Towards a Component-based Software Architecture for Genetic Algorithms

Leidy Garzón Rodríguez
lpgarzonr@correo.udistrital.edu.co

Henry Alberto Diosa
hdiosa@udistrital.edu.co

Sergio Rojas-Galeano
srojas@udistrital.edu.co

Universidad Distrital Francisco José de Caldas
Bogotá, Colombia

***Abstract*—We are motivated on the idea of whether a component-based software architecture for evolutionary algorithms would be feasible and advantageous. We believe that depending on the evolutionary computation model, software implementing these algorithms can be robustly built assembling loosely-coupled computational blocks, likewise hardware systems that are built gluing together prefabricated electronic components. We set about to develop an initial architecture with a focus on the genetic algorithm. The paper describes the analysis and design principles used, the obtained architecture, the resulting component specification and closes with a discussion about the benefits of this approach, as well as initial steps towards its implementation in a user-friendly platform for component-based visual programming. The complete portfolio of software models is available at:**

http://arquisoft.udistrital.edu.co/portal/web/guest/proy-compAG

***Index Terms*—Architecture and Software Engineering, Component-based Development, Genetic Algorithms**

I. Introduction

This study is grounded on the question of whether a component-based software architecture would be feasible and advantageous when developing evolutionary algorithms. Our premise is that for general-purpose optimization, these kind of algorithms share a common architecture consisting of self-contained loosely-coupled computational units. Thus, depending on the evolutionary computation (EC) model the algorithm implements, a system could be developed by reusing or refining self-contained units from a specialized library of EC software components, resembling how hardware systems are built from prefabricated electronic components kits (see Figure 1 for a fanciful depiction of a genetic algorithm (GA) system built following that approach).

We embarked on this initiative focused on GA, maybe the best-known instance of evolutionary algorithms. Several libraries and frameworks have been developed for GA most of them over the object oriented computation model or structured programming as GALib [12], CamGASP [14], JGAP [11], JCLEC [17], ECJ [19], Opt4J [9], DEAP [3], HeuristicLab [18] and Open BEAGLE [5]. Some few initiatives as OSGiLiath [6], MALLBA [1] and EO [7] in spite of being

978-1-4799-6717-9/14/$31.00

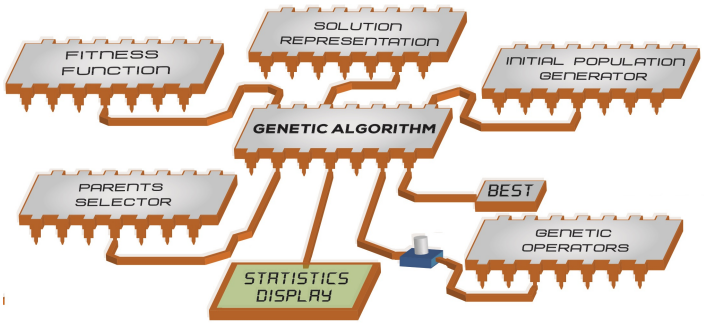

Figure 1. A fictitious depiction of an electronic component-based GA

component-based libraries, they only support an scripting mechanism for component assemblage. It should be pointed out, that, a software solution that meets the modularity non-functional quality aspect does not imply necessarily that it conforms to a component-based software architecture in the sense discussed in section II.

The paper is organized with the initial sections describing the analysis and design principles that guided our development, the middle sections describing the architecture and component specifications, and the final sections discussing benefits of this approach, as well as describing initial steps towards its implementation in a user-friendly platform of visual component programming. The complete portfolio of software models we developed is publicly available in [4].

II. Component-based Development

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [15]. In order to make a component available it is necessary to define either a required or provided interface, and in order to use it, it is only necessary to have other components complying with said interfaces. We have chosen UML2.0 [8] as a modeling language for our component architecture (see Figure 2), where an external representation (Component Diagram) depicts components as black-boxes exposing their contractual requiring and providing interfaces, and an internal representation (Use

Case and Activity Diagrams) specify in detail the structural and behavior models of the component.

Figure 2. UML Graphic representation of a software component

A. Component-based Development Workflow

In order to obtain the architecture, we adopted the workflow showed in Figure 3, originally proposed in [2]. The stages in the workflow are represented with boxes, the wide arrows indicate precedence order between stages. Each box correspond to a sequence of activities, with thin arrows representing the flow of input/output artifacts shared among them. The *Component specifications and architecture* of the GA system that we are looking for, is the output artifact of the Specification stage. In order to obtain this artifact we first need to define the *Use case model* and the *Business concept model* from the Requirements stage.

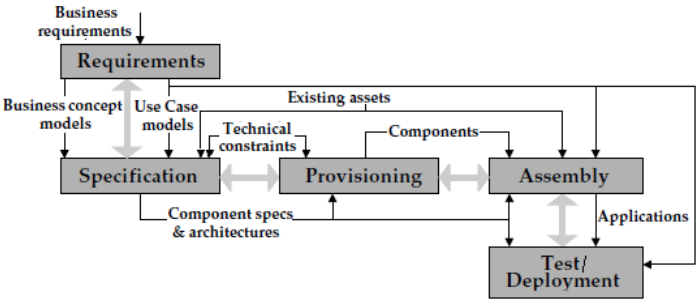

Figure 3. Component-based development workflow (adapted from [2])

III. A Component-Based GA

Evolutionary algorithms can be seen as general-purpose population-based stochastic search tools for optimization [10], sharing a common generic sequence of steps: (i) Encode the initial population, (ii) Evaluate population fitness using a cost function, (iii) Apply evolution operators, (iv) Go to step (ii) until termination, (v) Return the best solution emerged. In the specific case of GA, variations in the selection methods, genetic operators, solution coding/decoding and cost function types are associated to those generic steps. As said before, we believe at the end they can be seen as independent computational units that are assembled during the algorithm execution ("evolution"). We anticipate that reusing some of these components or refining their specifications would be useful in the design/implementation of other known or new evolutionary algorithms.

With a view to specify the initial version of our architecture, we consider the most relevant approaches, representations and operators found on the GA literature [10] which are summarized in Table I. Additionally, we contemplated some utility components needed for experimentation purposes (calibration and performance tester), other solution representations as permutations and discrete domains and theirs respective GA operators would be included in future versions of this work.

Table I
Relevant computational techniques associated to GAs

Name	Type
Solution encoding	Binary, Real
Evolutionary approach	Elitism, Steady-State
Initial population	Random, Seeded, Biased
Selection	Tournament, Roulette Wheel
Crossover	Binary:One point, Two points Real:AX, BLX-α, BLX -$\alpha - \beta$, WHX Custom
Mutation	Binary:One bit, Multiple bits Real:Truncate Gauss Custom

IV. Architecture Development

A. Requirements Stage

The purpose of this stage is to gather the description of the functionalities and information flow required to run a GA according to the expectations of a final user. Our analysis identified 58 requirements, which are summarised in Table II. Detailed technical documents describing these requirements are available in [4]. The elicited requirements were used to design the *Business concept model* and a *Use case models* that are the basis to build the *GA component specification*.

The *Business concept model* defines a mutually agreed vocabulary of concepts among the stakeholders of the system. For instance, in the model we specified the concept cost function as the function to be optimized by the GA; this concept would be also a surrogate of the more popular fitness function concept.

A *Use case* on the other hand, is a specification of the interactions between the users and the functional requirements of the GA. In our analysis we identified 38 use cases which are summarized in Table II (and also available in [4]).

Table II
Summary of elicited requirements and use cases

Type	Number requirements	Number use cases
Evolutionary approach	3	4
Initial population	7	5
Selection	3	5
Crossover	8	5
Mutation	5	5
Tester+Calibrator	13	3
GA Configuration	7	11
Non-Functional	12	NA
Total	**58**	**38**

Use cases represent low-level usage scenarios that can be combined to specify higher-levels of functionality. In order to fulfill a requirement, many use cases could be identified, and some of them may be used again to fulfill other requirements. This means that use cases can be packaged according to their

related functionalities. For instance, Figure 4 shows a use case model related to the creation of the initial population. It can be seen in this example that single use cases can be connected with ≪*include*≫ and ≪*extend*≫ relations. One use case may include another, meaning that at some point of the usage scenario the completion of the other use case is needed. In contrast, one use case may extend another, meaning that him is an alternative choice to complete the base use case. In the example, when *Generate a biased population* is carried out, the system reuses the *Generate random population* scenario in order to sample random chromosomes constrained to the bias criteria defined by the user. On the other hand, when *Generate initial population* scenario is carried out, the user must choose one of three available extensions representing different methods for creating the initial population. Notice that the model itself can be iteratively refined, that is, new versions of the released specification can be obtained by including or extending new additional use case models, in order to account for new or custom-tailored functionalities.

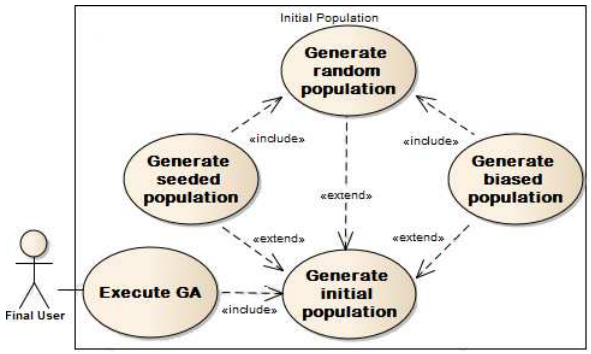

Figure 4. A Use case model

For each identified *Use case*, an *Activity diagram* specifies the actions and flow of information involved in the completion of the required functionality (for example Figure 5 is the activity diagram corresponding to the *Generate initial population* use case). Actions are represented as round corner boxes whereas input and output parameters as small rectangles. Notice that boxes with an ∞ indicate a drill-down specification is available.

B. Specification stage

This stage involves the identification of the components and the corresponding interfaces that will comprise the initial architecture also known as *prescriptive architecture*. These would be split into two layers: *GA System Components* and the *GA Business Components* (see Figure 6). The first layer of components and interfaces is derived from the *GA Use case model*, whereas the second layer is derived from the *GA Business concept model*. The *GA System Components* play a double role: interaction with the user (or other external systems) and interaction with the *GA Business Components*. As so, these are the components and interfaces responsible of collecting running parameters, coordinating the execution and visualising the results obtained by the GA.

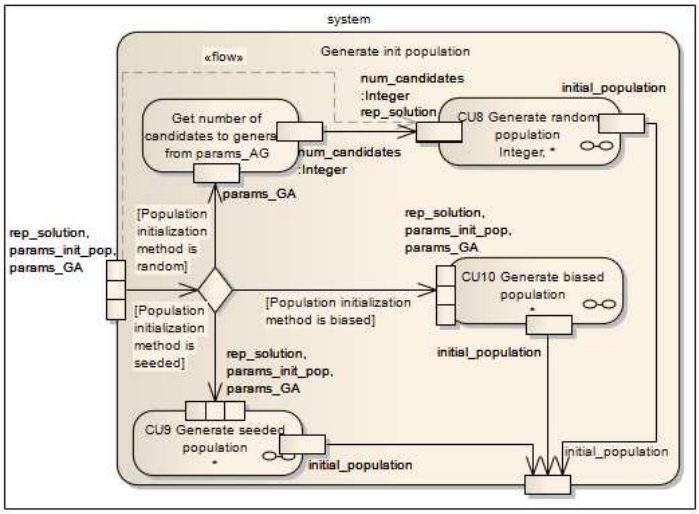

Figure 5. Generate initial population activity diagram

In order to identify the *Business Interfaces* we use the business concept model to help us focus on the information and associated processes that the system will need to manage, we refine the business model into a *Business type model* representing the specific business information that must be held by the system, next we decide which types we consider to be *Core Types*. A core type is defined as a business type that has independent existence within the business, the criteria to identified cores can be found in [2]. Finally, we create one business interface and assign responsibilities for each core type. As can be seen from Figure 7, the type *SolutionRep* has no mandatory associations, so we characterize it as a core, at that time we define the *ISolutionRepMgt* interface which will manage the information represented by the *SolutionRep*, note that this interface will have the responsibility of encode and decode solutions.

Otherwise, to identify the *System Interfaces* we define one interface for each use case that requires user interaction, then we go through the activity diagram which represent the steps of the use case considering whether or not there are system responsibilities that must be modeled. If so, we will need to include the *operations* in the interface.

The Figure 8 shows a subset of the use cases model, selecting those that will generate a system interface, observe that at the top of the Figure, we define the *ISolutionRepSetup* system interface according to the steps of the Use Case *Represent Solution*. In the first step we see that the system must provide a list of the available domains supported by the system, once the user define a specific domain, the system will provide a list of the available encoding types for the selected domain, as a final step the system interface provide the *apply_settings()* operation, this last interface operation will notifies if the Solution Representation was successful defined through the return parameters.

The *GA Business Components*, carry out the core computation of the GA (evolutionary approaches, genetic operators

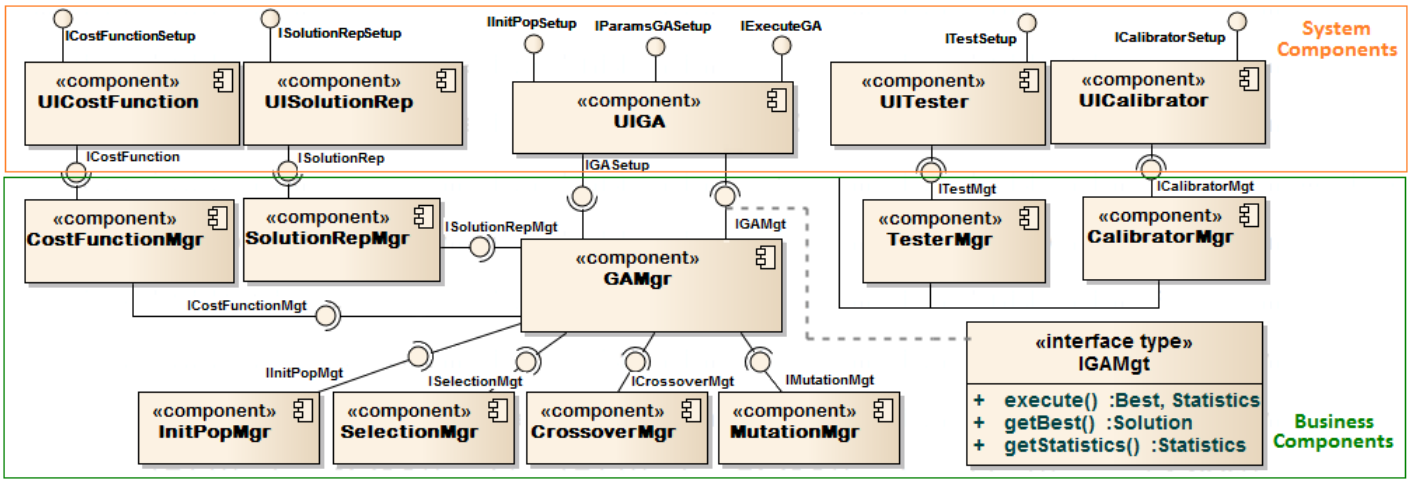

Figure 6. Component-based GA prescriptive architecture

and complementary techniques such as sampling, statistics gathering, etc). Observe that the core components in the Business layer (GA, cost function, solution representation, tester and calibrator) have companion user-interaction components. These are assembled together through their corresponding interfaces: provided interfaces for those components responsible of parameter collection, and/or required interfaces for those responsible of monitoring and visualising the execution of the GA.

The specification of the *GA System components* is dependent on the chosen user-interaction model (command-based, GUI, scripts, Web-service); we will defer the discussion about such model to Section IV-C.

In the remainder of this section we shall focus on describing briefly the *GA Business components*.

- `GAMgr`. The core component that controls the flow of execution of the GA. It comprises 7 required interfaces: `IGASetup` manages the setup of the running parameters for the algorithm, including choices for initial population method, genetic operators probabilities, number of generations and so on; `IInitPopMgt` manages the creation of the initial population; `IParentsSelectorMgt`, `ICrossoverMgt`, `IMutatorMgt` are managers of the genetic operators; `ICostFunctionMgr` manages the definition and evaluation of the cost function; `ISolutionRepMgt` manages the encoding and decoding of the candidate and best solution representation. Besides, this component comprises 1 provided interface, `IGAMgt`, which manages services for execution, statistics gathering and best solution retrieval (for illustration purposes the bottom-right corner of Figure 6 shows the service specification of the latter interface).
- `CostFunctionMgr`. The component that provides the `ICostFunctionMgt` interface to the `GAMgr`. It requires an `ICostFunction` interface from the `UIConstFunction` component in the user-interaction layer.
- `SolutionRepMgr`. The component that provides the `ISolutionRepMgt` interface to the `GAMgr`. It requires an `ISolutionRep` interface from the `UISolutionRep` component in the user-interaction layer.
- `InitPopMgr`. The component that provides the `IInitPopMgt` interface to the `GAMgr`. It is expected that the choice of creation method and its parameters should be supplied through the `IGASetup` interface of the `GAMgr` component.
- `SelectionMgr`, `CrossoverMgr`, `MutationMgr`. The components that provide the `ISelectionMgt`, `ICrossoverMgt` and `IMutationMgt` interfaces to the `GAMgr`. Again, the parameters of these operators should be supplied through the `IGASetup` interface of the `GAMgr` component.
- `TesterMgr`, `CalibratorMgr`. These components manage useful services for testing and calibrating the execution of a GA. The parameters of these components should be supplied through the `ITestMgr` and `ICalibratorMgr` interfaces respectively, which additionally are able to trace cumulative statistics of experiments including several runs of the algorithm.

C. Provisioning stage

The aim of the provisioning stage is to supply the components specified in the architecture so as to eventually be able to deploy the system. There are several ways of acquiring such components, either by building them, reusing from a third part or modifying existing components. In any case, at this point implementation decisions are to be taken related to the target runtime environment, including programming language, user-interaction technology, operating system, etc. Our first choice in this respect is keen towards an object-oriented implementation, with a GUI allowing interaction with the user, and ideally within a visual environment where components can be graphically assembled, i.e. glued together.

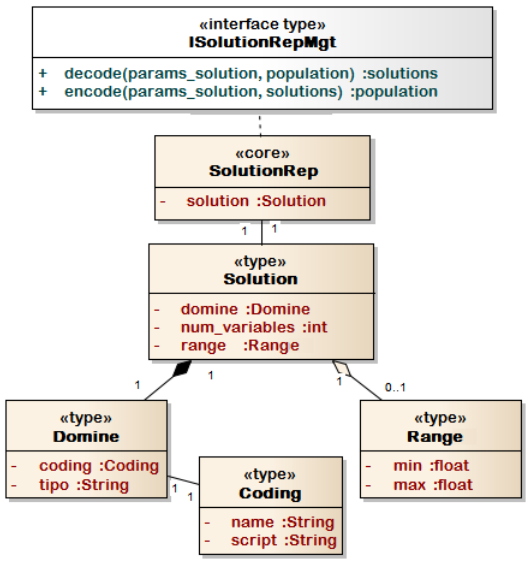

Figure 7. Identification of the ISolutionRepMgt Business interface

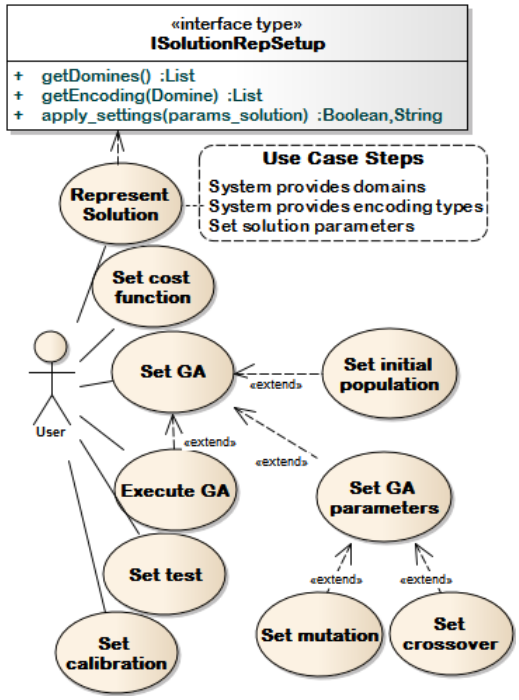

Figure 8. Use cases map to system interfaces

On the other hand, we were aware of an open-source component-based framework for machine learning called Orange [20] which meets two of our aforementioned expectations: support for component implementation using object-oriented programming in Python and a visual environment for components assemblage. Regrettably the Orange framework does not feature a library devoted to GAs or other stochastic search-based optimization algorithms, although recently, an extension toolbox known as Goldenberry was proposed to support Estimation of Distribution Algorithms [13].

As a result of this exploration we decided to carry out the provision stage in the following way: (i) Reusing visual canvas of Orange and GUI components also known as Orange Widgets; (ii) Reusing cost function and tester components from the EDA algorithms of Goldenberry; and (iii) Building the reminder components of the architecture. The latter is the aim of the second part of our project which is currently work in progress.

It is important to clarify that the provisioning stage involve a series implementation decisions that would define the *realization* of the prescriptive architecture into a descriptive architecture [16]. In our case, the following aspects were taken into account during the decision-taking:

- The concept of Widget is instrumental to enable user interaction within the Orange visual canvas; therefore it must be taken into consideration in our architecture.
- The latter was achieved by embedding our business components within widget components relatives, and *delegating* business interfaces to the containing widgets.
- Widgets expose system interfaces and GUIs to the user; those system interfaces are also *delegated* by the business components providing operations to setup the parameters required.
- We do not include internal design for the components to be reused (CostFunctionWidget and TesterWidget).

As a result, we obtained the Component-based GA descriptive architecture shown in Figure 9, where widgets components would be implemented as the visual elements that can be dragged onto the Orange visual programming canvas, wrapping the logic needed to provide graphical user interface to our business components. As a final note, observe that the system interfaces which are outside of the system boundary, keep correspondence with the use cases shown in Figure 8.

V. Conclusions

We have described a feasible architecture for general-purpose component-based GAs. The architecture is open and independent of runtime or implementation decisions. The modularity exhibited by this architecture makes it easily extensible to other kinds of stochastic-search population-based optimization algorithms, through the design of new components that can be replaced and/or assembled with the ones here described. For example, in a GA context the `SolutionRepMgr` component is intended to perform the genotype-phenotype mapping inherent to a chromosome representation, but it could also be straightforwardly redesigned to carry out the encoding/decoding computation of other type of Evolutionary Algorithms.

Our work is currently focused on the Provisioning stage, with an aim to build a library of GA software components that may be useful for developers and researches alike, within a friendly visual programming environment (Orange and Goldenberry 2.0).

As a final note, possible avenues of research may include extending and implementing the architecture with new components accounting for additional GA functionalities (preventing

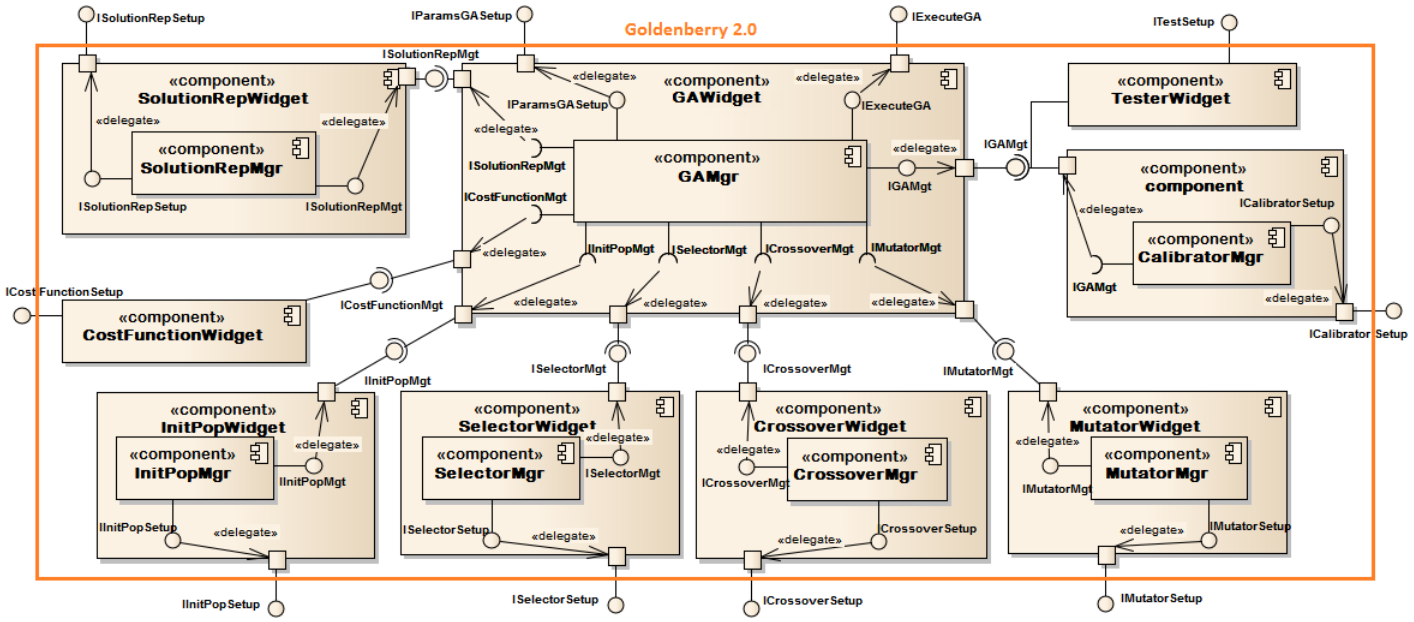

Figure 9. Component-based GA descriptive architecture

premature convergence, parallelism, multi-objective optimization). We are also considering in the future, the problem of evolving specific-purpose Evolutionary Algorithms by automatically orchestrating the software components available in such library.

REFERENCES

[1] E. Alba, G. Luque, J. Garcia;Nieto, G. Ordonez, and G. Leguizamon. Mallba, a software library to design efficient optimisation algorithms. In *Int. J. Innov. Comput. Appl.*, pages 74–85, 2007.

[2] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Component Software Series. Addison-Wesley, Londres, 2000.

[3] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné. Deap: A python framework for evolutionary algorithms. In *Proceedings of GECCO2012*, pages 85–92, 2012.

[4] H. Diosa, S. Rojas-Galeano, and L. Garzon. A component-based GA architecture, 2014. Available at http://arquisoft.udistrital.edu.co/portal/web/guest/proy-compAG.

[5] B. Dmitry. Open beagle: a generic framework for evolutionary computations. *Genetic Programming and Evolvable Machines*, 12(3):329–331, September 2011.

[6] P. García-Sánchez, M. I. García Arenas, A. M. Mora, P. A. Castillo, C. Fernandes, P. de las Cuevas, G. Romero, J. González, and J. J. Merelo. Developing services in a service oriented architecture for evolutionary algorithms. In *Proceedings of GECCO2013*, pages 1341–1348, 2013.

[7] M. Keijzer, J. J. M. Guervós, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *Selected Papers from the 5th European Conference on Artificial Evolution*, pages 231–244, London, UK, UK, 2002. Springer-Verlag.

[8] C. Luer and D. Rosenblum. Uml component diagrams and software architecture. In *Workshop at the 23rd International Conference on Software Engineering*, volume 1, pages 1–4, Irvine, California, 2001.

[9] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of GECCO 2011*, pages 1723–1730, Dublin, Ireland, 2011.

[10] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[11] K. Meffert and N. Rotstan. Jgap, java genetic algorithms package. Available at http://jgap.sourceforge.net.

[12] M. I. of Technology. Galib, a c++ library of genetic algorithm components. Available at http://lancet.mit.edu/ga/, 1995.

[13] S. Rojas-Galeano and N. Rodriguez. Goldenberry: Eda visual programming in orange. In *Proceedings of GECCO2013*, pages 1325–1332, Amsterdam, The Netherlands, 2013. ACM.

[14] R. N. Shaw, M. B. Grieve, A. T. Carpenter, and R. Ansorge. Cambridge genetic algorithmb software package. Recovered on July 3rd, 2013, from http://www.bss.phy.cam.ac.uk/˜rea1/camGASP/camGASP.pdf, 2001.

[15] C. Szyperski. *Independently Extensible Systems - Software Engineering Potential and Challenges*. Queensland University of Technology, Brisbane, Australia, 1996.

[16] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[17] S. Ventura, C. Romero, A. Zafra, J. Delgado, and C. Hervas. Jclec: a java framework for evolutionary computation. *Soft Computing*, 12(4):381–392, October 2007.

[18] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. *Advanced Methods and Applications in Computational Intelligence*, volume 6, chapter Architecture and Design of the HeuristicLab Optimization Environment, pages 197–261. Springer, 2014.

[19] D. R. White. Software review: The ecj toolkit. *Genetic Programming and Evolvable Machines*, 13(1):65–67, Mar. 2012.

[20] B. Zupan and J. Demsar. Orange: Data mining fruitful and fun. *15th International Multiconference on Information Society*, 2012.