Effective QoS aware web service composition in dynamic environment

Peter Bartalos and Mária Bieliková

Abstract Nowadays, several web services composition approaches, each considering various aspects of the composition problem, are known. Consequently, a lot of attention shifts to the performance issues of web service composition. This paper presents an effective QoS aware composition approach. Our work focuses to its performance, which is studied also in dynamically changing environment, where new services are deployed, some services are removed, or the QoS characteristics change in time. We analyze the impact of the need to manage these changes to the sojourn of the composition query in the system. The experiments show that the proposed composition system is handling these changes effectively and the sojourn time is not significantly affected.

1 Introduction

Web service composition [8] is a process of arranging several web services into workflows. The aim is to bring a utility, which cannot be provided by a single service. Its automation showed to be a challenging task. The research of service composition in last years tends to focus on issues related to QoS [3, 5, 6, 9, 13, 15], pre-/post-conditions [4, 6, 12], user constraints and preferences (e.g. soft constraints) [2, 10, 14], consideration of complex dependencies between services [7, 16].

Several approaches deal with performance and scalability issues of the composition process [3, 5, 6, 9, 12]. These use effective data structures created during preprocessing to speed up the composition. As it is true for the web in general, also web services change in time. New services are deployed, some of them are removed. Moreover, the non-functional properties of services may change frequently. The aim

Peter Bartalos and Mária Bieliková
Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Ilkovičova 3, 842 16 Bratislava, Slovakia
e-mail: {bartalos,bielik}@fiit.stuba.sk

is to find a composite service with the best global QoS characteristic, computed from the QoS attributes of single services based on aggregation rules [3, 13, 15]. The composition system must flexibly react to these changes. The solution should reflect the current situation in the service environment. If the changes are not managed, it may happen that the designed composition does not use the right services, it includes services which are already not executable, or it is not optimal from QoS point of view. Hence, we do not achieve the satisfaction of the user goal based on which the composition is realized. On the other side, the dynamic changes of services require updating the data structures of the composition system, before we start new composition. Thus, the dynamic changes affect the composition time. To avoid too much delay, the updates must be realized quickly. In this paper we deal with the performance and scalability of service composition operating in dynamic environment. We present our designed algorithms, which were already used in our previous work and extended to more effectively manage the dynamic changes.

2 Related work

Several approaches automating the web service composition had already been proposed. They showed to be useful when looking for a composition aware of pre-/post-conditions, QoS and satisfying the user constraints, preferences. Promising results were achieved also regarding the performance. There are approaches able to compose services in acceptable time, even if the number of services in the registry rises to some ten thousands. However, only little attention had been given to the research of approaches handling the dynamic changes in the service environment.

As we studied approaches in [5, 9, 12], we found out that there are some issues appearing in each of them. First, they use pre-processing during which effective data structures are created. These represent the services and the interconnections between them. Another similarity is in the base of the composition process. It is a forward search, starting with the concepts describing the inputs provided in the query (at semantic level). The set of these provided concepts is then incrementally extended. In each iteration, concepts associated with outputs of services having provided inputs are appended. An input is provided if it is associated with a concept, already included in the set of provided concepts. This continues in iterations until all concepts required in the user query are not provided and new concept can be appended. During the iterations, also an acyclic graph of services, depicting all possible solutions of the composition problem, is built. After all required concepts are produced, to remove redundant services, a backward search is performed, starting with services directly producing the outputs required in the user query.

In [12] the authors present a Prolog based, pre-/post-condition aware composition approach. It does not consider the QoS of services. The aim of pre-processing in this approach is to convert the service descriptions into Prolog terms. The semantic relations (e.g. subsumption) are also pre-processed. The built-in indexing scheme facilitates fast composition. The authors deal also with *incremental updates* required

when adding a new service. They results show that managing an update requires much more time, than the composition (e.g. 1 vs. 18 msec).

An effective, QoS aware composition approach is presented in [9]. It is based on modified dynamic programming, and a data structure having three parts. The first represents services, the second concepts, the third the mapping of the concepts to service parameters. The composition is effective due a filtering utilized to reduce the search space. The pruning removes services i) which have no inputs (thus cannot be executed), and ii) are not optimal from QoS point of view. During forward chaining, when adding new concepts into the set of provided concepts, the aggregated QoS values of services are updated to find the optimal value. Also the provided concepts are related to an optimal QoS value, calculated based on the optimal aggregated QoS value of services which are producing it.

In [5] we had presented a composition approach dealing with performance and scalability issues. The difference between our approach and [9] seems to be minor from the algorithm point of view. Due poor description of the algorithm in [9], we are not able to exactly state the difference. However, it is clear that in [9] different data structures are used to express the services, and their relation to other services and concepts. Moreover, our work deals also with pre-/post-conditions, just as [12]. These aspects of our composition system are described in [4, 6].

3 Service composition process

The aim of our composition approach presented in [4] was to *select usable* services. The service is usable if all its inputs are provided thus can be executed. The inputs are provided in the user query, or as an output of another usable service. To select usable services, forward chaining is realized. During it, we also select the best provider for each input of all services, considering QoS. This process becomes a high computation demanding task as the number of services, their inputs, and number of input providers rises. The complexity of the computation is also strongly dependent on the interconnections between the services.

To speed up the *select usable* services process, we propose search space restriction. Its aim is the selection of *unusable services*. A service is unusable, if at least one its input is not provided in the user query, neither as output of ancestor services. The process lies on identification of such services, for which there is at least one input not provided by any available service, i.e. the only case when it is usable is if the respective input is provided in the user query. These services are identified during preprocessing. We call them *user data dependent services*.

The selection of (un)usable service is done by marking a service. The marks are: *UNDEC* if it is undecided if the service is unusable, *USAB* if the service is usable, *UNUSAB* if the service is unusable, *UDD* if the service is user data dependent, *UDDPROV* if the service is *UDD* and all dependent inputs are provided in the query.

A lot of our work had been redesigning the effective data structures used in [4, 5], to be quickly modifiable when the service environment changes. In the following,

we present an overview of the data structures to be able to understand the algorithms introduced later. The most important are:

- *AllServices*: a collection of all available services.
- *ConceptProviders*: a collection, where an element is a key-value pair of i) *concept* and ii) list of services having an output associated with *concept*.
- *ConceptConsumers*: a collection, where an element is a key-value pair of i) *concept* and ii) list of services having an input associated with *concept*.
- *InputDependents*: the same as *ConceptConsumers*, but contains only services marked with *UDD*.
- *UserDataDependents*: a collection of services marked with *UDD*.
- *SuperConcepts*: a collection, where an element is a key-value pair of i) *concept* and ii) list of concepts (transitively) subsumed by *concept*.

A service is represented as a complex data structure having several attributes. Its aim is to store information about service's properties and interconnections to ancestor, and successor services. The main attributes are:

- *usability*: stores the mark of the service as presented before.
- *unprovidedInputs*: the number of unprovided user data dependent inputs.
- *inputs*: list of concepts associated with the inputs.
- *inputProviders*: for each input, it stores a list of services providing it.
- *bestQoSProviders*: for each input, it stores a reference to a service which provides it and has the best QoS.
- *successors*: list of successor services.

3.1 Restricting the search space

The *select unusable* services process performs as presented in Alg. 1. It can be started only after we select services having all inputs provided in the user query and mark them as *USAB*. This is the part of Alg. 2 at lines 1 to 12. Then, the process starts evaluating which services can be marked with *UDDPROV* (lines 1 to 6). For each *provided input* in the user query, we get the *list* of services from *InputDependents*, requiring the respective *provided input*. For each service in the *list*, we decrement the number of unprovided user data dependent inputs. If this number is zero, the service is potentially usable. We cannot be sure because it may have other inputs which do not rely on provision in the user query. This is evaluated later.

Next, we decide which user data dependent services are not usable (lines 7 to 9). We traverse *UserDataDependents* and if the respective service is not usable, either has not provided all user data dependent inputs, it is marked with *UNUSAB*. Then, we create a list of services still having undecided usability (lines 10 to 12).

Up to now, we know which user data dependent services are unusable. The (un)usability of these services influences also their successors. If there is a service having some input for which all providers were marked with *UNUSAB*, it is unusable too. In the rest, we evaluate this (lines 13 to 25).

Algorithm 1 Find unusable services: Input: *provided inputs*

```
1: for all provided input do
2:     list = InputDependents.get(provided input);
3:     for all service in list do
4:         decrement service.unprovidedInputs;
5:         if service.unprovidedInputs == 0 then
6:             service.usability = UDDPROV;
7: for all service in UserDataDependents do
8:     if service.usability ≠ USAB AND service.usability ≠ UDDPROV then
9:         service.usability = UNUSAB;
10: for all service in AllServices do
11:     if service.usability == UNDEC then
12:         toProcess.add(service);
13: while is service to process do
14:     for all service in toProcess which is undecided do
15:         for all input in service.inputs do
16:             isUnprovidedInput = true;
17:             if input provided in user query then
18:                 isUnprovidedInput = false;
19:             else
20:                 providers = service.inputProviders of input;
21:                 for all provider in providers do
22:                     if  provider.usability ≠ UNUSAB then
23:                         isUnprovidedInput = false;
24:             if isUnprovidedInput == true then
25:                 service.usability = UNUSAB;
```

While there is a potential service which may be marked unusable, we traverse services from *toProcess*. For each input of the respective service, we check if it is for sure unprovided, which a default assumption is. If it is provided in the user query, then *isUnprovidedInput* is false. If it is not, we traverse all the input provider services. If there is at least one which is not unusable, then *isUnprovidedInput* is false. This case does not mean that the input is provided, we just cannot be sure that it is not. If we are sure that the service has at least one unprovided input, i.e. *isUnprovidedInput* was not changed to false, it is unusable. If we mark some service as unusable, the evaluation starts again. The reason is that the found unusable service affects its successors, which may thus become unusable too.

After the *select unusable* services process finishes, several services were marked as unusable. These cannot be used in the composition. Thus, we restricted the search space and we will not waste time during the composition with several unusable services. Note that we did not find all unusable services. We found each unusable service which is user data dependent, or it is a (indirect) successor of it.

3.2 Discovering usable services

The *select usable* services process has two phases, see Alg. 2. First, we select services having all inputs provided in the user query. Second, we realize forward chaining to find other usable services. The first phase starts by collecting services which have some inputs provided in the user query (lines 2, 3). Then, for each potential service we check if it has provided all inputs (lines 5 to 12). If so, the service is marked as *USAB* and put in *toProcess*.

The second phase (lines 13 to 30) performs in a loop for all services in *toProcess*. For a respective *service*, we check if there is some provider of its inputs (lines 15 to 24). During this, we select the provider having the best QoS at the moment. If the service has provided all inputs, it is marked with *USAB* and we recalculate its aggregated QoS based on the best QoS providers. Then, for all its successors, we check if their usability is undecided or if the *service* can improve the aggregated QoS of the respective *successor*. If so, the *successor* is processed too.

Algorithm 2 Discover usable services: Input: *provided inputs*

```
 1: for all provided input do
 2:     services = ConceptConsumers.get(provided input);
 3:     potentialServices.putAll(services);
 4:     inConceptsMap.put(provided input);
 5: for all service in potentialServices do
 6:     allInputProvided = true;
 7:     for all input of service do
 8:         if inConceptsMap.contains(input) == false then
 9:             allInputProvided = false;
10:     if allInputProvided == true then
11:         service.usability = USAB;
12:         toProcess.add(service);
13: while toProcess.size > 0 do
14:     service = toProcess.first;
15:     allInputProvided = true;
16:     for all input in service.inputs do
17:         inProvided = false;
18:         providers = service.inputProviders of input;
19:         for all provider in providers do
20:             if provider.usability == USAB then
21:                 inProvided = true;
22:                 update best QoS provider;
23:         if inProvided == false then
24:             allInputProvided = false;
25:     if allInputProvided == true then
26:         service.usability = USAB;
27:         update aggregated QoS;
28:         for all successor in service.successors do
29:             if successor.usability == UNDEC OR service improves aggregated QoS of successor
                then
30:                 toProcess.add(successor);
```

4 Dynamic aspects of service composition

Our composition system can be seen as a queuing system with different type of requests [1]. It works as depicted in Fig. 1. When the system starts, it initializes its data structures based on currently available services. After, it is waiting for update requests, or user queries (1). These are collected in queues and processed regarding the *first in first out* rule. The updates are managed with higher, *non-preemptive* priority, i.e. the composition is not interrupted if an update request arrives. If an update request arrives (2), i.e. new service is available, some service become unavailable, or the QoS characteristics of some service had changed, we update the affected data structures. When all the changes are managed (3), the system is ready to process new user query. If a new query has already been received, the system processes it immediately (4). Otherwise, it goes to waiting state (5). Here, it again waits for an update request, or composition query (6). The composition is followed by the reinitialization (7). After, we manage update requests, if some had arrived (8). If not, we process a new user query for composition (9), or go to waiting state (10).

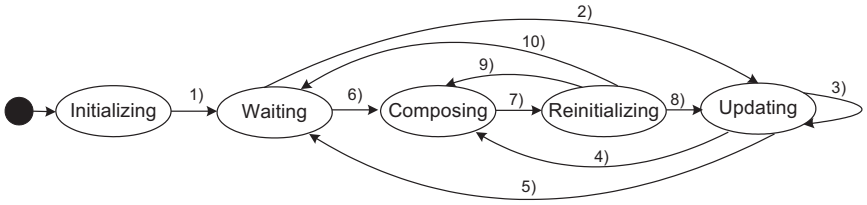

Fig. 1 Composition system life cycle.

The change of service QoS characteristics are managed in constant time, independently on the size of the service repository. It requires only changing the values of the QoS attributes in the object representing the corresponding service. The updates required because some service is made (un)available are divided into two cases. Adding a new service requires creating a new object representing it and discovering its interconnections with other objects. When the service is removed permanently, we have to remove the interconnections with other objects. The temporal removing of the service is possible, too. This is useful if we expect that it will be made available again. The temporal removing of the service is done in constant time, by setting a flag depicting the (un)availability of the service. To make such a service available again, we only have to set a flag once again. The adding of a new service, and permanent removal of some service are more complicated. They depend on the overall set of services available in the service repository, and their interconnection.

The adding of a new service starts with adding a service into *AllServices*, see Alg. 3. Then, we discover the interconnections with ancestor services (lines 2 to 10). For each input, we determine the services providing it. If there is no input provider, we add the service to *UserDataDependents*, and *InputDependents*. Otherwise, we add it to the list of successors of each provider. Finally, we add it to services requiring the given input.

Algorithm 3 Add service: Input: *service*

```
 1: AllServices.add(service);
 2: for all input in service.inputs do
 3:     providers = ConceptProviders.get(input);
 4:     if providers is empty then
 5:         UserDataDependents.add(service);
 6:         InputDependents.get(input).add(service);
 7:     else
 8:         for all provider in providers do
 9:             provider.successors.add(service);
10:     InputConsumers.get(input).add(service);
11: for all output in service.outputs do
12:     outputSuperConcepts = SuperConcepts.get(output);
13:     for all superConcept in outputSuperConcepts do
14:         allOutputSuperconcepts.put(superConcept);
15: for all superConcept in allOutputSuperConcepts do
16:     ConceptProviders.get(superConcept).add(service);
17:     consumers = ConceptConsumers.get(superConcept);
18:     for all consumer in consumers do
19:         service.successors.add(consumer);
20: for all successor in service.successors do
21:     for all input in successor.inputs do
22:         if allOutputSuperConcepts.contains(input) then
23:             successor.inputProviders.add(service);
24:         InputDependents.get(input).remove(successor);
25:     if UserDataDependents.contains(successor) then
26:         if successor has not unprovided inputs then
27:             UserDataDependents.remove(successor);
```

In the next, we create a collection of distinct concepts subsumed by some service outputs (lines 11 to 14). For each such *superConcept*, we add the service to a list of its providers (line 16). Each service for which *service* produces input data is added to its successors (lines 17 to 19). Finally, for each successor we add *service* to providers of all inputs produced by it (line 23). During this, we check if the successor is in *UserDataDependents*. If it is, and the *service* provides an input, which was the only unprovided input, the successor is removed from *UserDataDependents*.

The permanent removal of the services is a revert operation to adding a new service. It deletes the interconnections to other services, and the references of the object representing it, from each data structure. Due a straightforwardness of the process and a lack of space, we omit its detailed description.

5 Evaluation

The evaluation of our approach is split into evaluation of i) times required to add/remove services, compose services, and reinitialize the system, ii) behavior of the system due continuing query arrival, without or with dynamic changes in the

service registry. All experiments had been realized using a Java implementation of our composition system. The computations had been running on a machine with two 64-bit Quad-Core AMD Opteron(tm) 8354 2.2Ghz processors with 64GB RAM.

The experiments were realized using data sets generated by a third party tool (`http://ws-challenge.georgetown.edu/wsc09/software.html`) used to create data sets for *Web services Challenge 2009* [11]. We generated test sets consisting from 10 000 to 100 000 services. For each test set, the solution requires at least 50 services to compose. The number of concepts in the ontology is from 30 000 to 190 000. To allow comparison with others, the data sets are available at `http://semco.fiit.stuba.sk/compositiontestsets`.

In Tab. 1 and Fig. 2 we see how the complexity of the service set (e.g. its size) affects the time required to i) add a new service (as depicted in Alg. 3), ii) permanently remove a service, iii) compose services, and iv) reinitialize the system after composition. In our experiment we measured the time of adding 1 000 services into a repository, and permanent removing of the same services. As we see, removing a service is more time demanding. This is because the removal requires more traversal of data structures, to find the position of the service's object reference. The composition and update times do not necessary rise as the number of services rises. This is because they are strongly affected also by the interconnections of services, and QoS parameters. The reinitialization time is linearly dependent on the number of all services and the number of user data dependent services. In our experiments, it reached maximum of 53% of the composition time, 33% in average.

Table 1 Operation times.

Web services	Add	Remove	Reinitialization	Composition
10 000	0.84	1.68	1.86	4.95
20 000	0.92	2.84	4.46	14.8
30 000	1.02	4.82	10.2	46.3
40 000	1.53	7.88	13.8	35.9
50 000	1.13	5.81	19.6	37.3
60 000	2.12	10.3	25.6	93.6
70 000	1.39	9.11	27.1	88.0
80 000	1.48	13.3	29.5	64.3
90 000	1.86	9.46	30.0	271.8
100 000	1.89	12.0	51.1	152.4

In the next we present experimental results considering a composition system as presented in section 4. We measured the sojourn time, and the queue size. The sojourn time is a time between the generation of the update request, or user query and the end of its processing. We assume that the arrivals of both the update requests and user queries are continuous, independent, and occur according to a *Poisson process*. Hence, the interarrival times follow exponential distribution. Since there is no real application, we cannot evaluate these assumptions. Due this, we present also results where the interarrival times follow uniform distribution, to see the effect of different distribution on the measured parameters.

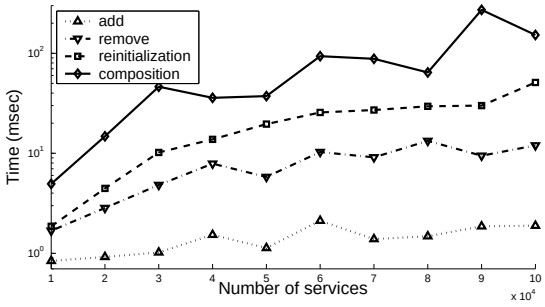

Fig. 2 Add/remove, reinitialization, and composition time.

Fig. 3 and 4 present the dependency between the mean interarrival time and sojourn time, and mean queue size, for data sets with 20 000 up to 100 000 services. As we see, there is a significant difference between the results when different distributions are used. In the case of exponential, the standard deviation of the results is higher. Moreover, it presents rising tendency as the complexity of the data set, and the sojourn time rises. Similar observations are present also regarding the mean queue size. The system tends to be in stable state when the mean interarrival time is more than a double of the composition time. In this case, the mean queue size (measured after the request is put in it, i.e. it cannot be less than 1) is less than 2. Adequately, the sojourn time is less than a double of the composition time.

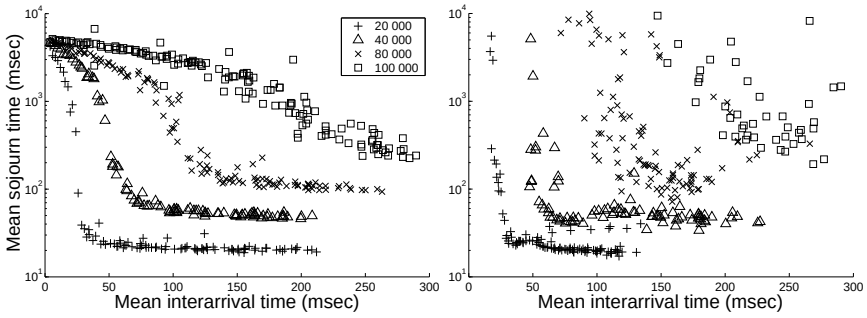

Fig. 3 Sojourn time: uniform distribution at left, exponential distribution at right.

Fig. 5 presents the effect of the dynamic changes in the service environment. We had been simulating arrival of requests to add a new, or permanently remove a service, for various interarrival times with exponential distribution. We used the test set with 20 000 services. The experiments show that the sojourn time is not significantly enlarged, even if the update requests are frequent. We had dramatically decreased the interarrival time of update requests, from 1 000 to 10 msec. Even in this case, we observed only a little delay in composition. The mean composition query queue size remained low and the stability of the system was not upset.

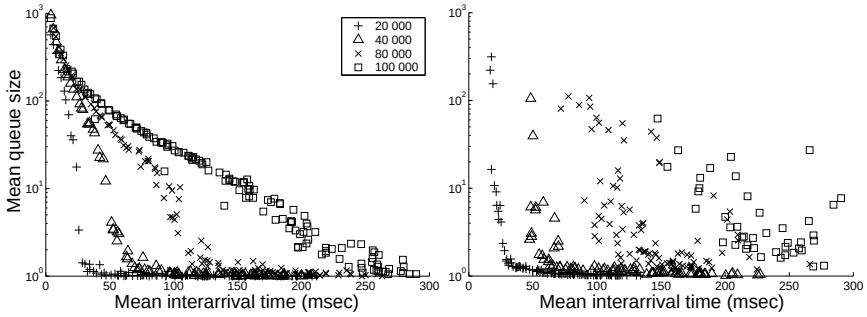

Fig. 4 Queue size: uniform distribution at left, exponential distribution at right.

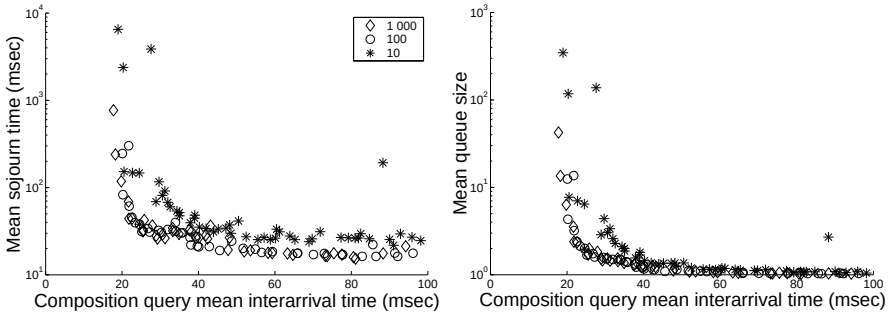

Fig. 5 Effect of dynamic changes in the service environment.

6 Conclusions

This paper presents a QoS aware composition approach focusing on efficiency, and scalability in dynamically changing service environment. We adapted our previous approach to effectively handle the update requirements. The previous approach proved to be fast as it was successful at *Web services Challenge 2009*, a world competition aimed at well performing service composition [11].

Our composition system manages the updates in much shorter time than composition queries (about one order of magnitude). We have investigated its behavior when simulating continual arrival of user queries collected in queues. Our experiments show that if the mean interarrival time of composition queries is more than a double of composition time, the sojourn time is not significantly higher. When the queries arrive more frequently, the sojourn time rises dramatically. Since our composition system manages effectively the update requests, these do not significantly affect the sojourn time. We have made also a simulation model of our composition system. It is an effective tool to analyze its behavior, without the need of running it.

Our future work is to study the behavior of the composition system regarding new simulation scenarios, such as arrival of batches of update requirements. We will study what are the bottlenecks of the composition system, and what modifications

could overcome them. Our assumption is that for example parallel processing of user queries significantly reduces the sojourn time. We will use also a simulation model of our composition system, to analyze what modifications are useful.

Acknowledgements This work was supported by the Scientific Grant Agency of SR, grant No. VG1/0508/09 and it is a partial result of the Research & Development Operational Program for the project Support of Center of Excellence for Smart Technologies, Systems and Services II, ITMS 25240120029, co-funded by ERDF.

References

1. I. Adan and J. Resing.: Queueing Theory. *Eindhoven University of Technology*, 180 pages, 2002. `http://www.win.tue.nl/%7Eiadan/queueing.pdf`, downloaded in April, 2010.
2. S. Agarwal and S. Lamparter.: User preference based automated selection of web service compositions. In *Proc. of Int. Conf. on Service-Oriented Computing 2005, Workshop on Dynamic Web Processes*, 1–12. 2005.
3. M. Alrifai, T. Risse, P. Dolog, and W. Nejdl.: A scalable approach for qos-based web service selection. In *Proc. of Int. Conf. on Service-Oriented Computing 2008 Workshops*, 190–199. Springer, 2009.
4. P. Bartalos and M. Bielikova.: Fast and scalable semantic web service composition approach considering complex pre/postconditions. In *Proc. of the 2009 IEEE Congress on Services, Int. Workshop on Web Service Composition and Adaptation*, 414–421. IEEE CS, 2009.
5. P. Bartalos and M. Bielikova.: Semantic web service composition framework based on parallel processing. In *Proc. of Int. Conf. on E-Commerce Technology*, 495–498. IEEE CS, 2009.
6. P. Bartalos and M. Bielikova.: QoS Aware Semantic Web Service Composition Approach Considering Pre/Postconditions. In *Proc. of Int. Conf. on Web Services*. IEEE CS, 2010. Accepted.
7. P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner.: Control flow requirements for automated service composition. In *Proc. of Int. Conf. on Web Services*, 17–24. IEEE CS, 2009.
8. Dustdar, S., Schreiner, W.: A survey on web services composition. In *Int. J. of Web and Grid Services*, (2005), 1(1):1–30.
9. Z. Huang, W. Jiang, S. Hu, and Z. Liu.: Effective pruning algorithm for qos-aware service composition. In *Proc. of Int. Conf. on E-Commerce Technology*, 519–522. IEEE CS, 2009.
10. E. Karakoc and P. Senkul.: Composing semantic web services under constraints. In *Expert Systems with Applications*, (2009), 36(8):11021–11029.
11. S. Kona, A. Bansal, M. B. Blake, S. Bleul, and T. Weise.: Wsc-2009: A quality of service-oriented web services challenge. In *Proc. of Int. Conf. on E-Commerce Technology*, 487–490. IEEE CS, 2009.
12. S. Kona, A. Bansal, and G. Gupta.: Automatic composition of semantic web services. In *Proc. of Int. Conf. on Web Services*, pages 150–158. IEEE CS, 2007.
13. F. Lécué and N. Mehandjiev.: Towards scalability of quality driven semantic web service composition. In *Proc. of Int. Conf. on Web Services*, 469–476. IEEE CS, 2009.
14. N. Lin, U. Kuter, and E. Sirin.: Web service composition with user preferences. In *Proc. of European Semantic Web Conf.*, 629–643. Springer, 2008.
15. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An End-to-End Approach for QoS-Aware Service Composition. In *Proc. of Int. Enterprise Distributed Object Computing Conf.*, 151–160. IEEE, 2009.
16. H. Q. Yu and S. Reiff-Marganiec.: A backwards composition context based service selection approach for service composition. In *Proc. of Int. Conf. on Services Computing*, 419–426. IEEE CS, 2009.