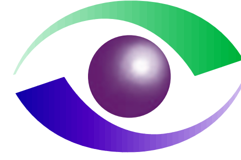

LABORATOIRE DE
VISION ET SYSTÈMES
NUMÉRIQUES

Open BEAGLE Manual

Christian Gagné[1,2] and Marc Parizeau[2]

For Open BEAGLE version 3.0.0
Document revision 1

Technical report RT-LVSN-2003-01-V300-R1

October 6, 2005

[1] Équipe TAO, INRIA Futurs,
LRI, Bat. 490, Université Paris Sud,
91405 Orsay Cedex, France.

[2] Laboratoire de Vision et Systèmes Numériques (LVSN),
Département de Génie Électrique et de Génie Informatique,
Université Laval, Québec (QC), Canada, G1K 7P4.

E-mails: christian.gagne@lri.fr, cgagne@gmail.com, parizeau@gel.ulaval.ca

Warning: outdated manual

Open BEAGLE has been significantly improved since version 1.0.0. I tried the best I can to keep the manual up to date, although it is now lagging behind the source code. This is not to say that the information is incorrect, the explanations given here are still exact, or at least pretty close to reality. The problem is more than there is a lot of things missing and several parts should be revised in order to keep the manual a precise and helpful reference on Open BEAGLE. Time is now cruelly missing and can't overextend myself anymore by trying to rewrite it a the short term.

This is why I am looking for help to update and improve the manual. It would also be very beneficial to the community to get contributions from different kind of users (beginners as well as experimented developers), in order to make this manual valuable for everyone. If you are interested to improve this manual, please contact me so I can supervise the global operation. The LaTeX sources are also publicly available in the Open BEAGLE CVS repository on sourceforge.net[1]. In the meanwhile, I annotated this manual in order to spot parts of the manual that should be updated and new elements that should be added.

Christian Gagné, October 6, 2005

[1] See page http://sourceforge.net/cvs/?group_id=91196.

Contents

1 Introduction **7**

1.1 What is Evolutionary Computations 7

1.2 What is Open BEAGLE 8

1.3 Survey of Existing Frameworks 10

1.4 Document Content 11

2 Open BEAGLE Tutorial **12**

2.1 Genetic Programming 12

2.2 Simple GP Example 13

2.2.1 Symbolic Regression 13

2.2.2 Function `main` 13

2.2.3 Datum and Primitives 16

2.2.4 Fitness Evaluation 18

2.2.5 Experimenting with the Example 20

3 Open BEAGLE Class Architecture **23**

3.1 Open BEAGLE C++ Naming Convention 24

3.2 Portable Agile C++ Classes 25

3.3 Generic OO Foundation 25

3.3.1 Objects 25

3.3.2 Smart Pointers 26

3.3.3 Allocators 28

3.3.4 Data Structures 29

3.3.5 XML Input/Output 31

- 3.3.6 Object Wrappers . . . 32
- 3.3.7 Exceptions . . . 33
- 3.3.8 Summary . . . 34
- 3.4 Generic EC Framework . . . 35
  - 3.4.1 Populations and Statistics . . . 35
  - 3.4.2 Internal System . . . 38
  - 3.4.3 Operators and Evolver . . . 41
  - 3.4.4 Breeder Model . . . 42
  - 3.4.5 Summary . . . 43
- 3.5 Miscellaneous Architecture Elements . . . 44
  - 3.5.1 Evaluation Operator . . . 44
  - 3.5.2 Termination Operator . . . 44
  - 3.5.3 Multiobjective Evolutionary Algorithms . . . 45
  - 3.5.4 Co-evolution . . . 45
  - 3.5.5 Standard Open BEAGLE Operators Library . . . 46
  - 3.5.6 GP Genotypes . . . 46
  - 3.5.7 GP Primitives and Sets . . . 51
  - 3.5.8 GP Primitives Library . . . 52

4 Open BEAGLE User Manual **54**

- 4.1 Guidelines . . . 54
- 4.2 Building a System for an Evolution . . . 57
- 4.3 Using the Register . . . 58
  - 4.3.1 Registering Parameters . . . 58
  - 4.3.2 Interaction with the Register . . . 59
  - 4.3.3 Modifying Parameters . . . 60
  - 4.3.4 Modifying Parameters on the Command-Line . . . 60
  - 4.3.5 Modifying Parameters Using a Configuration File . . . 61
- 4.4 Customizing the Evolutionary Algorithm . . . 62
  - 4.4.1 Building a Custom Evolver . . . 62
  - 4.4.2 Custom Evolver with a Breeder Tree . . . 63

4.4.3 Defining New Operators 65
4.5 GA User Manual 66
4.6 GP User Manual 68
4.6.1 Getting the Most of the Primitives 68
4.6.2 Strongly Typed GP 70
4.7 Multiobjective EA User Manual 72

5 Conclusion **76**

Bibliography **80**

List of Tables

2.1 Symbolic regression problem 14

3.1 Predefined wrapper types 33
3.2 Standard exception classes defined in Open BEAGLE 34
3.3 Log levels available in Open BEAGLE 39
3.4 Open BEAGLE functional operators 46
3.5 Open BEAGLE population manipulation operators 47
3.6 Open BEAGLE breeder and replacement strategy operators 47
3.7 Open BEAGLE initialization operators 47
3.8 Open BEAGLE selection operators 47
3.9 Open BEAGLE fitness evaluation operators 48
3.10 Open BEAGLE termination operators 48
3.11 Open BEAGLE crossover operators 48
3.12 Open BEAGLE mutation operators 49
3.13 `GA::EvolverBitString` default operator sets 49
3.14 `GA::EvolverFloatVector` default operator sets 49
3.15 `GA::EvolverES` default operator sets 50
3.16 `GP::Evolver` default operator sets 50
3.17 List of standard GP primitives 53

4.1 Open BEAGLE constrained operators 71

List of Figures

2.1 C++ `main` routine of the symbolic regression example. 15

2.2 Definition of an `Add` primitive. 17

2.3 Evaluation operator definition for the symbolic regression example. 19

3.1 Software architecture diagram 23

3.2 Relation between objects and smart pointers. 27

3.3 Generic object oriented foundation of Open BEAGLE 36

3.4 Generic EC framework architecture 37

3.5 Structure of the population in Open BEAGLE 38

3.6 Internal Open BEAGLE system 40

3.7 Relation between the GP sets, primitives and trees. 52

4.1 `GP::Primitive` class declaration 69

Chapter 1

Introduction

This document introduces the *Open BEAGLE* framework for Evolutionary Computations (EC). It contains a tutorial, an overview of the object oriented software architecture, and a complete user manual. It is aimed at both the novice and expert user of EC techniques. For more implementation details, the reader is referred to the *Open BEAGLE Reference Manual* which was distiled from the source code, using the *doxygen* [26] documentation generator. The detailed installation and compilation instructions are given in the *Open BEAGLE Compilation HOWTO*, available on-line on the project's Web page[1].

1.1 What is Evolutionary Computations

Evolutionary computations is a discipline that involves simulating the natural evolution process on computers [1]. This can be seen as an optimization process in which a population evolves over time, to fit a given environment. The EC principles have been successfully applied to numerous situations where classical methods of optimization, classification and automatic design were not able to produce adequate solutions. Generally, EC is presented in four major flavors: Genetic Algorithms (GA), Genetic Programming (GP), Evolution Strategy (ES), and Evolutionary Programming (EP).

Genetic Algorithms (GA) [13] involve the evolution of a population of individuals representing possible solutions to a problem. An individual is usually a string of symbols defined over a given alphabet, most often bits (individual = string of bits). The idea is inspired from genetic DNA that composes every living creature. The search process is made by an iterative application of genetic operators, such as crossover, mutation, and Darwinian selection operators biased toward the fittest individuals. Using this Darwinian paradigm, a population of solutions evolves until some stopping criterion is reached. GA has been widely used as a numerical method to optimize parameters of systems without having any *a priori* knowledge about the search space.

[1] http://www.gel.ulaval.ca/~beagle

Only a feedback criterion is needed to give an objective value that guides the search.

Genetic Programming (GP) [15, 2] is a paradigm that allows automatic programming of computers by heuristics inspired from the same evolution principles as GA: genetic operations of crossover and mutation, and natural selection operation. The difference between GA and GP lies mainly in the representation used, for which GP is similar to a computer program structure. Canonical GP was first formally expressed by Koza in the beginning of the 1990s [15, 16]. Koza's GP represents programs as trees, that is acyclic and undirected graphs, where each node is associated to an elementary operation specific to the problem domain. Others have experimented with different representations, such as linear programs [2] or cyclic graphs [25]. GP is particularly adapted to the evolution of variable length structures.

The Evolution Strategy (ES) paradigm was developed by I. Rechenberg and H.-P. Schwefel at the Berlin Technical University in the 1960s [1, 24]. In ES, each individual is a set of characteristics of a potential solution. This set is generally represented as floating-point vectors of fixed length. ES is applied to a parent population (of size μ) from which individuals are randomly selected to generate an offspring population (of size $\lambda \gg \mu$). Offsprings are modified by mutation, which consists in adding a randomly generated value that follows some parametrized probability density function. The parameters of this probability density function, called the *strategy parameters*, themselves evolve over time following the same principles. To engender a new population of size μ, the best μ individuals are chosen within either the λ offsprings (approach (μ, λ)), or the μ parents and λ offsprings (approach $(\mu + \lambda)$). Modern ES can also be nested.

Evolutionary Programming (EP) has been developed by L.J. Fogel in the 1960s and later by D.B. Fogel et al. in the 1990s [8, 1]. EP was initially designed to evolve finite state machines and has been later extended to parameter optimization problems. The approach is more focused on the relation between parents and offsprings than on the simulation of nature-inspired genetic operators. Contrary to the three first EC flavors, EP doesn't involve the use of a particular representation, but rather a high-level evolutionary model and a representation appropriate for the problem to solve. Only a mutation operator specific to the representation is needed. To do EP, a population of μ solutions is randomly generated. Each individual of the population produces λ offsprings resulting from mutation. Then, a natural selection operation is applied to produce a new population of μ individuals. The mutation - selection process is applied iteratively until a good solution is found.

1.2 What is Open BEAGLE

Since 1999, we have developed Open BEAGLE, a versatile C++ environment primarily designed to do GP, but general enough to do other kinds of Evolutionary Computations (EC). The

name BEAGLE stands for the acronym *the Beagle Engine is an Advanced Genetic Learning Environment*[2]. Beagle is also the name of an English vessel, the *HMS Beagle*, in which Charles Darwin engaged himself as a naturalist for a circumnavigation of the earth. The name *Beagle* was previously used in a very old (in term of technological advancements) software that does pattern recognition using EC principles [9, 17]. Although, the adjective *Open* was added to the current BEAGLE framework to distinguish the two designations, it also puts the emphasis on the open source aspect of the project. In this document, the term BEAGLE always designates the actual framework, *Open BEAGLE*.

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 is built on a generic OO foundation, independent for the different EC flavors. Over this foundation, both GP and GA (vector-based) frameworks are currently implemented. Open BEAGLE is made to be extensible, to allow the user to implement his own algorithm, with little efforts and minimal code writing.

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 only necessary condition is to have a population of individuals to which a sequence of evolving operations is iteratively applied. So far, Open BEAGLE supports most mainstream EC flavors such genetic programming; bit string, integer-valued vectors, and real-valued vectors genetic algorithms; and evolution strategy. It also includes support for advanced EC techniques such multiobjective optimization and co-evolution. The user can take any of these specialized frameworks and modify them further to create his own specialized flavor of evolutionary algorithms.

User Friendliness Considerable efforts were deployed to make the use of Open BEAGLE as easy and pleasant as possible. Open BEAGLE possesses several mechanisms that offer a user friendly programming interface. For example, the memory management of dynamically allocated objects is greatly simplified by the use of reference counting and automatic garbage collection. The programming style promoted is high-level and allows rapid prototyping of applications.

Portability The Open BEAGLE code is compliant with the C++ ANSI/ISO 3 standard. It requires the Standard Template Library (STL) [23]. The framework also includes a subset of the Portable Agile C++ Classes (PACC) collection, which encapsulates calls made to the operating system in a portable fashion.

[2] In French, BEAGLE stands for *Beagle est un Environnement d'Apprentissage Génétique Logiciel Évolué.*

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 verification and 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 energy was 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, eventually, modify. The use of XML as file format is also a central aspect of Open BEAGLE, which provide a common ground for tools development to analyze and generate files, and to integrate the framework with other systems.

Free Sourceness The source code of Open BEAGLE is free, available under the GNU Lesser General Public License (LGPL) [14]. It can thus be distributed and modified without any fee. Open BEAGLE is available on the Web at `http://www.gel.ulaval.ca/~beagle`.

1.3 Survey of Existing Frameworks

NOTE: This section can be reviewed taking the survey of existing frameworks available in the IJAIT paper.

Open BEAGLE shares a lot of features with other GP and EC systems[3] available on the Web. We have identified four of them for comparison with Open BEAGLE: *lil-gp* [27], *gpc++* [10], *EO* [21] and *ECJ* [19].

lil-gp is a C implementation of the original *little-lisp* code described in *Genetic Programming I* [15]. It provides the majors features of a complete GP environment. It is coded in the widely used C language, providing a fast and efficient implementation. However, it has several drawbacks, essentially from the functional structure of the code. Indeed, it is harder to extend this kind of GP implementation, compared to an object oriented one. There is a numbers of patches for lil-gp currently available[4] that are more or less compatible. It might be a long work to adapt and make them work together and add new functionalities.

[3] Because there are currently few generic EC systems that do GP, it is natural to compare Open BEAGLE to both GP and EC packages.

[4] We know the existence of four of them: a multi-threads safe implementation (*mtlilgp*), a constrained GP version (*cgp-lilgp*), a Windows 95 port, and mix of these patches (*patched lil-gp*).

gpc++ is one of the first known C++ framework for tree-based GP. 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 [18].

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). We are thus convince that dynamic polymorphism is necessary for a programmer-friendly EC environment and for rapid application developments.

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[5], and big memory footprints, which is indeed a serious limitation for resource-intensive tasks like EC.

1.4 Document Content

The document is divided into three parts. First is presented a tutorial on how to implement a GP application with Open BEAGLE. This section presents an overview on how to use the framework, and is targeted not only for the GP users, but also for any other EC users. Second, the object oriented architecture is specified. It presents the important entities and mechanisms deployed in Open BEAGLE, and gives a comprehensive description of the framework. Finally, the document ends with a detailed user manual where all essential details needed to rapidly implement an EC application with Open BEAGLE are given.

Users familiar with the C++ programming language can read the present document from the first page to the last. For those less familiar with C++ and object oriented concepts, the first reading should focus on chapter 2, and sections 3.5, 4.1, 4.2, and 4.5. Thereafter, with a good C++ reference manual on their side, they can read the entire document to better understand the general organization of the framework.

[5] 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.

Chapter 2

Open BEAGLE Tutorial

This section is a tutorial on Open BEAGLE as a genetic programming framework. It first summarizes GP basics, and then proceeds with a simple example in order to illustrate both the power and ease of use of the framework.

2.1 Genetic Programming

This sub-section is not a tutorial about tree-based GP. It is rather a summary of what needs to be defined to resolve a problem with tree-based GP. If you are not familiar with GP, references [2, 15] are good introduction books on the topic. To genetically program a computer for an application, three issues of the problem domain must be addressed:

- What type of data elements are handled by the genetic programs?
- What is the set of computation primitives?
- How does one measures the fitness of individual programs?

To address the first issue, one needs to define the data type that will be manipulated by the genetic programs. In Open BEAGLE, these data types are called *datums*. The datums are application specific. For example, if the goal of an evolution is to discover the mathematical expression of a hidden physical phenomenon, the datum could simply be floating-point numbers.

Once the datum type is defined, the primitives (more usually called functions and terminals in GP) that will compose the GP individuals must be specified. This step is important because these primitives are the building blocks of the genetic programs. A primitive is an application-specific operation to which the nodes of the GP trees are associated.

All primitives must process and return data of the specified datum type. A primitive is defined by three properties: its (unique) name, its number of arguments, and its characteristic function. These properties must be carefully established for all Open BEAGLE primitives.

Each primitives used are inserted into the set of usable primitives. This is a kind of instruction set for the genetic programs. A valid set of primitives must be made of at least one function primitive and one terminal primitive.

Finally, once the datum type and the set of primitives are established, the fitness function must be specified, that is the objective criteria that allow quality discrimination between solutions.

2.2 Simple GP Example

This section provides an implementation of a well known symbolic regression problem genetically programmed with Open BEAGLE. The discussion follows a top-down approach, starting with the `main` routine, followed by a description of the different implemented modules. Note that in this document, all C++ code is intentionally written in a compact format without separation between interface (class definition) and implementation (method definition). This is not a recommended practice for good C++ programming, nor for Open BEAGLE. Note also that header file `beagle/GP.hpp` must be included in the application code to get the declarations of the framework classes for GP. All generic Open BEAGLE classes are defined in the C++ namespace `Beagle`, while the GP specific classes are in namespace `GP`, which is in the global namespace `Beagle`.

2.2.1 Symbolic Regression

A classical GP problem is the symbolic regression of a set of data points where the computer searches for an analytical function that fits these data points. The problem is fully described in [15], pages 162 to 169. Table 2.1 also summarizes this problem.

2.2.2 Function main

To define the `main` routine of a Open BEAGLE GP application, the following five steps are needed:

1. Build primitives (`GP::Primitive`) and insert them into one or more primitive sets (`GP::PrimitiveSet`);
2. Build an evolutionary system (`GP::System`) using primitive sets;
3. Build evaluation (`GP::EvaluationOp`);
4. Build an evolver (`GP::Evolver`);
5. Build a vivarium (`GP::Vivarium`).

Objective	Find a function $f(x)$ of an independent variable x, in its symbolic form, that corresponds to a set of sampled points $(x_i, f(x_i))$. For the present case, the function to extrapolate is $f(x) = x^4 + x^3 + x^2 + x$.
Datum	Floating-point numbers.
Terminals set	x (the independent variable).
Functions set	$+$, $-$, $*$ and $/$. $/$ is a protected division returning 1.0 when the denominator is near to 0 (0 ± 0.001).
Fitness cases	A random sample of 20 points $(x_i, f(x_i))$, with $x_i \in [-1, 1]$.
Raw fitness	Sum of the square error for all the fitness cases: $F_{raw} = \sum_{i=0}^{19}(f_{obtained}(x_i) - f_{desired}(x_i))^2$.
Standardized fitness	Mean square error (MSE) for all the fitness cases: $F_{standard} = F_{raw}/20$.
Adjusted fitness	Root mean square error (RMSE) for all the fitness cases: $F_{adjusted} = \sqrt{F_{standard}}$.
Normalized fitness	$F_{normalized} = 1.0/(1.0 + F_{adjusted})$.
Hits	Number of fitness cases for which the difference between the desired and obtained value is less than 0.01.
Stop criterions	Whether the evolution reached the maximum number of generations allowed, or that an individual with 20 hits is found.

Table 2.1: Symbolic regression problem

Afterward, the genetic system is ready to evolve. Figure 2.1 presents the C++ code for the `main` function of the symbolic regression.

Step 1 (lines 7 to 12) defines the set of usable primitives. The `GP::PrimitiveSet` object is a container for our primitives. The GP package of Open BEAGLE predefines several common primitives (see Section 3.5.8). Here we use only basic arithmetic operators: add, subtract, multiply and divide. The generic primitive `GP::TokenT<Double>` is used to contain a floating point number that represents the independent variable of the regression. It is of type `Double` which is a simple Open BEAGLE wrapper for atomic type `double`. All GP primitives must be allocated on the heap (with a call to operator `new`), as shown in the example. Henceforth, they will be memory-managed by the primitives set (see Section 3.3.2 for more details). The primitives themselves are defined through inheritance from abstract class `GP::Primitive` (see Section 3.5.7).

Step 2 (line 14) builds the system, that is an object that centralize references to the important entities of the Open BEAGLE framework: the context, the register, the logger, the randomizer and, for GP, the set of usable primitives. These entities are described in more details in the architecture section (Section 3). In most cases, the Open BEAGLE user doesn't need to interact with these entities. They need to be used explicitly only for advanced applications, or for developing new evolutionary algorithms.

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

Figure 2.1: C++ `main` routine of the symbolic regression example.

Step 3 (lines 16) concerns the creation of evaluation operator. The evaluation operator is an object that implements the fitness evaluation method. The user needs to subclass abstract class `GP::EvaluationOp` and define an `evaluate` method. In this case, we use class `SymbRegEvalOp` to evaluate the fitness of an individual (i.e. a regressed function). This class is detailed in Section 2.2.4.

Step 4 (line 18) builds an evolver that encapsulates the operators of the evolutionary algorithm. These operators refer to all processes that evolve populations in a given paradigm. In the current example, we use the default GP evolver, class `GP::Evolver`. This includes the *ramped half-and-half* initialization, a tournament selection mechanism, standard crossover and three different mutation operators, and miscellaneous operations like saving populations milestone and computing generational statistics. The user can change the operations of predefined evolver from either a configuration file or some programming. This generic approach, which is one of the most powerful features of Open BEAGLE, when combined with generic population

allocators, enables the construction of any evolutionary paradigm.

Step 5 (line 20) builds the vivarium that contains the population. A vivarium is composed of a set of demes, which are composed of individuals that are themselves composed of genotypes. A deme, Open BEAGLE class `Deme`, is simply a set of individuals that evolve together in a closed circuit. An individual is composed by one or some genotypes and a fitness value. It can in fact contain more than one genotype. For instance, Automatically Defined Functions (ADF) [16] are coded using a set of genotypes (i.e. forest of trees). The genotype is the basic data structure used for coding individuals. For GP, this data structure is a rooted tree, defined by class `GP::Tree`. Using the `GP::Vivarium`, the vivarium is initialized using the default allocators of demes/individuals/genotypes for GP. However it is possible for the user to override the default setting of the vivarium and used allocators to desired population elements. More details about allocators can be found in Section 3.3.3.

At line 22, the evolver is initialized using both command-line arguments and the Open BEAGLE system. Through the command-line, it is possible to change the value any of the evolution parameters (how to do this is explained in section 2.2.5). A XML configuration file is also parsed by the initialize method (see Section 4.3 for more details). Line 23 then makes the vivarium evolving until the stopping criterion is reached.

Notice the fact that all instructions in function `main` are within a try-catch block. Open BEAGLE defines many standard exceptions for dealing with abnormal situations. See Section 3.3.7 for more details. The users are encouraged to proceed this way. Note also that every Open BEAGLE object is dynamically allocated (by a call to the C++ `new` operator) and its pointer is wrapped using a `Handle` object. The `Handle` type is a smart pointer that manages objects allocated on the heap. The use of smart pointers prevents some common memory leaks and facilitate the work of Open BEAGLE users. These handlers are fully described in Section 2.2.4.

2.2.3 Datum and Primitives

In the Open BEAGLE framework, every class inherits from an abstract base class named `Object`[1]. This class includes many virtual methods that makes the framework very powerful, based on strong abstractions and sophisticated OO mechanisms. But because of this, the GP primitives cannot directly use atomic C++ types like `int` or `double`. They must use an object wrapper that adapts the desired type to the `Object` class. Thus, GP primitives must be designed to manipulate object derived from type `GP::Datum` which is a simple typedef:

```
namespace GP {
  typedef Object Datum;
}
```

[1]There are some exceptions to this rule, but these will be covered later.

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

Figure 2.2: Definition of an `Add` primitive.

To simplify the wrapping of external types, Open BEAGLE defines a class template, named `WrapperT`, that implements the virtual methods of class `Object`. The methods of `WrapperT` are implemented as calls to the usual C++ operators (i.e. constructor, destructor, copy-constructor, assignment operator, operators `==` and `<`, etc). Moreover, casting operators are also defined in order to enable conversion between wrapper and wrapped types.

For the example of the symbolic regression, standard Open BEAGLE wrapper type `Double` (note the capital letter) is used to encapsulate floating point numbers:

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

If unsigned integers were needed, a corresponding wrapper type would be used. For instance, Open BEAGLE type `UInt` is defined as:

```
typedef WrapperT<unsigned int> UInt;
```

Other wrapper types such as `Float` and `Bool` are also defined in Open BEAGLE (see Table 3.1 for a complete listing).

After defining a common datum type for an application, the primitives themselves need to be defined. For the symbolic regression problem, we have chosen to use four of the predefined arithmetic operators: $+$, $-$, $*$, and $/$. Figure 2.2 gives the C++ code for class `GP::Add` that implements the $+$ primitive between two `Double` data. The definitions of the other three primitives are almost identical. Any primitive must inherit from abstract class `GP::Primitive` (line 3) and define method `execute` (line 6). It is this method that defines the characteristic function of the primitive. A primitive is usually constructed with a call to the constructor of its superclass, as in line 5. The first argument of this call specifies the number of arguments for the corresponding primitive, while the second argument assigns a string label (a name) to it.

Method `execute` (line 6) receives two arguments: a reference to a `GP::Datum` for returning its result, and a reference to the current context that holds the state of the whole evolutionary process, class `GP::Context`. The later is needed for evaluating the operands of the primitive. Although the Open BEAGLE context is a relatively complex object, the end user need not know anything about its internal working. He is only required to pass on this reference to method `get1stArgument` and `get2ndArgument` implemented in `GP::Primitive`, in order to retrieve the primitive's operands. This is done at lines 8 and 10 for the two operands of our primitive. The two arguments of methods `get1stArgument` and `get2ndArgument` are respectively: the datum for returning the value, and the current context. In order to return its own result, method `execute` also needs to cast the `GP::Datum` reference that it received at line 6, into an adequate reference for assigning the result of it computation, in this case a `Double`. This is done at line 11 using the Open BEAGLE `castObjectT` method that implements a dynamic cast.

In the listing of Figure 2.1, at line 12, a special primitive of type `GP::TokenT<Double>` was also inserted into the primitive set. This standard primitive is for encapsulating an input variable of the problem at hand. In this case, the datum is of type `Double`, and the variable is named `x` (for more details, the reader is referred to Section 3.5.7).

2.2.4 Fitness Evaluation

NOTE: The code of the symbolic regression example is now slighlty different from this one. It would be more consistent to update it here.

In this section, we examine the C++ code of class `SymbRegEvalOp` which implements the fitness evaluation function for our symbolic regression application (see Figure 2.3). This class is not a standard Open BEAGLE class, it is derived from abstract class `GP::EvaluationOp`, and especially designed for our application. The method `evaluate` is declared as pure virtual in the class `GP::EvaluationOp`. This mean that the method must be overridden in a concrete subclass. The method receives as argument a reference to an individual to evaluate, and a reference to the context to use. In the evaluation method, an individual is parsed by a call to the method `run` of `GP::Individual`. The result of the interpretation is put into the datum given as argument. The value of the input value of the individuals, can be modified for each evaluation case with a call to the method `setValue` of `GP::EvaluationOp`. The first argument of the call to the method is the name of the GP primitive to modify. The second argument is a reference to the datum that will contain the result of the evaluation. And finally, the third argument is the context to be used. The method `evaluate` terminates by returning a dynamically allocated fitness value object. The default fitness object type to use is `FitnessSimple`, which define a single fitness measure to maximize.

The method `postInit` (line 26 to 32) is a hook called one time by Open BEAGLE, just after the system has been initialized, but before fitness evaluation are done. In the actual case,

```
1  #include <cmath>
2  #include <vector>
3  #include "beagle/GP.hpp"
4  using namespace Beagle;
5  class SymbRegEvalOp : public GP::EvaluationOp {
6  private:
7    std::vector<Double> mX; // Sampled x-axis values
8    std::vector<Double> mY; // Sampled y-axis values
9  public:
10   SymbRegEvalOp() { }
11   virtual Fitness::Handle evaluate(GP::Individual& inIndividual,
12                                    GP::Context& ioContext) {
13     double lQErr = 0.; // square error
14     for(unsigned int i=0; i<mX.size(); i++) {
15       setValue("x",mX[i],ioContext);
16       Double lResult;
17       inIndividual.run(lResult,ioContext);
18       double lError = mY[i]-lResult;
19       lQErr += (lError*lError);
20     }
21     double lMSE  = lSquareError / mX.size();
22     double lRMSE = std::sqrt(lMSE);
23     double lFitness = (1.0 / (lRMSE + 1.0));
24     return new FitnessSimple(lFitness);
25   }
26   virtual void postInit(System& ioSystem) {
27     GP::EvaluationOp::postInit(ioSystem);
28     for(unsigned int i=0; i<20; i++) {
29       mX.push_back(ioSystem.getRandomizer().rollUniform(-1.,1.));
30       mY.push_back(mX[i]*(mX[i]*(mX[i]*(mX[i]+1.)+1.)+1.));
31     }
32   }
33 };
```

Figure 2.3: Evaluation operator definition for the symbolic regression example.

the method is implemented to sample the equation to regress. This could not have been done before, as the random number generator is not valid until the system is completely initialized. But this also cannot be done after, as the equation to regress must be sampled before any fitness evaluation. This method `postInit` is originally defined in the class `Operator`, which is a parent of `EvaluationOp`.

Three nested data types are defined for all object-derived Open BEAGLE classes: `Handle`, `Alloc`, and `Bag`. The handle type (`Handle`) defines a smart pointer of the Open BEAGLE object to which it belong. It allows the emulation of a garbage collection mechanism similar to the one of the *Java* language. The use of a smart pointer (or handle if you prefer) is equivalent to the use of a standard C/C++ pointer. The difference is that a smart pointers increments/decrements

the reference counter when a Open BEAGLE object is referred/unreferred. An handled object is *self-destructed* when its reference counter is equal to 0.

There is some rules to follow when using smart pointers in conjunction of Open BEAGLE objects:

1. An handled object must always be allocated on the heap, with a call to the `new` operator;
2. Programmers must *never* delete a smart pointed object. The deallocation of the instance is the responsibility of the handles;
3. The access to a pointed object, using a referring handle, is made with the same operators than the standard C/C++ pointers, the operators `*` and `->`.

The second nested data type is the allocator (the type `Alloc`). It defines a Open BEAGLE type able to allocate, clone and copy object of the associated type. For example an instance of `Double::Alloc` is an object that allocate, clone (on the heap) and copy `Double`. This data type is a central aspect of Open BEAGLE design and allows some sophisticated mechanism with little effort for the application developer. It is explained in more details in section 3.3.3.

Finally, the last nested data type is the container (the type `Bag`). It correspond to the Open BEAGLE basic structure containing data of a given type. A container of a given type is simply an array of smart pointers to the given type. It can contain instances of the associated type, and/or instances of types derived from the associated type. More details are given in Section 3.3.4.

2.2.5 Experimenting with the Example

The code of the symbolic regression example can be found in the folder `examples/GP/symbreg` of the Open BEAGLE distribution. Once the Open BEAGLE libraries built, you can compile the example `symbreg` by following the instructions given in the `INSTALL` file. Then, it can be very interesting to experiment with the example.

To do so, open a shell (or a DOS terminal under Windows) and `cd` to the `symbreg` directory of the example directory. If your built is successful, you should find here the binary `symbreg`. Execute the program with the following command.

```
./symbreg
```

At the console output, you get XML logs on the running evolution. Here is the typical vivarium statistics given by the logs.

```
<Log>
  <Stats id="vivarium" generation="21" popsize="100">
```

```
    <Item key="processed">96</Item>
    <Item key="total-processed">2031</Item>
    <Measure id="fitness">
      <Avg>0.583894</Avg>
      <Std>0.104593</Std>
      <Max>0.867907</Max>
      <Min>0.287821</Min>
    </Measure>
    <Measure id="treedepth">
      <Avg>4.74</Avg>
      <Std>2.39368</Std>
      <Max>11</Max>
      <Min>1</Min>
    </Measure>
    <Measure id="treesize">
      <Avg>10.4</Avg>
      <Std>6.98339</Std>
      <Max>33</Max>
      <Min>1</Min>
    </Measure>
  </Stats>
</Log>
```

The values in the XML tags `Log` indicate a log entry. The values in tags `Stats` are statistics of a population (a deme or a vivarium). The value of the statistics item named `processed` tags is the number of individuals processed (evaluated) during the actual generation, while the value associated to the item `total-processed` is the cumulative number of individuals processed since the beginning of the evolution. The values between each `Measure` tags are statistics of the given measure: the first is the simple fitness value and the two last are tree grow measure (i.e. maximum tree depth and total tree size). The values in the tags are respectively: `Avg` for the average, `Std` for the standard deviation, `Max` for the maximum value, and `Min` for the minimum value.

Once the evolution finished, a milestone is written on disk. By default, the milestone is written as the file `beagle.obm`[2] in the current directory. A milestone file contains a bunch of useful informations on the evolution: the evolver structure, the value of the parameters, some statistics, the best individuals of the evolution, and the population at the last generation. It is also useful to restart evolutions that were interrupted prematurely. Take a look on the milestone file and try to find the best-of-run individuals (it is between two `HallOfFame` XML tags).

Now, it can be interesting to change some evolution parameters. This can be done in two ways: on the command-line, or using a configuration file. To get the command-line usage of the example, type the following command.

[2] If an installation of zlib is detected on your system at the compilation time, which is usually the case with most modern Unix/Linux, the milestone might be written as compressed file `beagle.obm.gz`. If you want to take a look to it, you only need to uncompress it with the command gunzip (i.e. `gunzip beagle.obm.gz`).

```
./symbreg -OBusage
```

All the Open BEAGLE specific arguments start with the prefix `-OB`, followed by the tag, an equal sign and the values of the parameter (except for special arguments `usage` and `help`). For example, if you want to set the crossover probability to 0.75, the command-line would be the following.

```
./symbreg -OBgp.cx.indpb=0.75
```

You can specify several parameters on the command-line by separating them by a comma.

```
./symbreg -OBgp.cx.indpb=0.75,ec.term.maxgen=100
```

To get detailed about the different parameters available, use the argument `-OBhelp`.

```
./symbreg -OBhelp
```

The second way to set parameters is to use a parameter file. You can generate a default configuration file by using the command-line argument `-OBec.conf.dump` with the filename of your configuration file.

```
./symbreg -OBec.conf.dump=mysymbreg.conf
```

The generated configuration file, named here `mysymbreg.conf`, comprises all the available parameters set to their default values, and the evolver algorithm used. You can edit the file if you want to change the value of the parameters. Once you have a customized configuration file, you can start an evolution using this file by setting the `ec.conf.file` parameter on the command-line.

```
./symbreg -OBec.conf.file=mysymbreg.conf
```

The configuration file can also describe the form of the evolutionary algorithm used. For some examples distributed with the Open BEAGLE, there is two configuration files containing pre-configured evolvers: *name*`-generational.conf`, and *name*`-steadystate.conf`, where *name* is the name of the example. You can change the evolver structure in the configuration file, which mean changing the operation used in the evolutionary algorithm. This is explained later in the document, at section 4.4.1.

NOTE: It would be great to add some other tutorials on other representations such GA bitstring, GA floatvector and ES.

Chapter 3

Open BEAGLE Class Architecture

This section presents the global class architecture of Open BEAGLE. The framework has a three levels architecture, as illustrated in Figure 3.1. 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.

NOTE: Some general explanations on PACC and its introduction in Open BEAGLE is necessary here.

The discussion is divided in four parts: the naming convention, the generic object oriented framework, the generic EC framework, and the specialized frameworks.

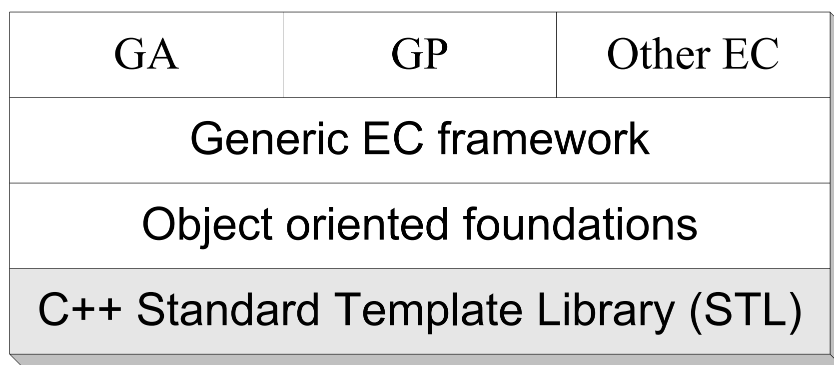

Figure 3.1: Software architecture diagram

3.1 Open BEAGLE C++ Naming Convention

All the Open BEAGLE framework has been implemented following a simple, but precise, naming convention. This naming convention is stated by the following rules.

1. An identifier is typically constituted of several concatenated words which all begins with a capital letter:

   ```
   MyIdentifier
   ```

2. As well are formed *namespace*, *class*, *struct* and *typedef* identifiers:

   ```
   namespace   MyNamespace;
   class       MyClass;
   struct      MyStruct;
   typedef int MyInt;
   ```

3. As well are formed *template* identifiers, with an additional capital T at the end of the identifiers:

   ```
   template<class T> MyTemplateT;
   ```

4. The enumerate types follow the basic rule, but the identifiers of the enum values start with a small *e*:

   ```
   enum MyEnum {eValue1, eValue2};
   ```

5. The methods identifiers are exceptions to the first rule by the fact that the first word of their name is entirely in small letters, including the first letter. Generally, the first word of a method is an action word.

   ```
   void MyClass::setValue(...);
   ```

6. The variable identifiers, class attributes or local variables are also exceptions to the first rule by the fact that they must start with a key word all in small letters. These key words specify the functionality of the variable.

   (a) *Class*, *struct* or *templates* attributes starts with a small *m*:

   ```
   class MyClass {
     int mAttribute;
   };
   ```

   If the attribute is also a *static* member, the identifier starts with a small *sm*:

   ```
   class MyClass {
     static int smStaticAttribute;
   };
   ```

(b) A local variable in a list of formal parameters of a method starts with the prefix *in* if it's an input parameter, *out* if it's an output parameter or *io* if it's a bidirectional parameter.

```
void simpleMethod(int inParam1, double &ioParam2, char *outParam3);
```

(c) A local variable that is not a member of a list of formal parameters starts with the letter *l*:

```
void MyClass::simpleMethod2(...) {
  ...
  int lLocalVariable;
  ...
}
```

(d) A constant (that is typed with the *const* qualifier) starts with the *c* letter. A constant *class* attribute starts with a *cm* attribute.

7. A macros identifier follows the first rule of the identifier and ends with a capital *M*:

```
#define MyMacroM(X) (...)
```

A macro specific to a namespace start with the namespace identifier, followed by an underscore (_):

```
namespace MyNamespace {
#define MyNamespace_MyMacroM(X) (...)
}
```

3.2 Portable Agile C++ Classes

NOTE: A detailed section on PACC is necessary here.

3.3 Generic OO Foundation

This section presents the generic object oriented (OO) mechanisms used in Open BEAGLE. Although these mechanisms were designed to be the foundation of Open BEAGLE, they could have been implemented in most OO frameworks. Some of our design choices are inspired from design patterns [11, 18]. Some concepts are also took from well-known OO environments such the *C++ Standard Template Library* (STL) [23], the *Java Library* [4], and *CORBA* [12].

3.3.1 Objects

In Open BEAGLE, most classes are derived from an abstract class called `Object`. An Open BEAGLE object comprises some general methods that are characteristic of a complete C++ object.

```
namespace Beagle {
class Object {
public:
  unsigned int getRefCounter() const;
  virtual bool isEqual(const Object&) const;
  virtual bool isLess(const Object&) const;
  virtual void read(XMLNode::Handle&);
  Object*      refer();
  void         unrefer();
  virtual void write(XMLStreamer&) const;
private:
  unsigned int mRefCounter;
};
}
```

The usual C++ objects comparison operators are defined, receiving arguments of the `Object` type and calling the appropriate comparison method (`isEqual` or `isLess`). The method `read` and `write` are used to read and write objects in the XML format. Section 3.3.5 gives more details about the Open BEAGLE I/O in XML.

NOTE: We are now using XML reading/writing facilities from PACC (`PACC::XML::Document` and `PACC:XML::Streamer`), which are in fact a more recent version of Open BEAGLE XML facilities. Readind/writing fonctions signatures are now slightly different and explanations on them should be updated.

An Open BEAGLE object also implements a reference counter that remembers the number of references that point to it. The C++ language defines two ways of allocating an object instance. First, the instance can be allocated on the stack, it will then be destructed at the end of the scope. Second, the instance can be allocated on the heap with a call to the `new` operator, which means that it is necessary to apply a symmetrical `delete` at the end of the instance lifetime. With Open BEAGLE, there is a variation on the second way to allocate objects: an Open BEAGLE object can be allocated on the heap and affected to Open BEAGLE smart pointers that interact with the object reference counter. The destruction of the object is then the responsibility of the smart pointers. This is the topic of the following section.

3.3.2 Smart Pointers

An Open BEAGLE smart pointer acts like a standard C++ pointer that manage the memory used by the pointed objects. It is implemented in class `Pointer`, which is tightly linked to the reference counter of class `Object`. The implementation of the `Pointer` class guarantees that the pointed object exists since there is a smart pointer that refer to it. It also guarantees that the object will be destroyed when its last reference will vanish. This emulates the appreciated garbage collection mechanism of some programming languages. In fact, the Open BEAGLE smart pointers are coherent with the creation and the garbage collecting of objects of the *Java*

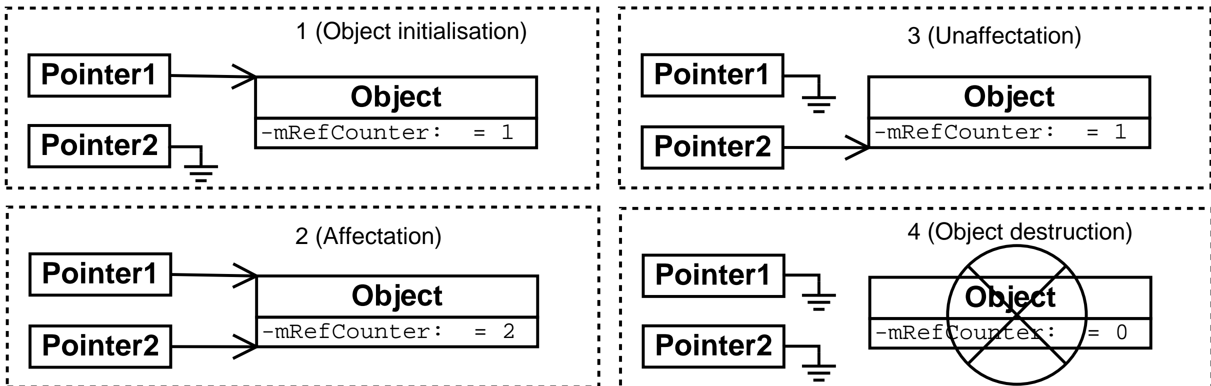

Figure 3.2: Relation between objects and smart pointers.

language. The two things that the user should remember are to allocate objects on the heap (with a `new`) and never interfere with object destruction after assigning them to smart pointers. Once an object is referred by smart pointer, the memory management responsibility is held by the smart pointer.

Exceptionally, the `Pointer` class and its subclasses do not inherit from superclass `Object`. Like a C++ pointer, an Open BEAGLE pointer can be assigned to a null pointer. The class also provides the two usual dereferencing operators, `operator*` and `operator->`, which return a reference to the pointed object. There is also two comparison operators (`operator==` and `operator!=`) between two `Pointer`, between a `Pointer` and an `Object*` and the null pointer testing operator (`operator!`). The relation between the objects and the smart pointers is presented in Figure 3.2.

Since any instance are concrete objects and smart pointers give references to the abstract `Object` type, each access to methods or attributes that are not declared in the object interface needs a cast of the reference given by the smart pointer to the desired derived type. This could lead to inelegant code, or even type inconsistencies if old C-style casts were used. To avoid these problems, there is a templated class, `PointerT`, that defines the pointer unreferencing operators to the desired type.

```
template <class T, class BaseType>
class PointerT : public BaseType {
public:
  inline T& operator*();
  inline T* operator->();
};
```

The `PointerT` template also emulates the mechanism of automatic pointer type binding for inheritance-related classes. This allows a compile-time binding of `T`-type smart pointers that inherit from the `BaseType`, when a `BaseType` object is needed. For example, suppose a method

taking an argument of the type `Base::Handle`, which is a smart pointer to an object instance of the class `Base`. With the automatic type binding of smart pointer, the method can get as argument a smart pointer to the type `Derived::Handle`, supposing that the class `Derived` inherits from the class `Base`. This can be done without any explicit cast of the smart pointers. `PointerT` has two templated parameters: the type of object handled, the `T`-type, and the parent type of the smart pointer, the `BaseType`. The `BaseType` is useful for the automatic type binding emulation by inheritance.

For each class of Open BEAGLE, the nested type `Handle` is declared. This is the type of handle to the class, that is, a smart pointer that gives exact reference type. Usually, this type is simply declared as a synonym of a parametrized `PointerT`.

```
class AnyClass : public SuperClass {
public:
  typedef PointerT<AnyClass,SuperClass::Handle> Handle;
  ...
};
```

Doing so, a smart pointer can be used to return reference to the right type, the type `AnyClass`, simply by working with the right handle type, `AnyClass::Handle`. It is a good practice to define the nested `Handle` type for every class that inherits directly or indirectly from an `Object`.

3.3.3 Allocators

Open BEAGLE is characterized by great versatility. This is achieved by different features and design choices, the first being an abstract object superclass that gives some essential generic mechanisms and from which almost every other Open BEAGLE class is derived. Another important design choice that gives flexibility to Open BEAGLE is the allocators, a kind of object factory that generates objects in an abstract fashion. An abstract allocator class named `Allocator` is implemented to define the methods to create objects on the heap.

```
class Allocator : public Object {
public:
  virtual Object* allocate() const =0;
  virtual Object* clone(const Object&) const =0;
  virtual void    copy(Object&, const Object&) const =0;
};
```

The purpose of the allocators is to provide factory methods to create and clone new types of objects derived from Open BEAGLE constituents. With such mechanism, any user could create new type of objects that redefine the default one and use associated allocators that return pointers to this new type of objects. This mechanism is similar and coherent with the *Factory Method* design pattern [11].

Usually, an allocator is used to allocate objects through their default constructor and clones the objects by calling their copy constructor. An allocator can also copy an existing object of a given type into an other. To simplify the task, a simple standard parametrized allocator named `AllocatorT` is defined to override the virtual method of the abstract class `Allocator`. Like the smart pointers `Handle` type, each component of Open BEAGLE has a nested allocator type called `Alloc`. The users are encouraged to define it in their classes.

```
class AnyClass : public SuperClass {
public:
  typedef PointerT<AnyClass,SuperClass::Handle>  Handle;
  typedef AllocatorT<AnyClass,SuperClass::Alloc> Alloc;
};
```

However, the template `AllocatorT` cannot be used to define an `Alloc` type for an abstract type. The reason is that the method `allocate` and `clone` cannot be implemented to instantiate object of the abstract allocated type. To solve this problem, another templated allocator type is implemented, `AbstractAllocT`. This allocator does not define the methods `allocate` and `clone`, but is usable in the same way than `AllocatorT` as an allocator type for abstract types.

3.3.4 Data Structures

It is natural to design data structures that take advantage of all the mechanisms exposed in the current section. Working with the new features of the C++, it is essential to exploit the *Standard Template Library* (STL) [23] which provides common, well known and portable data structures and algorithms. An Open BEAGLE specific, but STL compliant, data structures is extensively used: the container.

The Open BEAGLE container is a random access table of heterogeneous objects. It is implemented as a dynamic array of specified smart pointers. Any objects that is a specialization of the contained type could be put in a container. The constraints are the same as those of the smart pointers: the objects must be allocated on the heap (via the `new` operator) and these objects must not be unallocated explicitly (i.e. `delete` of the objects are forbidden). Our container is implemented as a Beagle object that is also a STL vector of object handles.

```
class Container : public Object, public std::vector<Object::Handle> { ... };
```

Built this way, our container could be manipulated by the STL generic algorithms. It is also possible to make container of containers, since a container is an object that can be smart pointed.

A drawback of this data structure is that every contained objects must have been allocated (on the heap) and thereafter inserted into the container. Knowing so, we add an allocator handle member attribute to the class `Container`. This member automatically allocates instance of objects when resizing. If no allocator are given to a container (the member value of the

allocator handle is null), resizing will simply resizes the vector of smart pointer and the newly added pointers are initialized to the null value. It is then possible to dynamically type containers by giving them allocator instances of the desired type.

As with the smart pointers and allocators, every Open BEAGLE type has a nested type that define a container of the specified type, the nested type named `Bag`. For example, the type that is a container of floating-point values, `Float`, is the type `Float::Bag`. The nested type is declared in classes as usual for the nested types `Handle` and `Alloc`, using a typedef to a template, in this case to template class `ContainerT`.

```
class MyClass : public SuperClass {
public:
  typedef AllocatorT<MyClass,SuperClass::Alloc> Alloc;
  typedef PointerT<MyClass,SuperClass::Handle>  Handle;
  typedef ContainerT<MyClass,SuperClass::Bag>   Bag;
};
```

The use of dynamically typed containers raises a problem when it is necessary to allocate containers of objects not from the basic usual associated type. Supposing a container of genotype, namely a `Genotype::Bag`. It may be possible for the container to contain a different type of genotype, for example genotypes of the type `SpecializedGenotype`, which is derived from the type `Genotype`. Then, the type allocator member of `Genotype::Bag` should be an instance of the type `SpecializedGenotype::Alloc`. This can be easily done by giving as argument an `SpecializedGenotype::Alloc` to the constructor of `Genotype::Bag`. But supposing now that a `Genotype::Bag::Bag` is created and that it should contain these containers dynamically typed, it is necessary to defined a new allocator of `Genotype::Bag` that instantiate the allocator of `Genotype` as instances of allocator of `SpecializedGenotype`. This can be very tricky to implement for a novice user.

This issue is solved by designing a new template of allocators specific to the Open BEAGLE containers. This specialized allocator has an attribute that is an object allocator handle compatible with the type contained in the associated bag. This allocator is implemented in the basic class `ContainerAllocator`.

```
class ContainerAllocator : public Object::Alloc {
public:
  ContainerAllocator(Object::Alloc::Handle);
protected:
  Object::Alloc::Handle mContainerTypeAlloc;
};
```

The type `Alloc` of `ContainerT` is declared as typedef of a templated allocator of container (`ContainerAllocatorT`). This template takes a third type, the type of the member allocator type of the container, in addition to the two first usual types.

```
template <class T, class BaseType>
class ContainerT : public BaseType {
public:
  typedef ContainerAllocatorT<ContainerT<T,BaseType>,BaseType,T::Alloc> Alloc;
  ...
};
```

Doing so, allocators of bags that are dynamically typed can be instantiated. A second template for the case of abstract bag type has also been declared, `AbstractContainerAllocT`. This is coherent with the discussion about the type `AbstractAllocT`, which is presented in section 3.3.3.

3.3.5 XML Input/Output

NOTE: As explained before, XML facilities are now provided by PACC. The way we use it is now fundamentally different from what is explained here, although the whole subsection should be completly revised.

Reading and writing files containing evolutionary data, such the population, the parameters, the algorithm, the results of the evolution, is an important aspect of an EC framework. The C++ already provides an interesting streaming mechanism to read and write objects into a file, the memory or the standard input/output. Although, there is also a very interesting technology that is more and more used as a file format, the eXtensible Markup Language (XML) [3]. Mixing these two concepts together lead us to an important mechanism of this framework: the Open BEAGLE input XML parser and output object streamer.

The XML format was first intended to be an improved generic replacement to HTML, but it appears to be so generic, and so well accepted by the industry, that it's now a central technology for domains such data modeling (as in the modern word processors), lightweight distributed computing (SOAP and XML-RPC), databases interactions and Web integration in general. Although the XML is a human understandable format, it is also a flexible and powerful file format, ideal for a to be used in conjunction of a framework such Open BEAGLE. A XML document can also be interpreted to screen by most modern Web browsers such *Mozilla* and *Microsoft Internet Explorer*.

The new Open BEAGLE XML input / output classes (since version 2.0.0) consists into a DOM-like XML parser and a XML streamer. The XML parser read file by constructing a tree with the XML files it reads, using object of the class `XMLNode` to build the tree. The XML parser uses a standard C++ input stream as input devices. The XML output streamer is a simple API class that can generate XML output into a standard C++ output stream.

For every Open BEAGLE objects that can read and written, the methods `read` and `write` must be over-defined consequently. For the `read`, this is done by a exploring the tree build by the XML parser. For the `write` method, the object must be serialized using the XML streamer

API.

To illustrate this, the following example presents the listing of the method `read` and `write` of the hypothetical class `DataValue`.

```
class DataValue : public Beagle::Object {
private:
  std::string mName;
  double      mValue;
public:
  virtual void read(Beagle::XMLNode::Handle& inNode) {
    if(inNode->getTagName() != "Data")
      throw Beagle_IOExceptionNodeM(*inNode, "tag <Data> expected");
    mName = inNode()->getTagAttribute("name");
    if(inNode->getFirstChild() == NULL)
      throw Beagle_IOExceptionNodeM(*inNode, "no value for tag <Data> found");
    mValue = uint2str(inNode->getFirstChild()->getValue());
  }
  virtual void write(Beagle::XMLStreamer& ioStreamer) const {
    ioStreamer.openTag("Data");
    if(mName != "") ioStreamer.insertAttribute("name", mName);
    ioStreamer.insertFloat(mValue);
    ioStreamer.closeTag();
  }
};
```

The XML output produced of an instance of a `DataValue` object that have as name `mydata` and as value 1.5 would be the following.

```
<Data name="mydata">1.5</Data>
```

In conjunction of the XML file format used in Open BEAGLE, it is planed to develop an infrastructure to analyze the files containing the populations or evolution statistics, into a standard Web browser, using modern Web technologies. This will eventually allows distant evolution management with simple Web interfaces and utility scripts to do some sophisticated analysis of the evolutions.

NOTE: A short overview (1-2 sentences) of Open BEAGLE file format is necessary here.

3.3.6 Object Wrappers

Given all this functionalities, the predominance of the object abstraction in Open BEAGLE's design is evident. But, if a user has some elaborated classes, or if a user wants to use other libraries in conjunction of Open BEAGLE, it might be restrictive that everything must inherit from the superclass `Object`. The user might not want to redefine everything or to use multiple inheritance to created hybrid classes. To enhance the user experience, a class template that would keep the use of Open BEAGLE simple is defined: the object wrapper. The wrapper is

a simple adapter of any type to the Open BEAGLE `Object` interface. The concept of object wrapper is based upon the *Adapter* design pattern [11].

The wrapper is defined in the C++ template `WrapperT`. The template maps the `Object` interface to the usual methods of the wrapped type. The template got some casting operators to indifferently use the wrapper as the wrapped type. Furthermore, some types that are wrappers of the C++ fundamental types are defined as standard types of Open BEAGLE. Table 3.1 presents all the standard wrapped type defined in Open BEAGLE.

C++ name	Wrapper name
`bool`	`Bool`
`char`	`Char`
`double`	`Double`
`float`	`Float`
`int`	`Int`
`long`	`Long`
`short`	`Short`
`std::string`	`String`
`unsigned char`	`UChar`
`unsigned int`	`UInt`
`unsigned long`	`ULong`
`unsigned short`	`UShort`

Table 3.1: Predefined wrapper types

NOTE: Some explanations on the `ArrayT` template is also necessary here.

3.3.7 Exceptions

Open BEAGLE integrates the C++ exceptions mechanism. Keeping it OO, an hierarchy of exceptions are defined. At the top there is the abstract exception superclass, **`Exception`**, from which every other Open BEAGLE exception classes inherit. This class inherit from the standard **`std::exception`** class, which allow catch of Open BEAGLE exceptions in a simple **`catch(std::exception&)`** expression.

```
class Exception : public Object, public std::exception {
public:
  explicit Exception(std::string inMessage="");
  void terminate(std::ostream& ioES=std::cerr) throw();
  virtual const char* what() const throw();
protected:
  std::string  mMessage;
};
```

Some other exceptions classes are defined into Open BEAGLE for different type of problem encountered. For most concrete standard Open BEAGLE exceptions, there is an associated macro that simplify the throwing of an exception. Table 3.2 presents the list of exceptions defined in Open BEAGLE, with a short description of the context they are used.

Exception class	**Parent class**	**Description**
`Exception`	`std::exception`	Base exception class
`TargetedException`	`Exception`	Family of exceptions relative to a specific piece of code. A targeted exception includes the file name and line number where the exception is thrown.
`ValidationException`	`Exception`	Generally related to the detection of invalid parameter values.
`AssertException`	`TargetedException`	Thrown when an Open BEAGLE assertion test fail. Several macros such `Beagle_AssertM` are also defined to do C-like assertion tests.
`BadCastException`	`TargetedException`	Thrown when a `castObjectT` or `castHandleT` operation fail.
`InternalException`	`TargetedException`	Thrown when unexpected situation denoting internal problems are detected.
`IOException`	`TargetedException`	Thrown when problems with the XML parser or streamer, or in the `read` or `write` methods, are detected.
`ObjectException`	`TargetedException`	Exception associated to a specific object. The error message usually display the serial representation of the associated object.
`RunTimeException`	`TargetedException`	Generic exception denoting a problem detected at the run-time that doesn't fall into the previous categories.

Table 3.2: Standard exception classes defined in Open BEAGLE

NOTE: New exceptions have been added to the GP framework, to interrupt the interpretation of an individual. They should be added to the table.

3.3.8 Summary

Figure 3.3 summarizes the Open BEAGLE OO foundation with a UML[1] diagram. It presents the different entities that compose the generic part of the framework. The most important of them is the class `Object`, the basic class from which most others Open BEAGLE class inherits.

[1] *Unified Modeling Language*. See [7] for further explanation about UML.

Any object subtype could be smart pointed by a Open BEAGLE pointer, which managed the deallocation of the object when it is appropriate. The allocators are objects that allocate, clone, and copy Open BEAGLE objects. The container is a flexible data structures tightly related with the objects, smart pointers and allocators. The Open BEAGLE XML parser and output streamer are objects to insert and extract objects into/from C++ streams. They allow a transparent reading and writing of Open BEAGLE object the powerful XML file format. There is some templates related to these entities that exploit great concepts of generic programming and make the programmer's life easier. Finally, a short discussion of the error handling mechanism deployed in Open BEAGLE terminates the section.

3.4 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 versatility to the framework and simplifies the implementation of any EA. See Figure 3.4 for the generic EC framework. The following discussion explains this generic framework and is divided into three sections: the populations and statistics, the internal system, and the operators and evolvers.

3.4.1 Populations and Statistics

The statistics of the fitness measurements of a population is held in the following structure.

```
struct Measure {
  std::string mId;
  float       mAvg;
  float       mStd;
  float       mMax;
  float       mMin;
};

class Stats : public Object, public std::vector<Measure> {
public:
  void          addItem(std::string inTag, double inValue);
  void          clearItems();
  double        deleteItem(std::string inTag);
  double&       getItem(std::string inTag);
  const double& getItem(std::string inTag) const;
...
protected:
  std::map<std::string,double>  mItemMap;
  std::string                   mId;
```

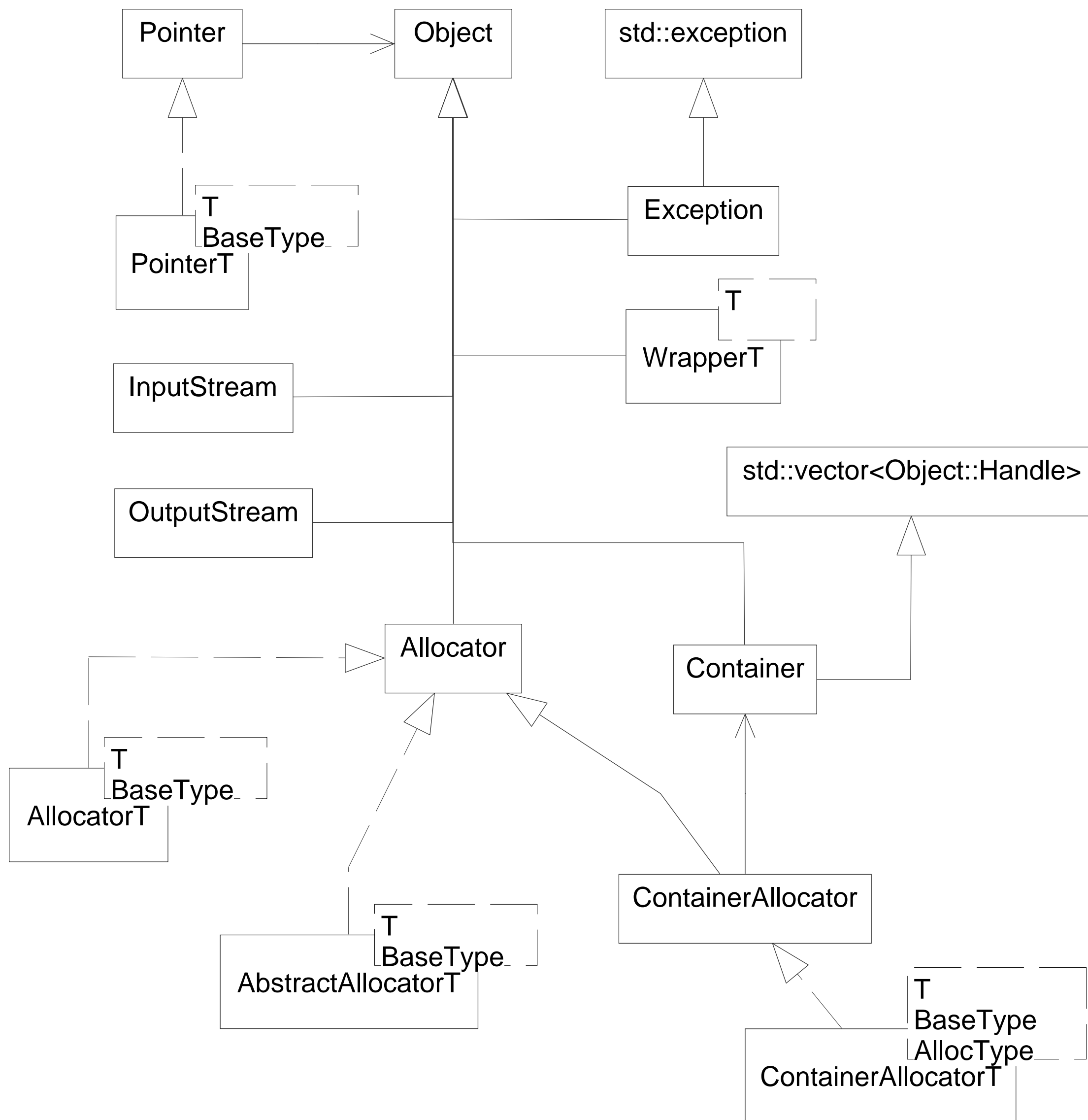

Figure 3.3: Generic object oriented foundation of Open BEAGLE

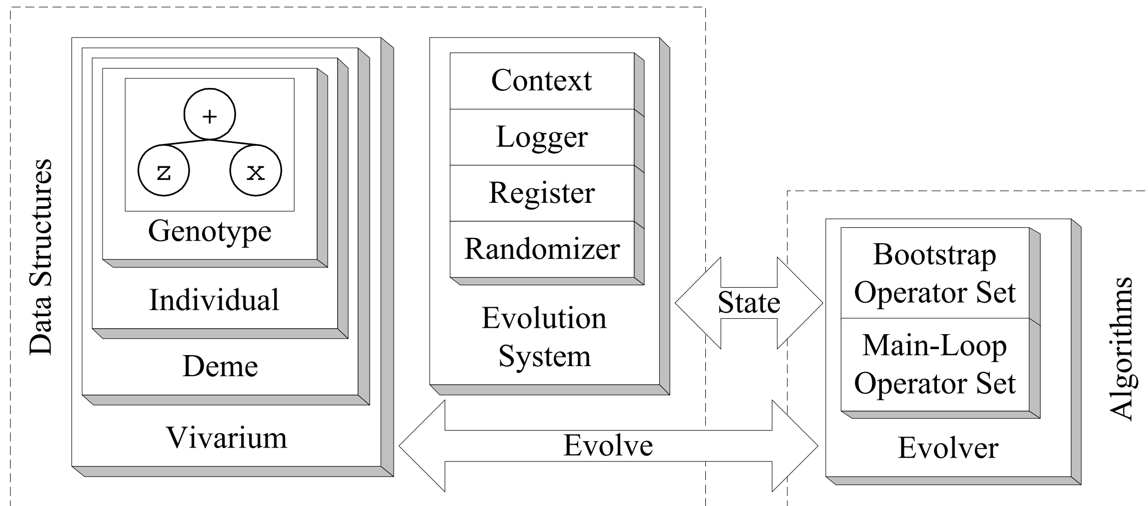

Figure 3.4: Generic EC framework architecture

```
  unsigned int                  mGeneration;
  unsigned int                  mPopSize;
};
```

For the usual EC algorithm, statistics for a population are calculated every generation. Two important elements compose the statistics: the *measures* and the *items*. A measure provides the statistical analysis of the distribution (average, standard deviation, maximum, and minimum) of a given value of interest taken on the population. On the other hand, an item is a single value represented by a floating-point number associated to a tag name. Items and measures are dynamically added to the statistics, and are generally computed by a statistics computation operators. These operators are specific to the fitness object and inherit from the abstract class `StatsCalculationOp`.

```
class StatsCalculateOp : public Operator {
public:
  virtual void calculateStatsDeme(Stats& outStats, Deme& ioDeme,
                                  Context& ioContext) const =0;
};
```

When a non-standard fitness object is used, a companion statistics computation operator must defined by inheriting from `StatsCalculationOp` class and over-defining the pure virtual method `calculateStatsDeme`.

The fitness holder of an individual is represented in an object derived of the type `Fitness`. When the user implements a concrete evaluation class, the method `evaluate` must returns a fitness object appropriate to the application. There is some of the predefined standard fitness measures, such `FitnessSimple`, `FitnessSimpleMin`, `FitnessMultiObj`, `FitnessMultiObjMin`

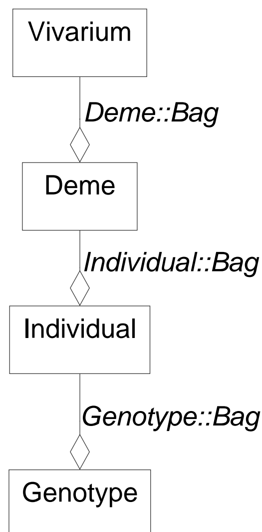

Figure 3.5: Structure of the population in Open BEAGLE

or `GP::FitnessKoza`, which defined respectively a fitness made of a single float, the minimization equivalent of single float, multiobjective maximization fitness measures, multiobjective minimization fitness measures or a more elaborated fitness taken from [15, 16]. A specialized fitness type could also be defined for more sophisticated evolution.

In Open BEAGLE, the population is structured into four layers: a vivarium, the demes, the individuals and the genotypes. The first layer, the vivarium, comprises the whole population of a specific evolution, that is, an aggregate of one or more demes. A deme [17] is a population that evolves in an independent environment. Generally, at each generation, there are some individuals that migrate between the demes that compose a vivarium. A deme is implemented in class `Deme`. The class provides and implements an interface of standard methods necessary to evolve its own population. The class `Vivarium` describes a bag of demes, which is itself a bag of individuals. Every single vivarium and deme also has an hall-of-fame (a `HallOfFame` object), where the best-of-run individuals are conserved, and its statistics (a `Stats` object).

The next underlying layer, the individual, is defined in class `Individual`. It is a bag of genotypes, the parts that compose a genome. The genotypes are tightly related to the representation of the individuals for a given EC algorithm. The genotype interface is declared in the abstract class `Genotype`. This hierarchical organization of the population is illustrated in Figure 3.5.

3.4.2 Internal System

This section presents the internal system of Open BEAGLE, which is the structure that holds and gives access to the state of the genetic engine. These structures are fundamental, because

Log Level	Value	Description
Nothing	0	Log nothing
Basic	1	Log essential informations
Stats	2	Log evolution statistics
Info	3	Log general informations (default)
Detailed	4	Log details on operations
Trace	5	Log trace of the algorithms
Verbose	6	Log details on everything (disabled in optimization mode)
Debug	7	Debug (enabled only in full debug mode)

Table 3.3: Log levels available in Open BEAGLE

they are used as entry points to the data of the evolution.

During the evolutionary processes, a context gives the current state of the evolution. A basic, general context is implemented in class `Context`. It provides some essential contextual informations such as the current deme, individual, genotype and generation. For a specific EC algorithm, a specific context could be used. For example, a GP specific context is defined, `GP::Context`, which contains the call stack with some other GP specific informations. An Open BEAGLE context is similar to the execution context of a computer that contains the different registers, counters and pointers to the actual state of the machine.

Given that the parameters of Open BEAGLE are distributed in the appropriate objects, an agent is implemented to take into account these parameter: the register. All the variables that are considered as parameters should be registered by giving the reference (object handle) of the parameter with the associated namespace and tag. The class `Register` can be seen as a centralized database from which any entity could dynamically add, delete, access or modify parameters. The register is also responsible of interpreting the parameters part of the configuration file[2].

All the output messages given to the user pass by the logger. It consists of an interface with the user, that receives all messages, associated with a type, a class name, and an output level, and output them in a given device if the used log level allow it. This is very interesting if a user want, for example, to use Open BEAGLE into a broader system using a graphical user interface. In such case, the user only need to define his own specialized logger that will intercept the messages and log them into the desired device, for example a specific graphical windows. There is actually one specialized logger, `LoggerXML`, that log messages to a file and/or the console (the STDOUT device) in a XML format. The other very interesting aspect of the logger is the possibility to choose the log level desired. The messages outputted are classified into eight categories, as depicted in Table 3.3. The registered parameters `lg.console.level` and `lg.file.level` allows the user to select the desired log level. For example, if the user choose

[2]See section 4.3 for further details.

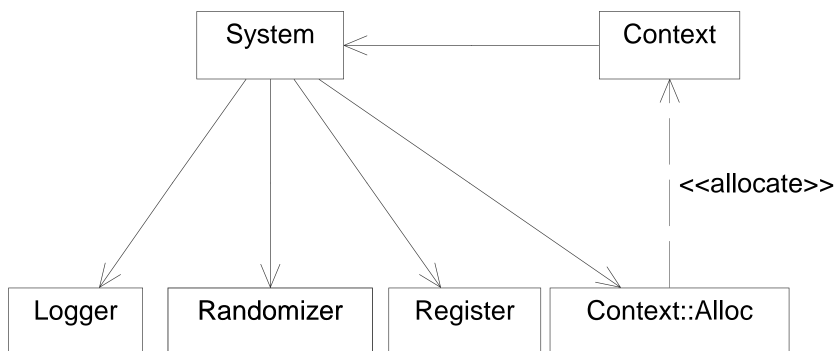

Figure 3.6: Internal Open BEAGLE system

the log level *info (3)*, all messages classified in categories *basic (1)*, *stats (2)*, and *info (3)* will be outputted. Log levels *basic (1)* to *detailed (4)* are appropriate for monitoring evolutions, log levels *detailed (4)* and *trace (5)* are good to get familiar with the internal functioning of Open BEAGLE, while log levels *trace (5)* to *debug (7)* may be useful to debug an user application, or the framework.

Class `Randomizer` provides a common pseudo-random numbers generator. The randomizer comprises two parameters that are registered (in the register): the internal state and the seed. The internal state is an integer that give the actual state of the randomizer. This value change at every generation of a random number. This value is useful mainly to restart an evolution from a milestone. The seed is the value of the first state of the randomizer. By default, the seed is initiated to the timer value. The seed can be set by the user to reproduce an evolution. Starting with version 2.0.0, the default random number generator used in the framework is based on the *Mersenne Twister* of Matsumoto and Nishimura [20].

NOTE: There have been several changes in the organisation of the randomizer, although it is still the Mersenne twister. All the previous paragraph should be updated accordingly.

The entry point to these resources is given by an extensible central repository: the system. This is simply an entry point that possessed some vital resources of the EC engine: a context allocator, a reference to the register, a reference to the logger, and a reference to the randomizer. This general system is implemented in the class `System`. The system is accessible by any context of a given evolution. The relations between the different constituents of the internal system is presented in Figure 3.6.

NOTE: A paragraph is necessary here in the concept of *system components*, new with version 3.0.0.

3.4.3 Operators and Evolver

The operator is a central concept of Open BEAGLE as an EC framework. The main-loop of operations executed on populations is dynamically defined. The operators are specified at runtime and the user is free to define them for his evolution. This give the opportunity to easily and rapidly experiment numerous variants of EC. The operator and evolver model is based on the *Strategy* design pattern [11], which is applied to the evolutionary algorithms. The operator interface is declared in the abstract class `Operator`.

```
class Operator : public Object {
public:
  Operator() { }
  virtual void initialize(System& ioSystem) { }
  virtual void postInit(System& ioSystem) { }
  virtual void operate(Deme& ioDeme,Context& ioContext) =0;
};
```

Before the characteristic method is applied to demes, method `initialize` is invocated. In this method, the operator usually registers its own parameters, probabilities or anything else, used by the characteristic operation. The characteristic operation is defined in the virtual method `operate`. There is a bunch of predefined operators in Open BEAGLE. To name a few of them, the tournament selection operator [17] (`SelectTournamentOp`), the GP tree crossover operator [15] (`GP::CrossoverOp`) and the milestone writing operator (`MilestoneWriteOp`).

The user that define his own operators must be aware that the system is not fully set-up when the `initialize` method is called. For example, the random number generator must not be used, as the seed can be modified thereafter on the command-line or in the configuration file. The rule is that the `initialize` method must be used only to add elements to the evolution system. There is also a second hook called `postInit`, which is also called once, just after the whole system is set-up. The `postInit` method can be use, for example, to configure some data using the random number generator, which is now fully configured.

The operators of a specific evolution are inserted into the evolver that supervises the evolution process. This object, implemented in class `Evolver`, comprises two major attributes: the bootstrap operator set and the main-loop operator set. The bootstrap operators set contains an ordered list of operators to apply on each deme, for the initial generation. The main-loop operators set is an ordered list of operators to apply iteratively, at each generation, on each deme. The user could launch an evolution, by calling the method `evolve` with the vivarium to evolve as argument.

```
class Evolver : public Object {
public:
  virtual void initialize(System::Handle ioSystem,int& ioArgc,char** ioArgv);
  virtual void evolve(Vivarium::Handle ioVivarium);
```

```
protected:
  Operator::Bag mBootStrapSet;
  Operator::Bag mMainLoopSet;
};
```

For common EC algorithms, the user usually needs not to create custom sequences of operators. In fact, some classes inheriting of `Evolver` can be used to create evolvers with predefined operator sets. If a special EC algorithm is needed, a custom building method can be invocated and the evolver should be configured properly. Section 3.5.5 presents the standard operator library with the methods that can be used to build standard EC algorithm.

3.4.4 Breeder Model

NOTE: The explanations on the breeder model are somewhat fuzzy. It is not easy to understand this thing, although this is an central mechanism of Open BEAGLE. More explanations are necessary in order to make it clear.

A conceptual layer is added over the operator/evolver model: the breeder model. This allow the manipulation of the population at the individual level, in comparison to manipulations at the deme level by the standard operators. Note that the use of the breeder model over the standard operator/evolver model is optional; users don't have to use it if they don't need it.

The breeder model includes a set of operators structured into a tree: the breeder tree. At the root of the breeder tree, there is a replacement strategy, which act as an interface between the operator/evolver mechanism and the breeder model. The function of a replacement strategy is to generate, by applying the breeder tree on a deme, the next generation deme. The direct children in the breeder tree to the replacement strategy are called alternatively following their respective breeding probability. The operators that compose the nodes of the breeding tree process the bred individual, while the leaves of the breeder tree are generally used to select individuals. The following snippet presents the XML configuration of a breeder tree for the `maxfct` example, using a steady-state replacement strategy [1].

```
...
<SteadyStateOp>
  <MaxFctEvalOp>
    <GA-CrossoverOp>
      <SelectTournamentOp/>
      <SelectTournamentOp/>
    </GA-CrossoverOp>
  </MaxFctEvalOp>
  <MaxFctEvalOp>
    <GA-MutationOp>
      <SelectTournamentOp/>
    </GA-MutationOp>
  </MaxFctEvalOp>
```

```
  <SelectTournamentOp/>
</SteadyStateOp>
...
```

Three new classes are added to the framework to support the new breeder model: `BreederOp`, `BreederNode`, and `ReplacementStrategyOp`. Class `BreederOp` designates operators that can, in addition to the facilities of the operators, process a pool of individuals to return one bred individual. The interface of the class `BreederOp` is the following.

```
class BreederOp : public Operator {
public:
  virtual Individual::Handle breed(Individual::Bag& inBreedingPool,
                                   BreederNode::Handle inChild,
                                   Context& ioContext) = 0;
  virtual float getBreedingProba(BreederNode::Handle inChild) = 0;
};
```

The pure virtual method `breed` is used to apply the characteristic breeder operation. It takes as first argument a pool of individual from which the breeding operation is done, as second argument a breeder node giving the position of the first child to the actual operator in the breeder tree, and as last argument the evolutionary context. The method `getBreedingProba` returns the breeding probability, which is used by the replacement strategy to decide which of its subtrees is called to generate a bred individual. The `BreederNode` class defines the breeder tree structure. It is made of three smart pointers, one to its first child, another to its next sibling, and one last to the associated breeder operator. Finally, the class `ReplacementStrategyOp` defines an item of the evolver's operator set that can be used as the root of the breeding tree, to generate a new generation of individuals. Five replacement strategies operator are actually implemented in Open BEAGLE: `GenerationalOp`, which defines a generational replacement operator using a breeder tree[3], `SteadyStateOp`, which defines a steady-state replacement operator, `NSGA2Op`, which defines the NSGA-II multiobjective evolutionary algorithm [5], and finally the `MuCommaLambdaOp` and `MuPlusLambdaOp` replacement operators, which define the (μ, λ) and $(\mu + \lambda)$ approaches commonly used with the evolution strategy EC paradigm.

3.4.5 Summary

This section had presented three important components of the EC architecture of Open BEAGLE. The first component is the structure of the population of the framework, that includes elements such the vivarium, the demes, the individuals, and the genotypes. The following is the internal system, where the parameters and some other entities are held. The third component is a central aspect of Open BEAGLE as EC framework, the operators and the evolver. An operator is simple operation, such writing a population to the disk or GP tree crossover operation, which

[3] Note that the standard evolver/operator model (not using the breeder model) is implicitly generational.

is applied sequentially and iteratively in conjunction of others operators to evolve a population. An evolver is an agglomerate of operators to apply on the population. An applications developers can use an evolver pre-packed with usual operators or instantiate his own custom evolver composed of more exotic operators. And finally, the breeder model represents a set of specific operators that allow the manipulation of population with a finer granularity.

3.5 Miscellaneous Architecture Elements

This section presents of different miscellaneous architecture elements more specific certain EC flavors or applications. It presents of some common application-specific elements such the evaluation and termination operators, and some other more EC flavor-specific elements such the GP primitives. It also discuss of a more general elements such the different standard operators defined in the Open BEAGLE framework.

3.5.1 Evaluation Operator

In most EC algorithms, modified individuals are evaluated at each generation. To do so, there is a common operator to evaluate individuals, the evaluation operator. This operator is implemented in class `EvaluationOp`, which is a standard abstract interface for the evaluation of an individual.

```
class EvaluationOp : public Operator {
public:
  virtual Fitness::Handle evaluate(Individual& inIndividual,
                                   Context& ioContext) =0;
  virtual void operate(Deme& ioDeme,Context& ioContext);
};
```

Usually, for a specific problem, the user must implements a concrete class that specialize `EvaluationOp`. In this concrete class, he must declare a definition of the method `evaluate`. This method provides the way an individual is evaluated.

3.5.2 Termination Operator

As explained in previous sections, the basic Open BEAGLE algorithm is an infinite loop over the main operator set. But, there is a standard Open BEAGLE operator, the termination criterion, that determine whether to stop or not the evolution process. In Open BEAGLE, the termination criterion is implemented in operator `TerminationOp`.

```
class TerminationOp : public Operator {
public:
  virtual bool terminate(const Deme& inDeme,const Context& inContext) = 0;
  virtual void operate(Deme& ioDeme,Context& ioContext);
};
```

The default termination criteria is the number of generations the algorithm iterates for an evolution. This criterion is defined in the operator class `TermMaxGenOp`. There is also several other termination operators such `TermMaxFitnessOp` (maximum fitness value), `TermMinFitnessOp` (minimum fitness value), `TermMaxEvalsOp` (maximum number of fitness evaluations), and `GP::TermMaxHitsOp` (maximum number of hits for GP Koza's fitness measure). The user could change or add termination criteria for his evolutionary application by redefining the method `terminate`.

3.5.3 Multiobjective Evolutionary Algorithms

Since version 2.1.0, Open BEAGLE includes support for Multiobjective Evolutionary Algorithms (MOEA). For this purpose, two fitness measure types have been defined in the framework, that is a maximization one named `FitnessMultiObj`, and its minimization equivalent `FitnessMultiObjMin`. In both these two fitness measures, a Pareto domination test has been added in method `isDominated`. If the user want to use a different multiobjective measure, for example a fitness measure comprising a mix of minimizing and maximizing objectives, the method `isDominated` of the new fitness measure class must be over-defined accordingly.

The other element of MOEA support in Open BEAGLE is the multiobjective selection operator. There is currently two of these implemented, that is NSGA-II [5], implemented as a replacement strategy of a breeder model, and the NPGA 2 [6], implemented as a classical selection operator. There is also a multiobjective-specific hall-of-fame class called `ParetoFrontHOF`, that contains the population's Pareto front.

3.5.4 Co-evolution

NOTE: This subsection on co-evolution is far to short. It should be rewritten with more explanations and use examples. Maybe a complete section in next chapter would be necessary.

Co-evolution framework is based on multi-threading programming, where each thread is associated to a population. The execution sequence of each thread is the same than what usually done in the main function of standard evolutions, as described in section 4.2 (i.e. build system, evaluation operator, vivarium, evolver, initialize evolver, and start evolution). The population in each thread evolves independently, with inter-thread synchronization only in co-evolutionary fitness evaluation operator. This operator behaves quite differently than usual mono-population evaluation operator. The co-evolution evaluation procedure starts by calling `Coev::EvaluationOp::makeSets`, which makes evaluation sets of the evolving population and add them into shared storage structure, using method `Coev::EvaluationOp::addSet`. When the desired number of evaluation sets is added (the trigger value) the co-evolutionary fitness evaluation method defined in `Coev::EvaluationOp::evaluateSets` is called. Pure virtual meth-

Operator	Superclass
RegisterReadOp	Operator
IfThenElseOp	Operator
MilestoneReadOp	Operator
MilestoneWriteOp	Operator
StatsCalculateOp	Operator
StatsCalcFitnessSimpleOp	StatsCalculateOp
GP::StatsCalcFitnessSimpleOp	StatsCalcFitnessSimpleOp
GP::StatsCalcFitnessKozaOp	StatsCalculateOp
ParetoFrontCalculateOp	Operator

Table 3.4: Open BEAGLE functional operators

ods `Coev::EvaluationOp::makeSets` and `Coev::EvaluationOp::evaluateSets` are problem-specific and must be defined by the user in its co-evolutionary fitness evaluation operators.

A trigger value is used to specify the number of evaluation sets needed to start a co-evolutionary evaluation. This value is usually equal to the number of threads/populations used, as usually each thread/population add one evaluation set before doing the co-evolutionary evaluation operation. But different trigger value can be used depending on the context.

Marc Parizeau's portable C++ classes for multi-threading (defined in namespace `Threading`) are also provided with co-evolutionary framework. These are used internally by the co-evolution framework. Users are advised to use them for their co-evolutionary applications.

3.5.5 Standard Open BEAGLE Operators Library

NOTE: Lot of new operators has been added to Open BEAGLE. They should be added to the different Tables of the actual subsection.

An important advantage of OO programming is the reuse of standard components with few or no modifications. Up to now, a bunch of generic and specific (to GP and GA) operators are integrated to Open BEAGLE. There is plans to integrate some more for doing other EC algorithms.

Tables 3.4 to 3.12 present the standard operators of Open BEAGLE. The Tables 3.13 to 3.16 present the bootstrap and main loops of the bit string GA, float vector GA, ES, and GP default evolvers.

3.5.6 GP Genotypes

For the GP specific individual implementation, the abstract interface `Individual` is sub-classed in the concrete class `GP::Individual`. An standard GP genotype, `GP::Tree`, is also defined.

Operator	**Superclass**
`MigrationOp`	`Operator`
`MigrationRandomRingOp`	`MigrationOp`
`DecimateOp`	`Operator`

Table 3.5: Open BEAGLE population manipulation operators

Operator	**Superclass**
`BreederOp`	`Operator`
`ReplacementStrategyOp`	`Operator`
`GenerationalOp`	`ReplacementStrategyOp`
`SteadyStateOp`	`ReplacementStrategyOp`
`MuCommaLambdaOp`	`ReplacementStrategyOp`
`MuPlusLambdaOp`	`ReplacementStrategyOp`
`NSGA2Op`	`ReplacementStrategyOp`
`OversizeOp`	`ReplacementStrategyOp`

Table 3.6: Open BEAGLE breeder and replacement strategy operators

Operator	**Superclass**
`InitializationOp`	`Operator`
`GA::InitBitStrOp`	`InitializationOp`
`GA::InitFltVecOp`	`InitializationOp`
`GA::InitESVecOp`	`InitializationOp`
`GP::InitFullOp`	`InitializationOp`
`GP::InitGrowOp`	`InitializationOp`
`GP::InitHalfOp`	`InitializationOp`
`GP::InitFullConstrainedOp`	`GP::InitFullOp`
`GP::InitGrowConstrainedOp`	`GP::InitGrowOp`
`GP::InitHalfConstrainedOp`	`GP::InitializationOp`

Table 3.7: Open BEAGLE initialization operators

Operator	**Superclass**
`SelectionOp`	`BreederOp`
`SelectRandomOp`	`SelectionOp`
`SelectRouletteOp`	`SelectionOp`
`SelectTournamentOp`	`SelectionOp`
`NPGA2Op`	`SelectionOp`

Table 3.8: Open BEAGLE selection operators

Operator	**Superclass**
EvaluationOp	BreederOp
GP::EvaluationOp	EvaluationOp
Coev::EvaluationOp	EvaluationOp
Coev::GP::EvaluationOp	Coev::EvaluationOp

Table 3.9: Open BEAGLE fitness evaluation operators

Operator	**Superclass**
TerminationOp	Operator
TermMaxGenOp	TerminationOp
TermMaxFitnessOp	TerminationOp
TermMinFitnessOp	TerminationOp
TermMaxEvalsOp	TerminationOp
GP::TermMaxHitsOp	TerminationOp

Table 3.10: Open BEAGLE termination operators

Operator	**Superclass**
CrossoverOp	BreederOp
GA::CrossoverBlendFltVecOp	CrossoverOp
GA::CrossoverBlendESVecOp	CrossoverOp
GA::CrossoverOnePointOpT<T>	CrossoverOp
GA::CrossoverOnePointBitStrOp	GA::CrossoverOnePointOpT<GA::BitString>
GA::CrossoverOnePointFltVecOp	GA::CrossoverOnePointOpT<GA::FloatVector>
GA::CrossoverOnePointESVecOp	GA::CrossoverOnePointOpT<GA::ESVector>
GA::CrossoverTwoPointsOpT<T>	CrossoverOp
GA::CrossoverTwoPointsBitStrOp	GA::CrossoverTwoPointsOpT<GA::BitString>
GA::CrossoverTwoPointsFltVecOp	GA::CrossoverTwoPointsOpT<GA::FloatVector>
GA::CrossoverTwoPointsESVecOp	GA::CrossoverTwoPointsOpT<GA::ESVector>
GA::CrossoverUniformOpT<T>	CrossoverOp
GA::CrossoverUniformBitStrOp	GA::CrossoverUniformOpT<GA::BitString>
GA::CrossoverUniformFltVecOp	GA::CrossoverUniformOpT<GA::FloatVector>
GA::CrossoverUniformESVecOp	GA::CrossoverUniformOpT<GA::ESVector>
GP::CrossoverOp	CrossoverOp
GP::CrossoverConstrainedOp	GP::CrossoverOp

Table 3.11: Open BEAGLE crossover operators

Operator	**Superclass**
MutationOp	BreederOp
GA::MutationESVecOp	MutationOp
GA::MutationFlipBitStrOp	MutationOp
GA::MutationGaussianFltVecOp	MutationOp
GP::MutationSwapOp	MutationOp
GP::MutationShrinkOp	MutationOp
GP::MutationStandardOp	MutationOp
GP::MutationSwapSubtreeOp	MutationOp
GP::MutationSwapConstrainedOp	GP::MutationSwapOp
GP::MutationShrinkConstrainedOp	GP::MutationShrinkOp
GP::MutationStandardConstrainedOp	GP::MutationStandardOp
GP::MutationSwapSubtreeConstrainedOp	GP::MutationSwapSubtreeOp

Table 3.12: Open BEAGLE mutation operators

Bootstrap Operators	**Main-loop Operators**
IfThenElseOp("ms.restart.filename")	SelectTournamentOp
MilestoneReadOp	GA::CrossoverOnePointBitStrOp
else	GA::MutationFlipBitStrOp
GA::InitBitStrOp	EvaluationOp
EvaluationOp	MigrationRandomRingOp
StatsCalculateFitnessSimpleOp	StatsCalculateFitnessSimpleOp
end if	TermMaxGenOp
TermMaxGenOp	MilestoneWriteOp
MilestoneWriteOp	

Table 3.13: `GA::EvolverBitString` default operator sets

Bootstrap Operators	**Main-loop Operators**
IfThenElseOp("ms.restart.filename")	SelectTournamentOp
MilestoneReadOp	GA::CrossoverBlendFltVecOp
else	GA::MutationGaussianFltVecOp
GA::InitFltVecOp	EvaluationOp
EvaluationOp	MigrationRandomRingOp
StatsCalculateFitnessSimpleOp	StatsCalculateFitnessSimpleOp
end if	TermMaxGenOp
TermMaxGenOp	MilestoneWriteOp
MilestoneWriteOp	

Table 3.14: `GA::EvolverFloatVector` default operator sets

Bootstrap Operators	**Main-loop Operators**
`IfThenElseOp("ms.restart.filename")`	`MuCommaLambdaOp`
`MilestoneReadOp`	`EvaluationOp`
`else`	`GA::MutationGaussianFltVecOp`
`GA::InitESVecOp`	`end replacement strategy`
`EvaluationOp`	`MigrationRandomRingOp`
`StatsCalculateFitnessSimpleOp`	`StatsCalculateFitnessSimpleOp`
`end if`	`TermMaxGenOp`
`TermMaxGenOp`	`MilestoneWriteOp`
`MilestoneWriteOp`	

Table 3.15: `GA::EvolverES` default operator sets

Bootstrap Operators	**Main-loop Operators**
`IfThenElseOp("ms.restart.filename")`	`SelectTournamentOp`
`MilestoneReadOp`	`GP::CrossoverOp`
`else`	`GP::MutationStandardOp`
`GP::InitHalfOp`	`GP::MutationShrinkOp`
`GP::EvaluationOp`	`GP::MutationSwapOp`
`GP::StatsCalculateFitnessSimpleOp`	`GP::MutationSwapSubtreeOp`
`end if`	`GP::EvaluationOp`
`TermMaxGenOp`	`MigrationRandomRingOp`
`MilestoneWriteOp`	`GP::StatsCalculateFitnessSimpleOp`
	`TermMaxGenOp`
	`MilestoneWriteOp`

Table 3.16: `GP::Evolver` default operator sets

This class is declared as genotype and a vector of nodes. The nodes are declared by the struct `GP::Node`, that comprises a smart pointer to a GP primitive and the size of the sub-tree.

```
namespace GP {
struct Node {
  Primitive::Handle mPrimitive;
  unsigned int      mSubTreeSize;
};
class Tree : public Genotype, public std::vector<Node> { ... };
}
```

One of the most notable points with Open BEAGLE is the possibility to redefine and modify almost everything. This is easily done by giving the appropriate allocator of a subclass to the upper layer class constructor. As example, when a `GP::Vivarium` is created, it creates by default `GP::Individual` by passing to its constructor an allocator of `GP::FitnessKoza` for the fitness value and an allocator of `GP::Tree` to allocate its genotypes.

```
namespace GP {
class Individual : public Beagle::Individual {
public:
  Individual(Fitness::Alloc::Handle inFitnessAlloc,
             GP::Tree::Alloc::Handle inTreeAlloc);
  ...
};
}
```

But, if the user want to use a different genotype, suppose a user defined custom genotype implemented in the class `MyGenotype`, he only needs to pass an allocator of his custom genotype to the allocator of `GP::Individual`:

```
Fitness::Alloc::Handle        lFitsAlloc     = new GP::FitnessKoza::Alloc;
MyGenotype::Alloc::Handle     lGenotypeAlloc = new MyGenotype::Alloc;
GP::Individual::Alloc::Handle lIndivAlloc    =
  new GP::Individual::Alloc(lFitsAlloc,lGenotypeAlloc);
```

The allocator of individual will then create individuals that are bags of `MyGenotype`. Furthermore, if the `GP::Individual` is copied into another `GP::Individual`, the reference to the allocator of custom genotype is copied and the genotypes of the bag are cloned.

3.5.7 GP Primitives and Sets

Open BEAGLE uses a real OO approach to implement the primitives that compose GP genotypes. The genotypes are composed of nodes that have one attribute, that is a smart pointer to an abstract primitive. Using this abstract interface, it is easy to implement primitives that have specific behavior without loosing any generality.

To make a concrete primitive that is usable to compose GP trees, the user needs to declare a concrete class derived from the abstract superclass `GP::Primitive`. Given that, existing primitives can be reused or specialized. These are some fundamental principles of OO programming. These also give us powerful mechanisms to defined atypical primitives without tweaking the internal structure. The section 4.6.1 at page 68 is a deep discussion on how to define GP primitives.

NOTE: The following explanations on association between the number of primitive sets in the super set and the number of trees in the GP individuals is no more true. This implicit link is broken and it is now possible to have GP individuals with several trees while having one primitive set. An index to the associated primitive set has been added to the GP trees. The following paragraph should be rewritten.

Once the primitives used for a genetic evolution of programs are determined, they must be packed into sets that are given to the GP system. In Open BEAGLE, the process is straightforward: the user directly creates the set of primitives by inserting references to primitives. The

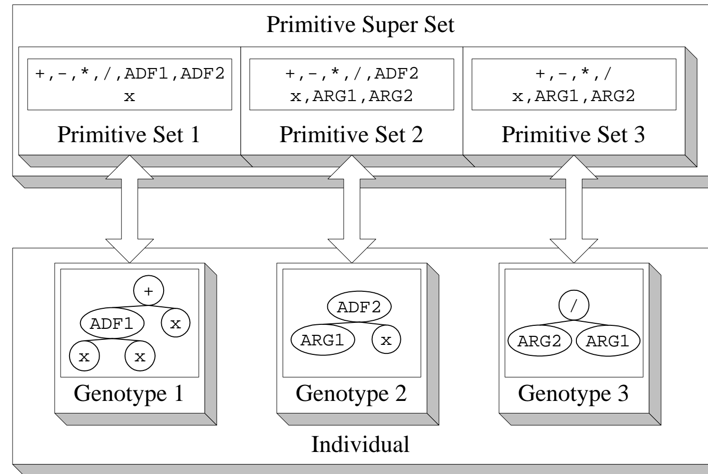

Figure 3.7: Relation between the GP sets, primitives and trees.

primitives set is implemented in the class `GP::PrimitiveSet`, a bag of `GP::Primitive`. When the primitives sets are made, they are put into the super set of primitives, implemented in the class `GP::PrimitiveSuperSet`. A `GP::PrimitiveSuperSet` is a bag of `GP::PrimitiveSet`. Finally, this super set of primitives is given to the system of GP. When only one set of primitives is used, which is usual when there is no use of automatically defined structures, the user could directly pass the simple primitive set as reference to the system. Then, the system build a super set of primitives that wraps the simple set. Each set is associated with corresponding trees of each individual, as presented in Figure 3.7 at page 52.

NOTE: Lot of improvements have been made to the GP framework concerning the individuals. For example, primitives with variable number of arguments, variable selection weight of primitives in primitive set, dynamic ADFs, and evolutionary module acquisition should be detailed.

3.5.8 GP Primitives Library

NOTE: The ADFs mechanisms has been completely changed.

Some common primitives are defined from the abstract interface `GP::Primitive`. As with the operators, there is some common GP primitives into Open BEAGLE. At this moment, there is three different series of standard primitives: the generic, the arithmetics and the Boolean functions. The generic series comprise a simple terminal containing a token, `TokenT`, a primitives to program automatically subroutines [16], `AdfT`, and a primitive for randomly generated constants, `EphemeralT`. The second bunch comprises usual arithmetics operators. The third bunch comprises some common logical operators that work on the Open BEAGLE `Bool` type.

Primitive	Datum	Description
`TokenT`	Any	Terminal that contain a value, the token.
`AdfT`	Any	Call to an associated ADF.
`EphemeralT`	Any	Generate and represents randomly generated constants.
`EphemeralDouble`	`Double`	Generate and represents randomly generated floating-point constants in $[-1, 1]$.
`AddT`[a]	Any[b]	Adds two floating-point numbers. $(x_1 + x_2)$
`SubtractT`[c]	Any[d]	Subtract two floating-point numbers. $(x_1 - x_2)$
`MultiplyT`[e]	Any[f]	Multiply two floating-point numbers. $(x_1 \times x_2)$
`DivideT`[g]	Any[h]	Protected division [15] of two floating-point numbers. (x_1/x_2)
`Sin`	`Double`	Sinus of a floating-point number in radians. $(sin(x_1))$
`Cos`	`Double`	Cosine of a floating-point number in radians. $(cos(x_1))$
`Log`	`Double`	Natural logarithm of a floating-point number. $(ln(x_1))$
`Exp`	`Double`	Exponentiation of a floating-point number. (e^{x_1})
`And`	`Bool`	And logical operator.
`Or`	`Bool`	Or logical operator.
`Not`	`Bool`	Not logical operator.
`Nand`	`Bool`	Nand logical operator.
`Nor`	`Bool`	Nor logical operator.
`Xor`	`Bool`	Xor logical operator.

Table 3.17: List of standard GP primitives

[a] Type `Add` is a synonym of `AddT<Double>`.
[b] The datum type must have an operator+.
[c] Type `Subtract` is a synonym of `SubtractT<Double>`.
[d] The datum type must have an operator-.
[e] Type `Multiply` is a synonym of `MultiplyT<Double>`.
[f] The datum type must have an operator*.
[g] Type `Divide` is a synonym of `DivideT<Double>`.
[h] The datum type must be comparable to the `float` type and have an operator/.

The Table 3.17 at page 53 presents and describe of all these Primitives.

Chapter 4

Open BEAGLE User Manual

This section is a extensive presentation of what an application programmer, which is called the user from now, needs to know to develop EC process with Open BEAGLE. Some important aspects of C++ programming using Open BEAGLE are presented here. The discussion starts with what the user should and should not do (the guidelines). Following this is presented a simple pattern of what is needed to build a system for an evolution. Thereafter the different ways to modify the parameters of the system are exposed. Finally is presented the different approaches to customize the evolutionary process and a specific manual for the different standard EC algorithms implemented in Open BEAGLE.

4.1 Guidelines

Open BEAGLE is a powerful framework for EC. It is based on different elaborated mechanisms that make it both a flexible and programmer-friendly environment. But, these mechanisms require that to the user follows some general guidelines. A little adaptation effort is necessary to make the experience with Open BEAGLE as pleasant as possible. There is these general guidelines, accompanied by a deeper discussion on each of them.

1. *Any instance (i.e. C++ objects) used in combination of Open BEAGLE must inherit, directly or indirectly, from the abstract* `Object` *class.*
   As explained in previous sections of the document, every Open BEAGLE classes are derived from the `Object` abstract class[1]. The Open BEAGLE class architecture can be see as a tree with the `Object` class as root. The internal mechanisms of Open BEAGLE usually manipulate data types by using the common `Object` interface. It is then necessary for the user to integrate his application classes to the basic architecture. This is often done implicitly by sub-classing some Open BEAGLE entities. But, in some circumstances, a

[1] With as exception the class `Pointer` and derived.

user may need to define a new class that do not have any relation with other existing Open BEAGLE classes. To do so, it is advised to use the `WrapperT` template to encapsulate these custom types.

2. *Any Open BEAGLE objects that are smart pointed must be allocated on the heap and memory-managed by object handles.*
   The smart pointer, also called handle, is a very useful mechanism that both simplify the task of the user and add global functionality and flexibility to the Open BEAGLE environment. But, users have to be cautious when they use smart pointers. They must be sure to allocate smart pointed objects on the heap, by a call to the `new` operator. This is fundamental because the smart pointed objects are automatically destructed, with a call to the `delete` operator, when their reference counter reach 0. This is incompatible with stack-allocated instances.

3. *The nested types* `Handle`*,* `Alloc`*, and* `Bag` *must be defined in any object classes.*
   In every classes of the Open BEAGLE framework, the types `Handle`, `Alloc`, `Bag` are defined. The handle type is a smart pointer to the type associated (i.e. a `Double::Handle` is a smart pointer of `Double`). An allocator is a type that can allocate, clone and copy the associated type. A bag is an Open BEAGLE container of the given type. Both these types are used extensively by the Open BEAGLE mechanisms. These three types can be easily defined by a synonym declaration to the associated template classes, as in the following listing.

```
class MyClass : public SuperClass {
public:
  typedef PointerT<MyClass,SuperClass::Handle>  Handle;
  typedef AllocatorT<MyClass,SuperClass::Alloc> Alloc;
  typedef ContainerT<MyClass,SuperClass::Bag>   Bag;
  ...
};
```

   The templates take as first parametrized value the type of the associated object. In this case, the associated object is the type `MyClass`. The second parametrized value is the equivalent nested type of the direct superclass. In this case, the class `MyClass` inherits from the class `Object` so the equivalent nested type to a `MyClass::Handle` is a `Object::Handle`.

4. *Any upcast of Open BEAGLE objects must be done by a call to the Open BEAGLE* `castObjectT` *operator for references and C-pointers, and by call to the Open BEAGLE* `castHandleT` *operator for smart pointers.*
   With the ANSI/ISO C++ standard, it is highly recommended to use the new style cast operators instead of the old C-style one. This new style cast operators, such the operators `static_cast` and `const_cast`, are more specialized cast operations. The use of different

cast operators prevent some unwanted casting operations that could lead to nasty hidden bugs. In the `Beagle` namespace, two new cast operator is defined, `castObjectT` and `castHandleT`. These operators must be used to cast any Open BEAGLE object type into another object type. The usage of this casting operator is identical to the usage of new style cast operators.

5. *Any call to Open BEAGLE methods must be into a try-catch block to handle properly any thrown Open BEAGLE exceptions.*

   Open BEAGLE defines his own exceptions in spirit of the modern C++ exception handling mechanisms. In fact, an hierarchy of classes are defined to cover different cases of error, with at the top the class `Beagle::Exception`. When programming an application, the user should be aware to catch any Open BEAGLE exceptions and treat them accordingly. At least, the user should catch them and call the exception aborting method. The following code is an example of an internal structure of Open BEAGLE that detects an irregular situation and throws an appropriate exception.

   ```
   if(n > size()) throw Beagle_RunTimeExceptionM("Out of bound!");
   ```

   The macro `Beagle_RunTimeExceptionM` is a wrapper to create an exception object of the type `Beagle::RunTimeException`, with the message given and the good file name and line number. For almost all the exception classes of Open BEAGLE there is a wrapper macro associated to construct exception with the appropriate file name and line number. A `Beagle_AssertM` macro is also provided to check some condition as with the useful standard C `assert` macro.

   A simple way to achieve the current guideline is to try-catch all the `main` routine of the user application, to intercept any Open BEAGLE exceptions. Using the following code, the exceptions not previously caught is displayed at the standard error output, just before the program exit.

   ```
   int main(int argc, char** argv) {
     try {
       ...
     }
     catch(Beagle::Exception& inException) {
       inException.terminate();
     }
     return 0;
   }
   ```

4.2 Building a System for an Evolution

To build an EC application with Open BEAGLE, some components need to be configured. Depending the type of evolutionary algorithm and the application, the configuration step is more or less demanding. Conceptually, the components to be configured are divided in three classes, the system, the population and the operator and evolver.

- **The system**
  The Open BEAGLE system of EC contains handles to some central entities used all over the evolution, such the register, the randomizer, the logger, and the context. Depending of the evolutionary algorithm used, the system can handles other specialized entities.

- **The population**
  In Open BEAGLE, the representation of the individual is not restricted. The application developer can use different standard population representation or implement his own. However the way to instantiate a population for an evolution follows the same bottom-up pattern:

  1. Instantiate an allocator of the fitness type used;
  2. Instantiate an allocator of the genotype type used;
  3. Instantiate an allocator of the individual type used with the genotype and fitness allocators;
  4. Instantiate an allocator of statistics type used;
  5. Instantiate an allocator of the deme type used with the individual and statistics allocators;
  6. Instantiate the vivarium with the deme and statistics allocators.

  As example, for multiobjective real-valued GA, the above pattern is translated into the following code.

  ```
  FitnessMultiObj::Alloc::Handle lFitsAlloc  = new FitnessMultiObj::Alloc;
  GA::FloatVector::Alloc::Handle lGenoAlloc  = new GA::FloatVector::Alloc;
  Individual::Alloc::Handle      lIndivAlloc =
    new Individual::Alloc(lGenoAlloc,lFitsAlloc);
  Stats::Alloc::Handle           lStatsAlloc = new Stats::Alloc;
  Deme::Alloc::Handle            lDemeAlloc  = new Deme::Alloc(lIndivAlloc,
                                                               lStatsAlloc);
  Vivarium::Handle               lVivarium   = new Vivarium(lDemeAlloc,
                                                            lStatsAlloc);
  ```

  For the other EC flavors the pattern is the same, only the types used must be changed. There is several other vivarium constructors, which can take directly, for example, alloca-

tors to genotypes and fitness values. You can use these instead of the previously exposed long pattern, if you only wanted to use a specific genotype of fitness type.

- **The operators and the evolver**
  As explained in section 3.4.3, the evolver applies iteratively, at each generation, the operators on each deme. The operators are packed in the evolver into the bootstrap and the main-loop sets. For usual application, the user can instantiate the default evolvers. If the user want to customize the evolutionary algorithm, he can build his exotic evolver, using the configuration file. This is the topic of section 4.4. In all case, once the operator sets is built, the whole EC system is initialized with a call to the evolver `initialize` method. The method takes as arguments an handle to the system and the command-line arguments. Once the EC system initialized, the evolution can then be launch with a call to the method `evolve` of the evolver, taking the vivarium to evolve as argument.

All the EC applications are built following the previously presented scheme. To get more details on how to develop an application for a specific EC flavor, take a look in the sections 2 and 4.6 for GP and 4.5 for GA. The section 4.4 gives also good insight on configuring a custom evolution.

4.3 Using the Register

The register is a central repository of all Open BEAGLE modifiable parameters. All these parameters must be registered when the EC system is initialized. The parameters are held in the register as a pair of tag and smart pointer. The tag is the unique identifier of a parameter associated to a smart pointer which is an indirection to the instance of the parameter.

4.3.1 Registering Parameters

When the EC system is initialized, the different parameters must be registered by the appropriate holding objects. As example, a probability of crossover is registered at the initialization time by a crossover operator. Usually, the parameters are registered when the `initialize` methods are called. The register is referred in the system object. The parameters are added in the register with a call to the method `addEntry`. The method take as arguments the tag of the parameter, the smart pointer the value of the parameter, and the description of the parameter. The following example add the parameters `ec.cx.indpb` with a reference to the floating point attribute `mMatingProba` of the crossover operator.

```
class CrossoverOp : public BreederOp {
public:
  CrossoverOp() { }
```

```
    virtual void initialize(System::Handle system) {
      BreederOp::initialize(ioSystem);
      if(ioSystem.getRegister().isRegistered("ec.cx.indpb")) {
        mMatingProba =
          castHandleT<Float>(ioSystem.getRegister().getEntry("ec.cx.indpb"));
      } else {
        mMatingProba = new Float(0.5);
        Register::Description lDescription(
          "Individual crossover probability",
          "Float",
          "0.5",
          "Single individual crossover probability for a generation."
        );
        ioSystem.getRegister().addEntry("ec.cx.indpb", mMatingProba, lDescription);
      }
    }
    ...
  protected:
    Float::Handle mMatingProba;
  };
```

In this example, a test is first done to check if the parameter is already registered. If it is registered, a reference is taken on it. If the parameter is not registered, the parameter is instantiated on on the heap and is then registered. When a parameter is registered, a short description must be given. This description is used by the usage of an application and to explain the meaning of the parameter. Note that the parameter referenced by the smart pointer attribute `mMatingProba` is a `Float` Open BEAGLE object instead of a C++ `float`. The type `Float` is a wrapper of `float`. All the parameters registered must be smart pointed, which mean that they must be Open BEAGLE objects.

4.3.2 Interaction with the Register

The application programmer can fetch parameters from the register with a call to the method `getEntry` (or the operator `operator[]`) of the class `Register`. This method takes as argument a tag and returns an handle to the parameter associated. The handle returned is a smart pointer of `Object`. The developer must upcast the handle referred into a exact derived type, if needed. If the named parameter is not registered, the method `getEntry` (or the operator `operator[]`) returns a null handle.

```
  void MyClass::MyMethod(System::Handle ioSystem) {
    Object::Handle lParameter = ioSystem->getRegister()["ec.cx.indpb"];
    if(lParameter == NULL) {
      std::cout << "No crossover probability registered" << std::endl;
    }
    else {
      try {
```

```
        Float::Handle lCastedParameter = castHandleT<Float>(lParameter);
        std::cout << "The crossover probability is "
                  << *lCastedParameter << std::endl;
      }
      catch(Beagle::BadCastException&) {
        std::cout << "Object registered is not of the Float type" << std::endl;
      }
    }
  }
```

The exception handling of the object casting is to intercept some abnormal cases, that is when the parameter fetched is not of the desired `Float` type.

4.3.3 Modifying Parameters

The parameters recorded in the register can be modified in three different ways: by directly modifying the value in the program, by passing specials arguments on the command-line and by using configuration files. The initialization sequence is the following:

1. Initialize the parameters to their default value (before the registration);
2. Read the default configuration file;
3. Parse the command-line, the parameters being modified in the order they are expressed (left to right);
4. Read the configuration files given on the command-line, if any. The files are read at the time the configuration file argument is parsed.

4.3.4 Modifying Parameters on the Command-Line

It is possible to modify the parameters registered with arguments on the command-line. When an evolver is initialized in the method `initialize`, it takes the command-line value (i.e the variables `argc` and `argv`) as method argument. This is done to parse the command-line and extract Open BEAGLE specific arguments. All the Open BEAGLE specific arguments start with the prefix `-OB`, followed by the tag, an equal sign and the values of the parameter. As example, if our executable is named `maxfct` and the crossover probability must be changed to 0.75, the command-line would be:

```
./maxfct -OBga.cx.indpb=0.75
```

If more than a parameter must be changed on the command-line, the user can specify numerous arguments starting with a `-OB`, or enumerate all of them following a single `-OB`, separated by comas:

```
./maxfct -OBga.cx.indpb=0.75,ec.term.maxgen=100
```

Note that all the command-line arguments starting with the `-OB` prefix are erased from the structure `argc/argv` once they are parsed.

There is four special parameters that can be given on the command-line. First, there is the special argument `usage`, that made the application to display the tag with a short description of the available parameters and exit normally without evolving.

```
./maxfct -OBusage
```

The second special argument, `help`, is very similar to the first one except that it display the tags with everything you want to know on the parameters and exit normally without evolving.

```
./maxfct -OBhelp
```

The third special argument, `ec.conf.file`, can be assigned to the configuration file to use to set the parameters of the evolution. This argument take as value a string that contains the name of the configuration file.

```
./maxfct -OBec.conf.file=maxfct.conf
```

And finally, the fourth special command-line argument, `ec.conf.dump`, can be used to generate configuration file with the default parameters value for a given application. This argument takes the name of the configuration file to generate as value.

```
~> maxfct -OBec.conf.dump=mymaxfct.conf
```

4.3.5 Modifying Parameters Using a Configuration File

Another interesting way to modify parameters is to use a configuration file. This file, formatted in XML, contains the new value of some or all of the parameters registered. The file format is quite straightforward in the XML style, the logical sequence being preserved. The following is a presentation of a short, but valid, Open BEAGLE XML configuration file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle version="2.1.1">
  <Register>
    <Entry key="ec.hof.demesize">0</Entry>
    <Entry key="ec.hof.vivasize">1</Entry>
    <Entry key="ec.mig.interval">1</Entry>
    <Entry key="ec.mig.size">5</Entry>
    <Entry key="ec.pop.size">100</Entry>
    <Entry key="ec.sel.tournsize">2</Entry>
    <Entry key="ec.term.maxgen">50</Entry>
```

```
    <Entry key="ga.cx1p.prob">0.3</Entry>
    <Entry key="ga.init.bitpb">0.5</Entry>
    <Entry key="ga.init.bitstrsize">125</Entry>
    <Entry key="ga.mutflip.bitpb">0.01</Entry>
    <Entry key="ga.mutflip.indpb">1</Entry>
    <Entry key="lg.console.enabled">1</Entry>
    <Entry key="lg.file.name">beagle.log</Entry>
    <Entry key="lg.log.level">3</Entry>
    <Entry key="lg.show.class">0</Entry>
    <Entry key="lg.show.level">0</Entry>
    <Entry key="lg.show.type">0</Entry>
    <Entry key="ms.write.interval">0</Entry>
    <Entry key="ms.write.over">1</Entry>
    <Entry key="ms.write.perdeme">0</Entry>
    <Entry key="ms.write.prefix">beagle</Entry>
  </Register>
</Beagle>
```

4.4 Customizing the Evolutionary Algorithm

The present section thread of important aspect of Open BEAGLE: how to customize an existing evolutionary algorithm (EA). This can be as simple as to modify a genetic operator of a standard EA to the more complex task of implementing a brand new EA. In any cases, this is an advanced topics that a newcomer can skip in the first time. This section presents the important concepts that is related to the customizing of the EA. It is advised to consult the reference manual for the implementation details.

4.4.1 Building a Custom Evolver

The evolver is a high level entity that apply the iterative evolution process on a population. An evolver is composed of two operator sets, the bootstrap and the main-loop. The bootstrap operators are applied once on the population, at the beginning of an evolution (at generation 0). The main-loop operators are then applied iteratively, until the termination state is reached. These iterations can be seen as generations in a generational evolution model or as certain amount of individual processing in the steady-state evolution model. In all case the basic algorithm is independent of the underlying evolution operations.

Evolvers can be dynamically configured directly by the configuration file. The set-up of the evolver is stated between two `Evolver` tag, with the list of the operators to used listed in the `BootStrapSet` and the `MainLoopSet` tags. The following XML snippet presents the default evolver of GA previously presented in Table 3.13, for the `maxfct` example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle version="2.1.0">
```

```
    <Evolver>
      <BootStrapSet>
        <IfThenElseOp parameter="ms.restart.file" value="">
          <PositiveOpSet>
            <GA-InitBitStrOp/>
            <MaxFctEvalOp/>
            <StatsCalcFitnessSimpleOp/>
          </PositiveOpSet>
          <NegativeOpSet>
            <MilestoneReadOp/>
          </NegativeOpSet>
        </IfThenElseOp>
        <TermMaxGenOp/>
        <MilestoneWriteOp/>
      </BootStrapSet>
      <MainLoopSet>
        <SelectTournamentOp/>
        <GA-CrossoverOnePointBitStrOp/>
        <GA-MutationFlipBitStrOp/>
        <MaxFctEvalOp/>
        <MigrationRandomRingOp/>
        <StatsCalcFitnessSimpleOp/>
        <TermMaxGenOp/>
        <MilestoneWriteOp/>
      </MainLoopSet>
    </Evolver>
  </Beagle>
```

When an user want to change the evolutionary algorithm by changing the order of operations, or using other standard operators, he only needs to modify the configuration file accordingly. If the user wants to use custom operators, he must first implement them, as explained at the following section, and then add an instance of them to the evolver, using the method `addOperator`, before initializing the evolver. Several typical configuration files are given with the examples to illustrate how these can be used to modify the EA used.

4.4.2 Custom Evolver with a Breeder Tree

The breeder model[2] allows the implementation of sophisticated EA. For example, the steady-state formulation of GA, you can use a specialized configuration file using the steady-state replacement strategy operator, with crossover, mutation, and selection operators structured under it as a breeder tree. The configuration file of a steady-state version of `maxfct` example, using a breeder tree would be the following.

```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <Beagle version="2.1.0">
    <Evolver>
```

[2] See section 3.4.4 to get a detailed explanation on the breeder model.

```
    <BootStrapSet>
      <IfThenElseOp parameter="ms.restart.file" value="">
        <PositiveOpSet>
          <GA-InitBitStrOp/>
          <MaxFctEvalOp/>
          <StatsCalcFitnessSimpleOp/>
        </PositiveOpSet>
        <NegativeOpSet>
          <MilestoneReadOp/>
        </NegativeOpSet>
      </IfThenElseOp>
      <TermMaxGenOp/>
      <MilestoneWriteOp/>
    </BootStrapSet>
    <MainLoopSet>
      <SteadyStateOp>
        <MaxFctEvalOp>
          <GA-CrossoverOnePointBitStrOp>
            <SelectTournamentOp/>
            <SelectTournamentOp/>
          </GA-CrossoverOnePointBitStrOp>
        </MaxFctEvalOp>
        <MaxFctEvalOp>
          <GA-MutationFlipBitStrOp>
            <SelectTournamentOp/>
          </GA-MutationFlipBitStrOp>
        </MaxFctEvalOp>
        <SelectTournamentOp/>
      </SteadyStateOp>
      <MigrationRandomRingOp/>
      <StatsCalcFitnessSimpleOp/>
      <TermMaxGenOp/>
      <MilestoneWriteOp/>
    </MainLoopSet>
  </Evolver>
</Beagle>
```

The important thing to understand is that the steady-state operator (`SteadyStateOp`) is a replacement strategy and that the XML tree under it (between the `<SteadyStateOp>` and `</SteadyStateOp>` tags)) is its breeding tree. The three sub-trees under it are called randomly following a uniform probability density function parametrized by their respective breeding probability. For the first breeding sub-tree to the steady-state replacement strategy, the breeding probability is given by parameter `ga.cx1p.indpb`, while for the second sub-tree the breeding probability is given by parameter `ga.mutflip.indpb`, and the last sub-tree breeding probability is given by parameter `ec.repro.prob`. The steady-state replacement strategy generates n individuals with its breeder tree at each generation, where n is the size of the population. Each newly generated individual replaces a randomly chosen individual existing in the actual population in a steady-state fashion. Once the configuration file set-up correctly, an evolution

using the given EA structure can be launched by given the good configuration file to use on the command-line. See previous section 4.3.4 for more details.

4.4.3 Defining New Operators

An operator is a specialized agent that apply an operation to an EC system. It is generally a genetic or utility operations such applying the crossover operation on a deme or calculating the fitness statistics for the actual generation. The operators are generally applied on one deme at the time for each generation, or sometime once a generation on all the population.

The definition of a new operator is not that complicated, usually the developer only needs to subclass the `Operator` class and over-define the method `operate`. The method received as argument a `Context` object. All the important contextual data are in the context object: smart pointers to the actual population, deme and system, the actual generation, etc. Furthermore, some EA use an extended context that haves some more specific informations in it. Several general pre-defined operators are already defined for most common operations such crossover, mutation, and selection. For these operators, the `operate` is usually already defined, and another operator-specific method need to be coded. This operator-specific method is usually the piece of coded very specific to the operation and representation used. Take a look into the code of existing operator to figure out the specificities of each case.

An usual operator also holds some evolution parameters, probabilities, and such. These parameters must be registered in the register of the evolutions system. This is done by redefining the `initialize` method by calling the appropriation registration method of the class `Register`. The method `initialize` is called once an evolution, before any the operator is applied to any population.

The user that define his own operators must be aware that the system is not fully set-up when the `initialize` method is called. For example, the random number generator must not be used, as the seed can be modified thereafter on the command-line or a configuration file. The rule is that the `initialize` method must be used only to add elements to the evolution system. If the user defines a new operator derived from an existing one, he must also call the `initialize` method from the super-class operator, as illustrated here.

```
class DerivedClassOp: public SuperClassOp {
  virtual void initialize(System& ioSystem) {
    SuperClassOp::initialize(ioSystem);
    // DerivedClassOp initialization code follow ...
  }
};
```

This is necessary as the super-class operator might do some specific operation such as registering its parameter.

The `postInit` operator method is the a companion method of `initialize`. The `postInit` method is called once for each operator, just after the whole system is initialized and the parameters are set. This method is best to do some parameters value checking and to setup operators internal structures. As with method `initialize`, if the `postInit` method is over-defined in a given operator, the parent's `postInit` must be called in the implementation before executing the actual operator post-initialization code.

There is a important hierarchy of standard defined operators in Open BEAGLE. The user is advised to interface his operators classes to already defined operator, using inheritance. This is in the spirit of OO programming and code reusing.

4.5 GA User Manual

NOTE: More explanations are necessary here on the integer-valued vector, real-valued vector, non-isotropic SA-ES and CMA-ES. Only the GA bit string representation is properly explained.

The Genetic Algorithm (GA) framework has been re-engineered in version 2.1.0 to extend its support over the basic binary representation to real-valued GA and simple Evolution Strategy (ES) representations[3]. The three standard representations of the GA framework share the namespace `GA` and a similar organization. Each representation has a genotype class that implement the linear structure which are derived from a `std::vector`. There is several crossover operators generic (1-point, 2-points, uniform) for vector-based representation implemented as class template, where the templated parameter must respect the STL *sequence* conceptual interface (the `std::vector` template respects this interface). A representation-specific type for each generic crossover operator is defined for each standard GA representation. There is also a representation-specific mutation operator defined for each standard representation, along with a per representation evolver. Every representation-specific evolver set the bootstrap and main-loop operators set with the default structure presented on Tables 3.13, 3.14 and 3.15.

To implement an application using a GA representation, we should follow the steps explained in section 4.2 by taking attention to specify the genotype used and the appropriate evolver. The following code snippet is extracted for the `onemax` example and illustrates this for a bit-string GA representation.

```
// 1. Build the system.
System::Handle lSystem = new System;
// 2. Build evaluation operator.
OneMaxEvalOp::Handle lEvalOp = new OneMaxEvalOp;
// 3. Instanciate the evolver.
```

[3]In fact, ES is not considered as a GA representation, but rather as a completely different EC flavor. But in Open BEAGLE, it is implemented as another EC linear representation in the existing GA framework. The accommodate the purists, we should in fact consider the GA framework a generic vector-based EC framework.

```
const unsigned int lNumberOfBitsPerGenotype = 50;
IntegerVector lEncoding(1, lNumberOfBitsPerGenotype);
GA::EvolverBitString::Handle lEvolver = new GA::EvolverBitString(lEvalOp, lEncoding);
// 4. Initialize the vivarium for bit string GA population.
GA::BitString::Alloc::Handle lBSAlloc = new GA::BitString::Alloc;
Vivarium::Handle lVivarium = new Vivarium(lBSAlloc);
// 5. Initialize the evolver and evolve the vivarium.
lEvolver->initialize(lSystem, argc, argv);
lEvolver->evolve(lVivarium);
```

Each of three GA standard evolvers have a constructor that take as argument an integer vector. The size of this vector states the number of genotypes in each individual, while the integer value of it specify the number of elements[4] of each linear genotype. So in the previous example, the integer vector passed to the evolver at step 3 configures the evolve to initialize the individuals with one bit-string of 50 random bits. The vivarium is initialized at step 4 with a bit string allocator to generate the population. For the two other representation, we would have used a `GA::FloatVector::Alloc` for real-valued GA or a `GA::ESVector::Alloc` for ES.

In the bit-string GA representation, the bits can be used as is (for example in the *OneMax* problem), or transformed into floating-point value in a given interval and a given precision, depending the number of bits used (for example in the function maximization problem). When the bits are converted to floating-point values, these are used as parameters to solve the problem, the fitness value is deduced from these parameters. The representation of a binary GA genotype is implemented in the class `GA::BitString`, which both inherits from the class `Genotype` and `std::vector<bool>`.

```
namespace GA {
class BitString : public Beagle::Genotype, public std::vector<bool> {
public:
  struct DecodingKey {
    double       mLowerBound;
    double       mUpperBound;
    unsigned int mEncoding;
  };
  void decode(const std::vector<DecodingKey>& inKeys,
              std::vector<double>& outVector) const;
  void decodeGray(const std::vector<DecodingKey>& inKeys,
                  std::vector<double>& outVector) const;
  ...
};
}
```

The method `decode` is used to decode binary-encoded bit strings, to translate it into a real-valued vector using a decoding key. The method `decodeGray` is similar to the `decode` method,

[4]That is the number of bits for binary GA, the number of floating-point values for real-valued GA, or the number of $(value, strategyparameter)$ pairs for ES vectors.

except that it supposes Gray-encoded bit strings. If the application needs to work directly on the bits, the interface of the class `std::vector<bool>` can be used.

4.6 GP User Manual

This section goes more deeply into the GP specific aspect of Open BEAGLE. This concern mainly the declaration of primitives by the user and the use of strongly typed GP.

4.6.1 Getting the Most of the Primitives

NOTE: The `Primitive` interface has slightly evolved since the writting of this section. New methods and changes to modified methods should be given here.

The declaration of the primitives are a central aspect of the implementation of GP in Open BEAGLE. An abstract class `Primitive` is declared with a standardized interface presented in Figure 4.1.

As explained in previous sections, the primitive represent the operation to be done by nodes of the GP trees. This operation is implemented the method `execute`, in classes derived from `GP::Primitive`. In this method, the user can get the value of the child subtrees to the node and apply the characteristic operation. The result must be returned in the datum given as argument. A primitive has a fixed number of arguments, the child subtrees, and a unique name. This attribute are generally given by the constructor of the derived subclass to the constructor of `GP::Primitive`, as in the following declaration.

```
class Add : public Beagle::GP::Primitive {
public:
  Add() : Beagle::GP::Primitive(2,"+") { }
};
```

The number of arguments and the name of the primitive can also be stated by a call to the methods `setNumberArguments` and `setName`.

The primitives are put into sets of usable primitives before creating any GP trees. In the process of trees creation, for every node, an associated primitive is chosen. Usually, the association is made by a call to the method `giveReference`, affecting the node's primitive smart pointer to the chosen primitives. But, for some special case, such when using ephemeral constants, the operation done must be different. In this case the method `giveReference` can be over-defined to desired behavior. By default, the method return an handle to the primitive instance, which is then put into the primitive smart pointer of the node. By over-defining the method, the user can return any other kind of primitive instance that will be put into the node's primitive smart pointer.

```
1  namespace GP {
2  class Primitive : public Object {
3
4  public:
5    typedef AllocatorT<Primitive,Object::Alloc> Alloc;
6    typedef PointerT<Primitive,Object::Handle>  Handle;
7    typedef ContainerT<Primitive,Object::Bag>   Bag;
8
9    explicit Primitive(unsigned int inNumberArguments=0,std::string inName="");
10   virtual ~Primitive();
11
12   virtual void execute(GP::Datum& outDatum,GP::Context& ioContext) =0;
13
14   virtual std::string  getArgType(unsigned int inN) const;
15   virtual std::string  getAttribute() const;
16           unsigned int getChildrenNodeIndex(unsigned int,GP::Context&) const;
17           std::string  getName() const;
18           unsigned int getNumberArguments() const;
19   virtual std::string  getReturnType() const;
20   virtual void         getValue(Object& outValue);
21   virtual Handle       giveReference(GP::Context& ioContext);
22   virtual bool         isEqual(const Object& inRightObj) const;
23   virtual void         initialize(GP::System& ioSystem);
24   virtual void         read(InputStream& ioIS);
25   virtual void         setAttribute(std::string inAttribute);
26   virtual void         setValue(const Object& inValue);
27   virtual bool         validate(GP::Context& ioContext) const;
28   virtual void         write(OutputStream& ioOS) const;
29
30 protected:
31    void setName(std::string inName);
32    void setNumberArguments(unsigned int inNumberArguments);
33    void getArgument(unsigned int inN,GP::Datum& outResult,GP::Context& ioContext);
34    void getArguments(GP::Datum outResults[],size_t inSizeTDatum,
35                      GP::Context& ioContext);
36    void get1stArgument(GP::Datum& outResult,GP::Context& ioContext);
37    void get2ndArgument(GP::Datum& outResult,GP::Context& ioContext);
38    void get3rdArgument(GP::Datum& outResult,GP::Context& ioContext);
39
40 private:
41   std::string  mName;             //!< Name of the primitive.
42   unsigned int mNumberArguments;  //!< Number of arguments of the primitive.
43 };
44 }
```

Figure 4.1: `GP::Primitive` class declaration

NOTE: Give explanations here on primitives with variable number of arguments and primitive with variable selection weight in primitive set.

NOTE: GP trees are no more written as Lisp-like string but rather as complete XML trees. The following paragraph should be completely rewritten.

The way the GP trees are inserted and extracted from a Open BEAGLE XML stream is standard. For each trees, the composing nodes are written into a Lisp-like string. By default, for each node, only the name of the primitive associated is written in the Lisp-like string. If appropriate when the primitive is written, some information can be added to the name of the primitive. This is called the attribute of the primitive. The attribute of the primitive is given by a call to the method `getAttribute` and the attribute can be changed by a call to the method `setAttribute`. When an attribute different from the null string is used, the value is put between braces just after the name of the primitive, in the Lisp-like string.

A primitive can have a value that is modified at the runtime by a call to the method `setValue`. A similar method exist in class `GP::Primitive`, taking a second argument that is the name of the primitive to modify the value. Actually, the method of `GP::EvaluationOp` only call the equivalent `setValue` of the variable primitive, with the `GP::Datum` given as argument. For the abstract class `GP::Primitive`, the method `setValue` does nothing. This method can be over-defined in the more specific subclasses to do what is needed to set the value of the primitive.

Three of the remaining primitive methods `getArgType`, `getReturnType`, and `validate` are related to the usage of strongly typed GP. This is the topic of the next section.

4.6.2 Strongly Typed GP

NOTE: Typing verification in STGP has been changed to compare references to `type_info` object instead of strings. The whole subsection on STGP should be updated accordingly.

Strongly typed GP (STGP) [22] is a standard Open BEAGLE GP feature. In every standard operators that modify the structure of the GP individuals, a dual operator supporting constraints is defined. These operator apply their operation while checking whether the typing of the nodes is respected when a GP genome is altered. Table 4.1 presents the list standard constrained GP operators actually implemented in the framework.

When an user want to take advantage of STGP, he first needs to use an evolver made of the constrained operators (instead of the simple unconstrained ones). The example `spambase` included in the distribution contains configuration files where evolver composed of operators that support constraints. Section 4.4.1 also explain how to change the set-up of evolvers.

The second point needed to do STGP is to set the typing in the primitives used. The typing is made by associating a char string to a type, generally the RTTI name of the datum type. To implement the typing of the individuals, two methods must be over-defined in the concrete

GP Operator	Constrained Dual Operator
GP::InitFullOp	GP::InitFullConstrainedOp
GP::InitGrowOp	GP::InitGrowConstrainedOp
GP::InitHalfOp	GP::InitHalfConstrainedOp
GP::CrossoverOp	GP::CrossoverConstrainedOp
GP::MutationSwapOp	GP::MutationSwapConstrainedOp
GP::MutationShrinkOp	GP::MutationShrinkConstrainedOp
GP::MutationStandardOp	GP::MutationStandardConstrainedOp
GP::MutationSwapSubtreeOp	GP::MutationSwapSubtreeConstrainedOp

Table 4.1: Open BEAGLE constrained operators

primitives classes, `getArgType` and `getReturnType`. The method `getReturnType` gives the type of the datum that is returned by the primitives. The method `getArgType` give the string type of the ith argument of the nodes associated to the primitive.

To illustrate this, suppose an application where two different datum types are used, `Double` and `Bool`. STGP is usually implemented by using the RTTI naming of the types used. Supposing a *if-then-else* primitive implemented in the class `IfThenElse`. The primitive received a Boolean as first argument and return a floating-point number, that is the second argument of the primitive if the Boolean is true and the third if not.

```
using namespace Beagle;
class IfThenElse : public GP::Primitive {
public:
  IfThenElse() : GP::Primitive(3,"if-then-else") { }
  virtual void execute(GP::Datum& outResult,
                       GP::Context& ioContext) {
    Bool lTest;
    get1stArguments(lTest,ioContext);
    if(lTest == true) get2ndtArgument(outResult,ioContext);
    else get3rdArgument(outResult,ioContext);
  }
  virtual std::string getArgType(unsigned int inN) const {
    if(inN == 0) return typeid(Bool).name();
    return typeid(Double).name();
  }
  virtual std::string getReturnType() const {
    return typeid(Double).name();
  }
};
```

The type of the root of a given tree is stated in the primitive set associated to the given GP tree. The constructor of the class `GP::PrimitiveSet` is defined in such way that you only need to pass the name of the type returned from the root of the tree when constructing the set.

This implementation of STGP is usually sophisticated enough to allow quite constrained

tree structures. But, the use of STGP does not allow all kind of structural constraints, and the user may want to apply to the GP individuals more elaborated constraints. The method `validate` of `GP::Primitive` is called to do the typing validation. As defined in `GP::Primitive`, this method implements STGP by checking the types returned by the methods `getArgType` and `getReturnType`. The user can over-define the method to do his custom structural checking. The method returns true if the structure imposed is respected, and false if not.

NOTE: More explanations should be given on general constraints that can be applied on GP trees. The use of constrained operator for dynamic ADFs should also be detailed.

4.7 Multiobjective EA User Manual

Multiobjective Evolutionary Algorithms (MOEA) are now supported in Open BEAGLE. To make use of it in your application, you must do the three following things:

1. Return a multiobjective fitness measure (class `FitnessMultiObj` or derived) in your evaluation operator;

2. Set the population allocators for the used multiobjective fitness measure allocators in the `main` routine;

3. Set-up a configuration file with the appropriate evolver structure, that is a multiobjective selection operator and a statistics calculation operator compatible with the fitness measure used.

The basic MOEA fitness measure is implemented in class `FitnessMultiObj`, which maximize all the objectives. A derived fitness measure is also defined in class `FitnessMultiObjMin` and consists to minimize the objectives. You should either use one of the two preceding standard MOEA fitness measure, or use a custom one derived from these as fitness measure. The class `FitnessMultiObj` is derived from a `std::vector<float>` and thus the user should use the STL vector interface to set the different objectives value.

Once the type of fitness measure used is stated in the evaluation operator, the population allocators must be set up to use it. This can be done by following the explanations given in Section 4.2, at the item population. This is necessary to clone/copy fitness value between individuals and to read a milestone from uninitialized population.

Finally, the evolutionary algorithm must be set-up for the use of MOEA. This first imply that a multiobjective selection operation must be used. There is actually two of these in Open BEAGLE, NSGA-II implemented as replacement strategy operator `NSGA2Op`, and NPGA 2 implemented as a more classical selection operator `NPGA2Op`. The following presents how the NSGA-II configuration file of the multiobjective bit string GA example `knapsack`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle version="2.1.0">
  <Evolver>
    <BootStrapSet>
      <IfThenElseOp parameter="ms.restart.file" value="">
        <PositiveOpSet>
          <GA-InitBitStrOp/>
          <KnapsackEvalOp/>
          <StatsCalcFitnessMultiObjOp/>
        </PositiveOpSet>
        <NegativeOpSet>
          <MilestoneReadOp/>
        </NegativeOpSet>
      </IfThenElseOp>
      <TermMaxGenOp/>
      <MilestoneWriteOp/>
    </BootStrapSet>
    <MainLoopSet>
      <NSGA2Op>
        <KnapsackEvalOp>
          <GA-CrossoverOnePointBitStrOp>
            <SelectRandomOp/>
            <SelectRandomOp/>
          </GA-CrossoverOnePointBitStrOp>
        </KnapsackEvalOp>
        <KnapsackEvalOp>
          <GA-MutationFlipBitStrOp>
            <SelectRandomOp/>
          </GA-MutationFlipBitStrOp>
        </KnapsackEvalOp>
      </NSGA2Op>
      <MigrationRandomRingOp/>
      <StatsCalcFitnessMultiObjOp/>
      <TermMaxGenOp/>
      <ParetoFrontCalculateOp/>
      <MilestoneWriteOp/>
    </MainLoopSet>
  </Evolver>
</Beagle>
```

The NSGA-II algorithm proceed first by generating a population of n children from a population of n parents. This is actually done here by calling the breeder tree of the `NSGA2Op` replacement strategy n times. The breeder tree generates the children one at the time by either applying one point crossover or bit mutation on randomly chosen individuals from the parent individuals, and then evaluating fitness of the newly generated individuals. When the n children are generated, the parents and children populations are merged and a non-dominated sort is done to keep the n best, which constitutes the new population. Note in the previous configuration file, the use of operator `StatsCalcFitnessMultiObjOp`, which is the appropriate statistics calculation operator for fitness measures `FitnessMultiObj` and `FitnessMultiObjMin`.

The NPGA 2 MOEA selection operator can be more simply used as a replacement of a usual mono-objective selection operators. The following configuration file shows the use of NPGA 2 in the `knapsack` MOEA example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle version="2.0.0">
  <Evolver>
    <BootStrapSet>
      <IfThenElseOp parameter="ms.restart.file" value="">
        <PositiveOpSet>
          <GA-InitBitStrOp/>
          <KnapsackEvalOp/>
          <StatsCalcFitnessMultiObjOp/>
        </PositiveOpSet>
        <NegativeOpSet>
          <MilestoneReadOp/>
        </NegativeOpSet>
      </IfThenElseOp>
      <TermMaxGenOp/>
      <MilestoneWriteOp/>
    </BootStrapSet>
    <MainLoopSet>
      <NPGA2Op/>
      <GA-CrossoverOnePointBitStrOp/>
      <GA-MutationFlipBitStrOp/>
      <KnapsackEvalOp/>
      <MigrationRandomRingOp/>
      <StatsCalcFitnessMultiObjOp/>
      <TermMaxGenOp/>
      <ParetoFrontCalculateOp/>
      <MilestoneWriteOp/>
    </MainLoopSet>
  </Evolver>
</Beagle>
```

This time the formulation of the evolver structure is more straight-forward, without the use of a breeder tree. Note that once again the multiobjective fitness measure statistics calculation operator `StatsCalcFitnessMultiObjOp` is used.

Attentive reader may note the use of an operator named `ParetoFrontCalculateOp` in the evolver's main-loop operator set. This operator is used to compute the Pareto front of the population into the hall-of-fame, just before exiting. This operator must be put in the main-loop operator set after the termination operator (in this case operator `TermMaxGenOp`), to intercept the moment that the evolution must stop, but before the milestone writing operator to get the Pareto front written into the milestone. It is not compulsory to use this operator in your evolver, although it may be very useful for result analysis.

NOTE: A section on co-evolution would be appreciated here.

NOTE: A detailed section on Open BEAGLE file format is necessary here.

Chapter 5

Conclusion

In this document was presented the main concept of the Open BEAGLE framework for evolutionary computations (EC). This framework has been designed following an Object Oriented (OO) methodology, built on strong foundations and abstract descriptions of general concepts. Open BEAGLE can do almost any evolutionary algorithms that fits into the following simple, general algorithm: an evolutionary process done by a series of operations applied iteratively on a population of individuals. The operations done can be anything and they can be apply on any kind of population entities.

Over this generic algorithm, a standard library have been defined to enable Open BEAGLE to do some classical EC flavor. With version 3.0 of the framework, Open BEAGLE support five standard EC representations: binary genetic algorithms, integer-valued vector genetic algorithms, real-valued vector genetic algorithms, evolution strategy, and genetic programming (GP). The four vector-based representations are relatively simple to implement on a framework such Open BEAGLE. On the other hand, GP is a quite more sophisticated EC paradigm and is provided in Open BEAGLE as a complex, fully-featured implementation. In our opinion, it is one of the most flexible and complete C++ implementation of tree-based GP available.

The Open BEAGLE characteristics and capabilities can be resumed in the following statements:

- *Open BEAGLE is a flexible, extensible EC framework.*
  As often stated in the actual document, the Open BEAGLE framework is built on a strong, sophisticated OO foundation. Many mechanism had been deployed to allow finely modularized interchangeable components. The design was inspired from some proved design patterns and existing generic OO frameworks. Every composing parts of the framework can be replaced by another, custom or defined in the standardized library, with little efforts.

- *Open BEAGLE is a programmer-friendly environment.*

Toward all these sophisticated mechanisms, a prospective Open BEAGLE user may think that the framework is usable only by C++ wizards. In fact, all the works to do by the user had been simplified by the usage of generic programming (i.e. templates) that encapsulated all the laborious programming tasks and letting the application programmer to implement only the essential, application-specific parts. The programming is usually very high-level and simple. Only some general guidelines, exposed in this document, needs to be followed.

- *Open BEAGLE is an open source and free software.*
  Open BEAGLE is an open source and free, as in *free speech*, framework that allow the user to do almost whatever they want with it. It is released under the *GNU Library (or Lesser) General Public License*, LGPL, which encourage sharing of the software. The limitations are common-sense: any individual or corporations cannot made any profit by selling it, except by charging a reasonable cost for the media, and it is impossible to release any closed-source version of the Open BEAGLE framework, or any derived products. This mean that the framework will stay forever free and open source.

- *Open BEAGLE is fast.*
  Although the Open BEAGLE framework is very flexible and user-friendly, the performance of the framework had never been neglected. During the implementation process, we look forward a quick, efficient implementation. Some optimization features had also been added to the framework and can be turned on once an application is known to be stable.

- *Open BEAGLE is a robust framework.*
  First, the framework had been developed using defensive programming principles, using deep checking to detect any irregular situations and signal it as necessary. Secondly, the framework intensively use the standards C++ exception mechanisms to handle erroneous situations and send to the user appropriate feedbacks. Third, Open BEAGLE had also been tested with specialized tools to ensure that there is no memory leaks, and successful evolutions, that were spanning some weeks of intensive computing, had been done without any crash. And finally, milestones can be written as often as necessary, up to one for each deme at each generation, from which recovery from interruptions of any source is possible, with a simple command-line argument.

- *Open BEAGLE is a system independent library.*
  Open BEAGLE had been designed using a standardized, modern C++ programming approach. It also only use standard, portable library and the core frameworks don't do system-dependent call. Is is a portable, "write once, compile anywhere" environment.

- *Open BEAGLE is well documented.*
  As you should realized, Open BEAGLE is supported by a quite big, comprehensive manual,

that you are actually reading, an extensive HTML reference manual that deeply document the implementation of the framework, and a maintained Web site.

Bibliography

[1] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol, UK, 2000.

[2] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, 1998.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - W3C recommendation 10-february-1998. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.

[4] Mary Campione and Kathy Walrath. *The Java Tutorial*. Addison-Wesley, Reading (MA), USA, 2 edition, 1998.

[5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA–II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[6] Mark Erickson, Alex Mayer, and Jeffrey Horn. The Niched Pareto Genetic Algorithm 2 Applied to the Design of Groundwater Remediation Systems. In Eckart Zitzler, Kalyanmoy Deb, Lothar Thiele, Carlos A. Coello Coello, and David Corne, editors, *First International Conference on Evolutionary Multi-Criterion Optimization*, pages 681–695. Springer-Verlag. Lecture Notes in Computer Science No. 1993, 2001.

[7] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit*. John Wiley Sons, 1998.

[8] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[9] Richard Forsyth. BEAGLE A Darwinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.

[10] Adam Fraser and Adil Qureshi. Genetic programming in C++. http://www-dept.cs.ucl.ac.uk/research/genprog.

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

[12] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, USA, 1999.

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

[14] Free Software Foundation Inc. What is free software? http://www.gnu.org/philosophy/free-sw.html.

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

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

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

[18] Tom Lenaerts and Bernard Manderick. Building a genetic programming framework: The added-value of design patterns. In *First European Workshop on Genetic Programming*, pages 196–208, 1998.

[19] Sean Luke. ECJ Evolutionary computation system. http://www.cs.umd.edu/projects/plus/ec/ecj.

[20] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8, 1998.

[21] J.J. Merelo, Maarten Keijzer, and Marc Schoenauer. EO Evolutionary computation framework. http://eodev.sourceforge.net.

[22] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

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

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

[25] 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.

[26] Dimitri van Heesch. Doxygen. http://www.doxygen.org.

[27] D. Zongker, B. Punch, and B. Rand. lil-gp 1.01 user's manual. ftp://garage.cps.msu.edu/pub/GA/lilgp/lilgp1.02.ps.

Index

ADF, 15, 50
allocator, 15, 18, 27, 53, 55
    container, 29
assertion, 54
automatically defined function, *see* ADF

Bag, *see* container
best-of-run individuals, *see* hall-of-fame
bit string, 65
    binary encoding, 65
    decoding, 65
    Gray encoding, 65
breeder, 40, 61
    breeding probability, 41, 62

cast operators, 53
command-line arguments, 15, 20, 58
configuration file, 15, 20, 56, 58–60, 68
container, 18, 28, 53
context, 17, 36, 55, 63

datum, 11, 15, 66
deme, 15, 36, 55

EP, *see* evolutionary programming
ephemeral random constant, 50
ES, *see* evolution strategy
evaluation operator, 12, 14, 42
    symbolic regression, 17
evolution strategy, 7, 41, 64
evolutionary programming, 7
evolver, 12, 14, 39, 56
    operator sets, 39, 44, 56, 60
exception, 31, 54
eXtensible Markup Language, *see* XML

fitness, 12, 36, 55
    Koza, 36, 48
    simple, 20, 36

GA, *see* genetic algorithms
generic EC framework, 34
generic foundation, 24
genetic algorithms, 6, 64
genetic programming, 7, 11, 66
genotype, 15, 36, 49, 55
GP, *see* genetic programming

hall-of-fame, 20, 36
Handle, *see* smart pointer

individual, 15, 36, 49, 55
initialization, 39, 56, 58, 63

logger, 19, 37, 55
logs, *see* logger

milestone, 20
MOEA
    seemultiobjective evolutionary algorithms, 43
multiobjective evolutionary algorithms, 43
multiobjective optimization, 41

naming convention, 22
NSGA-II, 41

object, 15, 24, 30, 52
    I/O, 30
operator, 38, 56, 63

parameter, 20, 56, 58, 63
Pareto front, 72
Pointer, *see* smart pointer
post-initialization, 17, 39
primitive, 11–13, 17, 49, 66
primitives set, 12, 13, 49
primitives super set, 49

random-number generator, *see* randomizer
randomizer, 38
reference counter, 25
register, 37, 55, 56, 63
replacement strategy, 40
    evolution strategy, *see* evolution strategy
    generational, 41
    NSGA-II, 41
    steady-state, *see* steady-state

smart pointer, 15, 18, 25, 28, 53
standard template library, *see* STL
statistics, 20, 34–36, 55
    computation operator, 36
steady-state, 21, 40, 41
STGP, 68, 69

STL, 22, 24, 28
strongly typed genetic programming, *see* STGP
symbolic regression, 12
system, 12, 13, 36, 38, 55

termination criterion, 42
TokenT, 17

vivarium, 12, 15, 36, 55

wrapper, 15, 16, 31

XML, 29, 37, 40, 59, 60