Available online at www.sciencedirect.com

ScienceDirect

Procedia Engineering 69 (2014) 1162 – 1171

Procedia Engineering

www.elsevier.com/locate/procedia

24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013

A Genetic Algorithm for Automated Service Binding

Raluca Iordache*, Florica Moldoveanu

University POLITEHNICA of Bucharest, Splaniul Independentei nr. 313,Bucharest, CP 060042, Romania

Abstract

Binding concrete web services to the tasks involved in an orchestration model is an important step in dynamic web service composition. In our previous work on automated service binding, we have described a method of combining a powerful technique for computing the QoS of a composite service with our own approach of expressing preferences related to the trade-offs between the various QoS parameters of a composite service. In this paper, we introduce a genetic algorithm that finds the best mapping of concrete services to the tasks involved in the composition. The algorithm uses the method presented in our previous work in order to estimate the fitness of the mappings that make up a population of candidate solutions. We describe how the challenges posed by the use of this method have influenced the design of our algorithm and report the experimental results showing the effectiveness of our approach.

Keywords: QoS preferences; service binding; QoS aggregation; genetic algorithms

1. Introduction

Composite web services that achieve advanced functionality can be constructed by integrating individual web services. Since web service composition is a complex task, there are ongoing efforts to automate this process. Web services have a dynamic nature: at every moment, a service may cease to exist or a new one may become available. Non-functional characteristics, such as the quality of service (QoS), are also subject to frequent changes. Therefore, much research is directed toward automated, dynamic web service composition approaches.

Web service composition is a complex task, which involves three challenging steps: (1) composite web service specification, (2) selection of the component web services and (3) execution of the composite web services [1]. In

* Corresponding author.Tel.: 0049-171-5349595
E-mail address: riordache@outlook.com

this paper, we focus on the second step, whose goal is to bind concrete web services to the activities involved in the composition, in order to produce the most suitable composite service.

In our previous work [2], we have presented a *binding-as-a-service* (BaaS) approach that combines two powerful technologies. The first one is our method of dealing with QoS preferences [3], which offers great flexibility in managing trade-offs, but is at the same time very intuitive. The second one is the QoS aggregation method of Yang *et al.* [4], which has the major advantage of being able to deal with unstructured orchestration models.

Optimizing the aggregated QoS of a composite service is an NP-hard problem. An exhaustive search is feasible only for simple compositions models, having a small number of tasks and a small number of available services for each task. We introduce a genetic algorithm that uses the above mentioned approach in order to find the best mapping of concrete services to the tasks involved in the service composition. The experimental results show the effectiveness of our algorithm.

The rest of this paper is organized as follows: Section 2 discusses the problem of estimating the QoS of a composite service and presents the aggregation technique chosen in this work. Section 3 describes our conditional lexicographic approach for expressing QoS preferences. Section 4 details our genetic algorithm and section 5 presents the experimental results. The last section concludes the paper.

2. The aggregated QoS of composite services

Various solutions have been proposed for estimating the aggregated QoS of a composite service, but they differ in the restrictions imposed on the topology of the composition. Most of them are limited to orchestration models that can be represented as well-structured workflows. Yang *et al.* [4] have introduced a method that overcomes these restrictions. This method, which is used in our *binding-as-a-service* (BaaS) approach, is presented in the remaining of this section.

The input of this method is an orchestration model together with a binding that maps tasks to component services. An orchestration model is a directed graph with execution probabilities attached to its edges. The orchestration models are decomposed into *orchestration components*, which are subgraphs with a single-entry and single-exit point. The QoS is computed in a bottom-up manner for each orchestration component. For well-structured orchestration models, different aggregation formulas are provided depending on the type of the QoS attribute, which can be classified into three categories: *critical path*, *additive* and *multiplicative*.

A preliminary step of the QoS aggregation method is to use the block-structuring technique introduced in [5] to transform an unstructured orchestration model into a *maximally-structured* orchestration model.

The components that are irreducible using this technique are called *rigid components* and they are of two types: irreducible Directed Acyclic Graphs (DAG) and irreducible multiple-entry, multiple-exit (MEME) loops. The authors of [4] provide an algorithm that transforms irreducible DAG components in equivalent choice components. Irreducible MEME loops can be transformed using the block-structuring technique into equivalent rigid components where the concurrency is fully encapsulated within child components. For these equivalent components, the expected number of times that a node in the MEME loop is visited can be calculated using standard methods. This allows computing the QoS of the irreducible component by applying the aggregation formulas characteristic to each category of QoS parameters.

3. QoSPref - The conditional lexicographic approach for the elicitation of QoS Preferences

Our approach to articulate the QoS preferences [3] is based on the observation that, when trying to find a set of rules allowing them to choose between several alternatives, people start by ranking their preferences, in accordance with their perceived importance. This action is equivalent to imposing a lexicographic order on the different criteria that have to be considered. Since using such a strict hierarchy is usually not sufficient to capture people's real preferences, they introduce additional rules that change the criteria priorities when some specific condition is met. Our method establishes a total order on the set of alternatives, by attaching conditions to lexicographic preferences and provides a preference specification language that can be used for authoring QoS preferences.

We illustrate our approach and the use of its associated specification language by considering an online trading system that offers services for trading various financial instruments. One of these services allows customers to buy domestic and foreign stocks. We consider that the following QoS attributes are interesting for the online trading system: execution time, cost, and reliability.

In order to be able to dynamically bind component services to the tasks specified in the composition model of the stock buying service, the online trading system must have an automated method of comparing composite services based on their QoS. The details provided in the next two paragraphs illustrate why this is not a trivial task.

The executives of this system try to maximize their profit, therefore they see the cost as the most important QoS parameter. However, they are willing to ignore small cost differences (not exceeding 10 cents) if the composite service with a higher cost has better values for reliability and execution time.

For the customers of this system, it is very important that trading orders are executed as soon as possible. Therefore, the online trading systems guarantees that the execution time of its stock buying service does not exceed 30 seconds. For every violation of this agreement, the owners of the online trading system must pay a penalty proportional with the delay. This means that, when comparing two composite services, the execution time becomes the most important parameter if at least one of the compared services has an execution time exceeding the 30 seconds limit.

In order to be able to articulate preferences for scenarios like the one above, our specification language provides four unary preference operators, which are shown in the table below :

Table 1. Preference operators

Preference operator	Meaning
AT_LEAST_ONE(condition)	condition(service1) OR condition(service2)
EXACTLY_ONE(condition)	condition(service1) XOR condition(service2)
DIFF(attribute)	|service1.attribute - service2.attribute|
ALL(condition)	condition(service1) AND condition(service2)

The first three operators take as argument a boolean formula, which usually involves one or more QoS attributes. The formula is evaluated twice, once for each of the web services to be compared. The two resulting boolean values are passed as arguments to the boolean operator (OR, XOR, or AND) associated with the given preference operator, in order to obtain the return value. The preference operator DIFF takes as argument a QoS attribute and returns the modulus of the difference of its corresponding values from the two web services compared.

Our specification language uses a *preferences* block that includes a comma separated list of entries, called *preference rules*, listed in the order of their importance.

A *preference rule* has three components: an optional *condition*, an *attribute* indicating the QoS dimension used in comparisons and a *direction* flag stating which values should be considered better.

In our specification language, the preferences corresponding to the above described scenario can be articulated as shown in Fig. 1.

```
preferences {
    [EXACTLY_ONE(execTime  > 30)] execTime : low,
    [DIFF(cost) > 10] cost : low,
    reliability: high,
    execTime: low,
    cost: low
}
```

Fig. 1. A specification of preferences.

In what follows, we use the notation $s_1 \succ s_2$ to indicate that the web service s_1 is preferred to the web service s_2, and the notation $s_1 \sim s_2$ to indicate that the service s_1 is indifferent to the web service s_2. Additionally, we introduce the notation $s_1 \succ_k s_2$ to indicate that the web service s_1 is preferred to the web service s_2 and that the preference rule k has been decisive in establishing this relationship. We also introduce the complementary operators $\prec$ and $\prec_k$, defined by the following relations:

$$s_1 \prec s_2, \text{ iff } s_2 \succ s_1$$

$$s_1 \prec_k s_2, \text{ iff } s_2 \succ_k s_1$$

An algorithm for comparing two web services based on the preferences expressed using our conditional lexicographic approach is shown in Fig. 2.

```
1   function compareServices(service1, service2, preferences)
2       for i ← 0 .. length(preferences) - 1 do
3           cond ← preferences[i].condition
4           attr ← preferences[i].attribute
5           dir ← preferences[i].direction
6           if cond = null OR cond(service1, service2)=true then
7               result ← compare(service1.attr, service2.attr, dir)
8               if result ≠ 0 then
9                   return {result, i}
10              end if
11          end if
12      end for
13      return null
14  end function
```

Fig. 2. Pairwise comparison of two web services.

The algorithm examines all entries in the *preferences* block in the order in which they appear (line 2). If the current preference rule has no attached condition or the attached condition evaluates to *true* (line 6), the values corresponding to the attribute specified by this entry are compared (line 7). The *compare* function returns a numerical value that is positive if the first argument is better, negative if the second argument is better and 0 if the arguments are equal. If the attribute values are not equal (line 8), the algorithm returns a tuple containing the result of the current comparison and the index of the preference rule that has been decisive in establishing the preference relationship (line 9). Otherwise, the algorithm continues its execution with the next preference rule. A null return value (line 13) indicates an indifference relation between the two web services, while a not-null tuple identifies a relation of type $\prec_k$ or $\succ_k$ between them.

An important feature of our specification language is its ability to capture intransitive preference. As a consequence of this, the pairwise comparison of the web service alternatives is not sufficient to impose a total order on these services. We illustrate this by considering a set of 5 composite web services with the aggregated QoS values specified in Table 2.

Table 2. Relevant QoS attribute values.

	WS_1	WS_2	WS_3	WS_4	WS_5
execTime	27	24	31	28	26
cost	536	548	520	525	540
reliability	0.97	0.96	0.98	0.98	0.96

Using the preferences specified in Fig. 1, the relations identified by the pairwise comparison of the 5 web services considered in our example are depicted in Table 3, where header notations use the format $i\ /\ j$ to indicate that the corresponding symbol in the line below represents the preference relation between the web service WS_i and the web service WS_j.

Table 3. Pairwise comparison of the 5 web services.

1/2	1/3	1/4	1/5	2/3	2/4	2/5	3/4	3/5	4/5
$\succ_2$	$\succ_1$	$\prec_2$	$\prec_3$	$\succ_1$	$\prec_2$	$\succ_4$	$\prec_1$	$\prec_1$	$\succ_2$

In the above table the following intransitive relationships can be observed:

$WS_1 \succ WS_2 \succ WS_5$

$WS_5 \succ WS_1$

In order to obtain a total order on the set of web service alternatives, we attach to each web service i a *score vector* of integer values: $V_i \in \mathbb{N}^{r+1}$, where r is the number of preference rules. The algorithm used to compute the score vectors is presented in Fig. 3, where n denotes the number of web service alternatives.

```
procedure createScoreVectors ()
    for i ← 1 .. n do
        for k ← 1 .. r do
            $V_i^k$ ← number of times service WS$_i$ is preferred to another web service
                                      due to the decisive rule k (i.e., due to a $\succ_k$ relation)
        end for
        $V_i^{r+1}$ ← number of times service WS$_i$ is indifferent to another web service.
    end for
end procedure
```

Fig. 3. Procedure to create the score vectors.

For the 5 web service alternatives considered in our example, the corresponding score vectors computed with the above algorithm are presented in Fig. 4.

	$\succ_1$	$\succ_2$	$\succ_3$	$\succ_4$	$\succ_5$	$\sim$
WS_1	1	1	0	0	0	0
WS_2	1	0	0	1	0	0
WS_3	0	0	0	0	0	0
WS_4	1	3	0	0	0	0
WS_5	1	0	1	0	0	0

Fig. 4. Score vectors of the 5 web service alternatives.

Using the score vectors, we are able to provide an algorithm for the ranking of web service alternatives. This algorithm is based on the function *compareScores*, described in pseudocode in Fig. 5. Again, r is used to denote the number of preference rules. The function takes as arguments two score vectors and returns a numerical value that is positive if the web service corresponding to the first score vector is preferred, negative if the web service corresponding to the second score vector is preferred and 0 if the corresponding web services are indifferent to each other.

```
1  function compareScores(V1, V2)
2      count1 ← $\sum_{i}^{r} V_1^i$
3      count2 ← $\sum_{i}^{r} V_2^i$
4      if count1 ≠ count2 then
5          return count1 - count2
6      end if
7      for i ← 1 .. r + 1 do
8          if $V_1^i \neq V_2^i$ then
9              return $V_1^i - V_2^i$
10         end if
11     end for
12     return 0
13 end function
```

Fig. 5. Function for score vector comparison.

For each of the two corresponding web services, the function computes the number of times it has been preferred to other web services (lines 2, 3). This computation does not take into account the number of times a web service has been found to be indifferent to another one (hence the sum is taken up to the value r, not $r + 1$).

If the previously computed values count_1 and count_2 are not equal (line 4), the web service with the higher value is chosen as the better one (line 5).

Otherwise, the algorithm scans each position in the score vectors (line 7) and if it finds different values, the web service corresponding to the higher value is chosen as the better one. The scanning of the values in the vector scores starts with the position corresponding to the first preference rule, because this is considered the most important one, and it ends with the position corresponding to the number of indifference relations (i.e., $r + 1$), because this is considered the least important one. If the score vectors are identical, the function returns 0 (line 12)

In contrast with the function *compareServices* presented in Fig. 2, the function *compareScores* induces a total order on the set of web service alternatives, thus allowing us to rank them accordingly. Using this algorithm, the 5 web service alternatives considered in our example will be ranked in the following order:

(WS_4, WS_1, WS_5, WS_2, WS_3),

with WS_4 being the best alternative.

4. Genetic algorithm

In this section, we describe the genetic algorithm used to find the best mapping of component services to tasks. The purpose of this algorithm is to be incorporated in a framework implementing our *binding-as-a-service* (BaaS) approach. A web service request sent to our BaaS provider must contain the following information:

- the orchestration model;
- the list of QoS attributes;
- for each task in the orchestration model, a list of concrete web services offering the required functionality;
- the QoS constraints;
- the QoS preferences.

The orchestration model is represented as a workflow with execution probabilities attached to its edges. If probabilities are missing, our implementation will assign default probabilities. Edges starting from an XOR gateway are assigned a probability of $1 / k$, where k is the number of outgoing edges of the given XOR gateway. All other edges are assigned a probability of 1.

The list of QoS attributes must contain information about the aggregation category of each attribute.

The list of concrete web services offering the required functionality of a given task must specify for each concrete web service its QoS values.

In the following paragraphs, we describe the algorithm used to find the best solution for a given BaaS request. A genetic algorithm maintains a population of chromosomes, where each chromosome encodes a possible solution of the problem. In our case, a chromosome encodes a possible mapping of web services to tasks, as shown in Fig. 6. The chromosome is structured as a vector of n elements, where n is the number of tasks in the orchestration model. The value of the element i in this vector is an integer indicating the index of the component service assigned to the task i. For example, in Fig. 6, there are 3 web services implementing the functionality of task 5: WS_5^1, WS_5^2 and WS_5^3. The solution encoded by the chromosome maps the second possible service (WS_5^2) to task 5. Therefore, the value of the 5th element of the chromosome is 2.

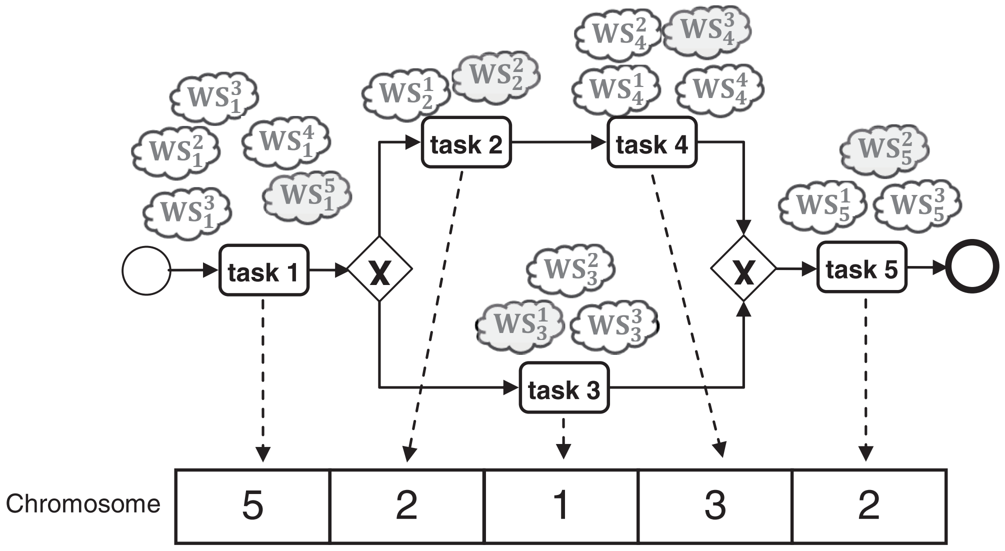

Fig. 6. The structure of a chromosome encoding a possible mapping of web services to tasks.

Our genetic algorithm uses the two-point crossover operator for recombination. Mutations are performed by randomly choosing a task and randomly changing the index of its assigned component service.

A peculiarity of our approach for preference specification is that the fitness of a solution can only be evaluated in the context of a given population, because the ranking algorithm performs pairwise comparisons of all candidate solutions. Therefore, it is not possible to offer an absolute value for the fitness of a solution. Our genetic algorithm computes the fitness of a solution based on its ranking in the current population, by assigning the maximum value to the top ranking solution and the minimum value to the solution at the last position in the ranking. The pseudocode of the fitness evaluation procedure is given in Fig. 7.

```
procedure evaluateFitness()
    for i ← 1 .. n do
        qos[i] ← computeAggregateQoS(solution[i])
    end for
    for i ← 1 .. n-1 do
        for j ← i+1 .. n do
            compResult[i][j] ← compareServices(solution[i], solution[j])
        end for
    end for
    createScoreVectors()
    rankList ← list of solutions sorted using compareScores()
    for i ← 1 .. n do
        fitness[i] ← populationSize - (index of solution[i] in rankList)
    end for
end procedure
```

Fig. 7. Procedure for evaluating the fitness of the candidate solutions.

The computeAggregateQoS() function in the pseudocode above uses the method described in section 2 in order to estimate the aggregated cost of a composite service based on the QoS values of its component services. The function compareServices has been defined in Fig. 2, the function createScoreVectors in Fig. 3, and the function compareScores in Fig. 5. The populationSize is a configurable parameter of the algorithm. In the next section we show experimental results for different values of populationSize.

Several conditions may be combined in order to trigger the termination of our genetic algorithm:

- a maximum number of generations has been reached;
- a maximum time limit for running has been reached;
- there has been no solution improvement during the last k generations.

Since there is no absolute value for the fitness of a solution, checking the occurrence of the third condition is not a trivial operation. In order to solve this issue, the genetic algorithm maintains a list of best-so-far solutions. At the end of each generation, the best solution in the current population is searched in the list of best-so-far solutions. If not already present, it is added to this list and ranked against the other elements. An improvement has occurred only if the current best solution is at the top of the resulting ranking. The size of the list of best-so-far solutions is limited by a value configured as a parameter of the algorithm. If, as a consequence of adding the current best solution to the list of best-so-far solutions, its size exceeds the limit, the element with worst ranking will be removed.

The prototype implementation of our algorithm is written in Java and it is available as open source at: http://baas.sourceforge.net/.

5. Experimental results

In our experiments, we have used 21 orchestration models from the public Oryx repository (available at https://code.google.com/p/oryx-editor/) and 10 from the IBM BIT process library (available at http://www.zurich.ibm.com/csc/bit/downloads.html). The number of tasks in the considered models is between 5 and 20, while the number of gateways varies between 0 and 16.

We have set the termination condition by 20 generations without solution improvement and the maximum size of the best-so-far list to 10.

We have performed experiments with the following population sizes: 20, 50 and 100.

For each population size and each orchestration model we have performed 100 runs, using a different set of candidate web services for each run. Each set of candidate services has been created by generating between 2 and 7 service alternatives for each task in the current orchestration model.

For each orchestration model and each population size, the average number of generations is given in Table 4.

Table 4. Experimental results.

Orchestration model	Number of tasks	Number of Gateways	Average number of generations		
			(population=20)	(population=50)	(population=100)
1	5	4	9.48	7.17	3.43
2	6	0	11.36	7.00	5.57
3	6	0	10.86	6.30	4.57
4	6	0	11.9	7.60	5.17
5	6	0	11.04	7.43	4.67
6	6	0	11.8	6.90	5.53
7	6	4	12.96	7.37	4.63
8	6	6	13.46	8.17	6.90
9	6	7	9.92	7.33	5.13
10	7	5	15.98	8.90	7.37
11	7	7	15.44	8.23	7.70
12	8	3	16.92	11.30	8.97
13	8	3	18.10	13.70	9.70
14	8	10	27.16	41.67	44.40
15	9	4	24.38	15.13	11.73
16	9	4	25.38	57.53	35.20
17	9	8	19.38	13.03	12.30
18	9	8	30.56	35.50	41.07
19	9	10	21.72	13.43	12.40
20	10	6	21.32	16.27	14.07
21	10	6	21.12	15.87	12.60
22	10	9	23.00	16.93	12.17
23	11	6	25.98	17.33	14.70
24	11	14	29.08	20.17	14.30
25	12	5	29.40	20.63	18.53
26	13	8	30.26	25.23	18.30
27	13	12	33.24	27.33	23.07
28	14	4	31.66	22.83	20.00
29	15	16	21.96	39.30	41.63
30	18	12	31.64	46.20	38.67
31	20	5	49.36	29.33	37.63

It can be observed that even a population of 20 individuals is enough to reach the termination condition in less than 50 generations. While using larger populations leads in most cases to a lower number of generations, the differences are not very significant and, in some cases, such as for the orchestration model 16, a larger population may actually lead to worse results.

Although a correlation between the number of tasks and gateways and the number of generations needed to reach the termination condition can be observed, other factors, such as the topology of the orchestration model, seem to have also a substantial influence.

6. Conclusion and future work

We have introduced a genetic algorithm for finding the best mapping of component services to the tasks involved in a service composition. The algorithm estimates the fitness of the mappings that make up a population of candidate solutions by using the service binding method described in one of our previous papers. The experimental results show that the algorithm is able to find the best mapping in a few number of generations even when small-sized populations are used.

We are currently devising a service composition framework combining our algorithm with a QoS-aware semantic selection algorithm that uses an ontology compatible with the preference approach described in this paper.

References

[1] J. El Haddad, M. Manouvrier and M. Rukoz, “Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition,” *IEEE Transactions on Services Computing,* vol. 3, no. 1, pp. 73-85, 2010.

[2] R. Iordache and F. Moldoveanu, “A web service composition approach based on QoS preferences,” in *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*, Kauai, Hawaii, 2013.

[3] R. Iordache and F. Moldoveanu, “A conditional lexicographic approach for the elicitation of QoS Preferences,” in *Lecture Notes in Computer Science Volume 7565, pp 182-193*, Rome, 2012.

[4] Y. Yang, M. Dumas, L. García-Bañuelos, A. Polyvyanyy and L. Zhang, “Generalized aggregate Quality of Service computation for composite services,” *Journal of Systems and Software,* no. 85(8), pp. 1818-1830, 2012.

[5] A. Polyvyanyy, L. García-Bañuelos and M. Dumas, “Structuring Acyclic Process Models,” *Information Systems,* vol. 37, no. 6, pp. 518-538, 2012.