Noname manuscript No.
(will be inserted by the editor)

Service Oriented Evolutionary Algorithms

P. García-Sánchez · J. González · P. A. Castillo · M. G. Arenas · J. J. Merelo

Received: date / Accepted: date

Abstract This work presents a Service Oriented Architecture for Evolutionary Algorithms (SOA-EA), and an implementation of this architecture using a specific technology (called OSGiLiath). Service Oriented Architecture is a computational paradigm where users interact using services to increase the integration between systems. The presented abstract architecture is formed by loosely coupled, highly configurable and language-independent services. As an example of an implementation of this architecture, a complete process development using a specific service oriented technology is explained. With this implementation, less effort than classical development in integration, distribution mechanisms and execution time management has been attained. In addition, steps, ideas, advantages and disadvantages, and guidelines to create service oriented evolutionary algorithms are presented. Using existing software, or from scratch, researchers can create services to increase the interoperability in this area.

Keywords Evolutionary Algorithms · Service Oriented Architecture · Service Oriented Science · Web Services · Interoperability · Distributed computing

1 Introduction

Research in Service Oriented Architectures (SOA) (Papazoglou and van den Heuvel, 2007) is an emerging field, as can be seen in Figure 1, obtained from the search terms *"service oriented OR service-oriented"* in the Scopus [1] database. Each year more papers about the area are published. This area seeks to promote services usage and adoption, and to improve the way to use them. For example, solving a problem combining existing services in an automatic way (Moussa et al., 2010).

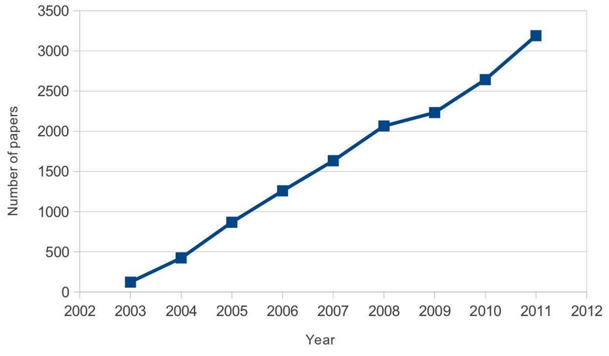


Fig. 1 Number of published papers (per year) about SOA (obtained from Scopus database)

Service Oriented Architecture is a computational paradigm where the users interact with each other using the concept of *service*. A service is a distributed entity (such as node, program, function), used to obtain a result, increasing the integration of heterogeneous systems (several operating systems, protocols or languages) due to this multi-platform nature. The service users do not need to know the language used to implement the service, and they are not forced to use a specific technology to access that service. For example, an evolutionary algorithms researcher could have access to a fitness function made publicly available by another researcher at the other side of the world without even

P. García-Sánchez
Dept. of Computer Architecture and Computer Technology, E.T.S. Ing. Informática y Telecomunicación and CITIC-UGR
University of Granada, Granada, Spain
E-mail: pgarcia@atc.ugr.es

[1] `http://www.scopus.com`

knowing which programming language has been used to implement it.

With the advancement of the Internet, new scientific communities, based on interoperable and distributed platforms are emerging. These communities allow scientist to collaborate in their research, sharing data and remote access to their programs. To achieve this, they use SOA, due to standards usage. Users publish and use flexible, interoperable and configurable services. These services can be created from scratch or by leveraging existing software. Foster (2005b) defines the term "Service Oriented Science" as the pursuit of scientific research using distributed and interoperable networks, being the uniformity of these interfaces the key to success. Thanks to it, researchers can discover and access the services without developing specific access for each data source, or program. Therefore, this paradigm has the potential to increase the scientific productivity due to these public and distributed services, and also to increase the data analysis automation in computing. There are many examples that attempt to boost this paradigm, like Open Science Grid (Altunay et al., 2011) and GLOBUS (Foster, 2005a). These projects are scientific communities and globally distributed infrastructures that support scientific and integrated applications of different domains. It is necessary to remark that services implementation technology is not the great challenge in SOA (neither the specific technology presented in this work). Its main goal is to increase the effort to migrate the existing work and to change the mind of researchers and practitioners.

The Evolutionary Algorithms (EAs) research area should adapt to SOA due to several reasons: this kind of algorithms usually require a lot of computational load, and they are inherently distributable and very configurable, due to the composition of a large amount of existing operators. For example, if the population is a service instead a normal data structure, it could be remotely accessed or undergo a change of its internal structure without modifying the code that implements it. The same happens with the rest of elements of an EA (such as *mutation* or *crossover*): they could be distributed and implemented in several languages, facilitating the modification and usage of the algorithms.

Moreover, there exist a lot of EAs applications available, but in general, they are incompatible with each other. Some of them use some kind of format to define the algorithm (Guervós et al., 2003), but the communication protocols are not usually well-defined.

Many researchers are wasting time in re-programing existing operators to be used in their frameworks, and these operators are also incompatible outside them (due to the programming language used, for example). If the EA developers use SOA they could make profitable the development effort: they could re-use existing components and reconfigure their applications in real time. Moreover, with minor changes, these services could be accessed remotely by another researchers or could be easily integrated in another frameworks, even with different programming languages.

In Gagné and Parizeau (2006), six criteria for qualify EA frameworks were presented: generic representation, fitness, operator, model, parameters management and configurable output. In this work we show how SOA follows these lines of genericity, but can also extend them:

- Genericity in the service interfaces: service interfaces are established to create new implementations. Furthermore, these interfaces must be abstract enough to avoid their modification.
- Programming language independence: for example, services implemented in Java can use services implemented in C++ and vice-versa.
- Distribution transparency: it is not mandatory to use a specific library for the distribution, or modify the code to adapt the existing operators.
- Flexibility: easy to add and remove elements to use the self-adaptation or other mechanisms.

As there is no general agreement over how to define interfaces to allow interoperability among researchers in this area, this work presents a possible architecture to implement Evolutionary Algorithms within a SOA. Using this architecture as a base, the EA researchers could integrate their algorithms with services and computation nodes over the world, increasing the development and capabilities of these algorithms. As an example, an implementation of the proposed architecture using a concrete SOA technology is presented.

With this implementation our aim is to prove that:

- The usage of services for developing EAs requires less effort in integration and code editing, with a lower number of lines of code.
- The distribution mechanisms for parallel EAs do not add extra source code, and can be modified without re-compiling the source.
- New services can be added in execution time and can be accessed independently of the programming language used to implement them.

The rest of the work is structured as follows: next, the State of the art in SOA and existing EA frameworks is presented. In Section 3, our SOA for EAs (SOA-EA) is explained, with a guide for creating services for Evolutionary Computation. Next, an implementation of these services using a specific SOA technology (OS-

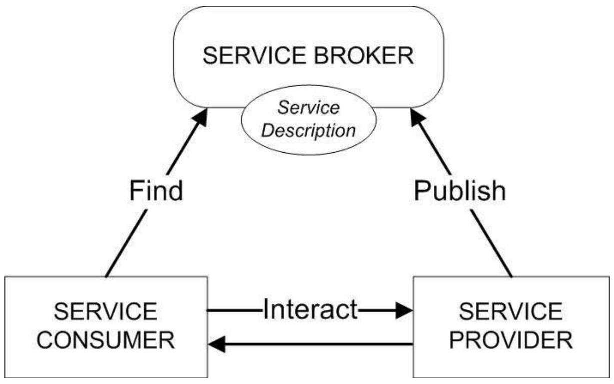


Fig. 2 Service interaction schema. The service provider publishes a service description that is used by the consumer to find and use the service

Giliath) is shown. Finally, the presented work and the conclusions are discussed (Section 6).

2 State of the Art

Figure 2 shows the basic interaction among services. First, the *service provider* exposes the service, publishing its interface in the *service broker*. The *service consumer* (or *requestor*) finds in the broker a service to be used and receives its interface. Then the request is performed by the *consumer* (which uses or consumes the service).

Moreover, several implementations of a specific service can exist (in one or several machines). The broker can choose which one to use each time, or offer another if a service is unavailable. Implementations may also have a different behavior, so the researcher can take advantage to create an auto-adaptive algorithm to select different implementations according to some criteria. Figure 3 shows this special interaction, where two different implementations of an operator interface exist (even using different languages) and the broker has chosen one of them.

The service broker in a SOA can be implemented in several ways and have different behaviors: for example, the implementations of the services to be used can be defined in a text file (if the services do not change in execution time). However, the broker can also assign implementations to interfaces in an automatic way, or using several rules, for example, to distribute a fitness between several machines activated while the algorithm is running.

An important SOA capability is that it is not focused on a specific implementation, but offers a set of guidelines to help the developers. In (Arsanjani et al., 2008) these guidelines and good practices, and also the differences between SOA and Object Oriented Programming (OOP) are explained: the main difference between SOA and imperative programming or OOP is the order of service execution. This order is not necessarily static, because the services are designed to be used in a non-established and configurable order. Furthermore, another important difference is that services can be dynamically discovered and used (while in OOP must be previously known and can not change during execution), being also one of the most important capabilities the (optional) distribution in a network. Finally, in OOP the programming language must be the same for each method call.

There exist many SOA implementations, being the most extended OSGi (OSGi Alliance, 2010b) and Web Services (Papazoglou and van den Heuvel, 2007).

OSGi (Open Service Gateway Initiative) (OSGi Alliance, 2010b) defines a SOA specification for virtual machines, like the Java Virtual Machine. In OSGi, the service providers just implement Java interfaces (the *service description* in Figure 2) with several implementations and the *service consumers* choose this implementations in several ways using SOA techniques. Thus no specific code is needed, and new services can be added and be discovered dynamically (even in a network).

Although SOA is the most relevant part in OSGi, it also includes other beneficial features (OSGi Alliance, 2010a), like package abstraction, life-cycle management, packaging or versioning. This reduces the complexity in the development, support and deployment of applications due to its plug-in based development. In Wagner et al. (2007) the definition and advantages of using these techniques in metaheuristic systems are explained. The benefits of the dynamism in OSGi can be used in dynamic EAs: the applications in OSGi can register, obtain, filter or wait for services (such as operators or fitness functions) that appear or disappear and the programmer does not need to write code for these actions.

The other most extended SOA technology are *web services*. A web service is a service whose interface is a WSDL file (*Web Services Description Language*, equivalent to the *Service Description* in Figure 2). The communication is performed with the interchange of messages described with the *Simple Object Access Protocol* (SOAP).

There exist many tools to convert existing code to SOA without too much difficulty, like Axis and JAX-WS[2] for Java, gSOAP [3] for C++, o ZSI [4] for Python, among others.

[2] `http://jax-ws.java.net/`
[3] `http://gsoap2.sourceforge.net/`
[4] `http://pywebsvcs.sourceforge.net/`

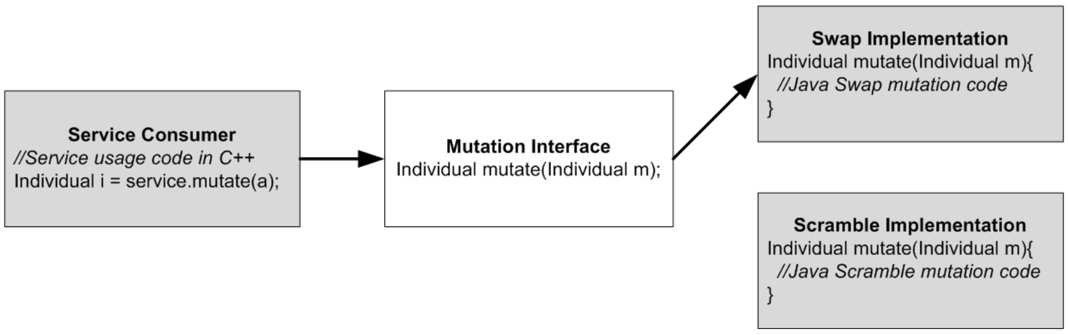

Service Consumer
//Service usage code in C++
Individual i = service.mutate(a);
Mutation Interface
Individual mutate(Individual m);
Swap Implementation
Individual mutate(Individual m){
//Java Swap mutation code
}
Scramble Implementation
Individual mutate(Individual m){
//Java Scramble mutation code
}

Fig. 3 Example of usage of a service implementation

Web services are also used in the GRID area for optimization problems, as can be seen in the works of Cox et al. (2001); Song et al. (2003, 2004); Jiao et al. (2004), where services are defined using WSDL interfaces and other transmission mechanisms (such as Remote Procedure Call (Ho et al., 2004; Xue et al., 2004)). Although there also exist EAs to be executed in GRIDs (Lim et al., 2007; Ng et al., 2005; Imade et al., 2004)) no information about how to design these services for EAs is provided in previous works.

Even as SOA is used extensively in software development, it is not widely accepted in the main EA software. Firstly, there exist Object Oriented frameworks, like Algorithm::Evolutionary (Merelo Guervós et al., 2010), JCLEC (Ventura et al., 2008) or jMetal (Durillo et al., 2010). Users implement specific interfaces of these frameworks (like *individual* or *crossover*) and they group them in the source code. For example, creating an operator object that groups several operators. However, these frameworks are not compatible among them. For example, the operators created in JCLEC can not be used in jMetal (despite both are programmed in Java). Also, they can not control the services (operators) outside the source code. Parallelism and distribution are added in other frameworks, like MALLBA (Alba et al., 2006), DREAM (Arenas et al., 2002) or ECJ (Luke et al., 2009), but using external libraries (such as MPI or DRM), so the code that uses these libraries is mixed with the algoritmh's code.

Even being distributed, these frameworks can not communicate with each other. HeuristicLab (Wagner and Affenzeller, 2005) is one of the few plug-in and service oriented frameworks. It uses web services for communication, but just to distribute the load, after consulting a central database of available jobs. Finally, the only service oriented optimization framework is GridUFO (Munawar et al., 2010), but it only allows the modification of the objective function and the addition of whole algorithms, without combining existing services. Table 1 shows a summary of the previous frameworks.

In brief, although these frameworks follows the six criteria for genericity of Gagné and Parizeau (2006), they present some shortcomings when it is needed to develop or add new features: the user is forced to modify the source code or stop the execution to add new functionalities (like load balancing, dynamic control of operators, or an user interface). The authors should improve their frameworks adding SOA technologies in order to facilitate the communication and integration among them. As Parejo et al. (2012) suggest, a standardization of the presented (and other) frameworks should be carried out.

Although all the approaches described above are focused on the implementation of distributed EAs, the abstraction level of each alternative can be quite different, as shown in Figure 4. As SOA is a methodology and not a technology, areas such as Evolutionary Robotics, or EA classic frameworks (i.e. OSGiLiath) can use SOA to be designed and developed. Implementation technologies, such as Web Services are the gap between SOA (abstract) and GRID (infrastructure) where interfaces are designed using SOA principles (dynamism, visibility, loose-coupling and heterogeneity). Finally, Cloud Computing can be seen as a combination that extends SOA adding the scalability of GRID (Jamil, 2009).

Finally, as stated in the introduction, there exist research lines related to the intelligent composition and discovery of services. The service composition is the aggregation of several distributed services in an intelligent way to solve a problem. To achieve this, the services information is used, like the execution cost, information, inputs and outputs and another features (see the work of del Val Noguera and Pedruelo (2008) for a survey). For example, some authors have used metaheuristics like the Genetic Algorithms (GAs) or Particle Swarm Optimization (PSO) to compose service sequences (Fan et al., 2011). So, these service composition techniques can be applied to a Service Oriented Architecture for EAs, like the one presented in this work.

Table 1 Comparison of EA frameworks. OO=Object-Oriented, SO=Service Oriented, PO=Plug-in Oriented

Name	Design	Language	Distribution	License	Other
ECJ	OO	Java	Sockets	Academic Free Lic.	Recently updated
MALLBA	OO	C++	MPI	Freeware	No new versions
jMetal	OO	Java	N/A	GNU/LGPL	
DREAM	OO	Java	DRM	GNU/GPL	No new versions
ParadiseEO	OO	C++	MPI	CeCILL	
HeuristicLab	OO/PO	.NET	Web-Services	GNU/GPL	Recently updated
METCO	OO	C++	MPI	N/A	
JCLEC	OO	Java	N/A	GNU/GL	
Algorithm::Evol.	OO	Perl	N/A	GNU/GPL	
GridUFO	SO	Java	Web Services	N/A	

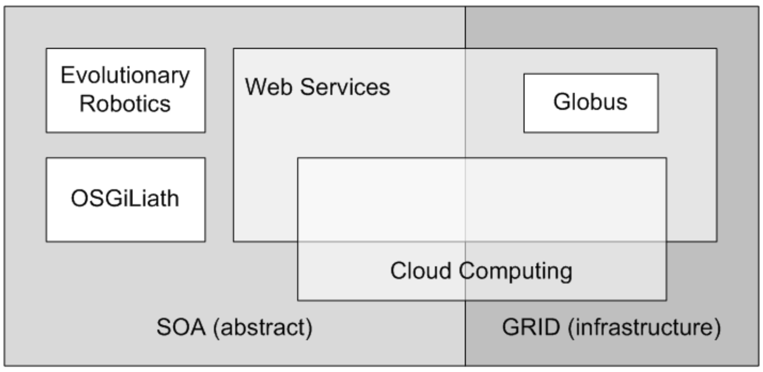


Fig. 4 SOA as abstract paradigm to develop in different EAs areas. Using especific technologies such as Web Services allows GRID integration.

3 Service Oriented Architecture for Evolutionary Algorithms: SOA-EA

The evolutionary algorithms research area is a propitious environment to migrate to SOA for several reasons:

– Firstly, as seen in the previous section, there are a large number of frameworks for EAs. Many of these frameworks are open source, but all of them are incompatible (different programming languages, operating systems or communication protocols).
– New research trends, like self-adaptation (Babaoglu et al., 2005), require many changes and modifications in the algorithms behavior in real-time.
– The increase of technologies like GRID and Cloud Computing (Buyya et al., 2009), where the computation elements are distributed in different machines, with many operating systems and programming languages.

When developing within a SOA, Papazoglou and van den Heuvel (2007) established that the services must be:

– Abstract: many different implementations can use the same interface.
– Well-defined: the interface of the service must be fixed, and it can not change in time, because the consumers or implementations of this interface should be modified with it.
– Encapsulated: services can use other services, but only the interface should be used to consume a service.
– Reusable: services should be designed to be used by as many applications as possible.

Starting from the general scheme (Eiben and Smith, 2005) of an EA, that can be seen in Figure 5, each step (and also the algorithm) can be developed as a service. Thanks to SOA, each part of the EA can be dynamically selected. This increases the flexibility in the development of new EAs or the modification of existing ones.

Figure 6 shows a basic EA that uses specific implementations of the services. Service implementations (gray blocks) are used accessing to their interfaces (white blocks, such as *Initializer* or *Mutator*). The services have been chosen from the schema in the Figure 5 because it is the most abstract scheme of an EA and services must be as abstract as possible. Gray blocks under white blocks are implementations bound with the interfaces in execution time. These implementations are

```
BEGIN
 INITIALISE population with random candidate solutions;
 EVALUATE each candidate;
 REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
   1 SELECT parents;
   2 RECOMBINE pairs of parents;
   3 MUTATE the resulting offspring;
   4 EVALUATE new candidates;
   3 SELECT individuals for the next generation;
 OD
END
```

Fig. 5 General scheme of an evolutionary algorithm in pseudo-code

used accessing to their interfaces (noted with continuous arrows) from other implementations. The implementations binding is not performed in the source code, but with a SOA mechanism (for example, automatically when the implementations are available, or in a configuration file) in a transparent way. Also, these implementations can be in different machines or different languages and the programmer does not need to know it. Although this is a basic example of how to apply SOA to EC, this work explains how to implement and develop it in a deeper way. Also more complicated examples and SOA mechanisms are presented.

As previously stated, Gagné and Parizeau (2006) discuss about the genericity in software tools for evolutionary computation. Although their work is based on Object Oriented Programming, genericity in a framework for EAs can be also applied in the development of a service oriented architecture for this kind of algorithms:

– Individual representation. Almost all services in an EA (like mutation or selection) will accept individuals as input data and produce/modify these individuals. Due to many kind of individuals may exist, the operators should be as abstract as possible to operate properly.
– Fitness evaluation. Each problem should implement an interface of the fitness service that receives the individual, allowing the distribution of this service (instead of being a method in the *individual* class, for example). However, to be more flexible, the *fitness service* must receive a list of at least one individual, to facilitate the parallelism.
– Definition or addition of every type of operator. Thanks to the loose coupling of services, several crossover or mutation implementations can be created. Moreover, new operators can be added in execution time, without re-compiling the existing ones, or combining them according to several parameters, for example.
– Adaptation of the evolutionary model. The user can manually select the services to be combined to create a Genetic Algorithm or an Evolution Strategy, for example.
– Dynamic adaptation of the parameters. Parameters can also be a service, thus the EA developer obtains two advantages: it is not mandatory to distribute the parameters among all services, and also they can be dynamically modified in execution time from an external service, facilitating self-adaptation (Babaoglu et al., 2005).
– Flexible output mechanisms. The developers do not need skills in GUIs (Graphical User Interfaces) or logs programming, because as this kind of services are not coupled to the operators, they can access their information without any modification in the operators.

As previously stated, the fitness should not be calculated within a method of an Individual class. To be less coupled, it should be implemented an an external service that receives a list of individuals (facilitating the load balancing). That way, the service is as abstract as possible. Also the parameters should be a service for the same reason, allowing the possibility of performing experiments related to the parameter control or tuning (Eiben and Smit, 2011) in an efficient way (being separated from the code of the existing operators). Services such as the *recombinator* or the *mutator* should not receive one or two individuals, since not all EAs have the same behavior. They should receive a list of individuals to be crossed or mutated each generation. On the other way, *population* should not be a list of individuals: it should be a service to access the individuals and allow the variation of its structure (for example, a change from an unique population to a distributed island model) without affecting the rest of the pieces of the algorithm. So, other services external to the EA could consult the *population* state and act accordingly to some rules.

Using the previous indications as a base, SOA-EA has been created. SOA-EA is an abstract architecture to develop service oriented EAs, independently of the technology to be used. Table 2 shows some reasons to migrate to SOA and how services in SOA-EA should be designed.

4 Example of SOA-EA usage

This section justifies the design of SOA-EA and the steps to create services within it.

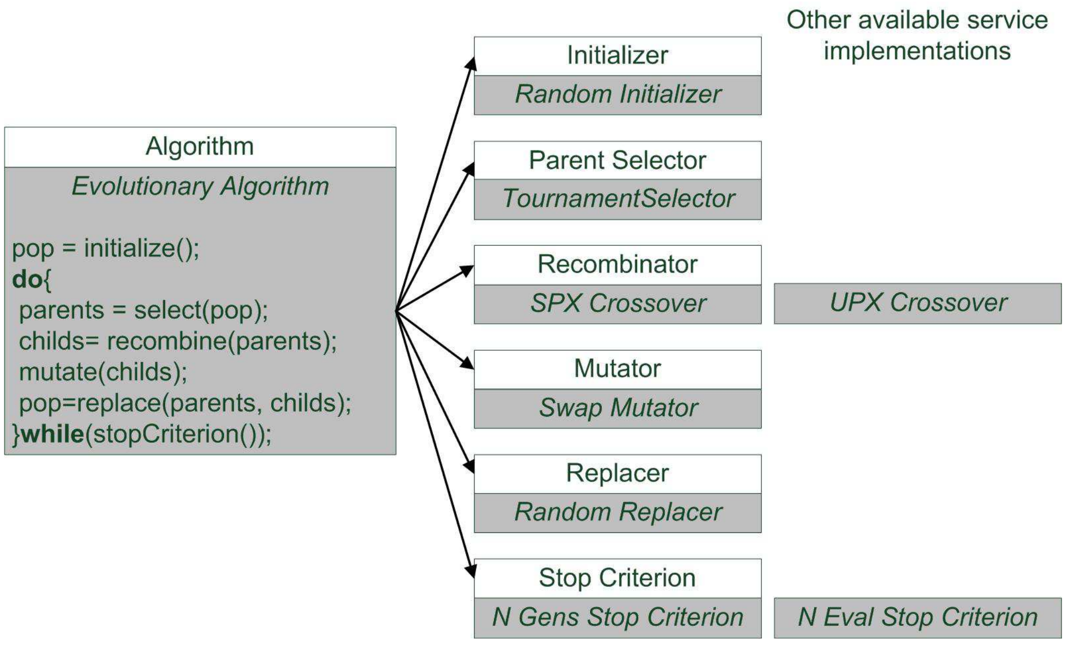


Fig. 6 Basic Service oriented Genetic Algorithm with example operators

4.1 Implementing a basic Genetic Algorithm

As stated above, a basic EA is formed by several steps. These steps are common to every EA, so this part must be fixed to allow the creation of services as abstract as possible. The differences between two EAs are in the operators, selectors or individual representation (as suggested by Eiben and Smit (2011)). Therefore, to accomplish with the genericity presented in the previous section the parameters and operators must be added dynamically. This is done with the SOA service binding. Users can specify the operators they need in several ways, for example, in a configuration file, or in an intelligent manner (an algorithm). It is important to remark that these "pieces" do not need to be modified and compiled again, because the loose coupling and the dynamic binding of SOA. Without SOA this behavior is very difficult to achieve or maintain.

Figure 7 shows a complete service oriented genetic algorithm, taking into account the proposed ideas. In this figure (and in the following ones) white blocks are the service interfaces. Gray blocks are specific implementations of these interfaces (that is, the source-code of the service), and arrows indicate how a service implementation can make use of other services via their interface. For example, almost all implementations access to the *Parameters* service using its interface. Service implementations (gray blocks) can be selected in a configuration file or be automatically bound when they are available (among other options).

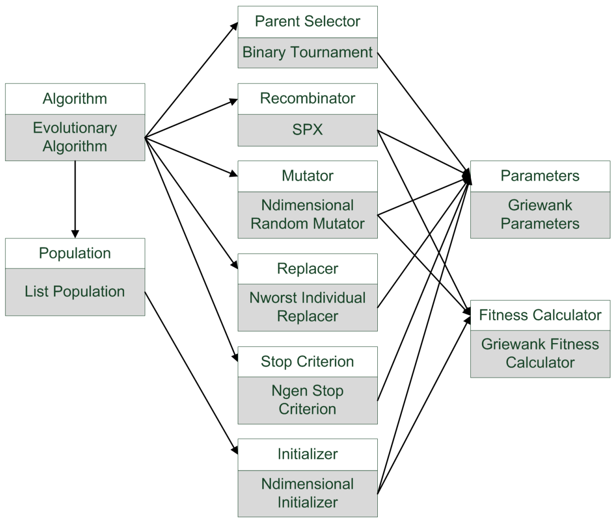


Fig. 7 Basic genetic algorithm. White blocks are interfaces and gray blocks are implementations. In this case, we are using specific implementations to solve the Griewank function problem

The change from a problem instance to another is quite simple. It is only necessary to notify the algorithm a change in the implementation of the service *Fitness Calculator* and the implementation of *Parameters* (because these can vary from a problem to another). Because some algorithms need to calculate the fitness every time an individual is modified (and not only at the end of a generation) the Figure 7 shows how the service *Fitness Calculator* may be used inside the implementa-

Table 2 Summary of migration from traditional EA programming to SOA

Element	**Current EAs development**	**Using SOA**	**Reason to migrate**
Programming language	Just one for all elements of the algorithm	Any	Services are independent of the programming language. Only the interface is required to use services
Operators	Methods or functions	Services	Services allow the selection of a specific implementation during the algorithm execution, and also different programming languages or distribution models
Operators behavior	Methods applied to a single individual	Service that receive individual lists	It allows load balancing and distribution, and also to modify the operators in execution time
Operator selection	Modifying the source code	In a flexible way outside the source code	It is not mandatory to recompile the source code to integrate new operators
Fitness	Method that evaluates an individual	Service that evaluates an individual list	It allows the distribution, load balancing and addition of new fitness calculators in real time
Population	Array or individual list	Population service	It allows to change the population type and topography, by selecting the service implementation
Self-adaptation	Modifying source code for a specific experiment	Self-adapting service that selects specific operator implementations	It does not modify the created services and brings more flexibility in the dynamic adaptation
Distribution	Libraries like MPI	SOA mechanisms	SOA technologies allow changing the transmission protocol and using extra technologies without adding extra code

tions that modify individuals (*Initializer*, *Mutator* or *Recombinator*). Moreover, each service can be in the local machine or distributed on the Internet, having the same behavior.

4.2 Implementing a service oriented NSGA-II

The difference between the previous version of a GA and the well known NSGA-II (Deb et al., 2002) lies in the selection operator. Therefore, to change from the basic GA to NSGA-II, the mutator and crossover are kept and new selection operators are added. Figure 8 shows the service oriented version of NSGA-II algorithm, where the new implementations are marked with a thick border. The problem has also been set to the multi-objective function MOP2 (Huband et al., 2006). New auxiliary services have been added, like *Crowding Distance Assignator* or *Pareto Assignator*. As these services may be used in other algorithms in the future, they must be designed as abstract as possible. These new services are called from the implementation (code) of the services *NSGA-II Replacer* or *Binary Crowding Distance Selector* (black arrows indicate an interface call).

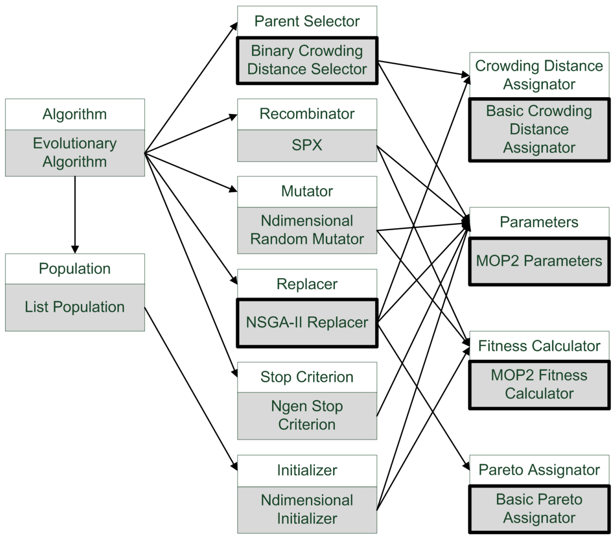


Fig. 8 Modification of the basic GA adding new service implementations (grey blocks with thick lines)

4.3 Adding basic distribution

As every service must keep the same behavior, independently of the machine that hosts it, distribution services for load balancing of a specific service can be easily created, for example, notifying the algorithm to use a distributed implementation for that service. As previously stated, the service *Fitness Calculator* receives a list of individuals to calculate their fitness, so, in this example, the new fitness implementation (*Basic Fitness Distributor*) binds with every fitness service available (in the same machine or in a network). The source code of this basic implementation simply distributes the list of individuals among the bound services and waits for their termination. Although more complex implementations probably will be more efficient, the objective of this section is to show how to distribute services, thus, this basic implementation is sufficient. Figure 9 shows the modification from a sequential fitness calculator to a distributed one. Thanks to SOA, the number of distributed fitness calculators is not fixed: calculators can be added o removed in real time without stopping the system. As can be see in the figure, if one of the nodes is a cluster, it could also implement another fitness distributor. This easy example can be adapted to more complex necessities depending on the infrastructure or the problem to be solved. More complex distribution services can be created, for example, taking into account communication latencies or computation capabilities of the nodes.

One of the most extended model in parallel EAs is the island model. Using SOA-EA, the *Population* service implementation can be modified to become a distributed population. Each certain time, this population could exchange individuals with other populations modified by other algorithms. These populations should be added or deleted in execution time without affecting the algorithm execution. Figure 10 shows this example, where the Island Population implementation maintains a list of references to other Population interfaces (which can be local or remote). Also other Population implementations exist (List Population is the usual list of individuals). If one of these population services drop, the others can continue working. The topology of these islands can also be managed from services (like the Island Population service, or another). The modification and dynamism of the population structure is difficult to apply in existing frameworks without using SOA because it is necessary to create mechanisms to modify the population behavior, the operators to modify it, the data structures, and also the code to manage all. With he usage of SOA, and due to the capability of accessing to a population via its service interface, it is not necessary to modify the source code to modify the population and its behavior. Also, to avoid bottlenecks in distributed executions, asynchronous communication must be provided to avoid idle time. This kind of communication offers excellent performance when working with different nodes and operating systems, as demonstrated by Alba et al. (2002).

4.4 Self-adaptation in SOA-EA

There are several ways to create self-adaptable algorithms using SOA-EA. For example, creating a service that modifies the parameters in the *Parameters* service, or activating and de-activating operators in real time. An easier way is to create a service that manages all available services of the same kind. For example a *Mutator* service that binds all the available mutation implementations and use the most adequate one depending on some rules during the execution (Serpell and Smith, 2010). This idea can also be extended to create a service that implements several interfaces and selects the most adequate implementation for each interface respect to some criteria, as can be seen in Figure 11, where thick lines represent the implementations used at the current moment (they vary as time passes).

Finally, another important usage of EAs is its hybridization with other metaheuristics, to obtain more effective search algorithms (Lozano and Garcia-Martinez, 2010), increasing the performance of intensification and diversification mechanisms. With traditional frameworks this task can be difficult, mainly because the source code for each metaheuristic must be modified. Nevertheless, using SOA a combination of loosely coupled services could be used.

5 Implementation of the proposed architecture

Although the previous examples can be developed using any SOA technology, this section presents OSGiLiath (*OSGi Laboratory for Implementation and Testing of metaHeuristics*), an implementation of SOA-EA based in OSGi. OSGi has been selected for the development of this service oriented architecture instead Web Services attending the following reasons:

- OSGi is faster, because it was designed for lightweight devices (Lim et al., 2008). Therefore, it can be used in embedded devices, like Evolutionary Robotics (Eiben et al., 2010). Web services were created to integrate complex data interchange among different companies.
- The transmission protocol in web services is SOAP, which implies the transmission of an XML (*eXtension Markup Language*) file (World Wide Web Consortium, 2006). This file is usually too large (for example, a complete list of workers in a company). EAs often need to send minimal information, but a large number of times (for example, the fitness of several individuals), so a complex transmission protocol is not recommended. OSGi includes a lot of mechanisms for data transmission, allowing more

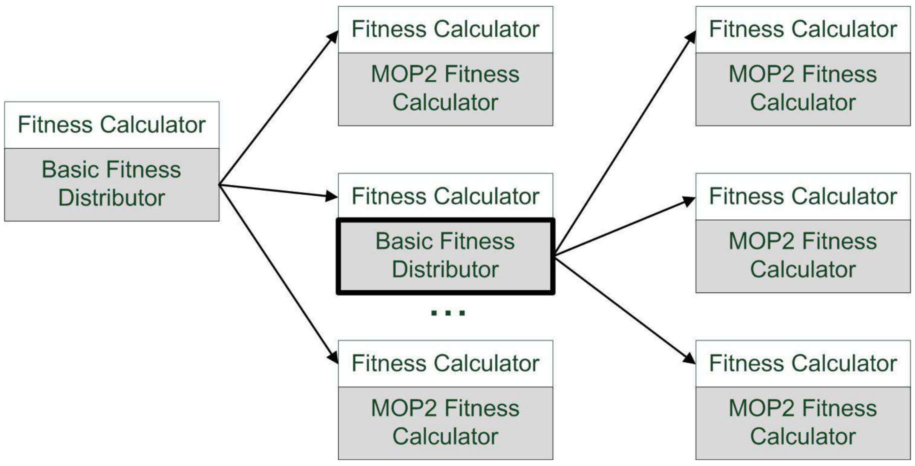


Fig. 9 Fitness distributor. The thick line implementation also re-distribute the individuals

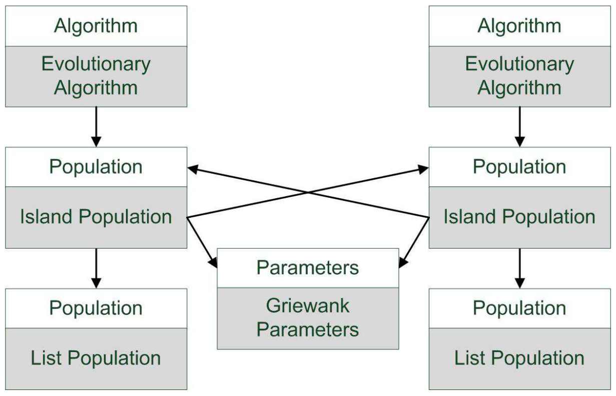


Fig. 10 Island model. From time to time, the implementation *Island Population* notifies the islands to initiate the migration

flexibility depending on the execution environment of the algorithms (for example, in a machine, in a local network, over the Internet, or even in more lightweight devices).
– Unlike web services, OSGi includes a blackboard event-manager, that is, services inform what they are doing without indicating any receiver. Other services can filter this information and actuate accordingly, so the synchronization is easier. For example, it is not mandatory to create a variable to count the number of times that the *Fitness Calculator* service is executed: an external service can track this number.
– Due to the separation between OSGi and the source code of the services, the code of OSGi-based applications can be used in other Java-based applications without OSGi. For the same reason, frameworks written in Java can be migrated into services in a easy way, because no specific code is required to combine code from different programs.
– Finally, OSGi includes other features that, although not related to SOA, facilitate the service development: version and package control, security and life-cycle management of the used components (as explained by Wagner et al. (2007)). These advantages can be used by the EA developers if they work in a team collaboration.

More information about the application of OSGi in other areas, with good practices, benefices and lessons learned are provided by the authors in (García-Sánchez et al., 2012).

The objective of this implementation is to promote SOA benefits and offer these features to programmers:

– Well defined interfaces. As previously stated, service interfaces must be as abstract as possible.
– Asynchronous data sending/receiving. Due to the distribution capabilities offered by OSGi, the presented implementation allows the direct distribution of services, without the need to implement specific functions in the source code, like MPI or other distribution mechanisms. EAs developers can use the existing distribution services or create new ones, if they want.
– Service oriented programming. New improvements can be added without modifying the existing modules, that is, adding or modifying only the affected service implementations without modifying the source code of the other services.
– Server/client or distributed model. All the components of this implementation can communicate in a bi-directional way, so it is not mandatory to use a central server to manage other nodes.

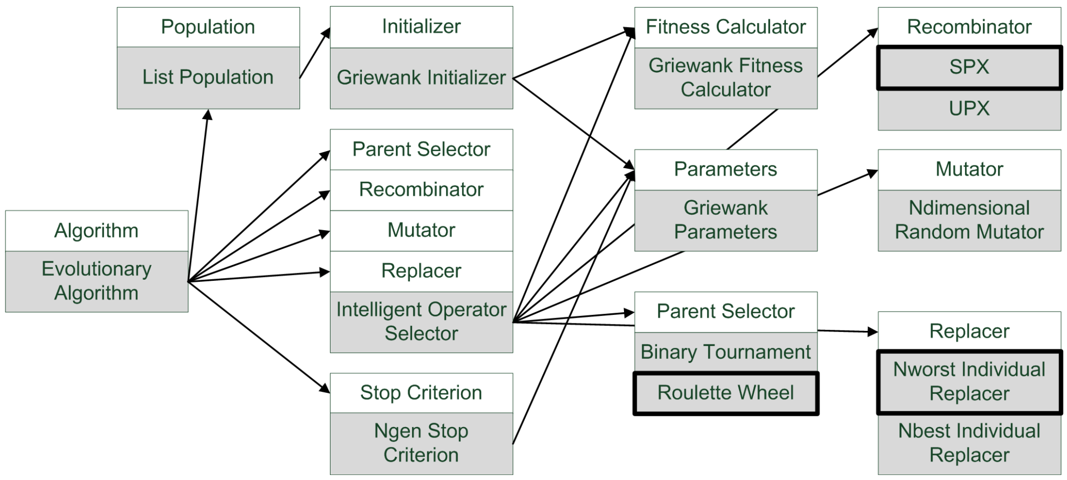


Fig. 11 Self-adaptable Algorithm. The *Intelligent Operator Selector* selects which service implementation is used each time

– Paradigm independent. Although the first developments are focused on EAs, this architecture can be extensible to other kind of metaheuristics.
– Remote event handling. Users can handle a powerful synchronization tool among distributed services using OSGi features.

The source code of OSGiLiath is available in `http://osgiliath.sourceforge.net` under a GNU/GPL license. This code is an updated version of the work published in (García-Sánchez et al., 2010).

5.1 Service development in OSGiLiath

Users have to define three elements to add a new service in OSGiLiath:

– *Service interface.* It is a Java interface. The user just needs to specify the operations that the service will perform.
– *Service implementation.* The programmer just writes the code of the interface methods.
– *Service description.* It is an XML file that indicates which interface is being implemented and which other services needs to be activated.

The presented implementation includes the interfaces defined in section 3, such as *Algorithm*, *StopCriterion*, *Population* or *Recombinator*. These interfaces are grouped with another interfaces that do not need to be a service. For example, the interface of the object *Individual*. This interface is used in the *Recombinator* interface, which receives a list of *Individual* objects to be recombined, and returns another list with the recombined ones. Also, several implementations are included, like *EvolutionaryAlgorithm* (implementing *Algorithm*) or the rest of services explained in previous section, like the services for NSGA-II.

The source code of the method that executes the algorithm in the class *EvolutionaryAlgorithm* (implementation) is shown in Figure 12. It also includes methods to bind the six references to the service implementations that are needed: Population (*pop* in the code), StopCriterion, ParentSelector, Recombinator, Mutator, and Replacer.

This is the code needed by every EA, so it not necessary to modify it. The *Service Description* appears when the service interfaces are bound to execute the service implementation. Each implementation of a service has an XML file indicating which interface is being implemented, and also other properties. This file is used by OSGi to automatically bind the services. The service descriptor of the EA is shown in Figure 13. This file describes that the *EvolutionaryAlgorithm* class is an implementation of the *Algorithm* interface, and that it needs implementations of the interfaces *Population*, *Mutator*, *ParentSelector*, *Replacer*, *StopCriterion* and *Recombinator* to be activated. It should be noted that this file usually can be modified using a friendly GUI, or from an assistant in Java IDEs, such as NetBeans or Eclipse (so, users do not have to care about its XML structure). The user interface to create this file in Eclipse is shown in Figure 14. The interface being implemented is set in the lower part (*Algorithm*). The necessary services to activate this implementation are indicated in the upper part (with the cardinality and functions to set and unset the service implementations in the implementation source code).

This XML file is read by the OSGi execution environment, which is the responsible to bind the available services to this implementation. For example, if

```
//References to the implementations to use
Population pop;
ParentSelector parentSelector;
Recombinator recombinator;
Mutator mutator;
Replacer replacer;

//Example of the method to obtain an implementation
//of the ParentSelector interface
//(one function per reference)
void setParentSelector(ParentSelector sel){
        this.parentSelector = sel;
        //now sel is a reference to an implementation
        //of ParentSelector
}

//Implementation of the start() method of the
//Algorithm interface
public void start(){
  pop.initializePopulation();
  actualIteration = 0;
  do{
    //SELECT parents
    List<Individual> parents
        = parentSelector.select(pop);

    //RECOMBINE parents
    List<Individual> offspring
        = recombinator.recombine(parents);

    //MUTATE offspring
    List mutatedOffspring
        = mutator.mutate(offspring);

    //SELECT new population.
    //pop is modified here
    replacer.select(pop, parents,
        offspring, mutatedOffspring);

    actualIteration++;

  }while(!stopCriterion.hasFinished());

}
```

Fig. 12 Java code of the class *Evolutionary Algorithm*. This class implements the *Algorithm* interface, which defines the operation *start()*

a *ParentSelector* is activated, automatically is bound to the variable *parentSelector* through the function *set-ParentSelector*. The *cardinality* is also set in the file, in this case, only one implementation is necessary (not multiple). This file can be modified in execution time, so it is not required to re-compile the Java code to use and set new services.

In brief, each implementation of a service (<*implementation*>) indicates the interface to being implemented (<*provide interface*>), and the other services this implementation needs (<*reference*>).

Moreover, each service can provide properties to be used by other services to obtain more information and filtering. For example, in this case only the *Replacers* whose property *replacerName=nsga2* are used.

5.2 Managing services: implementing the NSGA-II from the canonical GA

Following the development example showed in Section 4.2, some extra services have been developed to convert the basic GA into a NSGA-II (and have also been added to OSGiLiath to be available for users).

There exist many options for the EA to pick up the appropriate service. The first of them is modifying the source code of the implementations. Obviously this is not recommended, because the service would not be loose coupled due to the specific OSGi code, and this is not a good SOA practice. The following ways makes the service usage not code-dependent:

– De-activating the implementation *Binary Tournament* from the OSGi administration console, and activating the implementation *Crowding Distance Selector* (that is, manually). This technique is not recommended, because all services are then managed by hand, and this is very difficult with a large number of services. However, the OSGi console allows modifying services in execution time, so it can be used in some cases (for example, to stop the service in a machine while another big task is being executed, and activate it again when this task is over).
– Modifying the Service Descriptor of the *Evolutionary Algorithm* implementation to filter the desired implementations (for example, the attribute *target-="(selectorName=nsga2)"* in Figure 13). This option is used when the algorithm is fixed and does not need to be modified in execution time, and the number of operators and types are known in advance. However, as previously stated, new services can be added in execution time (for example, if the cardinality is set to multiple).
– Using an external service that activates or de-activates desired implementations or modify their status. This technique must be used when self-adaptation properties are used in the algorithm, and it is presented in next subsections.

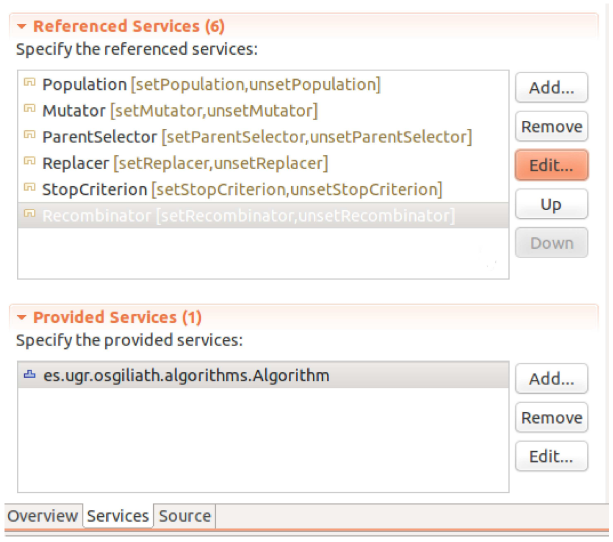
None of these options needs to modify the source code of the existing services: they just indicates which services uses each time.

5.3 Adding distributed capabilities

As previously stated in Section 4.3, one of the main advantages in SOA-EA is that services can be distributed, so the proposed implementation of the architecture should also allow the distribution and load balancing of the EA. In OSGiLiath all services can be distributed using

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" enabled="false"
  immediate="true" name="OsgiliathEvolutionary">
  <implementation class="es.ugr.osgiliath.evolutionary.EvolutionaryAlgorithm"/>
  <service>
    <provide interface="es.ugr.osgiliath.algorithms.Algorithm"/>
  </service>
  <reference bind="setPopulation" cardinality="1..1"
  interface="es.ugr.osgiliath.evolutionary.elements.Population"
  name="Population" policy="static" unbind="unsetPopulation"/>
  <reference bind="setMutator" cardinality="1..1"
  interface="es.ugr.osgiliath.evolutionary.elements.Mutator"
  name="Mutator" policy="static" unbind="unsetMutator"/>
  <reference bind="setParentSelector" cardinality="1..1"
  interface="es.ugr.osgiliath.evolutionary.elements.ParentSelector"
  name="ParentSelector" policy="static" target="(selectorName=nsga2)" unbind="unsetParentSelector"/>
  <reference bind="setReplacer" cardinality="1..1"
  interface="es.ugr.osgiliath.evolutionary.elements.Replacer"
  name="Replacer" policy="static" target="(replacerName=nsga2)" unbind="unsetReplacer"/>
  <reference bind="setStopCriterion" cardinality="1..1"
  interface="es.ugr.osgiliath.evolutionary.elements.StopCriterion"
  name="StopCriterion" policy="static" unbind="unsetStopCriterion"/>
  <reference bind="setRecombinator" cardinality="1..1"
  interface="es.ugr.osgiliath.evolutionary.elements.Recombinator"
  name="Recombinator" policy="static" unbind="unsetRecombinator"/>
  <property name="algorithmName" type="String" value="EvolutionaryAlgorithm"/>
</scr:component>
```

Fig. 13 Service descriptor of the Evolutionary Algorithm implementation. Figure 14 shows the friendly user interface to automatically create this file using the Eclipse program


Referenced Services (6)
Specify the referenced services:

Population [setPopulation,unsetPopulation]
Mutator [setMutator,unsetMutator]
ParentSelector [setParentSelector,unsetParentSelector]
Replacer [setReplacer,unsetReplacer]
StopCriterion [setStopCriterion,unsetStopCriterion]
Recombinator [setRecombinator,unsetRecombinator]

Provided Services (1)
Specify the provided services:

es.ugr.osgiliath.algorithms.Algorithm

Fig. 14 Graphic user interface in Eclipse that generates the Service Descriptor of Figure 13
.

the OSGi features. In this case, the distribution is performed using the service descriptor to set which service is distributable and which is the distribution technology that provides service discovering and data transmission.

OSGi allows several implementations for the service distribution. ECF[5] has been chosen because it is the most mature and accepted implementation (Petzold et al., 2011), and it also supports the largest number of transmission protocols, including both synchronous and asynchronous communication. ECF also separates the source code from the discovery and transmission mechanism, allowing users to apply the most adequate technology to their needs, and providing the integration with existing applications. For example, the lines of Figure 15 have been added to the service descriptor of *MOP2 Fitness Calculator* to distribute it in the local network.

In this case, it is only necessary to set the properties that ECF uses to identify the services being distributed in the network, indicating that all implemented interfaces are distributable (`service.exported.interfaces`). Also, the communication technology to be used is established (`ecf.generic.server`, although another kind of protocol could be used), and finally, the service URL (`ecf.exported.containerfactoryargs`). As previously stated, the service properties can be modified from other services, so this properties can be added outside the XML. It should be noted that the source code of the services has not been modified to distribute them

[5] `http://www.eclipse.org/ecf/`

```
<property name="service.exported.interfaces" type="String" value="*"/>
<property name="service.exported.configs" type="String" value="ecf.generic.server"/>
<property name="ecf.exported.containerfactoryargs" type="String" value="ecftcp://localhost:3787/server"/>
```

Fig. 15 Lines added to the service descriptor to be discovered by other services in a network (this can also be done in the GUI)

(as would happen if MPI had been used to perform the distribution, for example).

5.4 Converting a basic algorithm into a self-adaptive one

Previous sections remarked that the SOA-EA benefits are also related to self-adaptation. A simple example is presented here to demonstrate how easy is to convert a basic evolutionary algorithm into a self-adaptive one in OSGiliath. In this example, an intelligent service manages all the available operators. In this case, the service *IntelligentRandomManager* implements the interfaces *Parent Selector, Recombinator, Mutator* and *Replacer*. All operator implementations previously presented are added to the Manager in execution time (see Figure 11) when they become activated in the system. Every time the EA calls an operator, this simple manager chooses randomly one of the available implementations it controls. To create this manager no code has to be modified. The manager also does not need specific code to acquire all operators in execution time: it is done automatically thanks to OSGi. As the rest of services, these operators can be activated in execution time and added to the manager.

5.5 Increasing interoperability with other systems

As previously stated, another advantage of SOA is the programming language independence respect to the service interfaces. Although OSGi is a kind of SOA, it does not include the capability of interoperability with other kind of services by default. However, adaptation services can be added to transform OSGi interfaces into other SOA interfaces, such as Web Services (presented in section 2). So, services that are not written in Java, neither OSGi-based, could use services implemented in OSGiLiath (and vice-versa).

For example, using the Axis software inside OSGi all OSGi service interfaces could be transformed into WSDL interfaces automatically. Thus, these services could be used from other systems, that do not need to know the implementation language of the services in OSGiLiath. An example where an OSGi interface is transformed into a WSDL interface is shown in Figure 16. The computation node A, based on OSGi, uses the OSGi interface of the computation node C to calculate the fitness. Node B uses the WSDL interface to do the same task. It is not necessary to modify existing services source code to convert an OSGi interface into a WSDL interface. This transformation is bi-directional: given an WSDL interface, it can also be transformed into a service to use inside OSGi. For example, the algorithms in GridUFO framework (presented in section 2) could be used from OSGiLiath.

5.6 Experiments

One may think that working with services usually implies an overhead. This is true when communication protocols like SOAP are used, because the transmitted XML must be generated and parsed. However, as SOA is independent of the implementations, services also can behave as normal method calls in the same machine.

The first experiment of this section is focused on this issue, and will demonstrate that the use of a SOA oriented implementation of an EA doesn't have to affect the execution time of the algorithm. To carry out this experiment, the Java source code of OSGiLiath has been run outside of the OSGi framework, and a normal Java class has been used to integrate the interfaces and implementations "as is". The population has been set to 64 individuals, parents have been selected using Binary Tournament, and the mutation rate has been fixed to 0.1. Worst individuals (parents and off-spring combined) are replaced, and the stop criterion has been set to 200 generations. Each experiment has been launched 30 times to solve the OneMax problem Schaffer and Eshelman (1991).

Since the OSGi framework adds features to the implementation of the algorithm that are similar (and even superior) to those offered by several of the frameworks described in Section 2, the same algorithm (with the same operators and parameters) has been coded using several well known frameworks, such as Mallba (C++), Algorithm::Evolutionary (Perl), and ECJ (Java). Table 3 shows the execution time achieved, average solution, and Lines of Code (LoC) needed to integrate the algorithm for each framework. All the algorithm implementations have been executed on the same computer, an Ubuntu 12.04 Linux Machine with Intel Core2 Quad

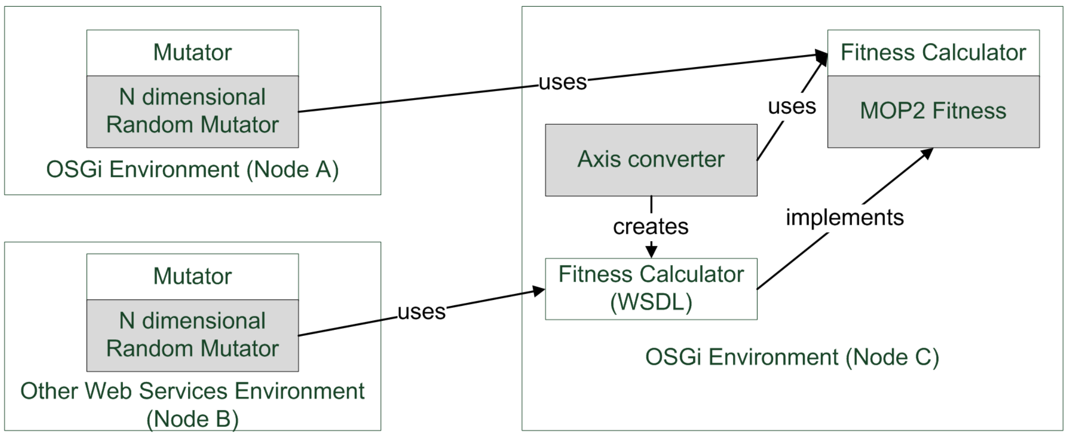

Mutator
N dimensional Random Mutator
OSGi Environment (Node A)
Mutator
N dimensional Random Mutator
Other Web Services Environment (Node B)
uses
uses
Fitness Calculator
MOP2 Fitness
uses
Axis converter
creates
implements
Fitness Calculator (WSDL)
OSGi Environment (Node C)

Fig. 16 Communication with other kind of services. Axis service automatically creates WSDL interfaces for the OSGi interfaces to be used from other environments

CPU Q8200 @ 2.33GHz, 4 GB RAM, without any distribution mechanisms. The LoC have been calculated using *sloccount* program.

Results show that time of services of OSGiLiath is not affected by the OSGi framework: times are almost identical to the integration with Java code. Note that, although are services developed under SOA, and bound in runtime, they are not distributed. Algorithmically, all frameworks behaves the same, and results are not quite different. The differences among frameworks are produced because the different implementations of random generators, operators or logs, for example. In Merelo Guervós et al. (2010), these different behaviours are also justified.

Regarding LoCs, MALLBA has the higher number: this is because every algorithm is created as a "skeleton" and a duplication of code exist for each algorithm and problem to execute. This is produced because many operations affect global variables: for example the method *select_offsprings()* affects the global variables *parents* or *aux*. Using this method as an external service would require a whole change in many parts of the code. Thanks the loose-coupling of Perl, many lines of code are saved using Algorithm::Evolutionary, mainly because many parameters and operators are defined by default.

ECJ and OSGiLiath do not require code to combine different operators, only modify configuration files without re-compilation. The difference is in ECJ the available operators must be known prior to execution (the interfaces are linked in the source code), while in OSGiLiath, all interfaces are bound in configuration files, or even without them (for example, appearing in the same network/machine). But there also exist limitations, because ECJ only provides a fixed ways of distribution mechanisms, and only certain parts of the framework can be accessed remotely, while in OSGiLiath all operators have the chance to be distributed if desired, modifying the configuration files.

It must be remarked that OSGiLiath does not try to compete with the other frameworks (they are widely accepted, completed and tested), it is only an example of how to develop EAs under the SOA paradigm.

6 Discussion

Previous sections have demonstrated that it is possible to create a service oriented architecture for EAs using a specific SOA technology. This architecture uses the features that SOA offers. To do this:

– In Sections 3 and 4 loose coupling services for EAs have been designed (SOA-EA), and they have been implemented in Section 5.
– These services can be combined in several ways to obtain different algorithms (from a canonical GA, a NSGA-II has been created just adding new services). These services are dynamically bound to change the needed EA aspects. The source code of the basic EA services have not been re-written or re-compiled to achieve this task.
– New services can be added in execution time using our implementation.
– No specific source code for a basic distribution have been added, neither the existing source code has been modified.
– Several techniques have been presented to combine existing services in a flexible way.

However, after the explanation of the most important issues of EAs in SOA (Algorithm representation, dynamism, load distribution and self-adapting), we want

Table 3 Comparison of tested EA frameworks in time and development.

Name	Average solution	Average Time (s)	LoC
OSGiLiath	612.36 ± 6.05	0.19 ± 18.21	10
OSGiLiath (without OSGi)	613.36 ± 4.50	0.19 ± 22.74	103
MALLBA	578.76 ± 7.48	0.16 ± 0.0003	2073
ECJ	602.76 ± 6.08	1.40 ± 0.03	5
Algorithm::Evolutionary	617.60 ± 12.92	7.78 ± 0.29	41

to share the benefits of SOA with the rest of EA researchers:

– Firstly, SOA fits with the genericity advantages in the development of software for EAs (Gagné and Parizeau, 2006) and adds new features, like language independence and distribution mechanisms.
– SOA allows the addition and removal of services in execution time without altering the general execution of the algorithm (that is, it is not mandatory to stop it or to add extra code to support new operators).
– It also increases the interoperability between different software elements (for example, it is possible to add communication libraries without modifying existing code).
– Related to the previous point, the existing EA frameworks could be re-used thanks to SOA, because it provides language independence.
– Easiness for code distribution: SOA does not require the use of a concrete implementation or library.
– Access to already created and operative services.
– Collaboration among geografically distributed work teams.

In brief, EAs users and practitioners should change their mind and make an effort to migrate the existing software to SOA, making their services publicly available and loosely coupled to support new research results.

7 Conclusions

Thanks to the Internet booming, there exist a paradigm change from Object Oriented Programming to Service Oriented Architectures, where the software is accessed as interoperable services that allow researchers sharing data and applications in a remote way. The usage of services does not imply remote accessing, but a way to develop and integrate without assumptions about implementation technology. The EAs research is a conducive area to migrate to SOA, because this kind of algorithms is inherently configurable and paralellizable. Moreover, there exist many software tools for EAs, although impossible to be integrated. Also, the booming of new trends, like Cloud Computing, and the usual high cost of this kind of algorithms makes them ideal to be transformed in loosely coupled and distributed services for an easy integration.

This work proposes a Service Oriented Architecture for EAs, with a specific implementation using a specific SOA technology, that takes advantage of this paradigm. Furthermore, new lines to follow in the development of services for EAs, and advantages and disadvantages have been presented.

Although the adoption of a new paradigm is not easy, the importance of the new emerging engineering problems must be taken into account. Therefore, it is necessary to use flexible tools that allow EA researchers to take advantage of all available computation nodes, and make possible the self-adaptation of EAs in every execution environment and for each problem type. SOA-EA tries to be a base for services development: even if the implementation technologies (like the ones used in OSGiLiath) are changed by new ones, the EA researchers should start to consider a migration from their actual software to be accessed as services. This work is the equivalent of what Gagné and Parizeau (2006) presented in the OOP for EAs, but extending their ideas to the SOA paradigm.

The software presented in this work (with all examples explained) is available in `http://osgiliath.sourceforge.net`, under a GNU/GPL license, available to any interested reader. A web portal to centralize new implementations of services being offered to the community will be created as a future task. Also, interviews with EA practioners with different skills in programming and areas will be performed, to validate if this change of paradigm is contributing to enhace their work.

Acknowledgements This work has been supported in part by FPU research grant AP2009-2942 and projects AmIVital (CENIT2007-1010), EvOrq (TIC-3903), and TIN2011-28627-C04-02. Authors wish to thank reviewers' comments, whose suggestion and guidelines have contributed to improve this work.

References

Alba, E., Almeida, F., Blesa, M., Cotta, C., Daz, M., Dorta, I., Gabarr, J., Len, C., Luque, G., Petit, J., Rodrguez, C., Rojas, A., and Xhafa, F. (2006). Efficient parallel LAN/WAN algorithms for optimization. the MALLBA project. *Parallel Computing*, 32(5-6):415–440.

Alba, E., Nebro, A. J., and Troya, J. M. (2002). Heterogeneous computing and parallel genetic algorithms. *Journal of Parallel and Distributed Computing*, 62(9):1362 – 1385.

Altunay, M., Avery, P., Blackburn, K., Bockelman, B., Ernst, M., Fraser, D., Quick, R., Gardner, R., Goasguen, S., Levshina, T., Livny, M., McGee, J., Olson, D., Pordes, R., Potekhin, M., Rana, A., Roy, A., Sehgal, C., Sfiligoi, I., Wuerthwein, F., and Open Sci Grid Executive Board (2011). A Science Driven Production Cyberinfrastructure-the Open Science Grid. *Journal of GRID Computing*, 9(2, Sp. Iss. SI):201–218.

Arenas, M., Collet, P., Eiben, A., Jelasity, M., Merelo, J. J., Paechter, B., Preuß, M., and Schoenauer, M. (2002). A framework for distributed evolutionary algorithms. In *Parallel Problem Solving from Nature, PPSN VII*, pages 665–675.

Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., and Holley, K. (2008). SOMA: A method for developing service-oriented solutions. *IBM Systems Journal*, 47(3):377–396.

Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., and van Moorsel, A. (2005). The self-star vision. *Self-star Properties in Complex Information Systems*, pages 1–20.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25:599–616.

Cox, S. J., Fairman, M. J., Xue, G., Wason, J. L., and Keane, A. J. (2001). The grid: Computational and data resource sharing in engineering optimisation and design search. In *30th International Workshops on Parallel Processing (ICPP 2001 Workshops), 3-7 September 2001, Valencia, Spain*, pages 207–212. IEEE Computer Society.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

del Val Noguera, E. and Pedruelo, M. R. (2008). A survey on web service discovering and composition. In Cordeiro, J., Filipe, J., and Hammoudi, S., editors, *WEBIST 2008, Proceedings of the Fourth International Conference on Web Information Systems and Technologies, Volume 1, Funchal, Madeira, Portugal, May 4-7, 2008*, pages 135–142. INSTICC Press.

Durillo, J. J., Nebro, A. J., and Alba, E. (2010). The jmetal framework for multi-objective optimization: Design and architecture. In *IEEE Congress on Evolutionary Computation*, pages 1–8.

Eiben, A., Haasdijk, E., and Bredeche, N. (2010). Embodied, on-line, on-board evolution for autonomous robotics. In Levi, P. and Kernbach, S., editors, *Symbiotic Multi-Robot Organisms: Reliability, Adaptability, Evolution*, volume 10, pages 361–382. Springer.

Eiben, A. and Smit, S. (2011). Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31.

Eiben, A. and Smith, J. (2005). What is an evolutionary algorithm? In Rozenberg, G., editor, *Introduction to Evolutionary Computing*, pages 15–35. Addison Wesley.

Fan, X.-Q., Fang, X.-W., and Jiang, C.-J. (2011). Research on Web service selection based on cooperative evolution. *Expert Systems with Applications*, 38(8):9736–9743.

Foster, I. (2005a). Globus Toolkit version 4: Software for service-oriented systems. In Jin, H and Reed, D and Jiang, W, editor, *Network and Parallel Computing Proceedings*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13.

Foster, I. (2005b). Service-oriented science. *Science*, 308(5723):814.

Gagné, C. and Parizeau, M. (2006). Genericity in evolutionary computation software tools: Principles and case-study. *International Journal on Artificial Intelligence Tools*, 15(2):173.

García-Sánchez, P., González, J., Castillo, P., Merelo, J., Mora, A., Laredo, J., and Arenas, M. (2010). A Distributed Service Oriented Framework for Metaheuristics Using a Public Standard. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 211–222.

García-Sánchez, P., González, J., Mora, A. M., and Prieto, A. (2012). Deploying intelligent e-health services in a mobile gateway. *Expert Systems with Applications*, (0):–. In press.

Guervós, J., Valdivieso, P., López, G., and Arenas, M. (2003). Specifying evolutionary algorithms in xml. *Computational Methods in Neural Modeling*, pages 1042–1043.

Ho, Q.-T., Ong, Y.-S., and Cai, W. (2004). "gridifying" aerodynamic design problem using GridRPC. In *Grid and Cooperative Computing*, volume 3032 of *Lecture Notes in Computer Science*, pages 83–90. Springer

Berlin Heidelberg.

Huband, S., Hingston, P., Barone, L., and While, L. (2006). A review of multiobjective test problems and a scalable test problem toolkit. *Evolutionary Computation, IEEE Transactions on*, 10(5):477 –506.

Imade, H., Morishita, R., Ono, I., Ono, N., and Okamoto, M. (2004). A grid-oriented genetic algorithm framework for bioinformatics. *New Gen. Comput.*, 22(2):177–186.

Jamil, E. (2009). White Paper: What really is SOA. A comparison with Cloud Computing, Web 2.0, SaaS, WOA, Web Services, PaaS and others. . Available at `http://soalib.com/doc/whitepaper/SoalibWhitePaper_SOAJargon.pdf`.

Jiao, Z., Wason, J. L., Song, W., Xu, F., Eres, M. H., Keane, A. J., and Cox, S. J. (2004). Databases, workflows and the grid in a service oriented environment. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*, volume 3149 of *Lecture Notes in Computer Science*, pages 972–979. Springer.

Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., and Lee, B.-S. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, 23(4):658 – 670.

Lim, J., Choi, O., et al. (2008). An evaluation method for dynamic combination among OSGi bundles based on service gateway capability. *IEEE Transactions on Consumer Electronics*, 54(4):1698 –1704.

Lozano, M. and Garcia-Martinez, C. (2010). Hybrid metaheuristics with evolutionary algorithms specializing in intensification and diversification: Overview and progress report. *Computers & Operations Research*, 37(3, Sp. Iss. SI):481–497.

Luke, S. et al. (2009). ECJ: A Java-based Evolutionary Computation and Genetic Programming Research System. Available at `http://www.cs.umd.edu/projects/plus/ec/ecj`.

Merelo Guervós, J., Castillo, P., and Alba, E. (2010). Algorithm::evolutionary, a flexible Perl module for evolutionary computation. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 14:1091–1109.

Moussa, H., Gao, T., Yen, I.-L., Bastani, F., and Jeng, J.-J. (2010). Toward effective service composition for real-time SOA-based systems. *Service Oriented Computing and Applications*, 4(1):17–31.

Munawar, A., Wahib, M., Munetomo, M., and Akama, K. (2010). The design, usage, and performance of gridufo: A grid based unified framework for optimization. *Future Generation Computer Systems*, 26(4):633 – 644.

Ng, H.-K., Ong, Y.-S., Hung, T., and Lee, B.-S. (2005). Grid enabled optimization. In *Advances in Grid Computing - EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 296–304. Springer Berlin Heidelberg.

OSGi Alliance (2010a). Benefits of using OSGi. Available at: `http://www.osgi.org/About/WhyOSGi`.

OSGi Alliance (2010b). OSGi service platform release 4.2. Available at: `http://www.osgi.org/Release4/Download`.

Papazoglou, M. and van den Heuvel, W.-J. (2007). Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415. 10.1007/s00778-007-0044-3.

Parejo, J., Ruiz-Corts, A., Lozano, S., and Fernandez, P. (2012). Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 16:527–561. 10.1007/s00500-011-0754-8.

Petzold, M., Ullrich, O., and Speckenmeyer, E. (2011). Dynamic distributed simulation of DEVS models on the OSGi service platform. *Proceedings of ASIM 2011.*

Schaffer, J. and Eshelman, L. (1991). On Crossover as an Evolutionary Viable Strategy. In Belew, R. and Booker, L., editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann.

Serpell, M. and Smith, J. E. (2010). Self-Adaptation of Mutation Operator and Probability for Permutation Representations in Genetic Algorithms. *Evolutionary Computation*, 18(3, Sp. Iss. SI):491–514.

Song, W., Keane, A., and Cox, S. (2003). Cfd-based shape optimisation with grid-enabled design search toolkits. In *UK e-Science All Hands Meeting 2003*, pages 619–627. EPSRC.

Song, W., Ong, Y. S., Ng, H. K., Keane, A., Cox, S., and Lee, B. S. (2004). A service-oriented approach for aerodynamic shape optimisation across institutional boundaries. In *Control, Automation, Robotics and Vision Conference, 2004. ICARCV 2004 8th*, volume 3, pages 2274 – 2279.

Ventura, S., Romero, C., Zafra, A., Delgado, J. A., and Hervas, C. (2008). JCLEC: a Java framework for evolutionary computation. *Soft Computing*, 12(4):381–392.

Wagner, S. and Affenzeller, M. (2005). HeuristicLab: A generic and extensible optimization environment. In Ribeiro, B and Albrecht, RF and Dobnikar, A and Pearson, DW and Steele, NC, editor, *Adaptive and Natural Computing Algorithms*, Springer Computer Science, pages 538–541. 7th International Conference on Adaptive and Natural Computing Algorithms (ICANNGA), Coimbra, Portugal, MAR 21-23, 2005.

Wagner, S., Winkler, S., Pitzer, E., Kronberger, G., Beham, A., Braune, R., and Affenzeller, M. (2007). Benefits of plugin-based heuristic optimization software systems. In Moreno Daz, R., Pichler, F., and Quesada Arencibia, A., editors, *Computer Aided Systems Theory EUROCAST 2007*, volume 4739 of *Lecture Notes in Computer Science*, pages 747–754. Springer Berlin / Heidelberg.

World Wide Web Consortium (2006). Extensible Markup Language (XML) 1.0 (Fourth Edition).

Xue, G., Song, W., Cox, S. J., and Keane, A. J. (2004). Numerical optimisation as grid services for engineering design. *J. Grid Comput.*, 2(3):223–238.