CORBA-based Composition Components in a Service Model for Distributed Genetic Algorithms

Victor Theoktisto[1]

[1] Universidad Simón Bolívar,
P.O. Box 89000, Caracas 1080A, Venezuela
vtheok@usb.ve

Abstract. This paper describes GADISC, a high-level CORBA-based service model for Distributed Genetic Algorithms frameworks. A CORBA service administers all participating network nodes with multithreaded DGA objects, each having its own population(s) and operators. A genetic algorithm problem is assembled as a component pipeline made by composing filters [operators], mappings [fitness functions] and totalizers [accumulators], coded as Java class instances acting on population pairs. Problems are set up and monitored through a web page, by selecting from a expandable database of basic encodings, high level operators, accumulators, and fitness functions. Access, manipulation and control are performed through a domain specific IDL. Each DGA can be stopped, resumed, and run remotely, storing all results as XML data. A servlet collects information and statistics, with a suitable user interface designed for process control and visualization. All services are implemented using freely available software, with such planned enhancements as MPI, JNI and a better GUI.

1 Introduction

Early in the development of Genetic Algorithms **[Gold89]** as a proper body of knowledge, there were already available source and binary implementations of GA environments. Used to tackle more complex problems, they soon were made inadequate by new genotype approaches (sharing), population strategies (niching) and more genetic operators. To account for this evolution, parallel implementations of GA started appearing in the optimization landscape, such as pGA **[Liep90],** GENITOR II **[Whit90],** DGENESIS **[Cant99]** and GALOPPS **[Good97].** Most used C or C++.

The advent of the Internet allowed a different strategy to deal with complex GA problems, by redistributing populations on a network **[Jose97]** using niching strategies, using the Island or Archipelago Model **[Whit98].** The advent of Java also brought Distributed Genetic Algorithms (DGA) implementations using multiple threads and RMI **[Smit98],** with very few attempts at interoperability. A more recent implementation uses XML-based communication protocols **[Mere01].**

The GADISC framework (Genetic Algorithms with DIstributed Software Componetns) is a framework for modeling, setting up and running Distributed Genetic Algorithms (DGA) components in a truly distributed, interoperable and interactive fashion, based on the standards of the CORBA distributed objects model. It is designed as an expandable and evolvable service, providing a platform for

different encodings, a class hierarchy of genetic operators, and an abstraction mechanism for GA's acting on distributed populations.

2 The CORBA Distributed Component Model

The Common Object Request Broker Architecture (CORBA) was created by the *Object Management Group (OMG)* **[OMG1999]** with the objective of promoting object-orientation in software engineering. This is accomplished by specifying a common standard for the development of distributed applications, relying on the reusability, portability, and interoperability of objects in heterogeneous environments.

The CORBA bus defines the components' structure and manner of interaction, allowing low-level communication by synchronized *remote procedure calls* or RPC's **[OSF91].** CORBA also allows asynchronous interaction, so clients can continue working after issuing requests without having to wait for an immediate result from the server. CORBA allows classes to be implemented in several languages, executed on different operating systems and platforms, and dispersed on a heterogeneous network.

To account for this, CORBA has centered on three concepts:

- *Separation between interface and implementation*. All CORBA components are specified by an Interface Definition Language (IDL), a purely declarative language close to C++, without statements or control structures. It is implementation independent, with existing bindings for C++, Java and others **[LBS1998].** A component can specify which classes it inherits from, method signatures, attributes, throwable exceptions, input and output arguments, return values, and data types.

- *Localization Independence*. The kernel of any CORBA implementation is the *Object Request Broker (ORB),* a naming service that locates objects transparently. It routes petitions in such a way so that objects can communicate among themselves, whether on the same machine or across a network.

- *Vendor Interoperability and Systems Integration*. The *Internet Inter-Operability Protocol (IIOP)* specifies how to change the messages from *General Inter-ORB Protocol (GIOP)* in TCP/IP networks, making the Internet a giant ORB. The IIOP has the advantage of operating on the *Secure Sockets Layer (SSL),* which allows secure (encrypted) passage of data.

The architecture allows the creation of simple objects that, when inheriting from the proper services, may have transactional, secure, locked or persistent attributes. These objects are able to interrelate and find themselves within the bus. CORBA specifies an extensive service set for the creation, disposal, name access, storage and retrieval of objects, and how to define relations among them **[Orfali1996], [Orfali1998].**

All CORBA Services are implemented in IDL (**Fig. 1**). The ORB's interface offers the *Stubs* service, and dynamic invocation interfaces to CORBA objects. From the client side, to invoke a remote object as local, it is first referenced, then the client is linked with the IDL-defined *Stub*, which transforms the ORB's requests into a real implementation, inheriting the interface methods. On the server side for each interface there is an implemented *Skeleton*. Its function is to transform the requests from the

ORB into invocation on the server object or real (remote) implementation. *Stubs* and *Skeletons* are generated by the IDL's compiler.

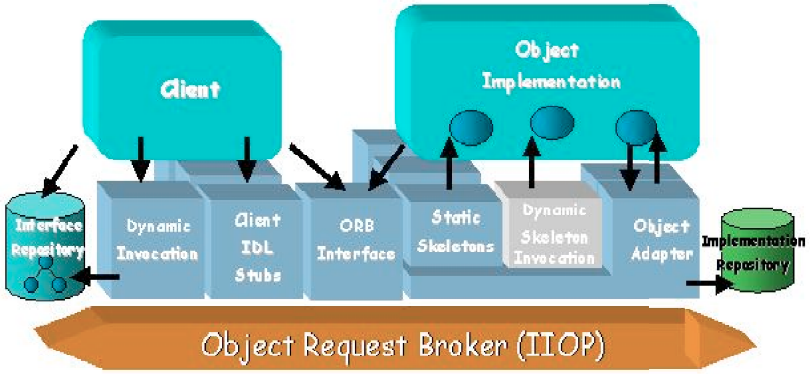

Fig. 1. The CORBA Service Architecture of the Object Management Group

3 The Component Pipeline Model

In a object computing model, several technologies must come into play: The first is a platform defining object streams, whether they are simple messages or actual sequential flows of structured data **[Chang1995].** Object streams are directed flows of objects of the same class (or kind) connecting two processes together. One acts as emitter, and the other as receiver. Object streams correspond to signal flows, and the applications correspond to composition filters and mappings operating on those object streams **[Aksit1994]**.

3.1 Composition Filters

A distributed application network is described as a directed graph, where at each node there is a sequence of transformation components. Components at nodes are connected using object streams **(Fig. 2a).**, and can then be expressed as the orderly composition of transformations (filters, mappings, and totalizers) **[Theo98b]**.

Each transformation T takes the form,
for each individual I □ P *do*
if (I *meets* filter criterion) *then* apply operation to I

A ***filter*** **(Fig. 2b)** uses a query method to detect those objects that fit a certain criterion. The output is an object stream of valid objects. A *pure filter* performs no transformations. In the example, the gray blocks are filtered out.

A ***mapping*** **(Fig. 2c)** is applied sequentially to all objects of the input stream, producing an transformed stream. A *pure mapping* has no criterion. In the example, blocks are transformed into pentagons.

A ***totalizer*** **(Fig. 2d)** returns a signal after processing a whole stream. In the example, it is counting the stream elements. A totalizer takes the form

initialize accumulator to some *default value*
for all (I *that meets* filter criterion) *do*
    obtain some partial value from I and *aggregate* it to accumulator
return value computed from accumulator

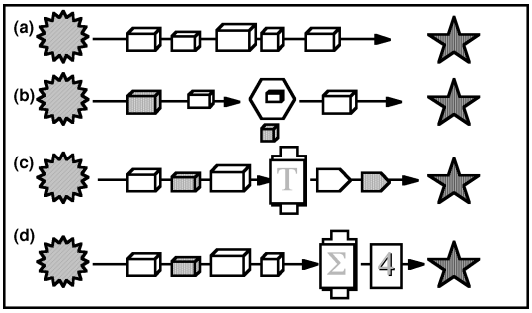

Fig. 2. Schematic component pipeline showing Filters, Mappings and Totalizers

All objects run concurrently, sequentially chained along the same object stream. Each filter repeatedly obtains an item from the stream, process it, and then passes it along to the next filter in the chain, in a pulsing manner akin to parallel digital signal processing. All instance objects are derived from one class, called the **STO** (Secure Threaded Object). Each object of this class is capable of locating, linking and secure object streams with each other **[Bena99].**

4 Distributed Genetic Algorithms

DAG models exploit new spaces in parallel. Individuals may migrate from one population to the next, even conforming new hybrid population nodes or islands.

The Island model is an abstraction used to represent connections between populations. The problem population is subdivided in several subpopulations or islands, with their own characteristics and operators, forming an archipelago (**Fig. 3**). Every number of generations, hybridization through migration performs a population exchange, exploiting differences among niches, a source of genetic diversity. Too much or too little migration might affect algorithmic convergence.

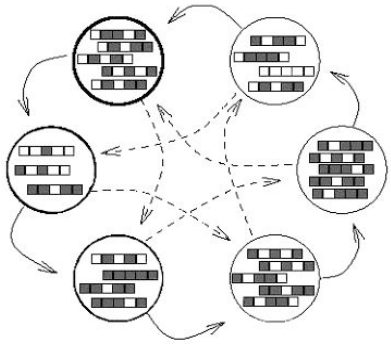

Fig. 3. Archipelago model of hybridization between island migrations during a DGA run.

There are several migration policies for the island model:

- *Star policy*: There is a master island, and the rest are slaves. All slave subpopulations send their h_1 best chromosomes ($h_1 >= 1$) to the master population, which correspondingly broadcasts its h_2 best chromosomes ($h_2 > 1$) to each slave subpopulation.
- *Ring policy*: Each subpopulation sends its h_3 best chromosomes ($h_3 > 1$) to a neighbor, in a one-way cyclic flow.
- *Network policy*: No hierarchy is specified among subpopulations, each sends its h_4 best chromosomes ($h_4 > 1$) to other population(s).

4.1 The Distributed Genetic Algorithm with Multiple Populations.

A general DGA is aptly described **[Theo98a]** by the following definitions:

A = Chromosome encoding (i.e. {0, 1}, real, string).
n = Number of alleles in a chromosome.
NP = Number of subpopulations.
Op = The set of genetic operators.
$\mathbf{P_i}$ = Population **i**.
$\mathbf{Os_i}$ = Offspring of Population **i**.
PT = $\{ \mathbf{P_1}, .., \mathbf{P_{NP}} \}$, The whole population set.
$\mathbf{G_{\%}}$ = Generation Replacement rate.
$\mathbf{K_{\%}}$ = Mortality Rate.
C = Convergence or Stop Criterion.
$\mathbf{F_x}$ = Fitness Funtion.
AFV = Average Fitness Value

The Operator Set **Op**:
$\mathbf{O_x}$ = Any Ordering (Ranking) operator
$\mathbf{RR_x}$ = Reproduction Ranking operator
$\mathbf{KR_x}$ = Ranking Removal operator
$\mathbf{S_x}$ = Selection operator
$\mathbf{X_x}$ = Crossover operator or Recombination.

$\mathbf{M_x}$ = Mutation operator
$\mathbf{K_x}$ = Culling operator
$\mathbf{J_x}$ = Join operator to merge populations.
$\mathbf{I_x}$ = Migration operator.

Algorithm
1. Generate **PT** with **NP** populations, having **L** individuals.
2. Repeat for each $\mathbf{P_i}$ in **PT**
  2.1. Apply $\mathbf{F_x}$ to compute fitness for each individual of $\mathbf{P_i}$
3. While **AFV**(**PT**) does not satisfy Convergence Criterion **C** do
  3.1. Repeat for each $\mathbf{P_i}$ from**PT**
    3.1.1. Apply $\mathbf{S_x}$ to $\mathbf{P_i}$, selecting $\mathbf{G_{\%}}$ individuals from $\mathbf{P_i}$ with $\mathbf{RR_x}$
    3.1.2. Apply $\mathbf{X_x}$ to $\mathbf{P_i}$, producing new individuals in $\mathbf{Os_i}$
    3.1.3. Apply $\mathbf{M_x}$ to each individual in $\mathbf{P_i}$
    3.1.4. Apply $\mathbf{F_x}$ to calculate fitness values for $\mathbf{Os_i}$
    3.1.5. Apply $\mathbf{J_x}$ to join populations $\mathbf{P_i}$ and $\mathbf{Os_i}$ in $\mathbf{P_i}$
    3.1.6. Apply $\mathbf{I_x}$ to migrate individuals from population $\mathbf{P_i}$ to $\mathbf{P_k}$, for some **k**
    3.1.7. Apply $\mathbf{K_x}$ to $\mathbf{P_i}$ to remove $\mathbf{K_{\%}}$ of individuals from $\mathbf{P_i}$ with $\mathbf{KR_x}$

4.2 A Pipeline of Genetic Operators

A Genetic Operator is defined as some transformation acting on the individuals of a population. They are designed as Composition-Filter objects for this implementation, having two streams instead of one (two input channels and two output channels).

$$T:P^2 \longrightarrow P^2 \qquad (\mathbf{1})$$

Using the same notation shown above, a DGA can be expressed in Composition-Filters form as follows:

1. Generate **PT** with **NP** populations, having **L** individuals.
2. *For each* $\mathbf{P_i}$ *in* **PT** *do*
  2.1. Apply $\mathbf{F_x}$ to {$\mathbf{P_i}$, *empty*} to calculate individuals fitness values
3. *While* **AFV**({$\mathbf{P_i}$, *empty*}) does not satisfy Convergence Criterion **C** *do*
  3.1. *For each* $\mathbf{P_i}$ *in* **PT** *do*
    3.1.1. *For each* operator $\mathbf{T_k}$ *in* **Op** *do*
      3.1.1.1. Apply $\mathbf{T_k}$ to channel pair {$\mathbf{Vp_i}$, $\mathbf{Wp_j}$}, where $\mathbf{Vp_i}$, and $\mathbf{Wp_j}$ are channels for some $\mathbf{P_i}$'s and/or $\mathbf{Os_i}$'s
4. Obtain best individuals from each $\mathbf{P_i}$

Although it seems logical that one population channel would suffice for a GA implementation, certain GA operators, such as Culling and Migration, require two separate channels **[Cast01],** one for objects that pass on to the next operator, and another for those that don't. Since all operators usually require access to the population as a whole, it is better to stream this population in its own channel, so no need arises for later global references. Operators are connected in a sequential pipeline by joining one-to-one the first's output channels to the second's input channels.

Only population sets of individuals flow at each channel. An operator pipeline results from the sequential composition of transformations on population pairs, as shown in **Fig. 4**

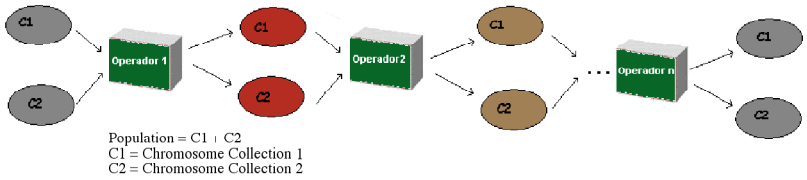

Fig. 4. Operator composition showing how to connect filters to implement the steps of a GA

The DGA is then expressed as a two-channel pipeline, in which component operators are plugged in and out, obtaining a test bed for whole classes of DGA's. Ordering and ranking transformations can also be expressed as Composition Filters, in effect establishing a design pattern for the whole family of Genetic Algorithms. As a lark, it is possible to design a DGA that mutates individuals before reproduction.

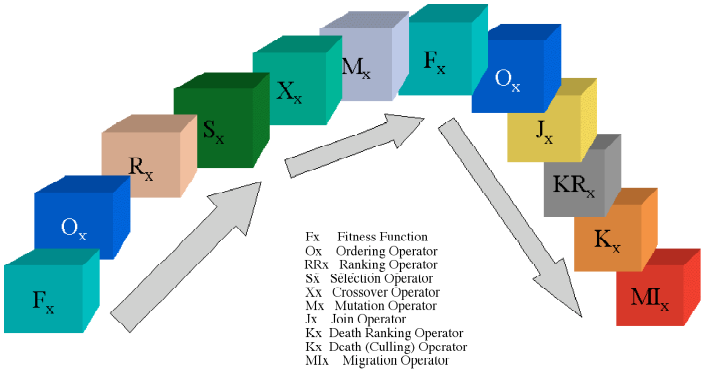

Fig. 5. Filter Composition showing sequential operator application

A typical assembly for a standard DGA setup run is shown in **Fig. 5**. Each algorithmic step is shown as the application of some operator. The first channel is usually the whole population set at that thread, and the second channel is usually the result of the transformation. The complete channel setup is shown on **Table 1**, and the UML class diagram of the GADISC implementation shown on **Fig. 6**.

Table 1. Common Operators families and their channels

Operator	1st. Input	2nd. Input	1st. Output	2nd. Output	Result
S Selection	*Pop*	*	*PopSelected*	*	-
X Crossover	*Pop*	*	*PopOffsrpg*	*	-
M Mutation	*Pop*	*	*PopMutated*	*	-
K Culling	*Pop*	*	*PopLive*	*PopKilled*	-
F Fitness Eval.	*Pop*	*	*Pop*	*	-
I Migration	*Pop*	*	*PopRemain*	*PopMigrant*	-
Avg. Func. Value	*Pop*	*	*Pop*	*	*AFV*
J Joiner	*Pop1*	*Pop2*	*PopJoined*	*	-

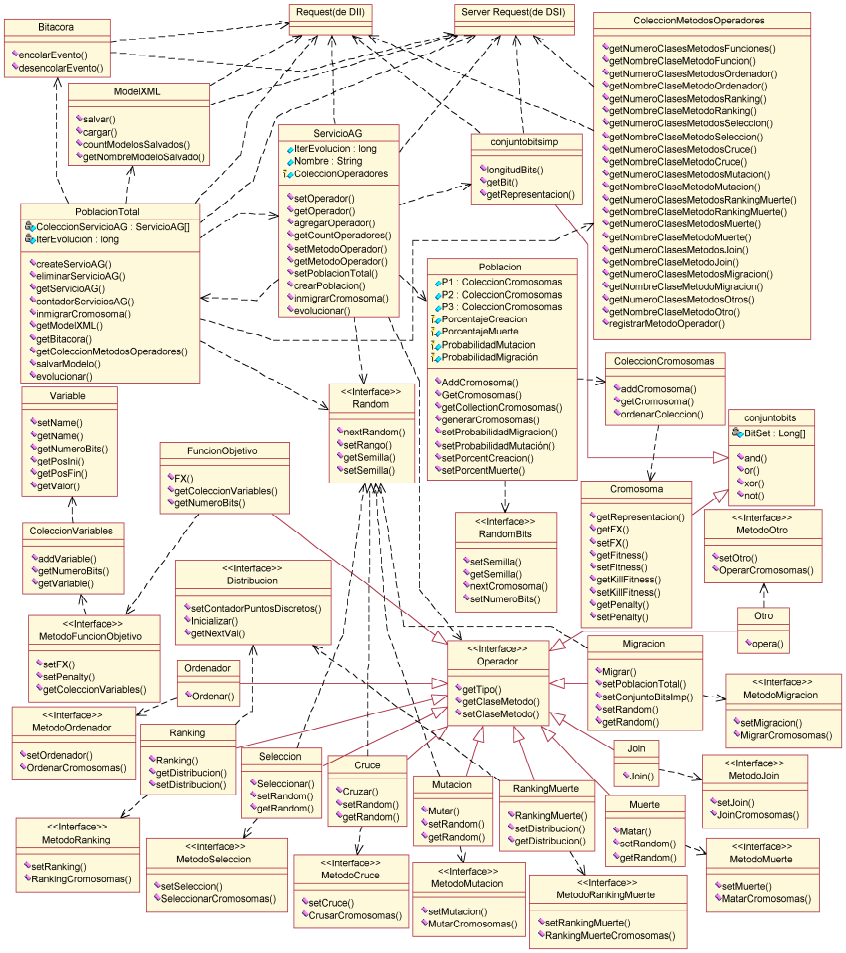

Fig. 6. UML Class Diagram of the operator class hierarchy of the GADISC System

5 Setting up and running a DGA problem in GADISC

The GADISC environment has been assembled from freely available software running on commodity hardware

5.1 Implemented with Free and Open Source Code

All specific programming was done in the Java™ language **[Bros01]**. The RedHat® Linux® platform was chosen because of its network centric services. CORBA® was implemented using the freeware Java IDL JacORB ORB, at http://www.inf.fu-berlin.de/~brose/jacorb. All web interaction is handled from a web server running Java servlets under RESIN® (http://www.caucho.com) to dynamically manage all setup and running of a DGA problem solving session from a JSP webpage. Operator classes are stored at a special directory, and their URI's stored in a Postgres® SQL database, freely accessible by JDBC® calls. Display of current DGA behavior is done by the free Java package JFreeChart® (http://www.jrefinery.com).

5.2 Interaction and Presentation

A webpage launches the Java front end for the GADISC system. The following screenshot (**Fig. 7**) shows all the functionality available to the practitioner in six different windows. At the top left window, all available genetic operators in the database are shown; At top right, their applying sequence and channel setup. At middle left, current best solutions from the population as a whole. At middle right, all CORBA nodes hosting DGA services. All services are multithreaded, so several populations may be evolving at the same node. At bottom left, a chart showing current fitness values and convergence. At bottom right, setup parameters and their current settings.

Any DGA can be started and stopped at will from the servlet. The current web session may be disconnected, leaving the CORBA services in the network to solve the problem, and connect later to see results. There is a notification service that alerts when a run has converged. All intermediate values and final are stored as XML encodings, and easily exported for further analysis.

Each problem must provide its own implementation of the Fitness Evaluation method, and its own encoder/decoder of genotype information. A XML specification for a genetic algorithms formulation has not been devised yet.
All results are stored by a software component implementing a log database, which can be turned on and off. It queues up all events generated at each operator. Stored information records have population ID, DGA ID, generation number, operator type, affected chromosome, fitness function ID and fitness.

5.3 Example Problem #1

The first run was made by using the following Fitness Function.

$$f(x,y) = (x^2 + 4y^2)e^{(1-x^2-y^2)}$$

It has two global maxima at (0,-1) and (0,1). It was implemented with real encoding. In 25 evolutions, half of the distributed island problems had converged to (0,-1) and the rest to (0,1). Further evolutions only caused random switches between the two groups. Since it was implemented in Java, speed was not a factor, although this problem was solved in less than a minute.

5.3 Example Problem #2

The second run was made by using the following Fitness Function, subjected to a restriction.

$$f(x,y) = (x-2.5)^2 + (y-1.5)^2$$

$$s.a.$$

$$sen(x+y) \geq 0$$

This is an optimization problem, with an unique minimum at (0.5, -0.5). A penalty function was incorporated to the fitness function. On average, it took around 250 evolutions to obtain this result, incrementing the penalty function every ten evolutions.

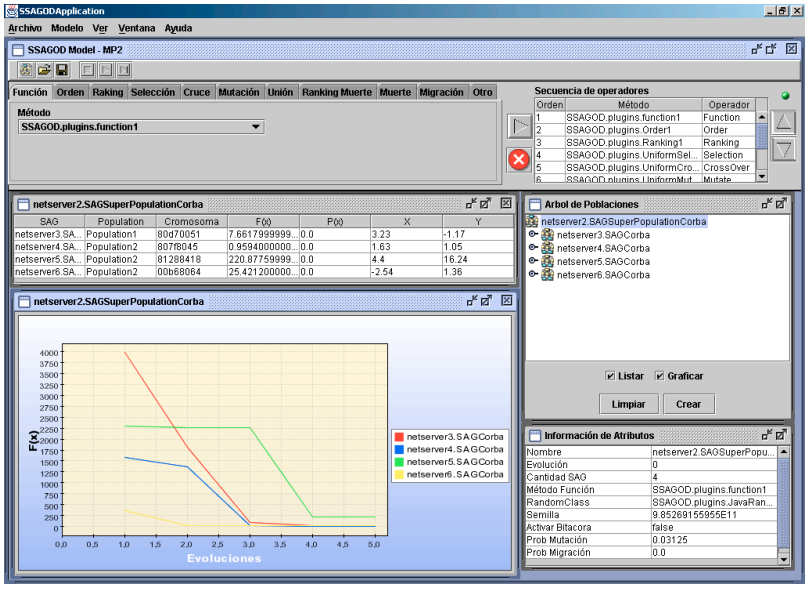

Fig. 7. UML Class Diagram of the operator class hierarchy of the GADISC System

6 Conclusion

The GADISC framework provides a versatile platform for setting up and running multithreaded Distributed Genetic Algorithms. The deployment cost is very low, since it was implemented with available free and open source code, or otherwise no

cost licensing. Derived genetic perators are created easily by subclassifying a similar operator class, added to the operator's database, and made available to the framework.

Future enhancements will include a measure of parallelization in multiprocessor machines using OpenMP, faster C++ implementation of operators (using the Standard Template Library, STL), better GUI and interaction for setting up encodings, and a generic fitness function parser linked via Java Native Interface (JNI). A plan to migrate to SOAP (XML based communications) has been proposed as an natural improvement for the openness and versatility of this framework.

The web site for GADISC is at http://www.ldc.usb.ve/~vtheok/GADISC.

References

[Aksi94] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object-Interactions Using Composition-Filters, in: "Object-based Distributed Processing", R. Guerraoui, O. Nierstrasz & M. Riveill (eds.), Lecture Notes in Computer Science 791, pp. 152-184, Springer-Verlag (1994)

[Smit98] Smith, J., Carreon, E., Sugihara, K.: A Web-accessible Tool for Design of Distributed Genetic Algorithms. Internal Report. Department of Information and Computer Sciences. University of Hawaii at Manoa (1998)

[Beld95] Belding, T. C. The distributed genetic algorithm revisited. Proc. 6th International Conf. on Genetic Algorithms. Morgan Kaufmann, San Francisco, CA, pp. 114–121 (1995)

[Bena99] Benarroch, S., Quijada, M.: Ambiente de Composición de Filtros para el Desarrollo de Aplicaciones Basadas en Objetos Distribuidos. Trabajo de Grado, Victor Theoktisto (tutor). Universidad Metropolitana, Venezuela (1999)

[Bros01] Brose, G., Duddy, K., Vogel A.: Java Programming with CORBA, 3rd Edition, John Wiley & Sons, Inc. (2001)

[Cant99] Cantu-Paz, E.: Designing Efficient and Accurate Parallel Genetic Algorithms. IlliGAL Report No 99017, Illinois Genetic Algorithms Lab., Univ. of Illinois at Urbana-Champaign (1999)

[Cast01] Castro Vielma, J.C., Samaniego Revilla, C.: Servidor de Soluciones de Algoritmos Genéticos para Optimización Distribuida. Trabajo de Grado, Victor Theoktisto (tutor). Universidad Metropolitana, Venezuela (2001)

[Chal99] Chalmers, R.: Does XML Need CORBA?. The Object Management Group. http://www.omg.org/xml/ (1999)

[Chan95] Wen-Tsung Chang, Chien-Chao Tseng. *Supporting Distributed Objects in FIFO-Based Message-Passing Systems*. Journal of Object-Oriented Programming, February 1995, Pages 56-64.

[Denq01] Denq, D., Dunietz, I., Eddy, J., Ehrlich, W., Gerth, D., Larson, B., Sivaprasad, G.: Certifying Component Performance in Synchronous Distributed Client/Server Systems. In Proceedings of ACM OOPSLA 2001, Tampa Bay, FL, October (2001)

[Gilb00] Gilbert, M.: Inside JfreeChart. Simba Management Limited. June 2000, <http://www.jrefinery.com>, (2000)

[Gold89] Goldberg, D. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989)

[Good97] Goodman, E.: An Introduction to GALOPPS – the "Genetic ALgorithm Optimized for Portability and Parallelism" System. Technical Report GARAGe 97-07-01, Michigan State University, Intelligent Systems Laboratory (1997)

[Holl75] Holland, J.: Adaptation in Natural and Artificial Systems. MIT Press (1975)

[Husb94] Husbands, P.: Distributed Coevolutionary Genetic Algorithms for Multi-Criteria and Multi-Constraint Optimisation. In: T. Fogarty (Ed.), Evolutionary Computing, Lecture Notes in Computer Science, Vol. 865, Springer Verlag, pp. 150-165 (1994).

[Jose97] Joseph, D., Kinsner, W.: Design of a Parallel Genetic Algorithm for the Internet. Proceedings of the 1997 Conference on Communications, Power and Computing IEEE WESCANEX'97, pp 333-343, May (1997)

[Larm99] Larman, G. UML y Patrones. Prentice Hall Hispanoamericana, S.A., México (1999)
[Lewi98] G. Lewis, S. Barber, y E. Siegel.: Programming with JavaIDL. John Wiley & Sons, Inc. USA (1998)
[Liep90] Liepins, G.E., Baluga, S.: pGA: an Adaptive Parallel Genetic Algorithm. Proceedings of ORSA-TIMS CSTS Conference, Williamsburg, VA (1990)
[Mend94] Mendes, R., Neves, J.: Genetic Algorithms, Classifiers and Parallelism – an Object Oriented Approach. Proceedings of the 2nd World Congress on Expert Systems, pp 1199-1206, Lisbon, Portugal (1994)
[Mere01] Merelo, J.J., Castellano J.G., Castillo, P.A., Romero, G.: Algoritmos Genéticos Distribuidos Usando SOAP. In XII Jornadas de Paralelismo pp. 99-103, ISBN:84-9705-043-6, Valencia, Spain, September, 2001.
[OMG98] Object Management Group. CORBA Main Suite. http://www.omg.org, (1998)
[OMG99] Object Management Group. The Common Object Request Broker: Architecture and Specification, [CORBA 2.3], (1999)
[Orfa96] Orfali, R., Harkey, D.: The Essential Distributed Objects Survival Guide New York. Wiley Computer Publishing (1996)
[Orfa98] Orfali, R., Harkey, D.: Client/Server Programming with Java and CORBA (2nd ed.) New York. Wiley Computer Publishing (1998)
[OSF91] The Open Software Foundation. Remote Procedure Call in a Distributed Computing Environment. Massachusetts USA (1991)
[Oste01] Ostermann K., Mezini, M.: Object-Oriented Composition Untangled. In Proceedings of ACM OOPSLA, pp. 283 - 299,Tampa Bay, FL., October (2001)
[Stee95] Steeb, W. H., Solms, F., Tan Kiat Shi: Genetic Algoryhms and Object-Oriented Programming. Int. J. Mod. Phys, Phys. Comput, 6(6):853-869, Singapore (1995)
[Theo98a] Theoktisto, Víctor S.: Order of Magnitude Reduction of Genetic Algorithms Operations. MPCS'98, III Conference on Massively Parallel Computing Systems, April 6-10, Colorado Springs, Colorado, (1998)
[Theo98b] Theoktisto, Víctor S.: A Composition-Filters Framework for Developing Distributed Java Applications on the Web". SCI'98/ISAS'98, World Multiconference on Systemics, Cybernetics and Informatics, Orlando, Fla., Julio 12-15 (1998)
[Whit90] Whitely, D. and Startkweather, T.: GENITOR II: a Distributed Genetic Algorithm. Journal Exprt. Theor. Artificial Intelligence, 2, 189-214,(1990)
[Whit98] Whitley, D., Rana, S., Heckendorn, R.: The Island Model Genetic Algorithm. On Separability, Population Size and Convergence. Nov, (1998)