Open BEAGLE: A New Versatile C++ Framework for Evolutionary Computations

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.
E-mail: {cgagne,parizeau}@gel.ulaval.ca

Abstract

This paper introduces a new Evolutionary Computation (EC) framework named *Open BEAGLE*, that we have been developing and improving since 1999. Coded in C++, this framework offers solid object oriented foundations based on design patterns. It contains a basic generic EC framework on which other specialized frameworks can easily be constructed. Release 1.0 of Open BEAGLE implements two specialized frameworks: a simple genetic algorithms framework, and a complete genetic programming framework. Its power and ease of use is demonstrated through an example of the latter for the classic symbolic regression problem.

1 Introduction

Open BEAGLE is an Evolutionary Computation (EC) framework entirely coded in C++. The recursive acronym BEAGLE means *the Beagle Engine is an Advanced Genetic Learning Environment*[1]. Beagle was also the name of the English vessel on which Charles Darwin embarked as a naturalist for his famous circumnavigation of the world. The name Beagle was previously used in the 1980's for pattern recognition software based on EC principles (Forsyth, 1981). The adjective *Open* was added to our framework to distinguish it from this software, and also to insist on the open source aspect of the project.

[1]In French, *Beagle est un Environnement d'Apprentissage Génétique Logiciel Évolué*.

The Open BEAGLE architecture follows the principles of Object Oriented (OO) programming, where some abstractions are represented by loosely coupled objects and where it is common and easy to reuse code. Open BEAGLE was designed with the objectives of providing an EC framework that is generic, user friendly, portable, efficient, robust, elegant and free.

Genericity With Open BEAGLE, the user can execute any kind of EC, as far as it fulfills some minimum requirements. The necessary condition is to have a population of individuals to which a sequence of evolving operations is iteratively applied. So far, two specialized frameworks were implemented using Open BEAGLE: Genetic Algorithms (GA) and Genetic Programming (GP). An Evolutionary Strategies (ES) framework is also planned for a future release. The user can take any of these specialized frameworks and modify them further to create his own specialized flavor of evolutionary algorithms.

User Friendliness Open BEAGLE provides several mechanisms that offer a user friendly programming interface. For example, memory management of dynamically allocated objects is greatly simplified by the use of reference counting and automatic garbage collection.

Portability The Open BEAGLE code is compliant with the C++ ANSI/ISO 3 standard. It requires the *Standard Template Library* (STL) (Musser and Saini, 1996). No specific calls are made to the operating system nor to the hardware.

Efficiency To insure efficient execution, particular attention was given to optimization of critical code sections. Detailed execution profiles of these sections were done. Also, the fact that Open BEAGLE is written in C++ contributes to its overall good performance.

Robustness Many validation statements are embedded into the code to ensure correct operation and to inform the user when there is a problem. Robust mechanisms for periodically saving the current evolution state have also been implemented in order to enable automatic restart of interrupted evolutions.

Elegance The interface of Open BEAGLE was developed with care. Great efforts were invested in designing a coherent software package that follows good OO and generic programming principles. Moreover, strict programming rules were enforced to make the C++ code easy to read, understand, and modify.

Free Sourceness The source code of Open BEAGLE is free, available under the GNU Lesser General Public License (LGPL) (Free Software Foundation Inc., 2000). It can thus be distributed and modified without any fee. Open BEAGLE is available on the Web at `http://www.gel.ulaval.ca/~beagle`.

2 Survey of Existing EC software

Although many EC software systems have been developed over the past decade, only a few are generic enough to implement different flavors of algorithms. We now briefly review three of the most significant ones, that were designed using OO programming paradigms, and that we feel are comparable to Open BEAGLE: *gpc++* (Fraser, 1994), *EO* (Merelo et al., 2001) and *ECJ* (Luke, 2001).

gpc++ is one of the first known C++ framework for tree-based GP[2]. Although gpc++ and Open BEAGLE share some philosophic and implementation aspects, gpc++ contains C-like constructs which do not promote good OO practices. For example, the GP tree is structured as a prefix list of the function and terminal names (a list of `char*`). To evaluate the trees, the user needs to define a complex switch case that is very hard to recycle. Also, gpc++ does not profit from design pattern (Lenaerts and Manderick, 1998).

EO, which stands for *Evolving Objects*, is a C++ framework for EC. Open BEAGLE and EO share some design concepts, essentially by separating the algorithms (like genetic operators) from the data structure (the populations for instance). But their implementations are somewhat different: EO uses generic programming extensively (sometimes called static polymorphism), as opposed to Open BEAGLE which uses some generic programming concepts to enhance the user experience, but mostly uses polymorphism by inheritance (sometimes called dynamic polymorphism). All evolutionary processes and parameters in EO must be fixed at compile time. This may enhance performance, but certainly degrades the user experience and reduces the overall flexibility of the architecture. At the time of writing, EO implements GA and ES. An experimental implementation of GP is also available.

[2]Strictly speaking, gpc++ is not a generic EC framework, but the comparison is still interesting.

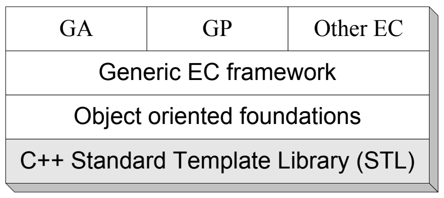
GA
GP
Other EC
Generic EC framework
Object oriented foundations
C++ Standard Template Library (STL)

Figure 1: Software architecture diagram.

Finally, ECJ is a complete, Java-based environment for EC. We chose ECJ among all available Java-based EC systems because it is a full featured, well-made OO system. Like Open BEAGLE, it is designed following an OO methodology, and uses polymorphism by inheritance extensively. However, although Java is a nice coherent language, it suffers from relatively poor execution speed[3] which is indeed a serious limitation for CPU-intensive tasks like EC.

3 Software Architecture

Open BEAGLE has a three level architecture, as illustrated in Figure 1. The foundations are located at the bottom, as an OO extension of the C++ language and STL. The EC generic framework is built on these foundations and is composed of features that are common to all Evolutionary Algorithms (EA). Finally, different independent modules specialize this generic framework, each module implementing a specific EA.

3.1 Object Oriented Foundation

The OO foundations are the basis of the Open BEAGLE architecture. They are inspired from *design patterns* (Gamma et al., 1994) and other environments such as STL (Musser and Saini, 1996), the *Java library* (Campione and Walrath, 1998), and *CORBA* (Henning and Vinoski, 1999).

[3]In principle, using an optimizing compiler that produce machine code, Java and C++ programs should be able to achieve comparable speeds. However, at the time of writing, several benchmarks on the Web report that this is not true in practice.

In Open BEAGLE, all classes are derived from an abstract `Object` class. As a complete C++ object, the Open BEAGLE object is an interface that includes a set of general operations. An object is composed of comparison functions, input/output functions, and functions to interact with the *reference counter*.

The use of reference counters and *smart pointers* simplifies the management of dynamically allocated objects. Indeed, each object owns a counter that keeps track of the number of existing references to itself. A smart pointer behaves like a standard C++ pointer, but also increments and decrements the object's reference counter when necessary. When an object's reference counter is equal to zero, the object is released. This mechanism allow the emulation of a *garbage collection* process, similar to the ones that can be found in higher level OO languages.

A smart pointer is associated to each object type. This association is done by declaring a `Handle` member type in all Open BEAGLE classes. For example, the smart pointer associated to type `MyClass` would be of member type `MyClass::Handle`. Furthermore, if there is an inheritance relationship between two classes, the same inheritance relationship exists between the two `Handle` member types of these classes. For instance, if class `DerivedClass` inherits from class `BaseClass`, then member type `DerivedClass::Handle` inherits from member type `BaseClass::Handle`. This very important mechanism in BEAGLE allows member type `Handle` to perfectly simulate the standard C++ pointers, even in the context of polymorphism by inheritance. This mechanism is implemented through templates that do most of the work. The programmer who creates a new subclass needs only declare a typedef using that template.

Open BEAGLE makes heavy use of polymorphism by inheritance, which means that object instances must be dynamically allocated. Moreover, it is difficult to copy or clone a given object when its exact type is unknown. Thus, *allocators* have been implemented, i.e. object factories that can allocate, clone and copy a given data type. There is an allocator, named `Alloc`, associated with each Open BEAGLE class. For example, a class named `MyClass` incorporates a member type `MyClass::Alloc`, that can be used to allocate, copy and clone objects of type `MyClass`.

A generic object container has also been incorporated into the OO foundation of Open BEAGLE: the *bag*. The bag is a random access table of BEAGLE objects. It is implemented as a dynamic array of smart pointers (i.e. `std::vector<Object::Handle>`). It is thus a container that can be manipulated by the generic algorithms of STL. Since the bag is also a BEAGLE object, it can be referred to by smart pointers, and it is possible to create bags of bags. Just like for smart pointers and allocators, a bag member type is associated with each object. For instance, for type `MyClass`, the associated bag member type is `MyClass::Bag`. Moreover, when there is an inheritance relationship between two classes, it is also respected by their embedded bag member types, just like for the `Handle` and `Alloc`.

Data input and output is also an important aspect of any high-level framework. The C++ language integrates I/O streams that allow objects to be inserted into and extracted from such media as files, console or memory. By combining the C++ I/O stream concept with the XML format, we developed XML I/O streams. XML (eXtensible Markup Language (Anderson et al., 2000)) is an ideal language for representing data because it is flexible, standardized, readable by humans, and easily editable.

Data insertion and extraction into Open BEAGLE XML streams are identical to data insertion and extraction into standard C++ streams. Operators `<<` and `>>` are defined for the `Object` class just like they are defined for C++ atomic types. Their calls are simply mapped to associated read and write virtual member functions. In these functions, class components are inserted and extracted by calling their respective insertion and extraction operators. In addition to these operators, some interface member functions are defined in the XML stream class to insert and extract XML tags and attributes.

The functionalities exposed here emphasize the importance of pure OO concepts in Open BEAGLE. However, forcing all types to inherit from superclass `Object` can be annoying, especially when the user has already developed his own classes or when other libraries are used. To simplify the use of Open BEAGLE in this context, a class template was written, named `WrapperT`, to adapt foreign types to the `Object` interface. Wrappers of C++ atomic types are also predefined in Open BEAGLE. The wrapper concept is based on the *adapter* design pattern.

3.2 Generic EC Framework

The generic EC framework is the extension of OO foundations. It offers a solid basis for implementing Evolutionary Algorithms (EA). It is composed of a generic structure for *populations*, an *evolution system* and a set of *operators* packed in an *evolver*. All components of the generic EC framework are integrated together as modules that can be replaced or specialized independently. This modular design gives much

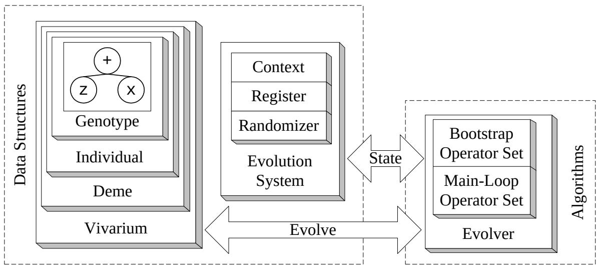

Figure 2: Generic EC Framework Architecture

Table 1: Generational evolver.

Bootstrap Operators	Main-loop Operators
1 `read parameters;`	1 `read parameters;`
2 `if(restart)`	2 `apply tournament selection;`
3 `read milestone;`	3 `apply crossovers;`
`else`	4 `apply mutations;`
4 `generate population;`	5 `compute fitness;`
5 `compute fitness;`	6 `apply random ring migration;`
6 `compute statistics;`	7 `compute statistics;`
7 `display statistics;`	8 `display statistics;`
`end if;`	9 `check for termination;`
8 `check for termination;`	10 `write milestone;`
9 `write milestone;`	

versatility to the framework and simplifies the implementation of any EA. In Open BEAGLE, evolving populations are structured into four hierarchical levels: *vivarium*, *demes*, *individuals* and *genotypes* (see Figure 2). The vivarium encompasses all individuals currently living in the evolving system. These individuals are aggregated into one or more demes (Langdon, 1998) that represent groups of individuals evolving in closed environments. A deme saves its statistics and has a *hall-of-fame* of the best individuals. In general, at each generation, some individuals can migrate in between the different demes of the vivarium. The vivarium is implemented as a bag of demes, and demes are implemented as bags of individuals.

The *individuals* represent potential solutions to a problem. An individual is defined by two types of data: the measure of its fitness (within its environment) and one or more genotypes. The *genotype* is the genetic data of an individual. The generic EC framework implements an interface describing a generic genotype model. This interface must be specialized in a specific framework. For example, in the case of GP, the genotype is specialized into an expression tree class.

In the generic framework, there is also an *evolution system* that contains the configuration of the genetic engine. It is made of three components: a *context* allocator, the *register*, and a *randomizer* (see Figure 2). During the evolving process, a *context* represents the current state of the evolution. The basic context of the generic framework gives some essential contextual information, like references to an actual deme, individual and genotype, and also the current generation number. For some EA, a more specialized context could be defined. For instance, in the GP framework a call stack related to the GP tree (the genotype) is added to the context. This context concept is similar to the execution context in a computer, which comprises the values of the different registers, counters and pointers of the CPU.

Given that parameters of Open BEAGLE are distributed in different entities, an agent named the *register* is used to centralize information. References to BEAGLE objects can be associated to tags in the register and entities such as operators can dynamically add, delete, access, and modify them. The register is also responsible for parsing XML configuration files.

The *randomizer* is the system's random number generator. It can generate integer or floating-point numbers following uniform or gaussian distributions. The generator seed can be assigned to an arbitrary value. This value is inscribed into the register so that the user can reproduce any evolution. By default, the seed is initialized with the current clock value of the computer.

The *operators* and *evolver* are central concepts of the generic EC framework. In Open BEAGLE, the evolving process is a sequence of operations that are iteratively applied on the demes of the vivarium. Each operation is defined as an operator. There are two operator sets in an evolver: the *bootstrap* and the *main-loop*. The bootstrap operator set is the list of operations to apply on each deme during initialization. The main-loop operator set is the list of operations iteratively applied on each deme, at each generation. This operators/evolver model is based on the *strategy* design pattern, applied to the specific case of EC.

For common EA, users need not define their own evolver and operators, standard ones can be used without modification. For example, Table 1 presents a standard generational evolver. Predefined operators include selection, crossover, mutation, statistics calculation, and many more. Only the fitness function requires to be specialized from an abstract evaluation operator.

In Table 1, operator 1 is called by the evolver at the start of each generation. It simply checks the register for the existence of a tag specifying the file name of an XML configuration file. If this file exists, then it is parsed and the register is updated with new tag values. In this way the user can modify parameters in between generations, without having to interrupt the evolution. In the bootstrap set, special `if-else`

Table 2: Steady-state evolver.

	Main-loop Operators
1	`read parameters;`
2	`steady-state loop;`
3	`apply random ring migration;`
4	`compute statistics;`
5	`display statistics;`
6	`check for termination;`
7	`write milestone;`

operator enables the construction of conditional operator sets. In this case, the executed set of operators is determined by the boolean value of the `restart` tag. In Open BEAGLE, a milestone represents the contents of the register and vivarium at a given point in time. Operator 3 is there to read a milestone in case of an evolution restart. Otherwise, operators 4 through 7 are executed in sequence to initialize the vivarium. The termination check operator determines if the evolution should be halted. If so, it sets a flag into its context, instructing the evolver to halt at the end of the current loop.

With the operators/evolver concept, most EA can be implemented easily, just by modifying the set of operators in the evolver. An example of this flexibility is presented in Table 2, where a steady-state (Langdon, 1998) evolver is built simply by replacing operators 2 through 5 in the main-loop of Table 1, with a special steady-state loop operator which implement the steady-state procedure: choose either reproduction, mutation, or crossover; select individual(s) using tournaments; apply chosen genetic operation; replace random individuals; and compute fitness. Operators are given access to the system by the evolver through a context handle which allows them to interact with the vivarium, the register, and the randomizer, in order to accomplish mostly anything.

We claim that this operators/evolver concept is an important feature that offers great versatility for rapid software design of complex EC systems. For instance, using this approach, an EC system could be made to run on a cluster of computers by replacing operators in the evolver, thus shielding the user from much of the underlying complexity. Moreover, Open BEAGLE is fully reconfigurable, all of its components can be specialized through dynamic polymorphism.

3.3 Specialized Frameworks

The specialized frameworks are at the top level of the Open BEAGLE architecture. For the current release, two specialized frameworks have been implemented: a GA framework and a GP framework. Open BEAGLE is built in such a way that the user can either implement his own EA flavor from an existing framework, or he can build directly on the generic framework.

The GA framework is very simple. It defines a GA specific genotype that consists of a bit string, and genetic operators enabling bit string crossover and mutation. The specific framework also has some functionalities that enable the mapping of a bit string into a vector of floating numbers defined over a given range. With this simple framework, it is possible to do standard GA as presented in (Holland, 1975).

The GP framework is more elaborate. New mechanisms specific to the paradigm need to be defined. To genetically program a computer for an application of GP, two issues of the problem domain must be addressed. First, the user needs to define the *datum* type, that is the type of data (variables) that will be manipulated by the genetic programs. Once the datum defined, the *primitives* used for building GP individuals must also be specified. A primitive is also an application specific operation associated with the nodes of the GP trees. The primitives must process and return variables of the specific datum type. All primitive used for a given problem are inserted into a primitive set.

In Open BEAGLE, the datum type must be derived from the `Object` class. Generally, this can be done using a predefined Open BEAGLE type, or by adapting a foreign type using a wrapper. To create a primitive that can be used in GP trees, the user must define a concrete class derived from the abstract `GP::Primitive` class. A pure virtual function in this abstract class must be overloaded in order to implement the characteristic operation of the primitive. The interface of the abstract primitive also allow variations from basic GP. For instance, strongly-typed GP (Montana, 1995), and randomly generated ephemeral constants (Koza, 1992) can be implemented by overloading appropriate functions.

Primitives must be packaged into sets of usable primitives. It is from these sets that the GP trees are generated. A primitive superset is an extension of the evolution system, and there can be different sets of primitives for a given evolution. The number of these sets specifies the number of genotypes in individuals, as illustrated in Figure 3. This feature allows for the implementation of Automatically Defined Functions (ADF) (Koza, 1992; Koza, 1994).

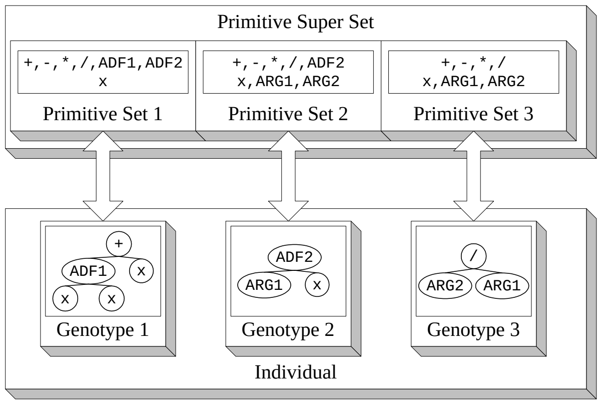

Figure 3: Relation between primitive sets and GP trees.

4 Simple GP Example

In less than a hundred lines of code, this section presents the implementation of a simple GP problem with Open BEAGLE. The aim is to demonstrate the ease of use of the environment, and at the same time to give additional insight into its programming philosophy. The problem is the classic symbolic regression of a simple polynomial function $p(x) = x^4 + x^3 + x^2 + x$ (Koza, 1992). The discussion follows a top-down approach. The `main` function is first exposed, then the implementation of a primitive, and finally the implementation of the fitness evaluation operator.

The `main` routine is presented in Figure 4. It follows the five steps needed to implement any GP application with Open BEAGLE:

```
1  #include <beagle/GP.hpp>
2  #include "SymbRegEvalOp.hpp"
3  using namespace Beagle;
4  int main(int argc, char *argv[])
5  {
6   try {
7     // 1: Build primitive set
8     GP::PrimitiveSet::Handle lSet = new GP::PrimitiveSet;
9     lSet->insert(new GP::Add);
10    lSet->insert(new GP::Subtract);
11    lSet->insert(new GP::Multiply);
12    lSet->insert(new GP::Divide);
13    lSet->insert(new GP::TokenT<Double>("x"));
14    // 2: Build a system
15    System::Handle lSystem = new GP::System(lSet);
16    // 3: Build operators
17    EvaluationOp::Handle lEvalOp = new SymbRegEvalOp;
18    TerminationOp::Handle lTermOp = new MaxGenerationTermOp;
19    // 4: Build an evolver and a vivarium
20    Evolver::Handle lEvolver =
21          new GP::GenerationalEvolver(lEvalOp,lTermOp);
22    Vivarium::Handle lVivarium = new GP::Vivarium;
23    // 5: Initialize and evolve the vivarium
24    lEvolver->initialize(lSystem,argc,argv);
25    lEvolver->evolve(lVivarium);
26  }
27  catch(Exception& inException) {
28    inException.terminate();
29  }
30  return 0;
31 }
```

Figure 4: `main` for the symbolic regression problem.

1. Build the primitives and insert them into the primitive set(s) (lines 8 to 12). Here we define a single set using built-in arithmetic functions; the `ADD` primitive is developed further in Figure 5. Problem variable x is also implemented as a primitive (line 13; a token is simply a named value wrapped into a primitive).
2. Build the GP evolutionary system using the primitive set(s) (line 15);.
3. Build the evaluation and termination operators (lines 17 and 18). Here an instance of class `SymbRegEvalOp` is used as the fitness evaluation operator; this class is detailed in Figure 6. An instance of class `MaxGenerationTermOp` is also allocated, even though this step is not really required since this is the default termination operator.
4. Build the evolver using the operators of the previous step (lines 20 and 21). Here we use the standard generational evolver that requires only a fitness operator with an optional termination operator. Also build the GP vivarium (line 22). An important feature of Open BEAGLE, which is not illustrated in this simple example, is the possibility for the user to build a vivarium of specialized demes, individuals, or even genotypes. This feature is implemented through the use of allocators (the default vivarium constructor of line 22 uses allocators for the default demes, individuals, and genotypes).
5. Launch the evolution (line 25) after initialization (lines 24). It is the `evolver::initialize` method that instructs all operators to initialize themselves, that is to register all their parameters into the *register*, before proceeding to command-line parsing (`argc`, `argv` arguments). Typically, the command-line specifies an XML configuration file which is in turn parsed to override default operator parameters with user defined ones.

Note that all object instances are dynamically allocated by calls to the C++ `new` operator, and referenced by smart pointers (`Handle` member types). This is the recommended procedure in Open BEAGLE.

The next step is to declare the datum type. For the symbolic regression problem, the data are floating-point numbers. In plain C++ we would use atomic type `double`. But in Open BEAGLE, data must be derived from the `Object` class. This can be achieved

```
 1  #include "beagle/GP.hpp"
 2  using namespace Beagle;
 3  class Add : public GP::Primitive
 4  {
 5  public:
 6    Add() : GP::Primitive(2, "+") { }
 7    virtual void
 8    execute(GP::Datum& outResult,GP::Context& ioContext)
 9    {
10      Double lArg1;
11      get1stArgument(lArg1,ioContext);
12      Double lArg2;
13      get2ndArgument(lArg2,ioContext);
14      Double& lResult = castObjectT<Double&>(outResult);
15      lResult = lArg1 + lArg2;
16    }
17  };
```

Figure 5: `Add` primitive.

by wrapping atomic type `double`:

```
typedef WrapperT<double> Double;
```

Thereafter, type `Double` can be used as a synonym for type `double`. In fact, type `Double` is pre-defined in Open BEAGLE, among several other standard wrapped types.

With the datum now defined, it is possible to implement any number of primitives. For the case of the symbolic regression problem, we only need the following primitives: $+$, $-$, $\times$, $/$, and token x. Primitive $+$ is defined in Figure 5. The implementation of the other primitives is very similar. All primitives used in the symbolic regression are in fact pre-defined in Open BEAGLE, because they are so common.

Class `Add` inherits from abstract class `GP::Primitive`. The default constructor instantiates the object using the `GP::Primitive` constructor by specifying its number of arguments (2) along with its name (`"+"`). Method `execute` is used to implement the characteristic function of the primitive, that is to add two `Double` values which are first extracted from the current *context* (lines 11 and 13). To return the result, output argument `outResult` needs to be cast from `Datum` to `Double` (line 14) using template function `castObjectT` which is a wrapper for either a dynamic of static cast, depending on debug flags.

Finally, when all primitives are established, either by using pre-defined primitives or by creating new ones, the evaluation operator must be implemented. This operator is always problem specific and must be implemented in every new application. For the symbolic regression example, the evaluator operator is named `SymbRegEvalOp` and is presented in Figure 6. Method `initialize` (called by the *evolver*) samples 20 points of the polynomial expression (i.e. $x^4 + x^3 + x^2 + x$), taken randomly in the interval $[-1, 1]$. Thereafter, the

```
 1  #include <cmath>
 2  #include <vector>
 3  #include "beagle/GP.hpp"
 4  using namespace Beagle;
 5  class SymbRegEvalOp : public GP::EvaluationOp
 6  {
 7  private:
 8    std::vector<Double> mX; // Sampled x-axis values
 9    std::vector<Double> mY; // Sampled y-axis values
10  public:
11    SymbRegEvalOp(int inN=20) : mX(inN), mY(inN) { }
12    virtual void initialize(System& ioSystem)
13    {
14      for(unsigned int i=0; i<mX.size(); i++)
15      {
16        mX[i] = ioSystem.getRandomizer().rollUniform(-1.,1.);
17        mY[i] = mX[i]*(mX[i]*(mX[i]*(mX[i]+1.)+1.)+1.);
18      }
19    }
20    virtual Fitness::Handle
21    evaluate(GP::Individual& inIndividual,GP::Context& ioContext)
22    {
23      unsigned int lHits = 0;  // number of hits
24      double       lQErr = 0.; // square error
25      for(unsigned int i=0; i<mX.size(); i++) {
26        setValue("x",mX[i],ioContext);
27        Double lResult;
28        inIndividual.run(lResult,ioContext);
29        double lError = std::fabs(mY[i]-lResult);
30        if(lError < 0.01) lHits++;
31        lQErr += (lError*lError);
32      }
33      double lMSE  = lQErr/mX.size(); // mean square error
34      double lRMSE = std::sqrt(lMSE); // root-mean square error
35      double lNorm = (1./(lRMSE+1.));
36      return new GP::KozaFitness(lNorm,lRMSE,lMSE,lQErr,lHits);
37    }
38  };
```

Figure 6: Symbolic regression evaluation operator.

operator is ready to evaluate the fitness of any individual. Each time the operator is executed, method `evaluate` will be called for each individual that needs evaluation. An individual is processed (`run` method, line 28) 20 times, once for each sampled value. Recall that problem variable x is just one of the system's primitive and that operators have access to the primitive set(s) through the *context.* Method `EvaluationOp::setValue` is thus called (line 26) to update the value of x before "running" the individual. The root-mean square (RMS) error is evaluated using the error between the desired and computed value for the 20 random samples. At the end of these computations, method `evaluate` returns the individual's five fitness values (normalized, adjusted, standardized, raw, and hits), as defined by Koza (Koza, 1992).

5 Future Work

Open BEAGLE currently implements GA and GP frameworks that were successfully used to tackle handwriting recognition (Lemieux et al., 2002) and lens system design (Beaulieu et al., 2002) problems. Plans for future releases include a specialized Evolution Strategies (ES) framework as well as new EA for genetic re-engineering of existing solutions. Moreover, we hope that new users will contribute to its development by suggesting their own specific frameworks. If Open BEAGLE is successful, we plan to initiate an open source project where interested people could freely continue the development of the software framework.

We are also currently developing a module named *Distributed BEAGLE*, that will enable the distribution of the evolving process on a cluster of networked computers. This module, independent of the used EA, will be user friendly. The migration of a mono-process application to a distributed model will be done by simply adding new distribution operators in the evolver.

Finally, we plan to develop a web interface to the system, named *BEAGLE Visualizer*. This system will allow visualization of evolution statistics and populations through a Web browser. This graphical interface will consist of CGI scripts that will analyze populations and statistics of the XML Open BEAGLE files.

6 Conclusion

Open BEAGLE is built on strong OO foundations, that augment the C++ language and STL to offer a solid and flexible basis for evolutionary computations. The EC framework is composed of a generic kernel that combines the population structure, the evolution system, the operators and the evolver. Specific frameworks allow different evolutionary algorithms. Open BEAGLE is thus a versatile, easy to use, portable, efficient, robust, elegant and free C++ environment for designing complex EC systems.

Acknowledgments The authors would like to express their gratitude to Julie Beaulieu for her assistance in the writing of this document. This research was supported by an NSERC-Canada scholarship to C. Gagné and an NSERC-Canada grant to M. Parizeau.

References

Anderson, R., Baliles, D., Birbeck, M., Kay, M., Livingstone, S., Loesgen, B., Martin, D., Mohr, S., Ozu, N., Peat, B., Pinnock, J., Stark, P., and Williams, K.: 2000, *Professional XML*, Wrox Press, Chicago, IL, USA

Beaulieu, J., Gagné, C., and Parizeau, M.: 2002, Lens system design and re-engineering with evolutionary algorithms, in *Genetic and Evolutionary Computations COnference (GECCO) 2002*, New York, NY, USA

Campione, M. and Walrath, K.: 1998, *The Java Tutorial*, Addison-Wesley, Reading (MA), USA, 2 edition

Forsyth, R.: 1981, *BEAGLE A Darwinian Approach to Pattern Recognition*, *Kybernetes* **10**, 159–166

Fraser, A. P.: 1994, *Genetic Programming in C++*, Technical report, Cybernetics Research Institute, University of Salford

Free Software Foundation Inc.: 2000, *GNU Lesser General Public License*, http://www.gnu.org/copyleft/lesser.html

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, USA

Henning, M. and Vinoski, S.: 1999, *Advanced CORBA Programming with C++*, Addison-Wesley, Reading, MA, USA

Holland, J. M.: 1975, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI

Koza, J. R.: 1992, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA

Koza, J. R.: 1994, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, USA

Langdon, W. B.: 1998, *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, Kluwer, Boston, MA, USA

Lemieux, A., Gagné, C., and Parizeau, M.: 2002, Genetical engineering of handwriting representations, in *International Workshop on Frontiers of Handwriting Recognition (IWFHR) 2002*, Niagara-on-the-Lake, ON, Canada

Lenaerts, T. and Manderick, B.: 1998, Building a genetic programming framework: The added-value of design patterns, in *First European Workshop on Genetic Programming*, pp 196–208

Luke, S.: 2001, *ECJ Evolutionary Computation System*, http://www.cs.umd.edu/projects/plus/ec/ecj

Merelo, J., Keijzer, M., and Schoenauer, M.: 2001, *EO Evolutionary computation framework*, http://eodev.sourceforge.net/

Montana, D. J.: 1995, *Strongly Typed Genetic Programming*, *Evolutionary Computation* **3(2)**, 199–230

Musser, D. R. and Saini, A.: 1996, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, Reading, MA, USA