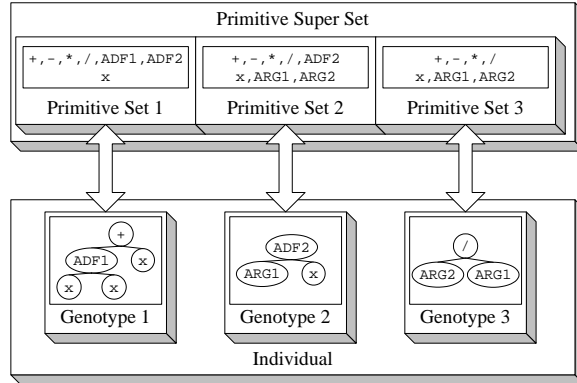
Fig. 5. Relation between primitive sets and GP trees.

specific operation of the primitive. The interface of the primitive class allows the use of advanced GP functionalities such as strongly-typed GP[31] and ephemeral random constants[21].

The primitives used in a given application must be inserted in the set of usable primitives. GP trees are generated from the primitives of this set. The super-set of primitives is an extension of the evolution system including one or more sets of primitives. The number of trees (genotypes) of GP individuals is determined by the number of sets in the super-set, as illustrated in Figure 5. Among other things, this mechanism allows the implementation of automatically defined functions[21,22].

Finally, the co-evolution framework[2,17] is different from the other two because it does not implement a new representation. It rather implements mechanisms for evolving simultaneously many species of individuals. This framework is based on concurrent programming, where each thread evolves one species. The co-evolution framework defines a fitness evaluation operator allowing the matching of individuals from different species (threads). Otherwise, individuals are evolved using standard mechanisms defined in the other specialized frameworks, or by the user.

3.5. *Code Example: OneMax*

Despite the inherent complexity of a generic EC framework, the use of Open BEA-GLE is relatively simple for a novice programmer. The components have default values and policies that are suitable for most simple applications. The user is only required to define a fitness evaluation operator and a main method that initializes different components. Figure 6 presents an evaluation operator implementation for the classical GA bit string example OneMax. The problem consists in searching for bit strings that have a maximum number of bits set to "1". The corresponding main routine is presented in Figure 7.

Line 5 in Figure 6 constructs a fitness operator named `OneMaxEvalOp`. Lines 6 to 15 corresponds to the function called to evaluate an individual fitness. Lines 9

18   *Christian Gagné and Marc Parizeau*

```
1    #include "beagle/GA.hpp"
2    using namespace Beagle;
3    class OneMaxEvalOp : public EvaluationOp {
4    public:
5      OneMaxEvalOp() : EvaluationOp("OneMaxEvalOp") { }
6      virtual Fitness::Handle evaluate(Individual& inIndividual,
7                                       Context& ioContext)
8      {
9        GA::BitString::Handle lBitString =
10         castHandleT<GA::BitString>(inIndividual[0]);
11       unsigned int lCount = 0;
12       for(unsigned int i=0; i<lBitString->size(); ++i)
13         if((*lBitString)[i]) ++lCount;
14       return new FitnessSimple(float(lCount));
15     }
16   };
```

Fig. 6.   Fitness evaluation operator for the OneMax problem.

```
1    #include <cstdlib>
2    #include <iostream>
3    #include "beagle/GA.hpp"
4    #include "OneMaxEvalOp.hpp"
5    using namespace Beagle;
6    int main(int argc, char** argv) {
7      try {
8        GA::BitString::Alloc::Handle lBSAlloc = new GA::BitString::Alloc;
9        Vivarium::Handle lVivarium = new Vivarium(lBSAlloc);
10       OneMaxEvalOp::Handle lEvalOp = new OneMaxEvalOp;
11       const unsigned int lNumberOfBits = 20;
12       GA::EvolverBitString lEvolver(lEvalOp, lNumberOfBits);
13       System::Handle lSystem = new System;
14       lEvolver.initialize(lSystem, argc, argv);
15       lEvolver.evolve(lVivarium);
16     }
17     catch(Exception& inException) {
18       inException.terminate(std::cerr);
19     }
20     return 0;
21   }
```

Fig. 7.   Main routine for the OneMax problem.

and 10 cast the generic individual to evaluate into a bit string individual. Lines 11 to 13 count the number of ones in the bit string while line 14 returns the fitness measure, that is a single real value to maximize.

For the main routine of the application presented in Figure 7, lines 8 and 9 build a bit string population. Line 10 instantiates the fitness evaluation operator

defined above. Lines 11 and 12 define a bit string GA evolver where individuals are initialized as a string of 20 bits each. Line 13 creates the evolution system as defined in Figure 5. Line 14 initializes the evolver and the evolution system, parses the command line, and reads configuration files. A generational GA similar to the one in Figure 3 is used by default. If the user wants to use a steady-state GA, for example, he must define a XML configuration file similar to the one in Figure 4[p] and execute the program with an option on the command line referring to the proper configuration file. Finally, the evolution is launched at line 15. The entire routine is in a try-catch block in order to intercept exceptions which may be thrown by Open BEAGLE, if a problem is detected at runtime. This example, as well as many others, are packaged together with the source code of Open BEAGLE.

4. Conclusion

Current and future needs of EC researchers include generic software tools that enable the rapid development of new paradigms or a mixture of old and new ones. Historical distinctions between genetic algorithms, genetic programming, evolution strategies, etc., are becoming less and less relevant, as not one of them is best suited for solving every possible problem. Different sub-problems may require different representations and/or different evolutionary models and heuristics. Different species of solutions may need to co-evolve. Some optimization problems may intrinsically involve multiple objectives. In order to develop prototypes of complex EC solutions that integrate all of these paradigms (and others), well-designed tools must promote code reuse and adaptability.

In this paper, six fundamental criteria were proposed to evaluate the genericity of existing EC frameworks, or guide the development of new ones: 1) possibility of replacing the internal representation of individuals, including the possibility of multiple representations; 2) possibility of replacing the fitness measure; 3) ability to define or add any type of operator; 4) capacity of changing the evolutionary model in order to enable different evolutionary algorithms; 5) capacity of dynamically changing any parameter, as well as adding or removing them, especially those of operators; and 6) availability of flexible output mechanisms to enable the periodical safeguard of the evolution state, and the retrieval of different types of statistics for its monitoring and control. These criteria have been used to evaluate six well-known and freely available EC frameworks. According to these criteria, results have shown that at most three of them can support a claim for a reasonable level of genericity.

One of these generic frameworks is Open BEAGLE which has been designed and implemented by the authors over a six year period. Its current release is in fact a third complete rewrite and re-design. The principles presented here stem from this experience. Through a case study of Open BEAGLE, we have also shown how the genericity principles can be instantiated in a concrete object oriented design.

---

[p]Taking care to replace `MyEvalOp` tags by `OneMaxEvalOp` tags.

Acknowledgments

References

1. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol, UK, 2000.
2. Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Evolutionary Computation 2: Advanced Algorithms and Operators*. Institute of Physics Publishing, Bristol, UK, 2000.
3. Thomas Bäck, Ulrich Hammel, and Hans-Paul Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, April 1997.
4. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, 1998.
5. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - W3C recommendation 10-february-1998. http://www.w3.org/TR/1998/REC-xml-19980210, 1998.
6. Mary Campione and Kathy Walrath. *The Java Tutorial*. Addison-Wesley, Reading, MA, USA, 2 edition, 1998.
7. James Clark. XSL transformations (XSLT) 1.0 - W3C recommendation 16-november-1999. http://www.w3.org/TR/1999/REC-xslt-19991116, 1999.
8. Carlos A. Coello Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, 2002.
9. Pierre Collet, Evelyne Lutton, Marc Schoenauer, and Jean Louchet. Take it EASEA. In *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, volume 1917 of *LNCS*, Paris, France, September 16-20 2000. Springer-Verlag.
10. John Cona. Developing a genetic programming system. *AI Expert*, pages 20–29, February 1995.
11. Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.
12. Richard Forsyth. BEAGLE A Darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.
13. Christian Gagné and Marc Parizeau. Open BEAGLE: A new C++ evolutionary computation framework. In *Proc. of the Genetic and Evolutionary Computations COnference (GECCO) 2002*, page 888, New York, NY, USA, July 9-13 2002.
14. Christian Gagné and Marc Parizeau. Open BEAGLE: An evolutionary computation framework in C++. http://beagle.gel.ulaval.ca, 2004.
15. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1994.
16. Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, USA, 1999.
17. W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42:228–234, 1990.

18. John M. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.
19. Maarten Keijzer, Juan J. Merelo, Gustavo Romero, and Marc Schoenauer. Evolving objects: a general purpose evolutionary computation library. In *EA-01, Evolution Artificielle, 5th International Conference in Evolutionary Algorithms*, 2001.
20. Mike J. Keith and Martin C. Martin. Genetic programming in C++: Implementation issues. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, 1994.
21. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
22. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.
23. Natalio Krasnogor and Jim Smith. MAFRA: A java memetic algorithms framework. In *Genetic and Evolutionary Computation 2000 Workshops*, pages 125–131, Las Vegas, Nevada, USA, 2000.
24. William B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!* Kluwer Academic Publishers, Boston, MA, USA, 1998.
25. Tom Lenaerts and Bernard Manderick. Building a genetic programming framework: The added-value of design patterns. In *Proceedings of EuroGP'98*, pages 196–208, 1998.
26. Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and Alexander Chircop. ECJ 13: A java-based evolutionary computation and genetic programming research system. http://cs.gmu.edu/~eclab/projects/ecj, 2005.
27. Robert M. MacCallum. Introducing a perl genetic programming system – and can meta-evolution solve the bloat problem? In *Proceedings of the Sixth European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 364–373, Essex, UK, 2003. Springer Verlag.
28. Nicholas Freitag McPhee, Nicholas J. Hopper, and Mitchell L. Reierson. Sutherland: An extensible object-oriented software framework for evolutionary computation. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 241, University of Wisconsin, Madison, Wisconsin, USA, 1998. Morgan Kaufmann.
29. Juan J. Merelo, Maarten Keijzer, Marc Schoenauer, Jeroen Eggermont, Sébastien Cahon, and Olivier König. Evolving Objects: Evolutionary computation framework. http://eodev.sourceforge.net, 2004.
30. Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, third edition, 1996.
31. David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.
32. David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, USA, 1996.
33. Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.
34. Bill Punch and Douglas Zongker. lil-gp 1.1 beta. http://garage.cse.msu.edu/software/lil-gp, 1998.
35. Eric S. Raymond. The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary. http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar, 2000.

36. Ingo Rechenberg. *Evolutionsstrategie*. Friedrich Frommann Verlag (Günther Holzboog KG), Stuttgart, 1973.

37. Jose L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. Genetic algorithm programming environments. *IEEE Computer*, 27(6):28–43, 1994.

38. Don Roberts and Ralph E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.

39. Andreas Rummler and Gerd Scarbata. ealib – a java framework for implementations of evolutionary algorithms. In *Computational Intelligence. Theory and Applications (Fuzzy Days 2001)*, volume 2206 of *LNCS*, pages 92–102, Dortmund, Germany, 2001. Springer Verlag.

40. Sara Silva. GPLAB: A genetic programming toolbox for MATLAB. http://gplab.sourceforge.net, 2005.

41. Sara Silva and Jonas Almeida. GPLAB – a genetic programming toolbox for MATLAB. In *Proc. of the Nordic MATLAB Conference (NMC-2003)*, pages 273–278, 2003.

42. Andy Singleton. Genetic programming with C++. *BYTE*, pages 171–176, February 1994.

43. K. C. Tan, Tong H. Lee, D. Khoo, and E. F. Khor. A multiobjective evolutionary algorithm toolbox for computer-aided multiobjective optimization. *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, 31(4):537–556, August 2001.

44. Astro Teller and Manuela Veloso. PADO: A new learning architecture for object recognition. In Katsushi Ikeuchi and Manuela Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

45. Zoltán Tóth and Gabriella Kókai. An experimental evaluation of the generic evolutionary algorithms programming library. In *FGML 2001, Treffen der Fachgruppe maschinelles Lernen der Gesellschaft fur Informatik*, Dortmund, Germany, 2001.

46. Róbert Ványi. Object oriented design and implementation of a general evolutionary algorithm. In *Genetic and Evolutionary Computation – GECCO-2004*, volume 3103 of *LNCS*, pages 1275–1286, Seattle, 2004. Springer-Verlag.

47. Christian Veenhuis, Katrin Franke, and Mario Köppen. A semantic model for evolutionary computation. In *6th International Conference on Soft Computing*, Fukuoka, Japan, 2000.

48. Matthew Wall. GAlib: A C++ library of genetic algorithm components. http://lancet.mit.edu/ga, 2000.

49. Garnet C. Wilson, A. McIntyre, and Malcolm I. Heywood. Resource review: Three open source systems for evolving programs – lilgp, ECJ and grammatical evolution. *Genetic Programming and Evolvable Machines*, 5(1):103–105, March 2004.