EASEA : un langage de spécification pour les algorithmes évolutionnaires

Pierre Collet, Marc Schoenauer, Evelyne Lutton, Jean Louchet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

EASEA : un langage de spécification pour les algorithmes évolutionnaires

Pierre COLLET — Marc SCHOENAUER — Evelyne LUTTON — Jean LOUCHET

N° 4218

June 2001

THÈME 4

ISSN 0249-6399

INRIA
ROCQUENCOURT

EASEA : un langage de spécification pour les algorithmes évolutionnaires

Pierre COLLET* , Marc SCHOENAUER† , Evelyne LUTTON‡ , Jean LOUCHET§

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Fractales

Rapport de recherche n° 4218 — June 2001 — 14 pages

Résumé : Contrairement aux apparences, il n'est pas simple d'écrire un programme informatique réalisant un algorithme évolutionnaire, d'autant que le manque de langage spécialisé oblige l'utilisateur à utiliser C, C++ ou JAVA. La plupart des algorithmes évolutionnaires, cependant, possèdent une structure commune, et la part réellement spécifique est constituée par une faible portion du code. Ainsi, il semble que rien ne s'oppose en théorie à ce qu'un utilisateur puisse construire, puis faire tourner son algorithme évolutionnaire à partir d'une interface graphique, afin de limiter son effort de programmation à la fonction à optimiser. L'écriture d'une telle interface graphique pose tout d'abord le problème de sauver et de recharger l'algorithme évolutionnaire sur lequel l'utilisateur travaille, puis celui de transformer ces informations en code compilable. Cela ressemble fort à un language de spécification et son compilateur. Le logiciel EASEA a été créé dans ce but, et à notre connaissance, il est actuellement le seul et unique compilateur de langage spécifique aux algorithmes évolutionnaires. Ce rapport décrit comment EASEA a été construit et quels sont les problèmes qui restent à résoudre pour achever son implantation informatique.

Mots-clés : algorithmes évolutionnaires, algorithmes génétiques, language de spécification

* CMAPX, École Polytechnique, 91128 Palaiseau cedex, France, Pierre.Collet@polytechnique.fr
† CMAPX, École Polytechnique, 91128 Palaiseau cedex, France, Marc.Schoenauer@polytechnique.fr
‡ Projet FRACTALES, INRIA, B.P. 105, 78153 Le Chesnay cedex, France, Evelyne.Lutton@inria.fr
§ ENSTA, 35 Boulevard Victor, 75011 PARIS, France, Louchet@ensta.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Specifying Evolutionary Algorithms with EASEA

Abstract: Evolutionary algorithms are not straightforward to implement and the lack of any specialised language forces users to write their algorithms in C, C++ or JAVA. However, most evolutionary algorithms follow a similar design, and the only really specific part is the code representing the problem to be solved. Therefore, it seems that nothing, in theory, could prevent a user from being able to design and run his evolutionary algorithm from a Graphic User Interface, without any other programming effort than the function to be optimised. Writing such a GUI rapidly poses the problem of saving and reloading the evolutionary algorithm on which the user is working, and translating the information into compilable code. This very much sounds like a specifying language and its compiler. The EASEA software was created on this purpose, and to our knowledge, it is the first and only usable compiler of a language specific to evolutionary algorithms. This reprot describes how EASEA has been designed and the problems which needed to be solved to achieve its implementation.

Key-words: evolutionary algorithms, genetic algorithms, specification language

1 Introduction

Not so long ago, evolutionary algorithms were considered as mere curiosities. Things have changed over the years and many end-users (chemists, physicists, applied mathematicians, aircraft designers, electrical engineers,...) have ended up selling their scientific souls to DARWIN. Unfortunately, taking this decision is not the hardest part of their ordeal: the evolutionary algorithm they have been dreaming of remains to be written and their specialty is not always computer science.

One way to speed up the process is to use one of the many existing evolutionary libraries which offer very powerful tools provided ... one is fluent enough with constructors, copy-constructors, destructors and such niceties involved by relatively low-level object languages.

The next hurdle is then to learn how to use the library, to understand the intricate data structures and to memorise the necessary several hundred object types, functions and variables and the way they are inter-related. This can be quite time consuming when all major evolutionary libraries are written in C++ or JAVA and make full use of object programming.

All in all, many physicists, chemists, mathematicians and other scientists who otherwise would be capable of writing extremely complex FORTRAN functions are denied experimentation of evolutionary algorithms due to the sheer complexity of their implementation.

The aim of EASEA (EAsy Specification of Evolutionary Algorithms) is to hide this complexity behind a relatively simple high-level language, allowing scientists to concentrate on evolutionary algorithms, rather than on their implementation.

Some research teams have already felt the need for a specific evolutionary language. They have however chosen a theoretical viewpoint, trying to enrich the evolutionary paradigm with new concepts or features not yet implemented [7, 9, 11, 12]. We have chosen a radically different approach, trying to be as pragmatic as possible. Our goal was to start with the realisation of a minimal working prototype, able to implement almost any optimization problem.

2 Presentation of evolutionary algorithms

Basic principles of Evolutionary Algorithms (EAs)[1] model some biological phenomena, and more precisely the ability of populations of living organisms to adapt to their environment, via genetic inheritance and Darwinian strife for survival. Resolution methods and stochastic optimisation methods have been designed according to these —of course, extremely simplified— biologically-inspired principles. The main characteristic of EAs is that they manipulate *populations* of points of the search space, and involve a set of operations applied (stochastically) to each *individual* of the population, organised in *generations* of the artificial evolution process. Operations involved are of two types : *selection*, based on the individuals' performance w.r.t. the problem being solved and *variation operators*, usually *crossover* and

1. The best known evolutionary algorithms are genetic algorithms; very often the terms *evolutionary computation* methods and *GA-based* methods are used interchangeably. Whether it is because of their fashionable name and concepts, the question is beyond the scope of this paper.

mutation, that produce new individuals. If correctly designed, a dynamic stochastic search process is started on the search space that converges to the global optimum of the function to be optimised. [2]

>From the point of view of optimisation, EAs are powerful stochastic zeroth order methods (i.e., requiring only values of the function to optimise) that can find the global optimum of very rough functions. This allows EAs to tackle optimisation problems for which standard methods (e.g., gradient-based algorithms requiring the existence and computation of derivatives) are not applicable.

Despite the apparent simplicity of an EA process —which has driven many programmers to first write their own EA adapted to their specific problem— building an efficient EA for an application is a rather tricky task. In fact, EAs are very sensitive to parameter settings and design choices. The current trend in EA applications is to use available toolboxes, containing a variety of operators and strategies, in order to easily test different combinations.

In the following paragraph, we succinctly describe the basic ingredients of a "canonical" evolutionary algorithm. This structure is only a framework, and an efficient EA application to a specific problem may be more complex.

Solving an optimization problem with Evolutionary Algorithms starts by choosing a *representation* for the problem at hand. This representation will be coded into a data structure called *genome*.

The EA will then evolve a *population* of *individuals*, i.e. a set of points of the search space.

The first step is the **initialization** of that populations, using a random generator of individuals. That initial population is then undergoes **evaluation**: the value of the *fitness* of each individual, i.e. the function to be optimized, is computed. The algorithm then enters the **generation loop**, the current population usually being called the *parents*:

1. **Selection:** This step selects which parents are going to reproduce. This operation implements artificial Darwinism, as it is based on the *fitness* of all individual, favoring the ones with better fitness values (with respect to the problem at hand).

2. **Variation:** Copies of the selected parents then undergo *variation operators*, that is are stochatically moved within the search space. In the simplest case, a *crossover* operator is first applied to pairs of individuals, with a given probability. A *mutation* operator is then applied to all result of crossover (or to the copies of the initial selected parents that have escaped crossover), giving a population of *offspring*.

3. **Evaluation:** The offspring population is evaluated by the fitness function.

4. **Replacement:** This step is used to create the population of parents for the future generation by selecting from the pool of offspring + parents which of them will survive (the others are discarded and disappear). The choice is another implementation of artificial Darwinism, in that it is biased toward hte individuals with better fitness.

2. A large part of EA theoretical research addresses this convergence problem, as well as understanding of the notion of EA-difficulty.

5. **Stopping criterion:** At some point, the algorithm stops, depending on user-defined criteria (total number of generations, fitness threshold on the best individual,...)

3 Design of the Graphic User Interface

Ideally, users should be able to use the Graphic User Interface to describe the structure of the genome they need, build some variation operators instanciating generic operators to their genome, choose the Darwinian operators (selection, replacement, ...) they would like to use on their population. Next, they need to somehow write the code for their specific fitness function. Finally, they should be able to set different parameters of the evolutionary algorithm (population size, offspring population size, crossover probability, mutation probability, ...), after which pressing the RUN button should magically run the evolutionary algorithm, and graphically display temporary results on their workstation.
There are three main issues in the above skeleton of an EA:

- the **evolution engine**, which contains all operations related to artificial Darwinism (i.e. the selection and the replacement steps). The evolution engine is conceptually independent of the genome.
- the **representation-dependent** part, namely the type of genome that represent potential solutions to the problem at hand, together with an initializer function, and corresponding variation operators.
- the **problem-dependent** part, that is the fitness function itself.

Ideally, users should be able to use the Graphic User Interface to describe both the representation-dependent part of their application, and the evolution engine they wish to use. The fitness function somehow has to be programmed.

3.1 The evolution engine

Its description is rather straightforward: it involves the choice of some high level parameters (e.g. the type of selection and replacement procedures together with their possible parameters, the number of offspring that will be created, how many of the best parent should be carried over to the next generation and so forth.

3.2 Genome and operators

The genome-related part of the algorithm is more difficult to describe in general. However, it is possible to offer to the user a number of constructs that cover most of the usual cases, leaving the text-window for the remaining cases.

The EASEA approach considers that a genome is built from a set of basic data types, namely boolean, symbolic values, integers and real numbers, and a set of constructors, namely aggregates (fixed length ordered lists of heterogeneous data), arrays (lists of homogeneous,

ordered or not, of fixed or variable length) , trees, graphs, ... These constructs can be iterated, using user-defined data types to build higher level genomes.

But such constructs would be almost useless if there was not the possibility to use **generic operators** without any programmation burden. Generic operators are based on well-known variation operators for the basic data types (e.g. arithmetic crossover and Gaussian mutation for real numbers) and hierarchical construction of operators for the higher level types.

For instance, when crossing over two fixed-length ordered arrays, one can either exchange some of the variables between both parents, or call a specific crossover for each corresponding pair of variables from both parents. Or a mutation of an aggregate can call in turn some specific mutation for each one of its components. Such definitions can be entered from the GUI, while still allowing the user to directly enter some code in a text window.

4 Presentation of EASEA

4.1 Introduction

In order to avoid creating an unusable superb piece of software, we instead decided to start with creating the specification language and its compiler and only create the GUI as a second step.

Several important specifications lie behind the EASEA language and compiler :

1. EASEA must be general/generic enough to be able to write virtually any evolutionary algorithm.
2. It is not the aim of EASEA to create yet another evolutionary library. Many already exist and work quite well. It was therefore decided that EASEA should *use* existing libraries, to avoid reinventing the wheel.
3. A generic language such as EASEA should not be tied to a specific evolutionary library but should be able to operate different evolutionary libraries.
4. EASEA should try to hide away all programming mechanisms not explicitly needed to describe the evolutionary algorithm and the problem to solve, and especially the complexity of object-oriented design.
5. EASEA source files must be simple enough to be written automatically using a graphic user interface.

4.2 Mode of operation

Specification 3 says that the EASEA compiler should be able to produce source code using different evolutionary libraries. This specification somewhat guarantees that the resulting language will comply with specification 1, that is: the aim of the EASEA language is to specify evolutionary algorithms in general, and not to drive a specific library.

Therefore, two independent libraries were chosen to start with: GAlib —a widely used C++ genetic library [5]— and EO (Evolving Objects [3][3]).

In parallel, EASEA will be used in the DREAM European project (Distributed Resource Evolutionary Algorithm Machine) [6], which works in a JAVA environment.

When the EASEA project started, EO was not stable enough to reasonably use it as a primary library, so GALib was selected. The first EASEA operational prototype (v0.1) was released in September 1999.

In January 2001 was released EASEA MILLENNIUM EDITION (v0.6) driving for the first time both GALib and EO.

The EASEA–DREAM compiler is still under development, along with the DREAM European project.

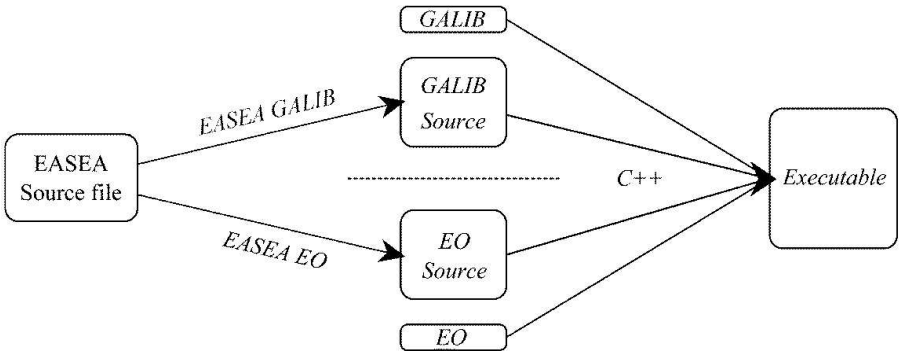

FIG. 1 – *EASEA mode of operation.*

Until the GUI is available, the EASEA compiler uses as input an ascii file with a `.ez` suffix. Its output is either a C++ source file driving the GAlib object-oriented library or a C++ source file driving the EO object-oriented library. The resulting C++ file must then be compiled by a C++ compiler and linked with the corresponding library (cf. figure 1). The produced executable implements the evolutionary algorithm described in the original EASEA source file.

4.3 Graphic User Interface

The GUI (under development) allows the user to graphically specify his evolutionary algorithm. Once this is done, the GUI temporarily saves the information input by the end-user into an EASEA source file, which is in turn automatically compiled into an object source file using the specified library. The object source file (in which an interface with the GUI has been inserted) is compiled to produce an executable file which is launched by the GUI. The

3. An open source project, originally developed at the University of Granada (Spain), and now continued at SourceForge.

GUI can then be used to control the running evolutionary algorithm and to display partial and final results thanks to the interface module that was embedded in the user application.

5 EASEA compiler

5.1 Description

EASEA is written in C++, using Lex and Yacc (in fact their excellent Windows C++ equivalent : ALex and AYacc [4]). The EASEA compiler is somewhat unusual in the sense that it produces source code in another language rather than microprocessor instructions.

As EASEA counts on the user to write the code of his evaluation function, a simple cut and paste strategy could have worked, taking pieces of code here and there and putting them together for compilation.

However, it was decided (specification 4) that EASEA would allow non state-of-the-art programmers to nevertheless specify their evolutionary algorithms. A cut and paste strategy would have required the user to know about object programming (as the underlying libraries are object-oriented) and would have required the user to know how to use the library. To him, the only advantage would have been the help of the Graphic User Interface, compared to his already existing C++ programming environment.

It was therefore decided to keep a pure C++ syntax for user-written functions, but hiding away all object-oriented concepts, which can be automatically inserted, provided that the software has a deep enough understanding of the code written by the user.

A real compiler was then needed, to create fully-fledged classes out of the data structures described by the user.

5.2 Source code analysis

Apart from sections containing pure C++, the syntax of the EASEA language must be simple enough to be generated and interpreted by the Graphic User Interface.

An EASEA source file is therefore composed of different areas, delimited by sections such as:

```
\User declarations:
  #define SIZE 10
  inline void swap(bool& a,bool& b) {bool c=a;a=b;b=c;}
\end
```

which can contain either pure C++ code (as in the above example), or pure EASEA code, such as:

```
\User classes :
  Elt { int Value; Elt *pNext; }
  GenomeClass { Elt *pList; int Size; }
\end
```

although the syntax is very much C/C++ like.

Pure C++ areas are inserted nearly *verbatim* in the `.cpp` target file, while EASEA areas are interpreted and compiled into symbol tables and data structures containing valuable information which is necessary to code generation.

As an example, let us take the (very simplified) grammar analysing the above `\User classes:` section:

```
ClassDeclarations: ClassDeclaration
     | ClassDeclarations ClassDeclaration ;

ClassDeclaration: Symbol {Add to symbol table w/type
                                                UserClass}
                 '{' VariablesDeclarations '}' ;

VariablesDeclarations: VariablesDeclaration
     | VariablesDeclarations VariablesDeclaration ;

VariablesDeclaration: BaseType {store CurrentType}
                      BaseObjects
     | UserType {store CurrentType} UserObjects ;

BaseType: BOOL | INT | DOUBLE | CHAR | POINTER ;

UserType: Symbol {Find the symbol in the symbol table and
                        return a pointer to the symbol};

Objects: Object | Objects ',' Object ;

Object: Symbol {Add to the symbol table with size, type,}
         | '*' Symbol {This is a pointer. Add to the
                 symbol table with size:(sizeof(char *)),
                 object type: pointer, pointing to
                 CurrentType (defined above)}
         | Symbol  '[' Expr ']' {This is an array. Add to
              symbol tbl w/size:Expr*(sizeof(CurrentType))
              object type : array of type CurrentType} ;
```

Such a minute analysis allows to automatically create full C++ classes out of the simple structure declaration of the user genome.

Here, for instance, are the recursive copy methods (called by the copy-constructor, and `operator=` methods) automatically created for the `Elt` and `GenomeClass` structures described above:

```
void Elt::copy(const Elt &EZ_Var) {
  pNext=(EZ_Var.pNext ? new Elt(*(EZ_Var.pNext)):NULL);
  Value=EZ_Var.Value;
}
void GenomeClass::copy(const GAGenome& g) {
  if(&g != this){
    if (pList) delete pList; // Destructing pointers
    pList=NULL;
    GAGenome::copy(g);    // copy the base class part
    GenomeClass & genome = (GenomeClass &)g;
    Size=genome.Size; // Memberwise copy
    pList=(genome.pList?new Elt(*(genome.pList)):NULL);
  }
}
```

And for each class, EASEA creates a constructor, a copy constructor, a destructor (which destructs thoroughly the `pList` linked list, for instance), an `operator=`, an `operator==`, an `operator!=`, an `operator<<` and an `operator>>`.

End-users (physicists used to FORTRAN) find it very interesting to produce automatically such pieces of code, as:

- they are not interested in spending a lot of time in learning object-oriented programming,
- EASEA introduces subtleties beyond the programming capacities of many end-users,
- it would take them a very long time to write such code properly for each new problem with a genome containing a different data structure, as is the case in real-world applications,
- the produced code is guaranteed to be bug-free (a quality that even experienced programmers may appreciate).

5.3 Code generation

Once compiled, the generated `.cpp` source file cannot[4] be totally hidden from the end user, as it may contain C++ errors. As the EASEA syntax is rather simple to follow, and as the code produced by EASEA is (generally) free of bugs, most real errors come in fact from C++ functions written by the user. The nice consequences are that:

- such errors are trapped by the very elaborate C++ compiler syntax analyser,
- whatever semantic errors (bugs) that can be detected are as elaborately dealt with by the host compiler symbolic debugger.

4. unfortunately :-)

The not so nice consequence is that the human end-user must somehow debug the C++ code produced by EASEA, which requires that the produced source code be highly readable. The main difficulty resides in the fact that humans usually find compiler-produced source code quite difficult to read.

EASEA can also be used as a primer: EASEA creates a C++ source file which can be a starting point for more experienced programmers to refine afterwards.

Our main concern is then to improve presentation and to have EASEA-generated C++ code look as human as possible.

This is mainly achieved thanks to:

1. man-made *template* files (`GALib.tpl` and `EO.tpl`),
2. very careful typesetting, whenever purely EASEA-generated code appears: indentation is respected, meaningful variable names are used and comments are generated from scratch to explain what the created code is supposed to do.

5.3.1 Using template files

As one can infer by their name, template files contain the framework of an instance of a generic evolutionary algorithm (in GALib or in EO), ready to be filled up with user-specific information found either directly in sections containing pure C++ code or in the EASEA-specific sections, such as the genome structure definition.

The mode of operation is very simple. The template file is read and copied *verbatim* in the target `.cpp` file until an EASEA token (preceded with \) is found. This token asks the EASEA compiler to replace it with significant code found in the `.ez` file. Let us take an example:

```
void EASEAGenome::copy(const EASEAGenome& genome) {
  if(&genome != this){
\GENOME_DTOR
\COPY_CTOR  }
}
istream& operator>>(istream &is,struct EASEAGenome &genome){
\READ
return is; }
```

This excerpt of EO.tpl shows the automatic creation of methods. The framework is present and EASEA is periodically asked to input the necessary pieces of code found in the `.ez` file.

The result is very readable, and looks very much like what a human user would have written.

5.3.2 Performance

The concern about performance surfaces whenever a piece of code is generated by a compiler. First of all, as far as syntax is concerned, EASEA-produced C++ files are not

that different from what human-produced code would have looked like ... after debugging. Semantically speaking, it is true that when writing minor classes, a human programmer will not take the pain of writing code for operators that he knows will never be called. Although such refinement could be included with much pain in EASEA (a first pass could determine which operators of which classes will be needed), the only drawback is that the evolutionary engine will deal with slightly larger objects than necessary.
However, this cost is negligible, mainly owing to two facts:

1. EASEA-generated code only concerns the manipulation of genome objects, which usually represents only *a few percents* of the total execution time of an evolutionary algorithm (usually dominated by the user-written evaluation function).

2. EASEA generates source code, which is then compiled by an extremely evolved C++ compiler. The code optimisation taking place in the C++ compiler will minimise the lack of optimisation of the EASEA output.

6 Real-world and academic applications

Several real-world applications have been written with EASEA. Papers assignment to reviewers of the *Parallel Problem Solving from Nature* sixth international conference were done with EASEA.

EASEA has been used to optimise airfoil shapes (with a FORTRAN evaluation function) over a network of computers in the INGENET european RTD Project. A small adaptation of the GALib template file allows to create very basic parallel code using the MPI library.

EASEA is used as the basic algorithm development language in student training at the French *École Polytechnique*, at the *Laboratoire d'Informatique du Littoral* where the Graphic User Interface is currently being developed, at the *École Nationale Superieure de Techniques Avancées*, where two students taking their first course on Genetic Programming had a paper accepted at the EuroGP'01 international conference, based on the results of their two-months project in EASEA. It has also recently been tested at General Electric Medical Systems as a prototyping tool in a Medical Imaging application.

7 Conclusion and future work

Many important fields in computer science have their specific languages (FORTRAN, C/C++, LISP, PROLOG, SMALLTALK, ...). Even complex applications such as databases or spread-sheets have developed their own languages! EA programmers remain however with C++, an inadapted and difficult to use low-level object-oriented language. As a result, many scientists have no other choice than spending a lot of time becoming computer programmers and rewriting their own evolutionary algorithms. Due to thoroughly different programming techniques and languages, their programs are barely comparable, which is a great obstacle to scientific cooperation and emulation.

Therefore, feedback from scientific users is quite positive although v1.0 is still far down the road. However, EASEAv0.6 is decisive in that it is the first version able to create indifferently `.cpp` files for EO or GALib out of the same EASEA source file, showing that EASEA has the capacity of being the generic specification language for evolutionary algorithms it aims to be.

The Graphic User Interface will be the next great step, as users will not need a text editor any more to write their evolutionary algorithms.

We hope that EASEA will be able to offer the scientific community the means to try out evolutionary algorithms with a minimal time investment as far as programming is concerned. The EASEA v0.6 compiler and its manual are available on the net [1].

Références

[1] *EASEA mailing list:*
`http://groups.yahoo.com/group/easea`.
EASEA home page:
`http://www-rocq.inria.fr/EVO-Lab/`.

[2] *EVONET home page:*
`http://www.evonet.polytechnique.fr`.

[3] *EO software:*
`http://eodev.sourceforge.net/`.
EO tutorial:
`http://www.eeaax.polytechnique.fr/EO`.

[4] P. Stearns, *ALex & AYacc home page* (Bumblebee Software Ltd.):
`http://www.bumblebeesoftware.com`.

[5] M. Wall, *GAlib home page:*
`http://lancet.mit.edu/ga/`.

[6] B. Paechter, T. Baeck, M. Schoenauer, A.E. Eiben, J.J. Merelo, and T. C. Fogarty, "A Distributed Resource Evolutionary Algorithm Machine," Proc. of CEC 2000.

[7] I. Landrieu, B. Naudts, "An Object Model for Search Spaces and their Transformations," Artificial Evolution conference, EA'99 France, 1999.

[8] Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs", Springer Verlag, 1992.

[9] N. J. Radcliffe, "Forma Analysis and Random Respectful Recombination," ICGA'91, proceedings pp222-229, 1991.

[10] N. J. Radcliffe and P. D. Surry, "Fitness variance of formae and performance prediction," FOGA'95, pp51-72, Morgan Kaufmann publ., 1995.

[11] P. D. Surry and N. J. Radcliffe, "Formal Algorithms + Formal Representation = Search Strategies," PPSN'96, proceedings 1141 pp366-375, 1996.

[12] P. D. Surry, "A Prescriptive Formalism for Constructing Domain-Specific Evolutionary Algorithms," PhD thesis, University of Edinburgh, 1998.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399