GUIDE System's Manual

Luis Da Costa and Marc Schoenauer

April 15, 2008

This document constitues is the System's Manual for GUIDE. An updated copy is kept on GUIDE's forge website, `http://gforge.inria.fr/projects/guide/`, under the *Documents* tab.

GUIDE is a software layer providing a common interface to different evolutionary libraries. The whole product comes accompanied with a Graphic User Interface (GUI) that extends the library's facilities to a wide range of users, expert practitioners of Evolutionary Computation (EC) or not. We think this is a very important point that deserves to be discussed and pinpointed, as we did in our latest communication[1] [?]. The center of our argument is presented in the following paragraphs.

One of the important research motivations in our research group (TAO, `tao.lri.fr`) is based on the idea of bringing the methods of our domain, Evolutionary Computation (EC), closer to general, *non-expert*, users. Indeed, despite impressive accomplishments, Evolutionary Algorithms (EAs) have still not been massively adopted as part of a general "*problem-solving toolkit*". We think that one of the main reasons for such "failure" is the lack of a clear, sensible unifying framework for the field. A proposal of a solution for this state of affairs gave birth to GUIDE.

There are (at least) two reasons for confusion among general users: first, until the publication of two books, one by Michalewicz [?] and the other by Eiben and Smith [?], it was very difficult to get a clear picture of the EC field, as there was not even a common terminology between papers in the domain. This terminology is very influenced by the history of the models that have been proposed over the years and, even though all models share a common structure, no existing software package allows the user to actually shift from one model to another [?].

The second problem is related to the specific tuning of this kind of algorithms. The description of a specific EA contains its components (the choice of representation, selection, recombination, and mutation operators) thereby setting a framework, while still leaving quite a few items undefined [?]. For instance, in a simple Genetic Algorithm (GA), while the representation is a simple bitstring with its associated operators, further details have to be given (population size and probability of application of these operators) before having a running version of the algorithm. These data, called the *algorithm parameters* or *strategy parameters*, greatly determine whether the algorithm will find an optimal or near-optimal solution, and whether it will find such a

[1] Latest GUIDE paper was sent on March 2008.

solution efficiently [?]. However, choosing such parameters is difficult, and a great deal of knowledge about EC is needed to do a sensible choice.

What would be needed for a "newcomer" to the field to use with ease an Evolutionary Algorithm? A user, advanced or not, is mainly concerned about the best possible way to solve their problem, so the historical differences that exist in the terminology will not matter at this stage [?]. Rather, as they are trying to solve a problem, they have the knowledge about two important points: first, they should be able to decide on an *encoding* for their problem. In other words, the problem to be solved needs to be adequately *represented*. Second, they have to somehow encode and define *what is a good solution for their problem*: this is what is called the definition of the *fitness function*. Other than that, they shouldn't be concerned with any of the details that we, practitioners in the field of EC, have to know about. GUIDE provides such a facility.

In this document you will find the description of GUIDE, from a designer's point of view. Writing this document was a great opportunity for us to revisit design choices we took for the implementation, so ideas and hints for improving them are added on the margins (like the one you see here), or as footnotes[2]. This document contains two big parts: one explaining the basis of GUIDE: it is what we called the *kernel*, and it is presented in Chapter . The GUI based on that kernel is in Chapter .

Hint-Hint!!

[2]This is a footnote - just in case you had forgotten.

Contents

1 How to use this guide **11**
1.1 Purpose and Audience 11
1.2 Scope 11
1.3 Suggested Reading Order 11
1.4 Notational Conventions 11
1.5 Associated Documents and Electronic Resources 12

2 The kernel of GUIDE **13**
2.1 Main classes description 15
2.1.1 Evolutionary Algorithm Class 15
2.1.2 Evolutionary Engine: EvolutionEngine class 17
2.1.3 Fitness Function: the FitnessFunction class 24
2.1.4 Representation of a problem in an EA 25
2.1.5 Type 26
2.1.6 Variable 30
2.1.7 GeneDefinition 31
2.1.8 Gene 33
2.1.9 Genome 34
2.2 Code Generation 35
2.3 A Complete Example 37

3 The GUI in GUIDE **39**

4 TODO's **41**
4.1 On this Document 41
4.2 On the implementation of GUIDE 41

List of Figures

2.1 GUIDE as a layer 14
2.2 Types, Genes, and others. 25
2.3 GUIDE: two ways of generating an EA 35
2.4 The engine and the controler idea 35

List of Tables

2.1 XML Structure of an EA 16
2.2 XML Structure of an Evolution Engine (EE) 17
2.3 XML Structure of a Fitness Function 24
2.4 Defining a Basic Type and its Translation 27
2.5 Parameters Definition Example 28
2.6 Example of the XML Representation for Type *enumeration* . . 29
2.7 Parameters Definition Example 30
2.8 Directory Structure for Types Plug and Play 31
2.9 Directory Structure for a Type 32
2.10 GeneDefinition XML Representation 32
2.11 Directory Structure for a Type 32
2.12 XML Structure of an EA 38

Chapter 1

How to use this guide

1.1 Purpose and Audience

This document is targeted to any user that wishes to understand the internal functioning of GUIDE.

1.2 Scope

1.3 Suggested Reading Order

This document is organized in a logical understanding order: we will explain what is our view of an evolutionary algorithm at the beginning of the document, and then proceed examining the different parts needed to present this view to an user. So the best way of reading this document is to follow the logical structure. However, the different parts been presented are also differentiated by Sections, contained in 2 main Chapters: chap. 2 explains the *kernel* of the application, and chap. 3 explains the Graphical User Interface associated with this kernel.

1.4 Notational Conventions

In general, the following conventions are used in this guide:

- **Java**. The term "Java" refers to any general version of the Java compiler. When functionality differs between compilers, the specific terms

will be used (for example, *Java 1.5* for Java v. 1.5).

- **C++**. The term "C++" refers to any general version of the C++ compiler.
- **Eclipse** refers to the working environment of the same name. IDEs for different languages are found here: `http://www.eclipse.org/home/categories/languages.php`

1.5 Associated Documents and Electronic Resources

GUIDE's Javadoc Documentation is in `http://www.lri.fr/~ldacosta/guideDoc/` and was developped in the TAO team (`http://tao.lri.fr`); now is it hosted at `http://gforge.inria.fr/projects/guide/`. There are 2 main papers, to this date, associated with it: the first was published in 2003, under the title *GUIDE: Unifying Evolutionary Engines through a Graphical User Interface*, written by Pierre Collet and Marc Schoenauer [**?**]. The second is currently being evaluated for publication in the Journal of Artificial Intelligence Tools, and is identified in the biliography as [**?**].

GUIDE generates code for two evolutionary libraries: EO and ECJ. EO's website is `http://eodev.sourceforge.net/` and a main communication describing its functioning was published by Keizer *et al.* and corresponds to reference [**?**]. ECJ's website is `http://www.cs.gmu.edu/~eclab/projects/ecj/`

GUIDE is actively used as product in the Evotest European Project `http://evotest.iti.upv.es/`

Chapter 2

The kernel of GUIDE

GUIDE implements an abstraction layer for the user (or for the application[1]) that wants to use a meta-heuristic as means of finding a solution (for a more detailed explanation, please see [?] and [?]). It uses a description language (see [?] for details) for choosing the basic types and associated initialization and variation operators, and fully implementing the different types of representation-independent operators.

GUIDE currently implements an interface towards 2 evolutionary libraries[2]: EO [?] and ECJ [?] (*Evolutionary Computation in Java*). As a good layer, the users of the layer above do not need to know about this!! See (Fig. 2.1)

[1]In this document we will use the term *GUIDE's user* to refer to an actual person, or to an application using GUIDE

[2]The reasons for such a choice are described in [?]

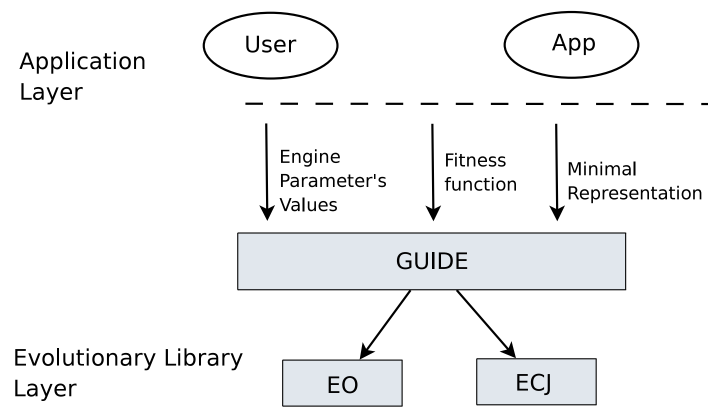

Figure 2.1: GUIDE as a layer

2.1 Main classes description

As an user is interested in developing an EA to solve its problem, the interaction presented in GUIDE will be with such an object. The description of the fonctioning of the system in this document will start by describing what is an EA.

For all class we five the Java *fully qualified name* and its JavaDoc documentation location, relative to a root. This root is called ROOTDOC, and currently it is on `http://www.lri.fr/~ldacosta/guideDoc`

2.1.1 Evolutionary Algorithm Class

- fully qualified name: `guide.kernel.EvolutionaryAlgorithm`
- JavaDoc Documentation: `guide.kernel.EvolutionaryAlgorithm`

JavaDoc wrong

This class contains the description of what en Evolutionary Algorithm (EA) is. As such, it has 3 private attributes describing the principal parts: an Evolutionary Engine (EE, described in 2.1.2), the Representation of the problem (described in 2.1.4), and the Fitness Function (described in 2.1.3). By the fact that it knows when the structure has changed, this class can offer advice as whether the code for this class should be generated, or is already up-to-date.

An EA is fully described by an XML Document, that has the structure shown in Table 2.1.

Each one of the 3 sections correspond to the three parts of an EA. They are going to be explianed in detail next. Evolutionary Engine (EE, described in 2.1.2), the Representation of the problem (described in 2.1.4), and the Fitness Function (described in 2.1.3).

```
<!--Instance saved on [2008/03/15 18:41:30]-->
<ALGORITHM>
  <EVOLUTION_ENGINE>
    .....evolutionary engine description.....
    <ELITISM>Strong</ELITISM>
  </EVOLUTION_ENGINE>
  <REPRESENTATION>
    .....representation description.....
    </CHROMOSOME>
  </REPRESENTATION>
  <EVAL_FUNCTION nbHelpLines="1">
    .....evaluation function description.....
  </EVAL_FUNCTION>
</ALGORITHM>
```

Table 2.1: XML Structure of an EA

2.1.2 Evolutionary Engine: EvolutionEngine class

- fully qualified name: `guide.kernel.EvolutionEngine`
- JavaDoc: `ROOTDOC/guide/kernel/EvolutionEngine.html`

Description The Evolution Engine (EE) of an EA contains the parameters of the engine shared by any kind of algorithm. They are described by an XML file (see Table 2.2)

```
<EVOLUTION_ENGINE>
  <P_MUT>0.1</P_MUT>
  <P_CROSS>0.85</P_CROSS>
  <MIN_MAX>Minimise</MIN_MAX>
  <NUM_GEN activated="true">500</NUM_GEN>
  <NUM_EVAL activated="false">1</NUM_EVAL>
  <MAX_TIME activated="false">1</MAX_TIME>
  <GRAPH_STAT>true</GRAPH_STAT>
  <ENGINE>SteadyState GA</ENGINE>
  <POP_SIZE>100</POP_SIZE>
  <ELITE TYPE="perc">15</ELITE>
  <FERTIL TYPE="perc">85</FERTIL>
  <NB_GEN TYPE="perc">100</NB_GEN>
  <GEN_SEL PARAM="2.0">Tournam.</GEN_SEL>
  <NB_OFFSPR TYPE="perc">100</NB_OFFSPR>
  <SURVIVING_OFFSPR TYPE="perc">100</SURVIVING_OFFSPR>
  <SURVIVING_PARENTS TYPE="perc">25</SURVIVING_PARENTS>
  <RED_PARENT PARAM="2.0">E.P. Trn.</RED_PARENT>
  <RED_OFFSPR PARAM="2.0">E.P. Trn.</RED_OFFSPR>
  <FINAL_RED PARAM="2.0">Sequent.</FINAL_RED>
  <ELITISM>Strong</ELITISM>
</EVOLUTION_ENGINE>
```

Table 2.2: XML Structure of an Evolution Engine (EE)

Each one of the tags of the XML file represents one parameter of the engine. These are all encapsulated in class `guide.kernel.EvolutionEngine`, who then provides methods to access and set the information. An explanation of the parameters is given here:

1. **General Parameters**:

   - Engine specification. GUIDE allows the choosing of a specific flavour of engine. The tag associated is **ENGINE**, and it can take one of these values:

     (a) **Generational GA**: this is probably the most popular GA algorithm. In this mode, all population is fertile, and is then potentially chosen for offspring generation. The new population (offspring) replaces the old population (parents).

     (b) **SteadyState GA**: on each generation, one offspring is generated and replaces one of the parents.

     (c) **Evolution Strategies, ES**: in this engine there is no crossover operator nor selection of the parents; instead, μ parents give birth to λ offspring. There is two flavours for this engine: either the (μ) individuals of the new population are only chosen from the (λ) offspring (in which case we talk about (μ, λ)-ES) or the (μ) individuals of the new population are only chosen from the sum of the old population with the offspring (so $\mu+\lambda$ individuals). In this case we talk about $(\mu + \lambda)$-ES.

     (d) **Evolutionary Programming**: this is basically $(\mu + \lambda)$-ES.

     (e) **Custom**: No restriction on parameters.

   - Problem is minimization or maximization: tag **MIN_MAX**. Value **Minimise** means that the goal is to minimize the fitness function. Value **Maximize** means that the goal is to maximize it.

   - Stopping criteria:

     – Num generations: tag NUM_GEN. Algorithm runs until the specified number of generations is reached. Only taken account if parameter *activated* of the tag is set to *"true"*

     – Num evaluations: tag NUM_EVAL. Algorithm runs until the specified number of evaluations is reached. Only taken account if parameter *activated* of the tag is set to *"true"*

    - Maximum time to run: tag MAX_TIME. Algorithm runs until the time specifed is reached. Only taken account if parameter *activated* of the tag is set to *"true"*
  - Feedback to the user: tag GRAPH_STAT. Gives graphical feedback to the user if the value is *true*

2. **Initial Population Parameters**:
   - Population size: tag POP_SIZE. Size of the initial population on each generation.
   - Elite value: tag ELITE. Specifies the number (tag attribute TYPE=*"abs"*) or percentage (tag attribute TYPE=*"perc"*) of the initial population that will intervene directly on the building of the final population.
   - Elite value: tag FERTIL. Specifies the number (tag attribute TYPE=*"abs"*) or percentage (tag attribute TYPE=*"perc"*) of the initial population that is eligible for giving birth to offspring.

3. **Intermediate Population Parameters**:
   - Choosing parents:
     - How many genitors to choose from the fertile: tag NB_GEN. Specified as a number (tag attribute TYPE=*"abs"*) or as apercentage (tag attribute TYPE=*"perc"*).
     - How to choose the genitors from the fertile: tag GEN_SEL. See below for details.
   - Creating offspring:
     - Mutation rate: tag P_MUT.
     - Crossover probability: tag P_CROSS.
     - How many offspring to choose from the genitors: tag NB_-OFFSPR. Specified as a number (tag attribute TYPE=*"abs"*) or as apercentage (tag attribute TYPE=*"perc"*).
   - Building intermediate population:
     - How many individuals to choose from the offspring to build intermediate population. Specified as a number (tag attribute TYPE=*"abs"*) or as apercentage (tag attribute TYPE=*"perc"*).

  – How to choose the individuals from the offspring: tag RED_-OFFSPR. See below for details.
  – How many individuals to choose from the fertile to build intermediate population: tag SURVIVING_PARENTS. Specified as a number (tag attribute TYPE=*"abs"*) or as apercentage (tag attribute TYPE=*"perc"*).
  – How to choose the individuals from the fertile: tag RED_-PARENT. See below for details.

4. **Final Population Parameters**:

   - Is the elite population going directly to the new population (in this case tag ELITISM = "Strong") or does it have to *fight* with the intermediate population (tag ELITISM = "Weak")?
   - How to choose the final population from the intermediate population: tag FINAL_RED. See below for details.

Selectors implemented in GUIDE A description of the Selectors implemented in GUIDE is given here. For a very interesting and detailed study of selection procedures, please see the work of Bäck [**?**]. For each Selection method we give the constant that represents it and its string descriptor (used in between the XML tags on the file description). All constants are defined in `guide.kernel.Util.EvolutionSelectionConstants`

1. *Roulette Wheel.* Represented in GUIDE by the constant `EvolutionSelectionConstar` `ROULETTE_WHEEL`. String descriptor: "*Roul. Wh.*"
   Also called *Proportional Selection*, this method was introduced by Holland for the Genetic Algorithm [**?**]. It assigns selection probabilites according to the relative fitness of individuals:

$$p_i = \frac{\Phi(a_i)}{\sum_{j=1}^{\lambda} \Phi(a_j)} \tag{2.1}$$

2. *(Linear) Ranking.* Represented in GUIDE by the constant `RANKING`. String descriptor: "*Ranking*"
   Defined by Baker [**?**], it uses a linear function to map indices i to selection probabilities p_i. Assuming that individuals are sorted according

to their fitness, the selection probabilities are given by

$$p_i = \frac{\eta^+ - (\eta^+ - \eta^-) * \frac{i-1}{\lambda-1}}{\lambda} \tag{2.2}$$

The couple (η^-, η^+) define the slope of the linear function. However, notice that the constraint $\sum_{j=1}^{\lambda} p_i = 1$ requires that $1 \leq \eta^+ \leq 2$ and $\eta^- = 2 - \eta^+$. So Eq. 2.2 can be rewritten as in Eq. 2.3.

$$p_i = \frac{\eta^+ - 2 * (\eta^+ - 1) * \frac{i-1}{\lambda-1}}{\lambda} \tag{2.3}$$

It is common to use a value of $\eta^+ = 1.1$. This is the value that GUIDE asks as a parameter of this Selector.

3. *Deterministic Tournament*: Represented in GUIDE by the constant `TOURNAMENT`. String descriptor: "*Tournam.*"
This selector chooses an individual by selecting a number q (≥ 2) of individuals at random and selecting the best one. q is usually 2, but can be configured in GUIDE.

4. *Stochastic Tournament*. Represented in GUIDE by the constant `STOCHASTIC_TOURNAMENT`. String descriptor: "*Stoch. Trn.*"
This selector behaves similarly of Deterministic Tournament. Stochastic Tournament of rate R first choses **two** individuals from the population, and selects the best one with probability R (the worse one with probability $1 - R$). Real parameter R should be in $[0.5, 1]$.

5. *Random*: Represented in GUIDE by the constant `UNIFORM`. String descriptor: "*Uniform*"
This selector is the *trivial* one: randomly selects the individuals.

6. *Sequential*: Represented in GUIDE by the constant `SEQUENTIAL`. String descriptor: "*Sequent.*"
It performs a deterministic selection, from best to wrost individuals, in turn.

Reductors implemented in GUIDE The way of reducing a population is implemented in GUIDE under the idea of "*Reductors*". Five of them are implemented: *Sequential*, *Random*, *Deterministic Tournament*, *Stochastic*

Tournament and *Evolution Programming Tournament*. The first 4 are identical to the ones with the same name described in the Selectors explanation. The fifth (*Evolution Programming Tournament*) is described here.

Represented in GUIDE by the constant `EP_TOURNAMENT` and with a String descriptor: "*E.P. Trn.*", it proceeds by taking the union of the initial population with the intermediate one. For each individual a in this (double) population q individuals are chosen at random and compared with a, with respect with fitness values. This gives a new value for a, the rank of this tournament. Once all individuals have been assigned values, a deterministic algorithm chooses the survivors. GUIDE expects the application to specify the value of q. A typical value is 10.

Default Parameters

TO FINISH. Those parameters are NOT GOOD

1. Population size = 100
2. GUIDE assumes the problem is of type Minimization (i.e., lower fitnesses are better)
3. the criterium stop is set at 500 generations
4. 10 per cent of the initial population will go down directly to the final population (so elitism=.1)
5. 85 percent of the initial pop is fertile (so the parents are chosen from there)
6. 33 percent of the fertile population will conform the parent population. The way of choosing is by Tournament, size 2
7. Mutation rate on parents: .1
8. Crossover probability: .85
9. Offspring population is 50 per cent as big in size as the parent population
10. There is an intermediate population that is formed by the sum of 50 per cent of the offspring population, chosen by Tournament, size 2, plus 75 per cent of the non-elite population, chosen by tournament, size 2

11. Finally, the Final population is chosen with the sum of the Elite (10 percent of the initial population), plus the rest coming from the intermediate population, chosen Sequentially.

Miscellanous Details about the Evolution Engine in EO can be found on `http://www.lri.fr/~marc/EO/eo/tutorial/html/eoEngine.html`

2.1.3 Fitness Function: the FitnessFunction class

JavaDoc wrong

- fully qualified name: `guide.kernel.FitnessFunction`
- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

An evaluation function provides the way of estimating the goodness of an individual in an EA. It is encapsulated by the class `guide.kernel.FitnessFunction`, and described by an XML file (see Table 2.3)

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Instance saved on [2008/03/15 18:41:30]-->
  <EVAL_FUNCTION nbHelpLines="1">
       return (theGenome.var5 ? 0.75 : 0.25);
  </EVAL_FUNCTION>
```

Table 2.3: XML Structure of a Fitness Function

2.1.4 Representation of a problem in an EA

- fully qualified name: `guide.kernel.Genome`
- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

JavaDoc wrong

In our view of an EA, the important structure to represent problems are Types (see 2.1.5). Those Types can be the base for an EA, but we also need to know how to combine them (as combination is at the base of an EA). For that, it is extended as GeneDefinition (see 2.1.7). The instantiation of a Type is a Variable (see 2.1.6), and the instantiation of a GeneDefinition is a Gene (see 2.1.8). The "top" Gene is called a Genome (see 2.1.9), who is then the actual representation of an EA. Fig. 2.2 shows what we mean.

Figure 2.2: Types, Genes, and others.

2.1.5 Type

- fully qualified name: `guide.kernel.Type`

JavaDoc wrong

- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

Being a Type one of the most important part of GUIDE, we will describe in quite some detail[3] the choices taken for its implementation.

Basic and Container Types

A Type can be either **Basic** or **Container**; it is "*basic*" when it represents a type present in the underlying programming language[4]. It is "*container*" if other types are needed to completely define it. In other words, if it *contains* other types.

- **Basic** If this type is basic, it has to provide what is the actual representation of the Type in the underlying language. That is achieved by an array with all Evolutionary Libraries for which code is generated, and, for each Library, the corresponding type (as a String) that it is. An example is given in Table 2.4
- **Container** Types.A *container* is a Type that can contain other types (the "*children*" types). We distinguish among Container types by the following features:
    1. A Container type can have fixed or dynamic size. For example, a *Set* is a container type that has dynamic size. A *Vector* has fixed size.
    2. The children Types can all belong to one specific Type, or they can be of different Types. For example, *Bag*s, *Vector*s and *Set*s are all container types where all elements belong to the same Type. A *Tuple* is a container types that can have different Types of elements.

[3]For basic description of the names of the methods and of the internal/external attributes, please consult the JavaDoc documentation.

[4]There is an implicit assumption here: *programming languages provide roughly the same types*. I am not so sure this is a fair assumption... Think, for example, of the distinction between Matlab and C++. Or Lisp and Java?

Let's take the instance of class `guide.kernel.Type` that represents a Floating-point precision numerical type is a Basic Numeric type. Its description in C++ is *double*, its description in Java is *float*. This is indicated in GUIDE by filling up the arrays *vEvolLibraries* and *vBasicTypeForEvolLibraries* such that it reads: reads, "*the Basic type defined by this class is translated, in library vEvolLibraries[i], by type vBasicTypeForEvolLibraries[i]*". We will then do vEvolLibraries = {EO, ECJ} vBasicTypeForEvolLibraries = {"double", "float"} EO and ECJ are defined as enumerate parts of `EvolLibsSupported`, in class `guide.kernel.Util`.

Table 2.4: Defining a Basic Type and its Translation

3. If a Container type has just one type of "children", they can repeated or not. For example, in a *Set* all elements have to be different. Not in a *Vector*.
4. If a Container type has just one type of "children", they can be ordered or not. For example, in a *Bag* the elements are ordered. Not in a *Vector*.

Parameters for Types

For completely defining a type we usually have to set some properties about it. For example, a *Vector* type needs a *size*; or we may be interested in assigning an interval of definition for an Integer (say $(-\infty, 34]$). This *parameters management* is done with the internally defined enumeration type called **BasicTypesForParams**[5]. The structure looks like this:

```
public static enum BasicTypesForParams {
INT, REAL, STRING, INTINTERVAL, REALINTERVAL
};
```

[5]Obviously, this sucks big time. Big TODO here. A parameter should be a Type, and then the actual value of that should be a Variable. The problem with that is that an Interval is NOT a Type... But this should be resolved, it is not THAT hard, I think...

So every time has a list of parameters, with names associated (mParamsNames), for each param, its type (mParamsTypes) and an explanation (mExplanations). An example is given in Table 2.5

Let's take the instance of class `guide.kernel.Type` that represents a Floating-point precision numerical type is a Basic Numeric type.It has a parameter of type Interval (BasicTypesForParams.REALINTERVAL). So mParamsNames[0] = "Interval"; mParamsTypes[0] = BasicTypesForParams.REALINTERVAL; and mExplanations[0] = "Interval of Definition"; The value taken by this interval is defined on the instanced Variable (see 2.1.6 and Example 2.7).

Table 2.5: Parameters Definition Example

Defining a Type

The main motivation for keeping all definition of a Type in an unique class (instead of doing a hierarchy of Types) is that now an XML representation can be defined for any Type we want to. It is quite straightforward to do so, and an example is given in Table 2.6.

This has one important consequence: any system that uses the class `guide.kernel.Type` for Type use can offer its users the possibility of *dynamically* defining Types: they only have to specify where the Type definitions are, and provide them. So *any* type can be used in this system. That is the philosophy behing Types management in GUIDE.

```
<?xml version="1.0" encoding="UTF-8"?>
<TYPE>
  <NAME>enum</NAME>
  <CLASSNAME>GuideEnum</CLASSNAME>
  <LONGNAME>enumerate</LONGNAME>
  <ISCONTAINER>false</ISCONTAINER>
  <ISBASICTYPE>true</ISBASICTYPE>
  <BASICTYPES>
    <EO>int</EO>
    <ECJ>int</ECJ>
  </BASICTYPES>
  <ISNUMERIC>true</ISNUMERIC>
  <ISLOGICAL>false</ISLOGICAL>
  <PARAMS>
    <PARAM type="intinterval">bounds</PARAM>
  </PARAMS>
  <EXPLANATIONS>
    <LINE>Enumerate : can take chosen values only</LINE>
    <LINE>Parameters: nominal values</LINE>
  </EXPLANATIONS>
</TYPE>
```

Table 2.6: Example of the XML Representation for Type *enumeration*

2.1.6 Variable

- fully qualified name: `guide.kernel.Variable`

JavaDoc wrong

- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

A object of type `guide.kernel.Variable` is an object whose definition is determined by an accompanying object of type instantiation of a `guide.kernel.Type`. It basically controls the specific values a Type has. For *Basic* types, just with this value it will be enough (for example, we may want to model a variable, type *int*, that takes a *value* of 3). But for *Container* types, we will also need to know who are our Children (Container Types have children, so they have to know what are the values of these children).

For anyone, we need to know who is my parent and what are the values of the parameters (defined in the Type. See 2.1.5).

(coming from Example 2.5) Let's take the instance of class `guide.kernel.Type` that represents a Floating-point precision numerical type is a Basic Numeric type.It has a parameter of type Interval (BasicTypesForParams.REALINTERVAL). The value taken by this interval is defined as a Value that can be $[-3.82, 34.1]$

Table 2.7: Parameters Definition Example

2.1.7 GeneDefinition

- fully qualified name: `guide.kernel.genome.GeneDefinition`
- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

JavaDoc wrong

A GeneDefinition is a `guide.kernel.Type` that has Operators for mutation and recombination. In other words, it is just a type that can be combined with others like him. That is why you basically see, in the JavaDoc documentation, methods to access those variation operators. WHERE exactly those operators are defined is given by a location where all the types are kept:TYPESLOC. the directory has a particular structure where the types' definitions are present, and also the variation operators that can be applied to them. The detail of that structure is on Table 2.8. We will examine it in detail next.

TYPESLOC/	
	operatorSchema.xml
	bag/
	bool/
	⋮
	all types here
	⋮

Table 2.8: Directory Structure for Types Plug and Play

As seen in Table 2.8, there is an important file in the directory: **operatorSchema.xml**, which contains the grammar definition of the operators for each gene definition. We will come back to the operators later. Additionally, the definition for a specific type (including its operators) are at TYPESLOC /name of the type. This definition is shown in Table 2.9 .

Ref. to operators Section

deftype.xml contains the definition for the Type that "defines" this GeneDefinition. Its structure follows the one presented at page 29, specifically on Table 2.6. **defgene.xml** defines specifically the GeneDefinition, and has exactly the same information as a Type, with an added reference to the definitions of Operators. See Table 2.10.

The structure of a GeneDefinition presented in Table 2.10 means that we can recover its definition, knowing from what Type it coms from, and where

<type-name>/
 defgene.xml
 deftype.xml
 cross/
 mut/
 init/

Table 2.9: Directory Structure for a Type

```
<?xml version="1.0" encoding="UTF-8"?>
<TYPEDEFINITION>
  <PLUGANDPLAYFOLDER>/tmp/typesPlugAndPlay/enum</PLUGANDPLAYFOLDER>
  <TYPE>
     ..... type definition here....
  </TYPE>
</TYPEDEFINITION>
```

Table 2.10: GeneDefinition XML Representation

are the definitions for its Operators. This is used in GUIDE to recuperate information that may have been damaged or lost.

Each type of variation operators (crossover, mutation and initialization) are represented by a directory with actual definitions (*i.e.*, code). There can be several operators to be applied for each type (*i.e.*, several crossover operators for an *int*, for example). The definition are language-dependent, so they are separated by evolutionary libraries, with the actual code for it. See Table 2.11

cross/
 EO/
 crossDef1.xml
 ⋮
 ECJ/
 file 1.xml
 ⋮

Table 2.11: Directory Structure for a Type

2.1.8 Gene

- fully qualified name: `guide.kernel.genome.Gene`
- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

JavaDoc wrong

In the same way that a `guide.kernel.Variable` is the encapsulation of a value for `guide.kernel.Type`, `guide.kernel.genome.Gene` is the encapsulation of `guide.kernel.genome.GeneDefinition`. It manages the specific Operators that apply on this Gene.

2.1.9 Genome

- fully qualified name: `guide.kernel.genome.Genome`

JavaDoc wrong

- JavaDoc: `guide.kernel.EvolutionaryAlgorithm`

The Genome is the representation of the problem. It is, in fact, just a Gene defined as a type Tuple - in that way, it accepts any combination of Types and Genes that the user may want.

2.2 Code Generation

The idea GUIDE can be used to generate an EA to be run by itself ("*standalone*") or it can also be used to produce an *evolution engine* to solve a problem as part of another application (see Fig. 2.3). As such, we need a centralized code generation scheme offering this flexibility.

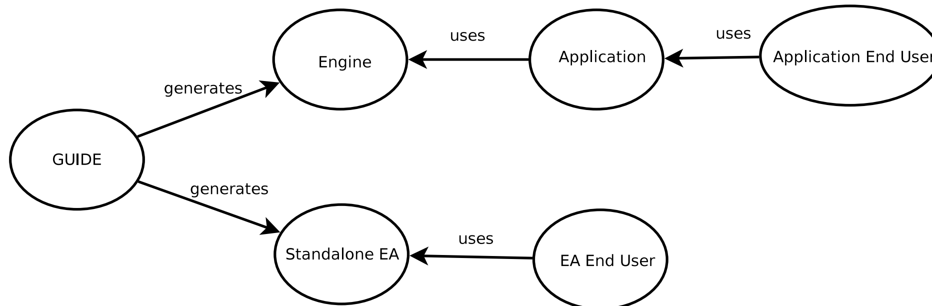

Figure 2.3: GUIDE: two ways of generating an EA

The more straightforward way of reaching this goal is by considering a standalone EA as a special type of engine: one that is run by a special application, *i.e.*, the end-user itself. GUIDE always generates an *engine*, that needs to be "controled" (*i.e.*, started, stopped, paused/resumed or given feedback) by another program, the *controler* (see Fig. 2.4).

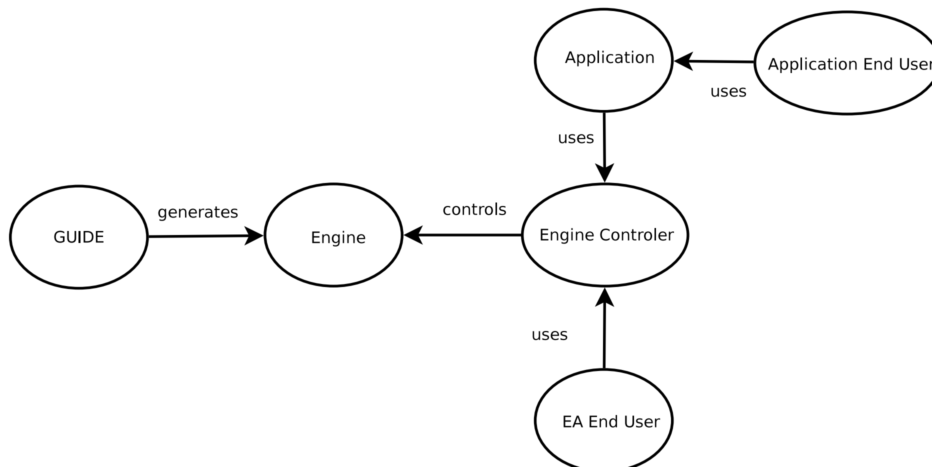

Figure 2.4: The engine and the controler idea

Under this scheme, the engine provides 2 things: (1) all the logic and important code to execute the generated EA, and (2) a way of creating itself. The controler has to create the engine, and to start it. On demand, it provides the fitness for the individuals (equivalently, population) and sends messages to the engine concerning the continuation or pausing of the execution. The important point here is that the engine gives total flexibility to the controler to perform those tasks depending on the logic of the application. A summarizing scheme of this idea is shown in Fig. 2.4

Practical implementation details. Implementing this idea is achieved by the following basic steps:

1. The code representing an engine is located on a directory called **src**, directly under the directory of the experiment. The directory of the experiment himslef will be where the specification file for the generation of the algorithm is located. For example, if the specification for the current experiment is at

   `/myDir/myExperiment.xml`

   then the code corresponding to the EA will be located at the following directory:

   `/myDir/myExperiment/`

   and the code corresponding to the generated EA will be at:

   `/myDir/myExperiment/src/`

2. GUIDE's distribution provides two interfaces ("header files"), called **IEOEngine.h IEOEngineControl.h**. These two files are provided in the distribution of GUIDE, and must exist in a directory specified in the settings file, under the tag DIR on INCLUDES[6]

3. The controler (see Fig. 2.4 for a quick reminder of what is a Controler):

[6] If this last sentence is confusing to you, or you would like more details, please refer to the Installation Manual, found in `http://gforge.inria.fr/projects/guide/`, Tab Docs, and here `http://www.lri.fr/~ldacosta/guide/installation/GUIDEInstallationManual.html`

(a) A general, abstract implementation of a Controler is generated by GUIDE at code-generation time: it is on the file **GUIDEGenericEngineControl.h**, on the source directory of the EA.

(b) The user (equivalently, application) must provide a specific implementation of a Controler. An example of a Controler is provided with GUIDE's distribution: an user can modify it to make it suitable for their needs. It must be kept in the directory specified in the settings file, under the tag CONTROLERDIR (see footnote 6)

4. Any other classes needed by the engine should be specified in the directory of the settings file identified by the tag DIR on EXTRACODE (see footnote 6)

An example Here I put the example that is in GUIDEKernel (for Evotest)

GUIDE provides pre-defined types

The *representation* of an EA

The *representation* of an EA is encapsulated in class `guide.kernel.Genome`. A Genome is, in fact, a special class of Gene; the important point is that a Genome is a Gene that contains other Genes, and that do not have a parent. Let's begin by explaining what a Gene is: we will need for that the class Type!!

described by an XML file (see Table 2.12)

2.3 A Complete Example

```
<?xml version="1.0" encoding="UTF-8"?>
  <REPRESENTATION>
    <INITIALISATOR id="init_root_default" name="default" type="root" userde
    <INITIALISATOR id="init_bool_false" name="false" type="bool" userdefine
    <MUTATOR id="mut_root_default" name="default" type="root" userdefined="
    <MUTATOR id="mut_bool_flip" name="flip" type="bool" userdefined="no" va
    <CROSSOVER id="cross_root_default" name="default" type="root" userdefin
    <CROSSOVER id="cross_bool_exchange" name="exchange" type="bool" userdef
    <CHROMOSOME name="theGenome" type="root">
      <CHROMO_INITIALISATOR id="init_root_default" weight="1.0" />
      <CHROMO_MUTATOR id="mut_root_default" weight="1.0" />
      <CHROMO_CROSSOVER id="cross_root_default" weight="1.0" />
      <CHROMOSOME name="var5" type="bool">
        <CHROMO_INITIALISATOR id="init_bool_false" weight="1.0" />
        <CHROMO_MUTATOR id="mut_bool_flip" weight="1.0" />
        <CHROMO_CROSSOVER id="cross_bool_exchange" weight="1.0" />
      </CHROMOSOME>
    </CHROMOSOME>
  </REPRESENTATION>
```

Table 2.12: XML Structure of an EA

Chapter 3

The GUI in GUIDE

Chapter 4

TODO's

4.1 On this Document

- All examples given here should be in an environment *example*, not in Tables!!! (see, for example, Table 2.4 page 27).
- Give the **exact** reference where the Type definition was used to recuperate the GeneDefinition (see page 32).

4.2 On the implementation of GUIDE

- Parameters management for types (mentioned in page 27).
- Multi-Objective Optimization

Acknowledgments

To the people that worked before?? Pierre Collet for GUIDE.

Funding information may also be included here.