ROS (Remote Optimization Service)

E. Alba, J.G Nieto

23 Nov 2005

Abstract. In this report the Remote Optimization Service (ROS) is described. This service allows to users the remote access with the possibility of executing and manipulating optimization algorithms based on heuristic techniques, located in stores with hardware and software resources to optimize practical and real problems. How this work relates to other similar work on the subject, and the main characteristics and ROS architecture is also presented. In addition, an introduction of the communication tools used, the XML specification of data types and algorithm encapsulation using the ROS service is outlined. Finally, a comparison of the different configurations of this system is shown, as well as the results obtained after making test evaluations of this service using typical configurations.

1 Introduction

At the present time, efforts are constantly being made to develop systems that integrate libraries of heuristic algorithms (like evolutionary algorithms [9]). Methodologies and services are also being developed to facilitate de use of these software repositories. A common characteristic in all almost works is the *XML*[2] */optimization algorithms* combination, by means of which, different applications are proposed in order to facilitate tasks such as: the creation of new algorithms, compilers will generate code from XML specifications, algorithms to be executed will be integrated into services, including XML objects and methods integration (see for example [8][12][13][17]).

In this document the Remote Optimization Service (ROS) is presented [1], which aims to provide an execution service of heuristic algorithms through the Internet. Using ROS it is possible to establish a system of servers which handle the inclusion and the operation of different algorithms, these being heterogenous algorithms, and possibly implemented with different programming languages.

By means of a client program, we can connect with the servers in order to interact remotely with the algorithms, which will already have been registered and active in the system. This interaction includes functions as: the selection of algorithms, parameters input, remote execution and the reception of results.

The ROS architecture has been designed to facilitate the incorporation of new servers and algorithms into the system, creating in this way an extensive set of heuristic algorithms accessible from different clients at distributed points of a network.

Next, in section 2, a summary of existing systems with characteristics related to this service is presented. ROS is defined in section 3 and in section 4 its architecture, components and main characteristics are described. In section 5 there is a brief introduction of the communication tools used by ROS, focusing specially on `WebStream`. Section 6 describes *Wrappers* tools, from which one can add algorithms to ROS. In section 7 the `optimization_algorithm` XML specification is described. Finally, in section 8 a series of evaluations of this service are made, concluding in section 9 where plans for future work are outlined.

2 State of Art

Some work has been done to address the necessity of specifying heuristic algorithms by means of XML. The most remarkable example is the *EAML* [11][12] language that specifies evolutionary algorithms by means of labels defined in its dictionary (EAML.DTD), and from which, it is possible to generate C++ code using the ECC (Evolutionary Computation Compiler) [12] compiler. EAML admits evident design improvements to avoid all the elements being defined as specific labels in the dictionary. In addition, it does not offer general purpose labels, so that whenever we want to specify a new element it is necessary to modify the dictionary. Nevertheless, in this work the target of XML is the specification, not the real execution and even less the parallelism.

A system that introduces some improvements on the limitations of EAML is the OPEAL [13] library, which groups together evolutionary algorithms implemented by means of XML mixed with Perl scripts called `Algorithm::Evolutionary` [13], so that it offers an alternative way of representing the data structures of the algorithms.

This library has been extended and includes the possibility of implementing some modules of OPEAL, particularly modules of distributed programming using SOAP [14]. In this case, specification and implementation are united and a limited definition of basic algorithms is possible. In the *XMLMen* wiki [16], paradigms which are different from specification of evolutionary algorithms using XML are explored (including a previous version of `Algorithm::Evolutionary`), using EAML finally to carry out a series of experiments within a system denominated Sutherland.

Other available services exist which integrate XML with Evolutionary Computation, for example, the Web service based on evolutionary algorithms *eaWeb* [15]. Using an XML template, a description of the functions to be optimized and the parameters of the evolutionary algorithm are provided to this system, in order to return the optimized function later. *eaWeb* works in a similar way as that outlined in this paper, nevertheless, it restricts the type of algorithms, data structures and parameters to be used, which are implemented by means of a set of XML files within the system itself. This characteristic makes the extension of de service more difficult, as in order to add another parameter or type of algorithm, new XML file must be created and the system updated.

Another available tool in this scope is *eaLib* [17]. It is a Java classes library through which, a developer with knowledge in the field of the evolutionary computation can create algorithms of this type easily. In addition, it includes an XML serialization method, but this is an automatic method of objects serialización in Java, not a specific one legible by experts in evolutionary algorithms. Unlike the ROS service (the object of study in this document), the target in *eaLib* is not to offer support to the remote execution of existing heuristic algorithms, but it is focused on the creation of new evolutionary algorithms.

One step ahead, we can find systems that offer a graphical user interface (GUI) to facilitate the creation and later execution of new algorithms from components grouped in libraries. For example, *Evolvica* [19], is the successive system of *eaLib* (also implemented in Java) and tries to go further because it allows the development of evolutionary algorithms to take place in a graphical context, Java editor, code *debugger* and visual analysis of results. *Evolvica* is based on the *Eclipse* [20] platform, and uses the foundations that this tool provides respect to the Java characteristics. Due to these characteristics *Evolvica* is aimed towards expert users with knowledge of Java programming and evolutionary computation. This system is in experimental phase still and is the subject of continuous updates.

Another example is the *EAVisualizer* [18] application, also implemented in Java, although the user does not need to have knowledge of Java or another programming language, because using this system, the execution of evolutionary algorithms can be carried out by means of an interactive selection of different previously predefined components. These components implement the parts of the mentioned type algorithm (operators, structures of data, parameters, etc). *EAVisualizer* is very extensible as much in new evolutionary components as in views of its graphical interface, which are incorporated in successive updates of the system.

The Table 1 shows the relation among the systems previously mentioned, emphasizing a series of characteristics from which we can make a comparison: method of operation, use of XML, programming language, specification of algorithms and whether or not it offers GUI. In the last row, the previously mentioned characteristics are also specified in relation to the ROS service.

In both of the first examples (EAML+ECC and OPEAL) the main characteristics are based on the use of XML (EAML especially) for the specification of evolutionary algorithms. That is, they implement the algorithm by means of EAML completely to compile and to execute it, locally (as in ECC) or in a distributed way (as in OPEAL).

The ROS objective on the matter is to use XML to contain the data that the algorithm uses (parameters, results, orders of execution, etc), without taking part in its implementation which could be done by means of some programming language.

System/ Characteristic	Description	Operation Method	Use of Xml	Programming Language	Algorithm Specification	Offers GUI
EAML + ECC	XML Specification and Compiler	Local	Yes	XML & C	Restricted to the XML Design	No
OPEAL	Library	Local & Distributes	EAML & SOAP	Perl	Restricted to the Programming Language	No
eaWeb	Web Service	C/S Web	Yes	ASPX	Restricted to the XML design	Yes
eaLib	Java API	Local	No	Java	Restricted to the API Classes	No
EAVisualizer	Application (Framework)	Local	No	Java	Restricted to the Available Operations	Yes
Evolvica	API & Edition Tool	Local	No	Java	Restricted to the API Classes	Yes
ROS	Distributes Optimization Service	Local & Distributes	Yes	Any	E/S Wrapper Interface (section 6)	Yes

Table 1: Table of characteristics of the systems related to ROS.

The following systems (*eaWeb*, *eaLib*, to *EAVisualizer* and *Evolvica*) offer a similar functionality. In these systems a series of functions, structures and data types to be used are established to specify algorithms and to execute them, either in a Web server such as *eaWeb*, or locally as in the other cases. ROS works in a different way, because its mission is to incorporate only implemented algorithms, and to provide mechanisms in order to the obtaining of parameters and the local or remote execution. Due to this characteristic, ROS provides a greater freedom in the implementation of the algorithms which it can work with.

In addition, in all of these systems (except *eaWeb*), the algorithms must be implemented in Java, using its own classes and/or components. The implementation of the algorithms in ROS is totally independent of the system, that is, algorithms implemented in practically any programming language can be added (by means of a `wrapper`), whenever they fulfill the minimum requirements of input/output data (see section 6).

As a final characteristic, it is observed in systems like *EAVisualizer*, *Evolvica* and also ROS, the facility of a graphical interface (GUI) is available in order to facilitate the management of the system by the final users. In the case of *eaWeb*, it is a Web service and incorporates (HTML-aspx) forms in its client application for data acquisition and presentation.

3 What is ROS?

ROS *(Remote Optimization Service)* born with the objective of establishing an optimization service on the Internet, that is, to facilitate the access and use of multiple algorithms (especially heuristic) created by different developers. It covers therefore, the necessity of grouping heterogeneous algorithms and those implemented in different programming languages in a system to which a multitude of users distanced in the network can gain access easily.

This way, two kinds of participants are established who work on ROS: the *developer*, that adds his algorithm to a server (Worker) by means of a *Wrapper* (section 4), and the *user*, who using the ROS Client program can work remotely with the algorithms available and thus solve a problem (typically of optimization).

However, it is possible to configure ROS so that it works locally or integrated in a LAN, establishing therefore a platform for the testing and execution of algorithms.

4 ROS Description

Due to its distributed and multiplatform nature, ROS is implemented with JAVA [6] and uses the `WebStream` communication tool (section 5) to establish a hierarchy of servers based on the Client/Server (C/S) model.

ROS connects four types of components: the client, the primary server, the distribution server and the process server or *Worker*. In figure 1 we can see a scheme of the ROS architecture.

1. The ROS client offers a graphical interface through which a user can execute a series of functions on the system. Such functions are: the user identification or registry in the system, the type of algorithm selection, server available selection (based on the type of algorithm), and the remote execution of the selected algorithm. Internally, the ROS client contacts the other components of the system (primary server and server of distribution) to carry out the exchange of necessary information in each function.

2. The primary server provides the identification service in the system (login and password) to the client, as well as the information service about the types of algorithms and addresses of accesible servers. It is the first service which a client accesses when a ROS execution begins.

3. With the distribution server (*Server*) all existing flows of information between the connected clients and the process servers are redirected. *Server* is therefore a fundamental piece for obtaining both the scalability and the distribution of the system in the network, because it organizes the communication between the client and the algorithm. That is, it works like a Proxy server with regard to the interaction between the client and the process server. This way, the incorporation of a new process server or algorithm into the system requires its registry in the distribution server. This registry includes the introduction of a new row in the configuration file of algorithm types and the process servers net address.

4. The process server (*Worker*) lodges the algorithms in the system and manipulates them according to the specified orders from the client. The procedure to lodge the algorithms is by means of *Wrappers* which encapsulate the code of these algorithms and offer the necessary methods for their manipulation from the outside.

   The main function of this server is to offer the execution service of the algorithms that it contains, whenever it receives a suitable request from a client. The execution request, is not carried out directly from the client, really it is the distribution server (previously commented) where it is produced.

Once a request has been managed and a selected algorithm executed, the process server will return a response to the distribution server through the same channel.

In the following picture (Figure 1), we can observe a summarized scheme of the ROS architecture in a hypothetical distribution of its components and the interrelation that exists among them.

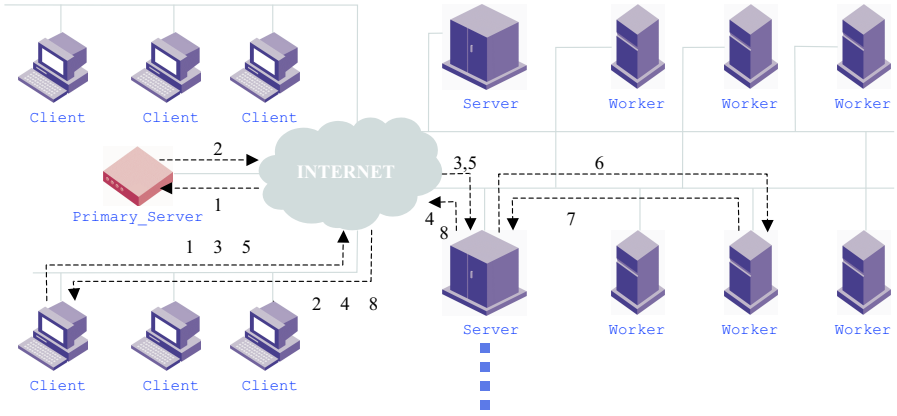

Figure 1: Summarized scheme of the ROS architecture.

Normally, the system is distributed by means of a set of connected subnetworks as shown in the figure above. The subnetworks that contain the clients and another subnetwork where the primary server is, are located on the left-hand side. The distribution servers (*Servers*) are in the center and from them the subnetworks with the process servers are organized (*Worker*). All subnetworks communicate with each other through the Internet. This is an ideal scheme, really different components in the same subnetwork can be arranged and it is even possible to configure ROS in order to work locally with all the components located in the same machine.

In order to follow a complete example of the operation we must look at the discontinuous lines that appear in the figure, because they show us an idea of the data flow sequence through the system in a cycle of execution. Next, we enumerated the basic steps of the ROS execution.

1. *User Identification.* The user introduces his login and password into the initial client form and connects with the primary server for his verification or registration. Once the user has been identified, the primary server sends back to the client a message with either the confirmation or rejection of the user. If it was a new registry, the primary server creates a new row in its users file and sends the confirmation to the client. After identification has been confirmed, the client gets to the phase of the selection type of algorithm and distribution server.

2. *Type of algorithm, server address and programming language selection.* In this phase, the client requests the types of algorithms from the primary server, which sends back a classification of the types of algorithms available at that moment, the distribution server addresses where they are located, and finally the programming languages in which the algorithms were implemented.

3. *Algorithm environment request.* The client, depending on the type of algorithm, server and programming language chosen in the previous step, sends a request to the distribution server. In this request, the client can establish the selected algorithm environment, that is, the types of XML file and graphical window corresponding to the type of algorithm.

4. *Obtaining the algorithm environment.* The distribution server generates the XML file adapted to the selected algorithm and sends it to the client. When the client receives the file, generates the graphical window suitable to the XML file specific format is generated automatically.

5. *Setting Input parameters.* In this phase, the client has the input parameters fields of the algorithm, so the user can now introduce these parameters and initiate a remote execution sending the request to the distribution server.

6. *Execution of algorithm.* The distribution server receives the request for execution from the client. It contains the type of algorithm to be executed, and based on this, the appropriate server process address is selected. Once connected with this server (*Worker*), it sends the data collected from the client in order for the remote execution to begin.

7. *Obtaining results.* When the execution of the algorithm finishes, the results are written in the XML file and it is returned to the distribution server again.

8. *Reading results.* The distribution server finally sends back to the client the XML file with the results of the execution, which shows the results on the screen or otherwise generates error messages.

We must to emphasize the role of XML within the information flow that circulates between the ROS components. It is also important that we look in depth at *Wrappers* because they are a fundamental piece in the operating of this service.

5 The Exchange of information in ROS

Due to its distributed nature, ROS requires connections to be established and information to be exchanged between components that will generally be installed in remote machines. Two different versions of ROS are available depending on the communication system used:

- ROS using information exchange by means of RPC with SOAP [4].
- ROS using information exchange by means of `WebStream`.

In the SOAP version of ROS, the information exchange is established by means of a remote procedure call between C/S machines. Remote objects in the accessible servers from the client are created, and through these objects the exchange of information takes place. The following figure shows the structure of SOAP services established in ROS.

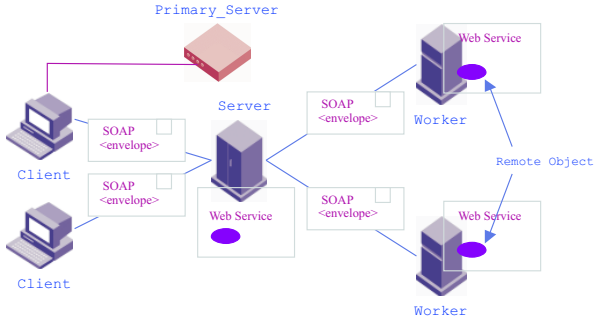

Figure 2: Summarized scheme of ROS SOAP architecture.

In this summarized example (Figure 2), ROS/SOAP consists of two process servers (*Workers*). These are implemented by means of Web services (Apache [5]), where the remote object which carries out the function of the process server is registered. The distribution server has access to each remote object, (that fulfills the function of the distribution server) to the client by means of a Web service. Thus the graph of ROS communications is established through C/S sequences until the algorithm (located in *Worker*) "connects" with the client.

This implementation of ROS using SOAP has the advantage of making the system very scalable and offers a very standardized medium (XML envelope) for a possible adhesion with other components or applications that use the same standard.

On the other hand, the necessity of installing additional Web services in order to register remote objects involves an added complexity. In addition, the system must interpret the SOAP RPC protocol meaning that the process load of the service is increased.

In situations where a smaller process load is required, or the installation of additional Web services is complicated, a second version exists that uses `WebStream` [7].

`WebStream(WS)` is a new tool for the C/S processes communication based on *Synchronous Message Passing* and by means of which it is possible to establish a *Connection Oriented* service. The target of `WebStream` is to offer a series of improvements with regard to other remote process communication systems (sockets, RPC, RMI, etc), in order to facilitate the implementation of C/S systems to the developer. This improvement includes the possibility of exchanging constructed data types such as arrays, files, even generic objects, without the necessity of computational hard systems or the use of XML.

`WebStream` is implemented by means of a Java class that encapsulates the `sockets TCP` service of the `java.net` library, so providing a method interface similar to the `sockets` interface as methods for the establishment of connection, message sending and reception, connection finalization, etc. This service is available in communication port number 80 by default thus the usual security hindrance will not be a problem now.

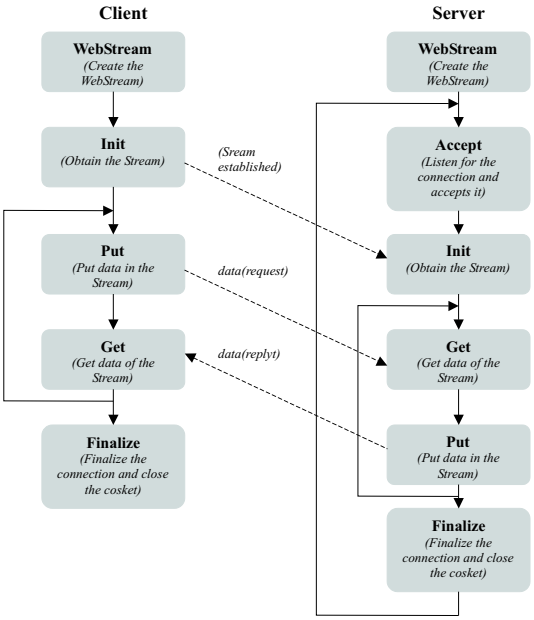

Figure 3: WebStream (C/S) basic operation.

To implement a communication system between C/S processes using `WebStream`, it must create constructor instances and methods are invoked to establish the connection as shown in figure 3.

`WebStream` provides the possibility of *full-duplex* information interchange, so that writing and reading in both directions is carried out. Although one aspect to be taken into account is the data types that are being written and read, since to avoid reception errors, the method which receives a fixed data type from the `stream` must correspond with the method which sent these data, for example:

if we do `put(float [] p)` [7] in the client, it must do `get(float [] p')` correspondingly in the server, otherwise the results could be unexpected.

ROS uses `WebStream` to establish connections and information exchange between components which are integrated in it, in such a way that it implements the client as `WS` client, the primary server and process server as `WS` servers and the distribution server as a `WS` server with respect to the client and as a `WS` client with regard to the primary server.

6 Wrappers

By means of Wrappers, the algorithms that are trying to added to in the system are encapsulated, specifically in the process servers (*Workers*). This algorithm can be very heterogeneous with respect to the programming language (C, C++, Java, Modula 2, etc) and so too can the paradigm used for its development (Object Oriented, Modular, Imperative, etc).

Due to the great variety of different kinds of algorithms, we must develop a specific `Wrapper\_` (Java) class for each algorithm, although all of this must to be inherited from the father `Wrapper` class (see Figure 4) since it offers the same methods and is instantiated in the same way.

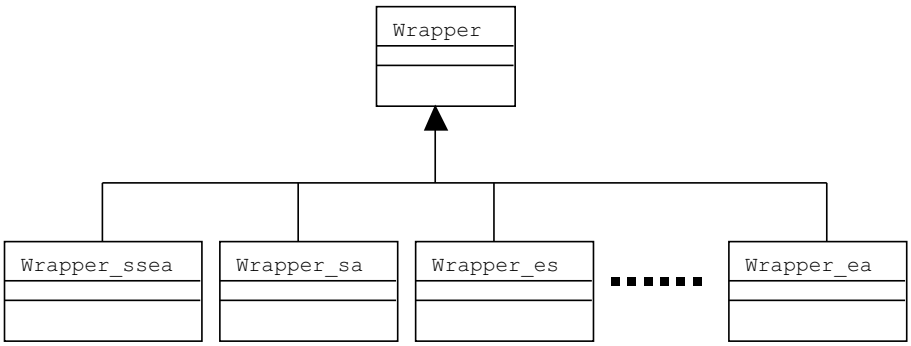

Figure 4: Simplified design of the Wrapper class diagram.

Each *Wrapper* offers in its interface the public method `run()` that at the same time calls to the rest of methods dedicated to carry out the "tunneling" of the algorithm.

- `private void getConfigurationFile(String xml_file_name, String config_file_name)`. Generates the suitable configuration file for the algorithm from the XML file (see section 7) with the parameters.
- `private void compile()`. Executes the appropriate compilation commands to the algorithm code.

- `private void execute()`. Executes the appropriate execution commands to the algorithm code.

- `private void getXmlResult(String results_file_name, String xml_file_name)`. Inserts into the XML file the results of execution from the results file of the algorithm.

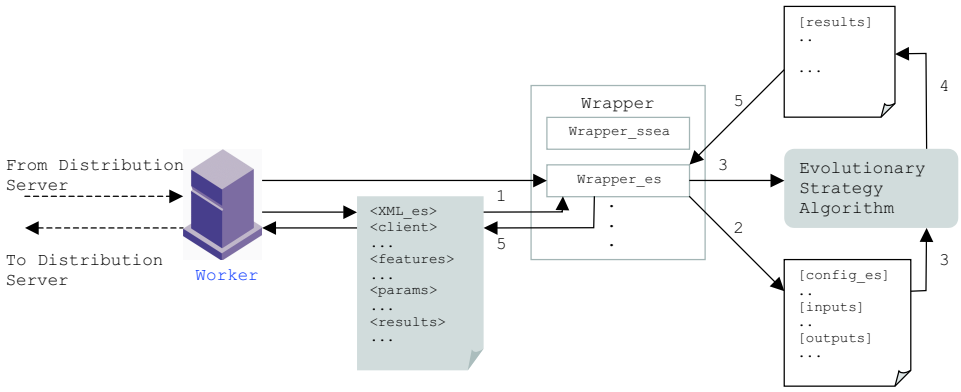

Figure 5: Position of a Wrapper with respect to the encapsulated algorithm.

To understand way the *Wrappers* operate we must look at a typical example of (illustrated in the figure 5), since the behavior is similar in all cases. Specifically we fixed in a Wrapper for an algorithm of *Evolutionary Strategy* [26] whose name is `Wrapper_es`:

1. The server process (*Worker*) obtains the XML file (sent by the distribution server) and makes it available to the `Wrapper_es`.

2. From the XML file with the input parameters, the `wrapper` generates the specific configuration file to the algorithm (`getConfigurationfile` method).

3. `Wrapper_es` extracts the compilation (if it is necessary) and execution order of the algorithm, running this compilation/ejecution order and waits until this process finishes (`compile` and `execute` methods). The algorithm internally obtains the parameters of the specific configuration file generated in the previous steps.

4. When the execution of the algorithm finalizes, it writes the results parameters in the appropriate file of the algorithm.

5. The `wrapper_es` adds the results to the XML file (obtained from the results file). After this, the server process has the complete XML file with parameters and results, so it is returned to the distribution server.

Due to the automatic way of working of the `wrappers`, the algorithms which being added to the system must fulfill the input/output data restrictions imposed on this scheme. That is to say, the input data are provided to the algorithm by means of a file (in the previous figure `[config_es]`), and the output data (results) will be written by the algorithm in another different file (in the figure `[results]`).

7 XML in ROS

It is not necessary in the ROS service, for the XML specification to describe the algorithm which it is trying to manipulate completely. In this system, XML is used to specify the configuration of an algorithm, thus the information that it contains is aimed to: input parameters, output parameters, information about the client (user), features about the implementation of the algorithm, programming language, etc. Figure 6 shows a typical ROS XML document expressing the configuration of an *Evolutionary Strategy* algorithm continuing thus with the example from the previous section.

In this sense, the dictionary (*DTD - Data Type Document*) has been defined as `optimization_algorithm.dtd` [21][22] created for this service, and from this, all the XML files that ROS uses are specified. Each XML file is organized hierarchically so `<optimization_algorithm>` is the root element which contains the following four main elements:

- `<client>`, contains elements referred to the data about the client (IP, ID, etc).
- `<features>`, their elements describe the characteristics of the algorithm (language, orders, etc).
- `<params>`, elements that specify the input parameters.
- `<results>`, elements that specify the output parameters (results).

In addition, it incorporates general purpose elements called `<others \_...>` elements, by means of with we can specify other data types not-contemplated in the DTD.

The ROS service uses this format of XML files to contain the interchange data between their different components. Initially, an XML file is generated in the distribution server in agreement with the type of algorithm selected by the user (in the client stage). In this phase the client information and the characteristics of the algorithm selected in the labels `<client>` and `<features>` are introduced respectively, in addition to the type of algorithm in the attribute `type` of the label `<params type=‘‘tipo of algorithm’’>`.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE optimization_algorithm
  SYSTEM "optimization_algorithm.dtd"
<optimization_algorithm>
                  <client>
client                <client_name>jnieto</client_name>
                      <client_ip>150.214.214.26</client_ip>
                      <client_id>null</client_id>
                  </client>
                  <features>
                      <language>C++</language>
                      <compilation>make MainSeq</compilation>
features              <execution>make SEQ</execution>
                      <online />
                      <comment />
                  </features>
                  <params type="es">
                      <population_size>100</population_size>
                      <offsprings_size>50</offsprings_size>
                    <crossover>
                        <crossover_prob>1.0</crossover_prob>
                    </crossover>
                    <mutation>
                        <mutation_prob>0.1</mutation_prob>
params                <number_of_generations>100</number_of_
                    genertons>
                    <migration>
                        <migration_rate>4</migration_rate>
                        <migration_number>20</migration_number>
                    </migration>
                    <fitness_function>rastrigin</fitness_function>
                    <replacement type="inclusion" />
                    <selection type="roulette_wheel" />
                  </params>
                  <results>
                      <best_fitness>0.00288847</best_fitness>
                      <worst_fitness>546.003</worst_fitness>
results               <generation>72 </generation>
                      <computation_time>1751ms</computation_time>
                      <error>null</error>
                  </results>
</optimization_algorithm>
```

Figure 6: XML document of an *Evolutionary Strategy* algorithm.

This way the file is sent to the client where the input parameters of the algorithm are introduced, each one in corresponding sub-labels within `<params...>`. When the client initiates the execution, the file is returned to the distribution server in order to it to be passed on to the server of the corresponding process.

When an XML file arrives at the distribution server, `wrapper` obtains the information and inserts results in the appropriate `<results>` label, each one within its corresponding sub-labels. Once finalized, the process server returns the complete XML file through the same channel to the client, who then shows the results to the user. This process is described in the previous section.

This XML specification about evolutionary algorithms looks for simplicity and ease of use, including a complete set of the necessary arguments in this matter.

8 ROS Evaluation

Two types of network configurations have been considered in order to carry out an evaluation of the ROS service: through a LAN and Locally, working in each case on a common set of algorithms:

- In the LAN configuration, a 100Mbps *Ethernet* network has been used, the servers were executed in machines with 2.4 GHz *Intel* processors and RAM memory of 512MB and the client phase was executed in a machine with 2.6 GHz *Intel* and RAM memory of 512MB.

  As regards the Operating Systems used, the distribution server and primary server were executed on *Linux Suse*, the client on *Windows* and the distribution servers were executed on different Operating Systems (*Linux/Windows*) depending on the requirements of each algorithm handled.

- In the Local Configuration, all the servers and the client operations were executed on one machine with a 2.6 GHz *Intel* processor and RAM memory of 512MB. In this case, the evaluation of all the components of ROS was carried out on *Windows* and *Linux* separately, depending on the the algorithm requirements.

The algorithms executed by means of these evaluations are very diverse in nature and they are implemented in different programming languages, which are described in the following list:

- *Steady State Algorithm* [23]. Implemented in *Java* for Operating System *Widows/Linux*. Genetic Algorithm [9] with the main characteristic that not many individuals are replaced in each generation (typically only one or two). It solves the *OneMax* [29].

- *Cellular Evolutionary Algorithm* [24]. Implemented in *Modula 2* for Operating System *Windows*. Genetic Algorithm that uses a special disposition for individuals in a simple population. These individuals are strongly connected. In this case a *Steady State* process is carried out on each population in parallel. It solves the *Rastrigin* [30] problem.

- *Distributed Evolutionary Algorithm*[25]. Implemented in *Modula 2* for *Windows* Operating System. The Genetic Algorithm is composed of an interconnected sub-algorithm (weaker than cellular case) which handles several sub-populations (>>1) each one. The algorithm used is made up of several distributed *Steady State* sub-algorithms. It solves the *Rastrigin* problem.

- *Evolutionary Strategy* [26]. Implemented in *C++* for *Linux* Operating System. Evolutionary algorithm that works on a population of individuals which belongs to a real numbers domain, and by means of mutation and a recombination process evolves to get the optimal objective function. It solves the *Rastrigin* problem.

- *Simulated Annealing* [27]. Implemented in *C++* for *Linux* Operating System. *Simulated Annealing* is a classic Meta-heuristic algorithm, and is based on a local search that allows ascending movements in order to avoid being caught prematurely in a local optimum, using for this reason a function of temperature.It solves the *OneMax* problem.

- *Genetic Algorithm (Mareas)* [28]. Implemented in *C++* for *Windows* Operating System. Genetic algorithm based on *Steady State* selection for the temporal series prediction. Due to its nature, this algorithm uses a great amount of data in its process. It solves the problem of *Temporal Series Prediction*.

The evaluation of these algorithms depends on the complexity of the problems solved. Specifically the next three kinds of problems are used:

- *OneMax* (or *BitCounting*). It consists of maximizing the number of *ones* in a bit chain. the objective is to find a bit chain $\vec{x} = \{x_1, x_2, \ldots, x_N\}$, with $x_i \in \{0, 1\}$, that maximizes the next equation:

$$F(\vec{x}) = \sum_{i=1}^{N} x_i \tag{1}$$

- *Rastrigin*. Consists of finding the global minimum within a search space with a great number of local minimums. Due to the search space dimensions and the great number of local minimums, *Rastrigin* is an extremely complexity function ($\Theta(n \cdot \ln(n))$ where n is the problem dimension).

$$F(\vec{x}) = A \cdot n + \sum_{i=1}^{n} x_i^2 - A \cdot \cos(\omega \cdot x_i) \tag{2}$$
$$A = 10 \ ; \ \omega = 2 \cdot \pi \ ; \ x_i \in [-5.12, 5.12]$$

  The evaluation of this function is fixed to external variables A and w, which control the amplitude and modulation frequency respectively.

- *Temporal Series Prediction*. The target is to predict the evolution of a tide, according to a consecutive sample of hours. A sample of values of 24 hours consecutively is introduced, to predict the values of the next hour (25).

The individuals will be values of 50 elements:
$\vec{x} = \{(c\_sup1, c\_inf1, \ldots, c\_sup24, c\_inf24, prediction, error)\}$
This pattern indicates to us that for a sample of 24 consecutive hours *(h1, ..., h24)* if all of i is fulfilled so that $c\_infi < hi < c\_supi$ then the next hour will be *prediction* with an approximate error of *error*.

Tables 2 and 3, show the results with respect to the *execution time* of the algorithms and the *communication time* from the client initiates the execution when it receives the results. The values are calculated based on the arithmetic average and the standard deviation on a set of fifty evaluations in each case. [1]

As can be observed in these evaluation tables, the execution times averages (t.e aver) are generally loser in Local Configuration than LAN configuration of the system, due to the difference of processor power where the algorithms were executed. However, there is an exception in the genetic algorithm *Mareas* execution time, which requires a greater amount of resources and its Local configuration execution must coexist with the other ROS process (client, distribution server, primary server and worker server).

For this specific algorithm this involves, the execution time in the Local configuration being longer than in the LAN configuration, where the algorithm is executed in a remote server without an external process load.

Algorithm/ features	O.S	Prog Len	e.t aver	e.t std-dev	c.t aver	c.t std-dev
Steady State Algorithm	Win Linux	Java	296.2800 ms	68.9493 ms	1.51e+003 ms	549.919 ms
Cellular Evolutionary Algorithm	Win	Modula 2	50.4200 ms	13.2574 ms	446.4000 ms	294.8466 ms
Distributed Evolutionary Algorithm	Win	Modula 2	2.08e+003 ms	129.1549 ms	370.8800 ms	147.1280 ms
Evolutionary Strategy	Linux	C++	3.44e+003 ms	92.8843 ms	52.8476 ms	256.076
Simulated Annealing	Linux	C++	126 ms	43.5328 ms	264.6400 ms	28.9191 ms
Genetic Algorithm (Mareas)	Win	C++	1.48e+005 ms	699.41 ms	652.42 ms	139.70 ms

Table 2: Table of ROS evaluations using LAN configuration.

[1] e.t aver (arithmetic average of execution time), c.t aver (arithmetic average of communication time), e.t std-dev (standard deviation of execution time), c.t std-dev (standard deviation of communication time), Prog Len (programming language).

Algorithm/ Features	O.S	Prog Len	e.t aver	e.t std-dev	c.t aver	c.t std-dev
Steady State Algorithm	Win Linux	Java	201.760 ms	18.0832 ms	995.960 ms	61.3501 ms
Cellular Evolutionary Algorithm	Win	Modula 2	139.840 ms	40.1763 ms	339.020 ms	40.298 ms
Distributed Evolutionary Algorithm	Win	Modula 2	3.63e+003 ms	61.2494 ms	355.10 ms	83.912 ms
Evolutionary Strategy	Linux	C++	2 ms	0 ms	2.53e+003 ms	48.1349
Simulated Annealing	Linux	C++	1 ms	0 ms	1.45e+003 ms	42.1422 ms
Genetic Algorithm (Mareas)	Win	C++	1.98e+005 ms	1.55e +004 ms	1.05e+003 ms	729.704 ms

Table 3: Table of ROS evaluations using Local configuration.

From the point of view of communication time (t.c aver) a greater difference can be observed. The communication time required in the LAN configuration for Steady State, Cellular and Distributed evolutionary algorithms evaluated in the tables above, is greater than in the Local configuration of ROS. This is logical because in the LAN configuration, communication takes place among processes which are to be found at different locations in a network, while in Local configuration all processes run in the same machine.

Nevertheless, the three last algorithms in the tables (Evolutionary Strategy, Simulated Annealing and G.A Mareas) shows greate communication time than the LAN configuration. Possibly this is because the communication time includes XML analysis pre-processes and file writing, which increases the communication time in Local configuration since it requires a larger amount of processing.

The next graphics (Figure 7 and 8) illustrate the computation time (LAN and Local configuration respectively) obtained in 50 tests carried out on the algorithms describe above.

In addition to what has been commented on previously about execution time, it is shows in these graphics show a longer deviation in execution times of the LAN configuration. This is due to the fact that the executed algorithms must coexist with others independent of ROS applications, unlike Local configuration where only ROS processes are in executed.

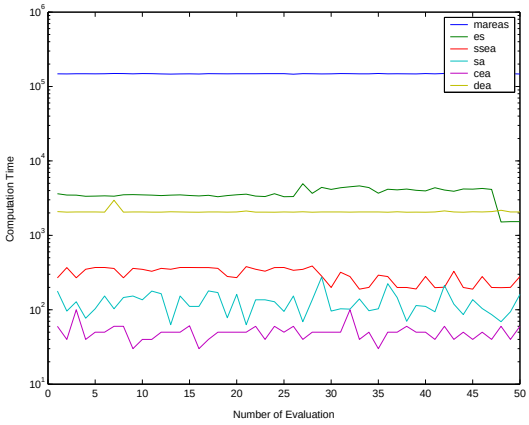

Figure 7: Evaluations graphic: computation time in the LAN configuration of ROS.

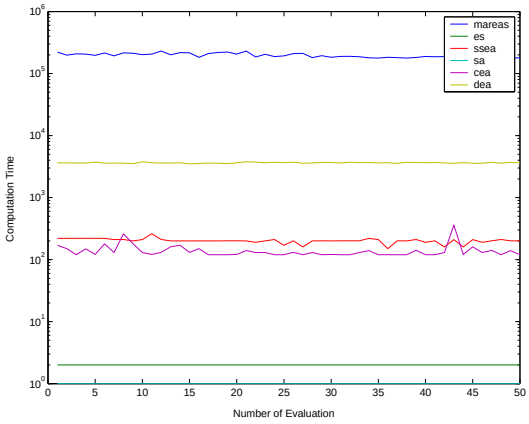

Figure 8: Evaluations graphic: computation time in the Local configuration of ROS.

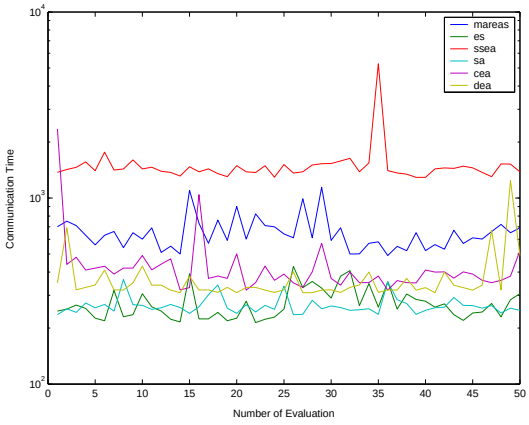

Figure 9: Evaluations graphic: communication time in LAN configuration of ROS.

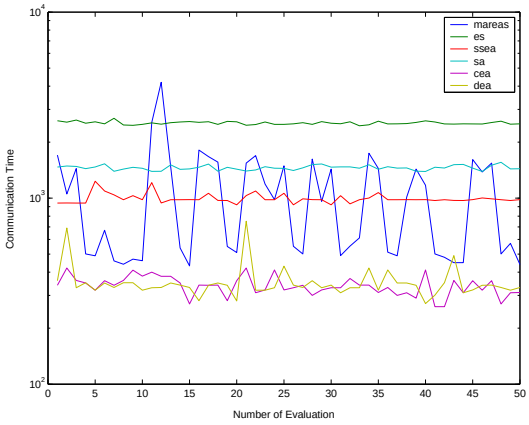

Figure 10: Evaluations graphic: communication time in Local configuration of ROS.

The graphics in figures 9 and 10 show the results of the algorithm evaluations with respect to the communication time. From this point of view, the graphics show a longer deviation in LAN configuration results, since the communication between processes depends on the traffic and congestion of network conditions.

One of the most important issues when studying the ROS behavior in a network is to measure the flow of data that arises in the exchange of information between the different system components.

This way, we can see the interactions between several ROS processes as soon as the flow of data through the network, in addition to its coexistence with other applications that use the same machines and communication channels.

In figures 11 and 12, graphics (generated by "Etherape" [31]) are shown on the traffic and direction of data that the ROS processes exchange in a typical evaluation through a LAN.

The servers are available in a LAN with all of the workers running on internal machines, the primary server and distribution server staying in a machine which os accesible from outside, since this is the one that offers service to the clients. The machine addresses in which the processes are executed are:

- *Client* ROS: `c24.lcc.uma.es`.
- *Primary Server*: `mallba11.lcc.uma.es`.
- *Distribution Server*: `mallba11.lcc.uma.es` (external address) y `192.168.0.17` (internal address).
- *Process Server (Worker)*: `192.168.0.13` (Linux) y `192.168.0.16` (Windows).

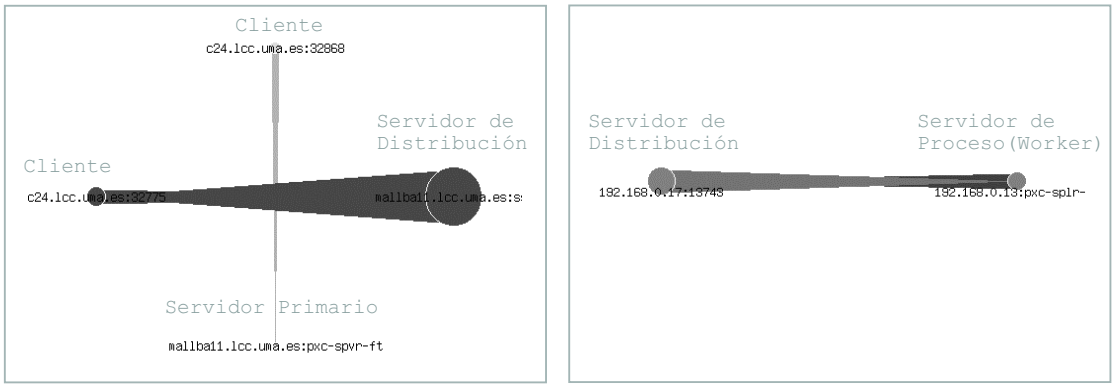

Figure 11: Traffic generated in a ROS execution with respect to the TCP protocol.

In figure 11 on the left, it is possible to see the traffic generated by the information exchange carried out between the client and primary server (vertical flow) and between the client and distribution server (horizontal flow). The heaviest outline belongs to the communication between the distribution server and the client, specifically about the sending of an XML file generated initially in the distribution server. However, the communication between the client and the primary server gives off a thinner outline (vertical), as soon as this exchange of data belongs to the information generated by the users identification and algorithm selection operations, which require a smaller volume of data.

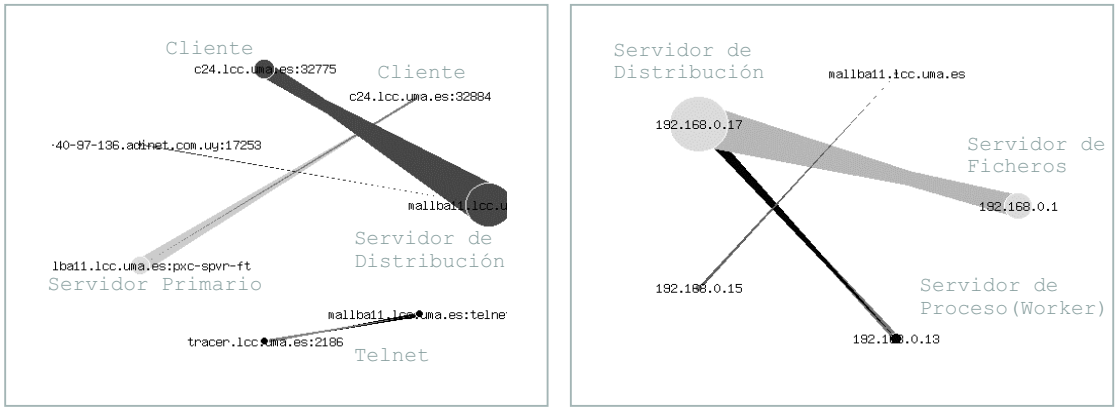

Figure 12: Traffic generated in a ROS execution with respect to the IP protocol.

In figure 11 on the right, it is possible to see the flow between the distribution server (once the client request has been made) and the process server selected, which executes the algorithm and returns the results. In this case, the thick outline is similar in both directions since the amount of data exchange is practically the same through the use of an XML file.

In this representation the communication with respect to the TCP protocol has been into account. In figure 12 the same operation representing the graphic with respect to the IP protocol can be seen. Now, the traffic generated by other applications that coexist in the same environment in which ROS ran is present.

The figure 12 on the left is similar to its comparable one in figure 11, however a flow which is independent of ROS appears because of a Telnet execution. In the figure 12 on the right, the distribution server (internal IP `192.168.0.17`) carries out the request to the process server located in `192.168.0.13`, as well as a reading from `192.168.0.1` which is the LAN file server (independent if ROS).

Although in this example, the coexistence with other applications which also work on the network is not detrimental to the operation of ROS, in other conditions where applications require the exchange of a greater volume of data (FTP for example), the speed of communication be negatively influenced.

Finally, it is possible to indicate that as well as configuring ROS in order to work in a local or distributed way, it is also possible to configure this service to work through a WAN and the Internet inclusive. It is simply necessary to install the required components in several machines located in remote networks (as shown in figure 13), and to set the corresponding entries (machine address and types of algorithms) in the configuration files.

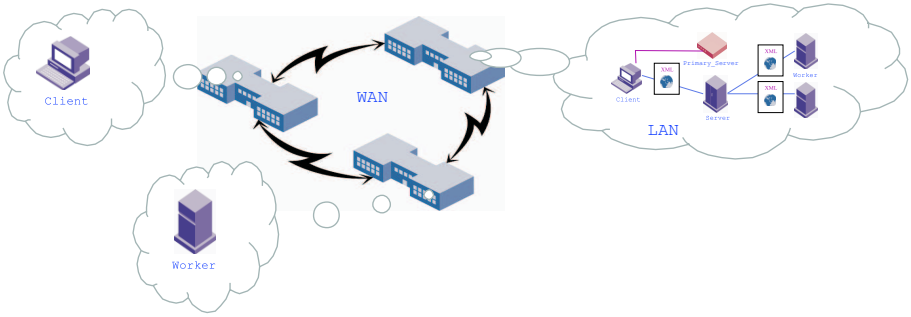

Figure 13: LAN/WAN ROS configuration.

In order to obtain the necessary software to work with ROS, a web site [1] was created. In addition it is possible to test the system by connecting to the ROS active servers whose addresses are configured in that service.

9 Conclusions and Future Work

In this document a general description of ROS was presented, this is a service that provides the access and remote execution of heuristic algorithms via the Internet. It is based on a client/server architecture and joined to the `optimization_algorithm` XML specification developed for this system. In addition, it offers functionalities that facilitate the utilization of the service, as much for final users (by means of the client graphic interface), as the algorithm developers (using the *Wrappers* tool).

Due to the wide range of algorithms that the service aims to deal with, and the wide number of different parameters that each one requires, it is necessary to generalize the XML specification and with this the functionality of the service. This is reflected in the necessity to generate a kind of XML file and a `Wrapper` sub-class for each algorithm, with the inherent risk of making maintenance complicated.

In this sense, in future phases of this project, methods will be developed to speed up the incorporation of new algorithms into the system, by means of the imposition of restrictions in the development of these algorithms. A class will be implemented to manipulate groups of algorithms and to incorporate new graphic interfaces in order to facilitate use by developers and users.

References

[1] ROS, Remote Optimization Service. Available in `http://tracer.lcc.uma.es/ros/index.html`.

[2] E R Harold, XML Bible, IDG Books, 1999.

[3] B McLaughlin, Java and XML, O'Relly, 2001.

[4] R Englander, Java and SOAP, O'Reilly, 2002.

[5] Web site of Apache Web Server. Available in `http://www.apache.org`.

[6] A Pew, Instant Java, The Sunsoft Press - Pretince Hall, 1996.

[7] J.G. Nieto, E. Alba, The WebStream Class, March 2005.

[8] J.J. Merelo, J.G. Castellano, P.A. Castillo, y G. Romero, Algoritmos Genéticos Distribuidos Usando SOAP, published in Actas XII Jornadas de Paralelismo, Universidad Politécnica de Valencia, 2001, pp. 99-104.

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

[10] E. Alba, Análisis y Diseño de Algoritmos Genéticos Paralelos Distribuidos, Tesis Doctoral, Universidad de Málaga, Marzo 1999, Capítulo 1, pp. 2-14 .

[11] Christian Veenhuis , Katrin Franke , y and Mario Köppen. A semantic model for evolutionary computation. En Proceedings IIZUKA, 2000, pp. 68-73.

[12] EAML web site. Available in `http://vision.fhg.de/~veenhuis/EAML/intro.html`.

[13] Juan J. Merelo-Guervós . OPEAL, una librería de algoritmos evolutivos en Perl, published in Actas del Primer Congreso Español sobre Algoritmos Evolutivos y Bioinspirados, AEB'02, Mérida, Febrero 2002, pp. 54-59.

[14] Juán J. Merelo-Guervós , Pedro Ángel Castillo Valdivieso, y Javier García Castellano. Algoritmos evolutivos P2P usando SOAP . Published in Actas del Primer Congreso Español sobre Algoritmos Evolutivos y Bioinspirados, AEB'02, Mérida, 2002, pp. 31-37.

[15] Vladimir Filipovic. Evolutionary Algorithm Web Service. Available in `http://www.matf.bg.ac.yu/~vladaf/EaWeb/index_e.html`. June 2002.

[16] David Gjerdingen. XmlMen Project Report. Available in wiki `http://tyvex.mrs.umn.edu/twiki/view/CSci4553/XmlMenProjectReport`. May 2002.

[17] Andreas Rummler. EALib - a Java Evolutionary Computation Toolkit. Available in the autor web site `http://www.evolvica.org/ealib/index.html`. 2002.

[18] P. Bosman. Evolutionary Algorithm Visualization - EAVisualizer GUI. Available in `http://www.cs.uu.nl/people/peterb/computer/ea/eavisualizer/EAVisualizer.html`. 2001.

[19] Evolvica, Department of Electronic Circuits & Systems of the Technical University of Ilmenau. Available in `www.evolvica.org`. 2002.

[20] Eclipse web site. Available in `http://www.eclipse.org`.

[21] E. Alba, J. G. Nieto. Optimization Algorithm DTD v.2, Data Tipe Document. Available in `http://tracer.lcc.uma.es/ros/documents/optimization_algorithm.dtd`, 2004.

[22] E. Alba, J. G. Nieto, A. J. Nebro. On the Configuration of Optimization Algorithms by Using XML Files, Tecnical Report. Available in `http://tracer.lcc.uma.es/ros/documents/xml-config.pdf`. Marth 2003.

[23] Frank Vavak and Terence C. Fogarty.: Comparison of Steady State and Generational Genetic Algorithms for Use in Nonstationary Environments,Faculty of Computer Studies & Mathematic, University of the West of England, Bristol BS16 1QY, UK, 1996, 1-3.

[24] Alba E. and Troya J.M.: Cellular Evolutionary Agorithms: Evaluating the Influence of Radio, Proceedings of the Parallel Problem Solving from Nature VI, Schoenauer M. et al, 2000, 29-38.

[25] E. Alba and J.M. Troya.: An Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands, Parallel and Distributed Processing, 1999, 1-5.

[26] T.Back and F.Hoffmeister and H.Schewefel.: A Survey of Evolution Strategies, Dpt. of Computer Science XI, University of Dortmund, D-4600 , Dortmund 50, Germany, 1991.

[27] Theodore W. Manikas and James T. Cain.: Genetic Algorithms vs. Simulated Annealing: A Comparison of Approaches for Solving the Circuit Partitioning Problem, University of Pittsburgh, Dept. of Electrical Engineering, 1997, 1-4.

[28] P. Isasi, C. Luque, J.Hernandez, J. Valls.: Predicciones de series temporales (mareas de Venecia) mediante algoritmos genéticos, Tecnical Report of Tracer project. Available in `http://tracer.lcc.uma.es/problems/tide/tide.html`.

[29] J.D. Schaffer and L.J. Eshelman.: Proceedings of the 4th International Conference on Genetic Algorithms: On Crossover as an Evolutionary Viable Strategy, R.K. Belew and L.B. Booker, Morgan Kaufmann, 1991, 61-68.

[30] A. Törn and A. Zilinskas.: Global Optimization: Global Optimization, 1989, volume 350.

[31] Etherape, a Graphical Network Monitor. Available in `http://etherape.sourceforge.net/`.