A methodology to develop Service Oriented Evolutionary Algorithms

P. García-Sánchez, A. M. Mora, P. A. Castillo, J. González and J.J. Merelo

Dept. of Computer Architecture and Technology and CITIC-UGR
University of Granada, Spain pablogarcia@ugr.es

Abstract. This paper proposes a methodology to design and implement Evolutionary Algorithms using the Service Oriented Architecture paradigm. This paradigm allows to deal with some of the shortcomings in the Evolutionary Algorithms area, facilitating the development, integration, standardization of services that conform a evolutionary algorithm, and, besides, the dynamic alteration of those elements in runtime. A four-step methodology to design services for Evolutionary Algorithms is presented: identification, specification, implementation and deployment. Also, as an example of application of this methodology, an adaptive algorithm is developed.

1 Introduction

New trends in Evolutionary Algorithms (EAs) [1], such such as P2P or pool-based EAs [2], lead to its implementation in dynamic and heterogeneous environments. Service Oriented Architecture (SOA) has been proposed [3] as a possible solution to facilitate the creation of applications in these kind of environments. In this paradigm, a service is a loose, coarse-grained, and autonomous component that allows interactions of all the elements that conform the system [4]. SOA could facilitate the creation and control of services for EAs in several issues addressed in other works, such as their use in *development*, as there exist several methodologies to model and design services, or the usage of techniques such as versioning, packaging or life-cycle control. Services also facilitate the *integration*, as they are independent of the programming language. Furthermore, services allow distribution transparency: it is not mandatory to use a specific library for the distribution, or modify the code to adapt the existing operators. As SOA is based in public standards (such as WSDL[1] or OSGi[2]), its use promotes the *standardization* of the service interfaces, facilitating the Open Science

This work has been supported in part by FPU research grant AP2009-2942 and projects SIPESCA (G-GI3000/IDIF, under Programa Operativo FEDER de Andalucía 2007-2013), PYR-2014-17 of CEI BIOTIC Granada (GENIL) and ANYSELF (TIN2011-28627-C04-02).

[1] http://www.w3.org/TR/wsdl

[2] http://www.osgi.org/

[5]. Finally, as services are not aware of the order of execution, the *dynamism* of this paradigm can fit with new parallel approaches for EAs, where the control of the nodes is not centralized. For example, new operators in different nodes can be bound and used during the run of an algorithm. Also, there should be easy to add and remove elements to achieve self-adaptive mechanisms. All previous issues are taken into consideration in this paper to propose the methodology.

SOA has been previously used in the EA area. For example, web services have been used in the grid area for optimization problems [6], where services are defined using WSDL. In our previous work [3], an introduction to the usage of SOA to develop EAs, with some advantages, guidelines and examples was presented. However, previous works do not count with a guided step by step methodology for EAs, as proposed in this paper.

The rest of the paper is structured as follows: the description of the proposed methodology (called SOA-EA) is presented in Section 2. Then, in Section 3 an example of application of the methodology to create a service oriented evolutionary algorithm is performed. Finally, conclusions and future works are discussed.

2 Methodology

This section presents all the steps to design and implement Service Oriented Evolutionary Algorithms (SOEAs), that is, evolutionary algorithms whose elements are services. The selected steps have been adapted from SOMA [4], a methodology to develop services focused on business environments and adapted to be used in the EA research area.

This methodology also takes into account the work of Gagné and Parizeau [7]. The authors established six criteria to qualify the genericity of a framework for EAs: generic representation, generic fitness, generic operations, generic evolutionary model, parameter management and configurable output. Also, according to Valipour [8], services developed must follow several characteristics: they must be discoverable and dynamically bound, self-contained and modular, interoperable, loose-coupled, transparent to location and composable.

2.1 Identification

This phase is focused on the identification of the three constructs of SOA: services, components and flows. The developers should ask themselves several questions to facilitate the identification about the problem to solve, such as the elements and operators needed by the EA, the extension capabilities and how to parametrize of the EA. To facilitate this task three different domains are proposed. In the first one, the **algorithm domain**, the services are those that conform the EA. For example, operators of individuals, stop criterion, or populations. The second one, the **problem domain**, comprises services to address the elements of the problem (for example, the fitness function). There are also other services that depend on the problem, such as an initializer of individuals. Finally,

the **infrastructure domain**, identify services that deal with the specific infrastructure that will be used to execute the algorithm. For example, services for user control, load balancing or logging. Depending on the environment where the EA is going to be developed, other services need to be modelled. For example, user control in cloud environments, different mechanisms for logging or interconnection with other systems (such as external databases).

2.2 Specification

The questions to solve prior to this phase are related with the operations and their inputs/outputs, the flow (order) of services, different kinds of available implementations and adaptation of services (metrics or behaviours).

First, the EA **operators** should not be be modelled to receive one or two individuals, but a list of individuals to be modified, as not all EAs have the same behaviour. Since many types of individuals may exist, the operators should be as abstract as possible to work properly. Therefore, services must accept interfaces of individuals as inputs, not concrete implementations, such as vectors or lists (generic representation). This is also applied to the **fitness**: it should not be calculated within a method of an *Individual* class. To be less coupled, it should be implemented as an external service that receives a list of individuals (facilitating the load balancing).

The **population** should be a service to access the individuals and allow the variation of its structure (for example, a change from an unique list population to a cellular 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.

Also the **parameter** set should be a service for the same reason, allowing the possibility of performing experiments related to parameter control or tuning in an efficient way (being separated from the code of the existing operators).

A SOEA can be seen as a service flow. The **flows** should be designed to reduce the impact of potential future changes. An example of service flow would be an implementation called *Evolutionary Algorithm* with all the steps common to all EAs and with independence of the implementations of these steps (generic evolutionary model). This allow the 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. Finally, the **infrastructure services** should be designed as flexible output mechanisms. For example, a GUI or logging should be independent of the services.

2.3 Implementation and deployment

The last two steps SOA-EA are explained together because the decisions about the technological solution to be used is bound to both phases. The questions to solve in these steps are related with the technological implementation of the services and its execution (locally or remotely). Different technologies should be compared to address the publication of interfaces, overload and dynamic control.

Also, considerations about security, persistence, benchmarking and monitoring are taken into account in this step.

The first step is to **select the technology to expose the interface**, depending on the use of the services. For example, a service that is going to be used remotely and publicly from any programming language should export its interface with WSDL publicly available with an URL, to allow users to automatically generate the client for that service. On the other side, interfaces could be previously known, and it is not necessary to export them to the public. This is the case of OSGi, where the interface is exposed only to the OSGi service registry.

The **selection of the communication mechanism** must be considered depending on the system to deploy the services. In the case of EAs, where the performance is important, usually the most efficient transmission mechanism should be preferred. In this step, issues related with testing, user control, security and persistence should be taken into account.

3 Example of creating a service oriented evolutionary algorithm

This section justifies the use of the proposed methodology and the steps to create services with it. In this example, a basic genetic algorithm to solve the MMDP problem [9] will be designed. This algorithm needs to automatically bind new operators (not previously known) when the algorithm has found a local optimum. No extra code should be added to the algorithm to bind/unbind operators or check the population.

In the *identification* step a number of abstract services have been identified. In the *algorithm domain*, the Algorithm, Population, Parent Selector, Recombinator, Mutator, Replacer, Stop Criterion and Parameters. In the *problem domain*, the Fitness Calculator and Initializer. Finally, a *infrastructure domain* service called Asynchronous Enabler, which is in charge of activating new operators when a local optimum is found (for example, no changes in best individual during certain time).

Concrete implementations are defined in the *specification* step: *N Tournament* and *Roulette* are implementations of parent selectors and *Optimum Found* the desired stop criterion. Also, to address the problem to be solved, such as *MMDP Fitness Calculator* or *Binary Initializer*. As we need a fixed set of steps, a *Evolutionary Algorithm* service is created to model the flow of services. The discovered services have been specified to accept a list of individuals. A service to gather all selectors in the environment is used. Infrastructure services to deal with control of the population and to enable other operators are also implemented (Asynchronous Enabler Imp). Figure 1 shows this designed configuration.

Since this example requires automatic binding and asynchronous and parallel control of the population services, OSGi is the technology proposed in the *implementation* step. The reasons of using these technologies are explained

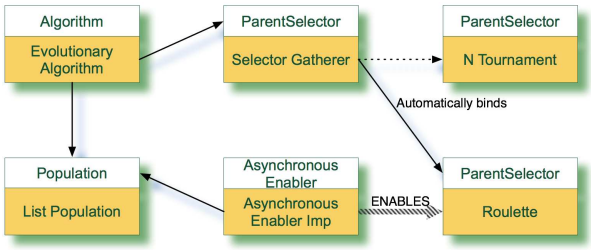
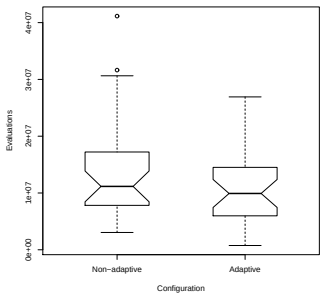


Fig. 1. Automatic binding of new operators. White boxes are interfaces and shaded boxes are implementations of the services.



Non-adaptive	
Generations	219403,10 ± 141692,16
Evaluations	14041926,40 ± 9068298,82
Time	68766,40 ± 45073,04
Adaptive	
Generations	167166,66 ± 93594,37
Evaluations	10698794,66 ± 5990039,68
Time	51710,40 ± 29329,21

Fig. 2. Results obtained. The version with automatic binding of services achieved significant better results.

in [3], but summarizing, OSGi allows automatically binding of implementations to interfaces without extra code or recompiling, and dynamic discovery of services, as this example requires. The source code of the proposed implementation is available under a GNU/LGPL V3 license in our repository at `http://www.osgiliath.org`.

Two configuration have been compared: a non-adaptive version that only uses a Binary Tournament for Selection, and an adaptive one, which automatically enables a Roulette Selection when a local optimum is found. The parameters used in this comparison (accessed from the Parameters service) are a population of 64 individuals, selector rate of 0.5, TPX crossover, bit flip mutation, and individual length of 60 genes. The Roulette selector is enabled when the best individual of the population has not changed in 10 seconds (checked every 2 seconds). Figure 2 shows the results obtained from the 30 executions of the two configurations tested. As it can be seen, automatic and adaptive enabling of selection operators has allowed an increase of performance, reducing time and evaluations (both significantly with a p-value$<$0.05 of a Wilcoxon test). This example has been used to demonstrate that applying a methodology to develop loose coupled services that can be dynamically bound, without modification of the existing services, can be used to achieve better results.

4 Conclusions

New trends in distributed Evolutionary Algorithms, such as P2P, imply to deal with heterogeneous and dynamic environments, with different programming languages and transmission technologies. This fact motivate the creation of a proper way to define service oriented evolutionary algorithms (SOEAs) to facilitate the development, integration, standardization and dynamism of the EA components in this kind of environments. In this paper the requirements in EA design, with the requirements in SOA, have been taken into account to propose a methodology to model the services that compose a service oriented EA, and several guidelines about the design of these services have been explained. This methodology, called SOA-EA, proposes four iteratively and incremental phases: identification, specification, implementation and deployment. SOA-EA has been used to create a SOEA that takes advantage of the SOA capabilities, such as loose-coupled services and automatic binding of new operators.

In future work, this methodology will be used to create new examples of SOEAs and refined to deal with other shortcomings. Other technologies available in SOA will be also tested and analysed.

References

1. Eiben, A., Smith, J.: What is an Evolutionary Algorithm? In: Introduction to Evolutionary Computing. Springer (2003)
2. Meri, K., Arenas, M.G., Mora, A.M., Merelo, J., Castillo, P.A., García-Sánchez, P., Laredo, J.L.J.: Cloud-based evolutionary algorithms: An algorithmic study. Natural Computing (2013) 1–13
3. García-Sánchez, P., González, J., Castillo, P.A., Arenas, M.G., Merelo-Guervós, J.J.: Service oriented evolutionary algorithms. Soft Comput. **17**(6) (2013) 1059–1075
4. Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., Holley, K.: SOMA: A method for developing service-oriented solutions. IBM Systems Journal **47**(3) (2008) 377–396
5. Foster, I.: Service-oriented science. Science **308**(5723) (2005) 814
6. Imade, H., Morishita, R., Ono, I., Ono, N., Okamoto, M.: A grid-oriented genetic algorithm framework for bioinformatics. New Generation Computing **22**(2) (2004) 177–186
7. Gagné, C., Parizeau, M.: Genericity in evolutionary computation software tools: Principles and case-study. International Journal on Artificial Intelligence Tools **15**(2) (2006) 173
8. Valipour, M., Amirzafari, B., Maleki, K., Daneshpour, N.: A brief survey of software architecture concepts and service oriented architecture. In: Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on. (2009) 34–38
9. Goldberg, D.E., Deb, K., Horn, J.: Massive multimodality, deception, and genetic algorithms. In R. Männer, Manderick, B., eds.: Parallel Problem Solving from Nature, 2, Amsterdam, Elsevier Science Publishers, B. V. (1992) 37–48