An Approach for QoS-aware Service Composition based on Genetic Algorithms

Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Maria Luisa Villani
canfora@unisannio.it, dipenta@unisannio.it, r.esposito@unisannio.it, villani@unisannio.it
RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano 82100 Benevento, Italy

ABSTRACT

Web services are rapidly changing the landscape of software engineering. One of the most interesting challenges introduced by web services is represented by Quality Of Service (QoS)–aware composition and late–binding. This allows to bind, at run–time, a service–oriented system with a set of services that, among those providing the required features, meet some non–functional constraints, and optimize criteria such as the overall cost or response time. In other words, QoS–aware composition can be modeled as an optimization problem.

We propose to adopt Genetic Algorithms to this aim. Genetic Algorithms, while being slower than integer programming, represent a more scalable choice, and are more suitable to handle generic QoS attributes. The paper describes our approach and its applicability, advantages and weaknesses, discussing results of some numerical simulations.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Performance, Reliability

Keywords

Service–oriented software engineering, QoS–aware composition

1. INTRODUCTION

The rapid diffusion of web services and of service–oriented systems constitutes an important step towards a radical change in the software development process. Service–Oriented software engineering represents a natural evolution of Component-Based Software Engineering (CBSE). In CBSE, a component integrator searches for reusable components and then *physically* integrates them into a new system using some glue code. A web service exports a piece of functionality on a network server. Web services are used by (remotely) invoking service operations. Thus, there is no need to integrate the service with the application under development. Service interface publication, service discovery and service invocation are performed using XML-based standards, known as WSDL, UDDI and SOAP [16]. In particular, SOAP is a XML-encoded RPC protocol built over well–known TCP/IP application protocols such as HTTP or SMTP. A service oriented system is therefore composed of some service invocations, orchestrated using glue code [14] or some specific web service orchestration language (e.g., BPEL4WS [2] or WSCI [15]).

One of the biggest promises of service–oriented systems is the use of late–binding mechanisms. Given a specific feature needed in a service orchestration (this feature will be hereby referred to as an *abstract service*), several services (referred to as *concrete services*) realizing such a feature may be available. In Figure 1, circles represent an orchestration of *abstract services*, while the corresponding *concrete services* are represented as rectangles.

All *concrete services* corresponding to an *abstract service* are functionally equivalent and thus are replaceable by each other. The choice between them can be dictated by non–functional properties, referred to as Quality of Service (QoS) attributes. One may decide to choose the cheapest service, the fastest, or maybe a compromise between the two. According to Std. ISO 8402 [11] and ITU [12], QoS may be defined in terms of attributes such as price, response time, availability, reputation (further details can be found in Cardoso's PhD thesis [3]). Moreover, it may be possible to have some domain-specific QoS attributes (e.g., a temperature service could have QoS attributes such as precision or refresh frequency). Finally, an user may specify constraints on the values of some attributes (e.g., the price cannot be greater than a given value), which could influence the choice. On the other hand, the service provider can estimate ranges for the QoS attribute values as part of the contract with potential users (i.e., the Service Level Agreement (SLA)).

As it will be detailed in Section 3, given an orchestration, a relevant problem is to determine the set of concretizations (i.e., bindings between abstract and concrete services) that satisfy the QoS constraints imposed by the SLA, and optimize some fitness criteria chosen by the service integra-

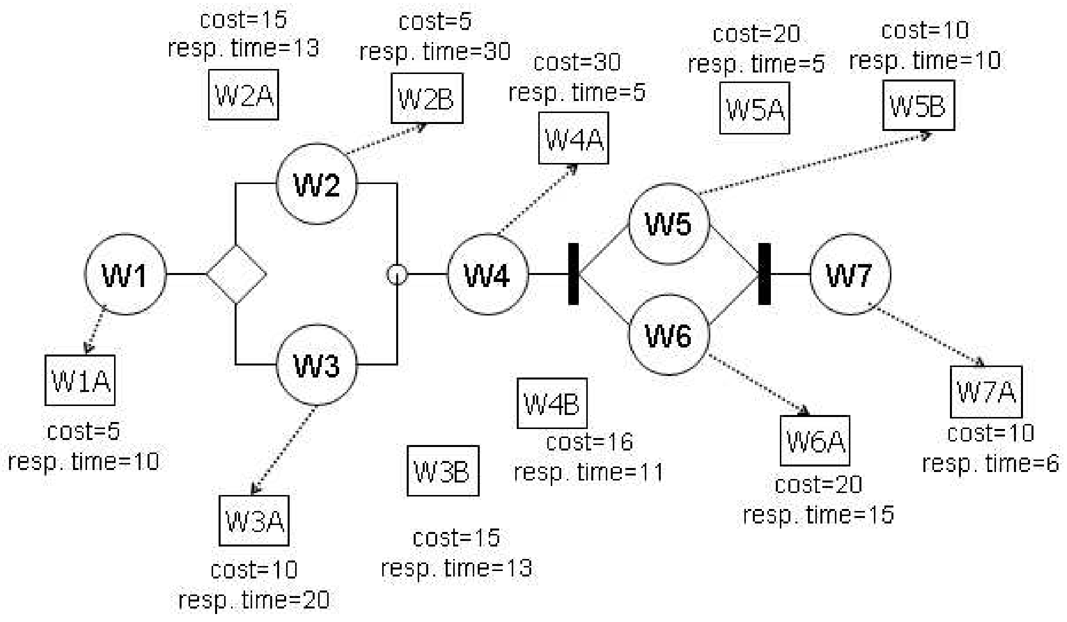

Figure 1: Example of workflow with bindings between abstract and concrete services

tor (e.g., minimize the cost). Finding the solution of such a problem, known as QoS–aware composition, is NP–hard. Some approaches, mostly based on linear integer programming, have been proposed in literature (e.g., the approach of Zeng et al. [17]).

It is particularly important for the QoS aware composition to be fast. Especially for interactive systems, long delays may be unacceptable. For example, the user of a booking ticket system might not want to wait for a long time while the system searches for candidate services offering flight tickets with the lowest booking fare. Gaining a few cents after several minutes of waiting may make the user disappointed.

A fast composition is also required to replan a service composition during the execution, because the actual QoS deviates from the estimated one and this could cause constraint violation, or simply because some services might not be available. In this case the composition time influences the overall service response time, thus it should be kept as low as possible.

This paper proposes to tackle the QoS–aware composition problem using Genetic Algorithms (GAs). Advantages and weaknesses of GAs will be compared with those of linear integer programming, that is the most widely adopted approach.

The remainder of this paper is organized as follows. After a review of the literature in Section 2, Section 3 details the proposed approach. In particular, after the QoS composition rules are shown, the evolutionary approach for QoS-aware composition is described. Section 4 reports and discusses results obtained in the simulations. Finally, Section 5 concludes.

2. RELATED WORK

Dynamic service discovery and QoS–aware service composition offer interesting applications of constraint handling methods and search strategies from operational research or artificial intelligence. In fact, in service-oriented architectures, constraint sets are used to express functional and non-functional properties of the services to search for or to monitor, and the service may be selected according to some QoS optimum criteria ([4], [5], [13]).

More challenging is the case of composite services whose structures are described by abstract processes, i.e, processes containing abstract services, where the aim is to find the best combination of concrete services, from the QoS point of view, at run time. This combination may change during execution due to the dynamic nature of the web services environment, therefore one important requirement is that a feasible, SLA-compliant solution should be found in a short time. Integer programming (IP) solutions to this problem have been proposed in some recent papers ([1], [17]). Both works assume linearity of the constraints and of the objective function, whose expression is similar to that described in Sub-section 3.4. In fact, Zeng et al. [17] essentially focus on the cost, response time, availability and reliability attributes, where logarithmic reductions are used for the multiplicative aggregation functions, and the model is claimed to be extensible with respect to other similarly behaving attributes. The need to deal with more general constraint criteria, such as service dependencies or user preferences, is strengthened in the work of Aggarwal et al. [1]. However, they do not explain how the optimization problem could be solved. In this paper, we advocate the use of GAs instead, as any kind of constraint could be handled, and provide some empirical data to assess the performance of our method against the (linear) integer programming solution in the service composition setting. An alternative to GAs could be to use nonlinear integer programming techniques, but the maturity of the available tools is questionable, and the application of these methods in our setting requires further investigation. A survey of some nonlinear techniques is contained in a paper by Grossmann [9].

GAs are unconstrained procedures by their nature, and so it is necessary to find ways to integrate constraint-handling techniques during the search process. Usually, this is achieved through penalty functions, i.e., constraints are incorporated in the fitness function so that their satisfaction is guaranteed for the optimal solution. An extensive survey of the constraint-handling techniques proposed in literature in the context of evolutionary algorithms is contained in a paper by Coello Coello [6]. In particular, several kinds of penalty functions are listed in the paper, together with an analysis of advantages/disadvantages for each of them. To solve our problem, we have used a simple distance-based penalty approach, that has turned successful also in scheduling problems presented in a work by Fang [8]. More complex dynamic or adaptive functions cited in the survey are not really worth in our case where the amount of constraints is usually small and so extra computational costs can be avoided.

Another approach for constraint handling derives from Artificial Intelligence, namely Constraint Logic Programming (CLP). CLP has the advantage of a richer modeling capability, but providing a good model for the problem at hand has implications on the performance of the approach. The peculiarity of CLP is constraint propagation, i.e., constraints are used to identify allowed subdomains for the variable values, which is continuously applied during the search. Obviously, this approach can lead to a more efficient search but it is expensive itself. Again, as the priority here is optimization, while constraints are expected to be few and rather simple, we think that a GAs-based approach is more appropriate. Comparative studies of all these approaches on some specific problems are described, for example, in papers by Craenen et al. [7], and by Helm et al. [10].

3. APPROACH DESCRIPTION

As described in the introduction, this work aims to propose an approach, based on GAs, to quickly determine a set of *concrete services* to be bound to the *abstract services* composing the workflow of a composite service. Such a set needs to:

1. meet QoS constraints, established in the SLA. For example, the service user may have a limited budget and thus the cost is constrained, or he/she cannot accept a response time above a given limit. Often, both local constraints (e.g., a particular operation could not have a cost above a given limit) and global constraints (e.g., the total response time is constrained) need to be met;

2. optimize a function of some other QoS parameters. For example, the user may want to minimize the response time while keeping the cost below a limit.

In the sequel we shall consider a composite service S of n *abstract services*, $S \equiv \{s_1, s_2, \ldots, s_n\}$, whose structure is defined through some workflow description language. Each component s_i can be bound to one of the m *concrete services* $cs_{i,1}, \ldots, cs_{i,m}$, which are functionally equivalent.

Before describing the GA used to find solutions to the optimization problem, we need to describe how to compute the QoS of a composite service, starting from the QoS attribute values of the component services.

3.1 Computing the QoS of Composite Services

The approach for computing the QoS of a composite service is similar to what proposed by Cardoso [3]. For a Switch construct in the workflow, each Case statement is annotated with the probability to be chosen. For example, for a workflow containing a Switch composed of two Cases, with costs C_1 and C_2 respectively and probabilities p and $1-p$, the overall cost is computed as follows:

$$p\,C_1 \;+\; (1-p)\,C_2 \tag{1}$$

Clearly, probabilities are initialized by the workflow designer, and then eventually updated considering the information obtained by monitoring the workflow executions.

Loops are handled differently from Cardoso [3], that basically proposes to adopt a mechanism (based on the probabilities of entering/exiting the Loop) as for the Switch construct. Our approach is more similar to what proposed by Zeng et al. [17], i.e., Loops are annotated with an estimated number of iterations k. Instead of unfolding Loops (like Zeng et al.), here the QoS of the Loop is computed taking into account the factor k. For example, if the Loop compound has a cost C_l, then the estimated cost of the Loop will be $k\,C_l$.

This approach for handling Loops presents two advantages:

- It allows for a quick computation of the overall workflow QoS, without the need to unfold Loops;

- The estimated QoS accounts for the estimated number of Loop iterations.

Given a *concretization* of a composite service, i.e., a composite service description where each *abstract service* has been bound to one of its corresponding *concrete services*, the overall QoS can be computed by applying the rules described in Table 1, which shows an aggregation function for each pair workflow construct and QoS attribute. While for some standard QoS attributes the aggregation function has been explicitly specified ([3], [17]) there may be other attributes (for example, domain-dependent attributes) for which the aggregation function is user–specified (see the last row of Table 1).

The table is not complete (it only contains rules to be used in our examples) and, except that for Loops, the aggregation functions correspond to those proposed by Cardoso [3]. These functions are recursively defined on compound nodes of the workflow. Namely, for a Sequence construct of tasks $\{t_1, \ldots, t_m\}$, the *Time* and *Cost* functions are additive while *Availability* and *Reliability* are multiplicative. The Switch construct of Cases $1, \ldots, n$, with probabilities $p_{a1}, \ldots, p_{an}$ such that $\sum_{i=1}^{n} p_{ai} = 1$, and tasks $\{t_1, \ldots, t_n\}$ respectively, is always evaluated as a sum of the attribute value of each task, times the probability of the Case to which it belongs. The aggregation functions for the fork (named Flow in BPEL4WS) construct, are essentially the same as those for the Sequence construct, except for the *Time* attribute where this is the maximum time of the parallel tasks $\{t_1, \ldots, t_p\}$. Finally, a Loop construct with k iterations of task t is equivalent to a Sequence construct of k copies of t.

QoS Attr.	Sequence	Switch	Flow	Loop
Time (T)	$\sum_{i=1}^{m} T(t_i)$	$\sum_{i=1}^{n} p_{ai} * T(t_i)$	$Max\{T(t_i)_{i\in\{1\ldots p\}}\}$	$k * T(t)$
Cost (C)	$\sum_{i=1}^{m} C(t_i)$	$\sum_{i=1}^{n} p_{ai} * C(t_i)$	$\sum_{i=1}^{p} C(t_i)$	$k * C(t)$
Availability (A)	$\prod_{i=1}^{m} A(t_i)$	$\sum_{i=1}^{n} p_{ai} * A(t_i)$	$\prod_{i=1}^{p} A(t_i)$	$A(t)^k$
Reliability (R)	$\prod_{i=1}^{m} R(t_i)$	$\sum_{i=1}^{n} p_{ai} * R(t_i)$	$\prod_{i=1}^{p} R(t_i)$	$R(t)^k$
Custom Attr. (F)	$f_S(F(t_i))$ $i \in \{1\ldots m\}$	$f_B((p_{ai}, F(t_i)))$ $i \in \{1\ldots n\}$	$f_F(F(t_i))$ $i \in \{1\ldots p\}$	$f_L(k, F(t))$

Table 1: Aggregation functions per workflow construct and QoS attribute

3.2 Searching for a solution with Genetic Algorithms

Differently from other approaches proposed in literature, such as linear integer programming, GAs do not impose constraints on the linearity of the QoS composition operators (and thus of objective function and constraints). This permits the use of our approach for all possible (even customized) QoS attributes, without the need for linearization.

To let the GA search for a solution of our problem, we first need to encode the problem with a suitable genome. In our case, the genome is represented by an integer array with a number of items equals to the number of distinct *abstract services* composing our service. Each item, in turn, contains an index to the array of the *concrete services* matching that *abstract service*. Figure 2 gives a better idea of how the genome is made.

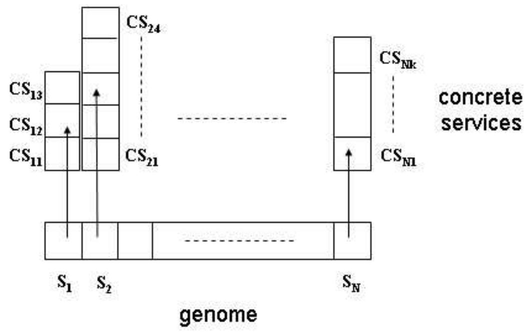

Figure 2: Genome Encoding

The crossover operator is the standard two-points crossover, while the mutation operator randomly selects an *abstract service* (i.e., a position in the genome) and randomly replaces the corresponding *concrete service* with another one among those available. Clearly, *abstract services* for which only one *concrete service* is available are taken out from the GA evolution.

The problem can now be modeled by means of a fitness function and, eventually, some constraints. The fitness function needs to maximize some QoS attributes (e.g., reliability), while minimizing others (e.g., cost). When user–defined, domain–specific QoS attributes are used, the specification of the fitness function is left to the workflow designer.

In addition, the fitness function must penalize individuals that do not meet the constraints and drive the evolution towards constraint satisfaction. Let us suppose that the composite service QoS has a set of constraints defined as follows:

$$cl_i(g) \leq 0, \; i = 1, \ldots, n \tag{2}$$

We define the *distance from constraint satisfaction* as:

$$D(g) = \sum_{i=1}^{n} cl_i(g) * y_i \tag{3}$$

where:

$$\begin{cases} y_i = 0 & cl_i(g) \leq 0 \\ y_i = 1 & cl_i(g) > 0 \end{cases} \tag{4}$$

The fitness function for a genome g is then defined as follows:

$$F(g) = \frac{w_1 \; Cost(g) \; + \; w_2 \; Response\ Time(g)}{w_3 \; Availability(g) \; + \; w_4 \; Reliability(g)} + w_5 \; D(g) \tag{5}$$

QoS attribute factors (i.e., Availability(g), Reliability(g), etc.) are normalized in the interval $[0, 1)$. $w_1, \ldots, w_5$ are real, positive weights of the different fitness factors. In particular, $w_1, \ldots, w_4$ indicate the important a service integrator (or user) gives to a particular QoS attribute, while w_5 weights the penalty factor.

Finally, it is necessary to define a stop criterion for the GA. One possible criterion is to fix a maximum number of iterations. Alternatively, it is possible to:

1. Iterate (with a maximum number of generations equal to $maxgen_{constr}$) until the constraints are met (i.e., $D(g) = 0$). If this does not happen within $maxgen_{constr}$ generations, then no solution has been found;

2. Once $D(g) = 0$, iterate over a further, fixed number of generations $maxgen_{fitness}$, that may be a percentage of $maxgen_{constr}$. Alternatively, iterate until the best fitness individual remains unchanged for a given number of generations.

3.3 Dynamic Fitness Function

The fitness function defined in equation (5) contains a static penalty for individuals that violate constraints. In other words, the penalty is the same at each generation. If, as usual, the weight w_5 for this penalty factor is high, there is a risk that also individuals violating the constraints but "close" to a good solution could be discarded.

The alternative is to adopt a dynamic penalty, i.e., a penalty having a weight that increases with the number of generations. This may allow, for the early generations, to also consider some individuals violating the constraints. After a number of generations, the population should be able to meet the constraints, and the evolution will try to improve only the rest of the fitness function.

The dynamic fitness function (to be minimized) is defined as follows:

$$F(gen, g) = \frac{w_1\ Cost(g)\ +\ w_2\ Response\ Time(g)}{w_3\ Availability(g)\ +\ w_4\ Reliability(g)} + \quad (6)$$
$$w_5\ D(g) * \frac{gen}{maxgen}$$

where *gen* is the current generation, while *maxgen* is the maximum number of generations.

3.4 QoS–aware Composition using Integer Programming

As described in the introduction and in Section 2, linear integer programming is one of the most adopted tools to solve a QoS–aware composition problem. An analytic description of this approach is out of the scope of this paper, for details see Zeng et al. [17] or Cardoso [3].

However, it is necessary to highlight advantages and weaknesses of this solution, before performing the performance comparison in Section 4.

Given n *abstract services* $S_1, \ldots, S_n$ invoked in our application (or composite service), and suppose that each *abstract service* S_i can be bound to m_i *concrete services*, we need to define our integer programming problem in terms of $\sum_{i=1}^{n} m_i$ integer variables $y_{i,j}$ such that:

$$\begin{cases} \sum_{j=1}^{m_i} y_{i,j} = 1 \\ y_{i,j} \in \{0, 1\} \end{cases} \quad (7)$$

Variables $y_{i,j}$ indicates whether the *abstract service* S_i is bound to the *concrete service* $CS_{i,j}$. As a consequence of what described above, the number of required variables tends to explode with the number of service invoked and, above all, with the number of concretizations available. On the contrary, for GA the genome size is bound to the number n of *abstract services*. The number of possible concretizations only augments the search space.

The second weakness is represented by the need to have linear aggregation functions for the QoS attributes. Even for some standard attributes, such as the availability or reliability, this is not the case. And, while for the latter a linearization can be adopted [17], this is quite difficult in other cases (e.g., computing the response time for a Flow). Finally, any user-defined attribute needs to have a linear (or at least linearized) aggregation function. An interesting alternative would be the use of non–linear integer programming, however scalability problems would still arise.

On the other hand, integer programming is often faster than GAs. When the workflow size and the number of concretizations are limited, and there is no need to use non–linear aggregation functions, integer programming is therefore preferable.

4. EMPIRICAL STUDY

In this section we analyze the performances of the proposed approach:

- observing the evolution of fitness factors over the GA generations;
- comparing different fitness functions (i.e., with static and dynamic constraint penalty); and
- comparing GAs with integer programming.

We used an elitist GA where the best 2 individuals were kept alive across generations, a crossover probability of 0.7, a mutation probability of 0.01 and a population of 100 individuals. The selection mechanism adopted was the roulette wheel selection. In our fitness function, we gave equal weights to the different QoS attributes, eventually disabling some attributes in some case studies.

For each experiment, GA was executed 50 times and average values are reported. Standard deviation was always below the 5% of the mean values.

Finally, since the purpose of our experiment was to compare different fitness functions and to compare integer programming with GAs, we used as a stopping criterion:

- A number of generations (100 in our case study) after which no fitness improvement was observed; and
- when comparing integer programming with GAs, a number of generation the GA required to reach the same solution as of integer programming (or a solution close to it, as explained in Section 4.2.

When integrating the GA composition approach in a service execution tool, however, it is necessary to adopt the approach detailed in Section 3.2, that automatically calibrates the stopping generation.

4.1 Comparing the fitness functions

To compare the different fitness functions, we present here results obtained optimizing a workflow containing 25 invocations of 16 distinct *abstract services*, and having a cyclomatic complexity equals to 17.

Figure 3 reports the evolution of different QoS parameters for both fitness functions. As shown, the optimization problem is constrained on *cost* and *response time*. The evolution shows how the GA is able to find a solution that meets the constraint and, at the same time, optimizes the different fitness parameters (i.e., maximizing the availability while keeping low cost and response time).

For our optimization problem, the *dynamic* fitness does not outperform the *static* fitness. Even different calibrations of the fitness weights did not help. t-tests with significance level $\alpha = 5\%$ showed that differences were not significant, except for the availability where the dynamic fitness started faster, although at the end of the evolution the result achieved was not different.

Experiments were also replicated on workflows of different sizes and complexity, basically confirming the results reported above.

4.2 Comparing GAs with Integer Programming

To compare performances of GAs and Integer Programming, we computed the time to optimize a static fitness function (5) on cost and time (i.e., with *Reliability* and *Availability* constantly set to 1[1]).

[1]To correctly perform the comparison, we did not consider availability QoS factors, nor workflow containing the Flow construct (that is non–linear when aggregating response–times).

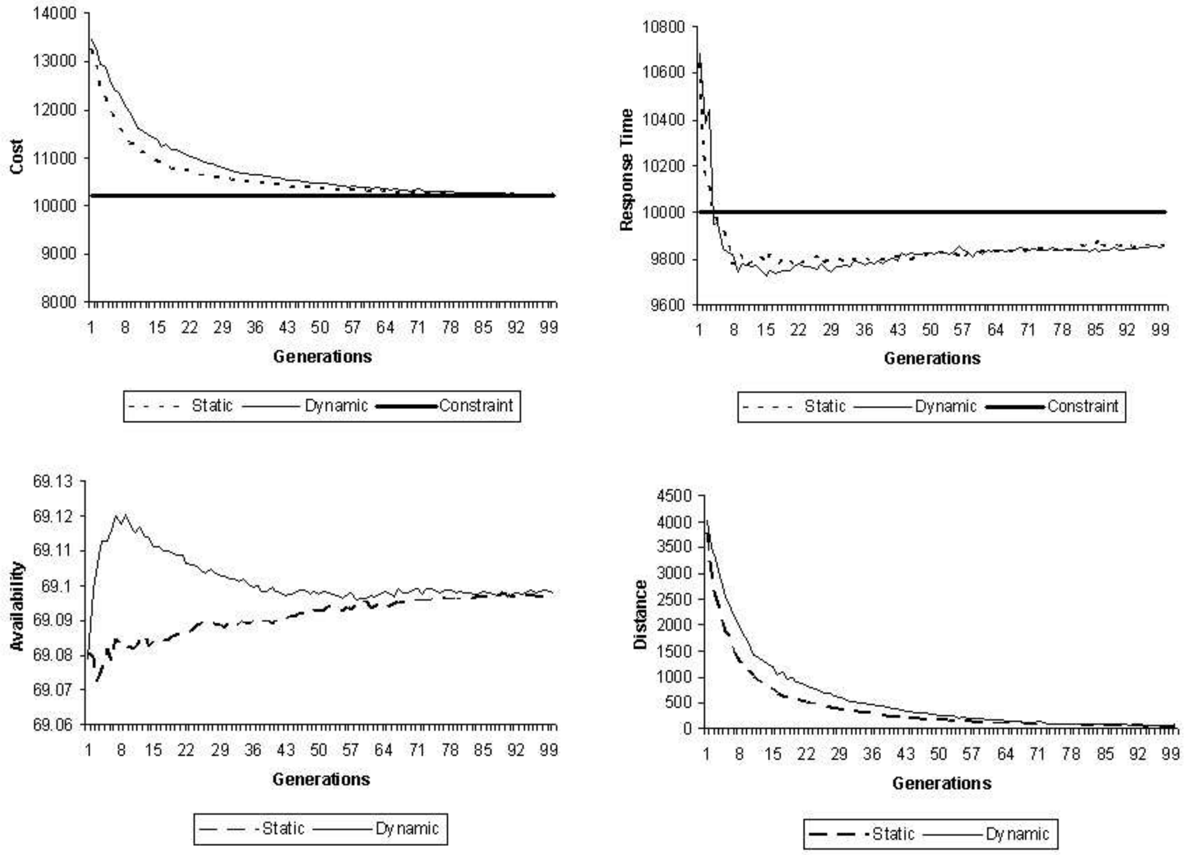

Figure 3: Evolution of fitness parameters - Comparing static and dynamic fitness

As a case study for comparison, we considered a workflow containing 18 invocations of 8 distinct *abstract services* and having a cyclomatic complexity 5 (i.e., containing 2 nested loops and a two–way switch). For each *abstract service*, we considered a variable (average) number of available *concrete services*, i.e., 5, 10, 15, 20, 25. Then, we compared the performance of integer programming and GAs as follows:

- For *Integer Programming* we measured the CPU *user time* to produce the solution;
- For *GAs*, we measured the CPU *user time* to reach a solution meeting the constraint and with *cost* and *response time* differing of no more than 1% from the *Integer Programming* solution. In other words, we wanted to compare convergence times of *Integer Programming* and *GAs* for the same (or almost the same) achieved solution.

Both approaches were executed 50 times, and then the average values were computed. In both cases, the standard deviation was less than 5% of the mean value.

Our GA was implemented in Java using a freely available library[2], while *LPSolve*[3] was used for *Integer Programming.* Experiments were performed on a 3 GHz Intel Pentium^{TM}, 512 Mb of RAM, Microsoft Windows XP^{TM} and J2SDK 1.5.

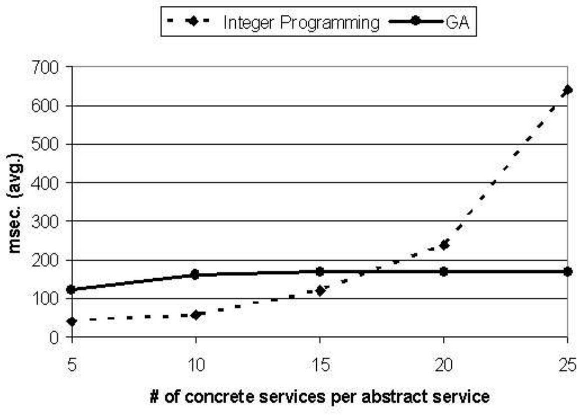

Figure 4: Comparison between Integer Programming and GAs

Figure 4 shows the results of our comparison. When the number of *concrete services* is small (5, 10) Integer Programming outperforms GAs, substantially confirming the choice made by other works that adopted this kind of approach [17,

[2] *http://sourceforge.net/projects/java-galib*
[3] *http://www.cs.sunysb.edu/ ~algorith/implement/lpsolve/implement.shtml*

4]. For about 17 *concrete services* the performances of the two approaches tend to be the same. Then, while the GA is able to keep its timing performance almost constant, this is not the case for Integer Programming, for which we see an exponential grow due to the corresponding increment of the number of variables needed to represent the problem (see Section 3.4).

Then, we investigated whether the performance variation was due to the increase of the search space, or just to the increase of the number of variables for integer programming. To this aim, we increased the number of *concrete services* for a few *abstract services* only. Results obtained basically confirmed what was described above. Also, results were also confirmed by other experiments performed with workflows having different size and cyclomatic complexity.

The lesson learned from our esperimentation is basically that, when we have a large number of *concrete services* available for each *abstract service*, *GAs* should be preferred instead of *Integer Programming*. This, in the authors' opinion, will be the case of widely used services, such as hotel booking, weather services or ecommerce services. On the other hand, whenever the number of *concrete services* available is limited, *Integer Programming* is to be preferred. This would be the case of very specific (e.g., scientific computation) services. Finally, we think that a binding mechanism should be able, time to time, to select the best approach to be adopted, to always ensure a reasonable binding time. The latter constitutes an important requirement for many scenarios, such as interactive or (soft) real time service–oriented systems.

5. CONCLUSIONS

This paper proposed a GA–based approach for QoS–aware service composition, i.e., to determine a set of *concrete services* to be bound to *abstract services* contained in a orchestration to meet a set of constraints and to optimize a fitness criterion on QoS attributes. Compared with linear *Integer Programming*, the most widely adopted approached, GA permits to deal with QoS attributes having non–linear aggregation functions. Also, GA is able to scale–up when the number of concretizations increases. Finally, to deal with constraints, it is possible to adopt a fitness function with both *static* or *dynamic* penalty, even if we did not experience a significant difference, at least for our case studies and calibration mechanisms.

Future work will aim to apply the proposed approach to some large–scale service–oriented system, and to perform a thorough comparison of GA with other non–linear technique, such as non–linear integer programming. Multi–objective fitness functions will also be considered as an alternative to single–objective fitness functions where factors are aggregated using a weighted sum.

6. ACKNOWLEDGMENTS

This work is framed within the European Commission VI Framework IP Project SeCSE (Service Centric System Engineering) (*http://secse.eng.it*), Contract No. 511680.

7. REFERENCES

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in METEOR-S. In *Proc. IEEE International Conference on Services Computing (SCC'04)*, pages 23–30, Shanghai, China, Sept. 2004.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. T. D. Smith, I. Trickovic, and S. Weerawarana. Business process execution language for web services. *http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/*.

[3] J. Cardoso. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, Univ. of Georgia, 2002.

[4] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, April 2004.

[5] F. Casati and M. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–162, May 2001.

[6] C. A. Coello Coello. Theoretical and numerical constraint-handling tehniques used with evolutionary algorithms: A survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12), January 2002.

[7] B. Craenen, A. Eiben, and J. van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 7(5):281–308, October 2003.

[8] H. Fang. *Genetic Algorithms in Timetabling and Scheduling*. PhD thesis, Univ. of Edimburg, 1994.

[9] I. Grossmann. Review of nonlinear mixed-integer and disjunctive programming techniques. *Optimization and Engineering*, 3(3):227–252, September 2002.

[10] T. Helm, S. Painter, and W. Oakes. A comparison of three optimization methods for scheduling maintenance of high cost, long-lived capital assets. In *Proc. of the 2002 Winter Simulation Conference (WSC'02)*, pages 1880–1884, San Diego, California, Dec. 2002.

[11] ISO. *UNI EN ISO 8402 (Part of the ISO 9000 2002): Quality Vocabulary.*

[12] ITU. *Recommendation E.800 Quality of service and dependability vocabulary.*

[13] E. R. U.Greiner. Quality-oriented handling of exceptions in web-service-based cooperative processes. In *Proc. EAI-Workshop 2004 - Enterprise Application Integration*, pages 11–18. GITO-Verlag, 2004.

[14] J. Voas, A. Ghosh, G. McGraw, and K. Miller. Glueing together software components: How good is your glue? In *Proceedings of the Pacific Northwest Software Quality Conference*, pages 90–97, Oct 1996.

[15] W3C Working Group. Web Service Choreography Interface. http://www.w3.org/TR/wsci/.

[16] W3C Working Group. Web services architecture. *http://www.w3.org/*.

[17] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5), May 2004.