Chapter 1
Effective QoS aware service composition based on forward chaining with service space restriction

Peter Bartalos and Mária Bieliková

Abstract Several approaches dealing with the performance of QoS aware semantic web service composition have been proposed. We describe an approach which took part at Web Services Challenge 2009. It showed good performance even if large service repositories were processed. We discuss the main principles of the used approach and also its further enhancement. The enhancement includes shorter composition time, consideration of the pre-/post-conditions of services, and adaptation to real conditions where changes of the service set, or the QoS attributes must be effectively managed. Moreover, the lessons learned during the development and the participation of the competition is discussed too.

1.1 Introduction

Automatic dynamic web service composition is showing to be an effective way how to deal with the dynamic character of the web service, and business environment, while providing a mechanism capable to supply varying composition goals [6]. Several research results concerning different aspect of the overall problem had been proposed. These present promising results and also address existing problems.

Several tasks, required to perform during the service composition, are NP-hard in general. These include for example the construction of the control-/data-flow together with the QoS optimization and evaluation of the compatibility between the pre- and post-conditions of services. To be able to handle large service repositories, sophisticated methods must be developed. These must provide good performance and scale well as the number of web services in the repository rises. There are al-

Peter Bartalos · 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

ready several works presenting promising results considering effective QoS aware service composition [1, 5, 8, 10, 13].

In this chapter, we deal with our approach for QoS aware web service composition. Our work focuses on issues related to the description of the functional aspects of the web services, QoS optimization, reaction to the changes in the service environment, performance, and development of a composition system able to handle continuing arrival of the composition queries.

1.2 Approach: QoS and pre-/post-condition aware web service composition in dynamic environment

The overall web services composition process is a complex task including several sub-processes. The general aim of service composition is to arrange multiple web services into a composite service to supply more complex functionality. Beyond this basic objective, there are a lot of aspects of the problem, which are showing to be inevitable, or have the potential to make the composition much usable. These include the consideration of the QoS, pre-/post-conditions [2, 5, 9], user preferences, constraints and context [11, 15], user assistance [14], and transactional behavior of service compositions [7, 12]. The main objectives of our work, presented in [2, 3, 4, 5], are related to the following issues:

- *Functional aspects of web services*. Proper representation of the functional aspects of web services is crucial for automatic web service composition. The existing approaches exploit additional meta-data depicting the semantics of the I/O parameters to describe the service behavior. This approach is showing to be insufficient. The proposed solutions are oriented to express the pre-/post-conditions of web services. Our work deals with the pre-/post-conditions aware composition and shows the feasibility of this approach.
- *QoS optimization*. Beside the functional requirements, the user is usually interested also in non-functional properties of web services. Hence, the QoS optimization during service composition is important. We deal with a service composition aware of the QoS and capable to find the best solution considering them.
- *Changes in the service environment*. The web service environment is frequently changing in time. New services are deployed, some of them are removed. The changes relate also to the QoS attributes, whose values might evolve in time. We developed a composition approach capable to react to these changes and thus providing a solution reflecting the actual situation in the service environment.
- *Effectiveness*. As the Web in general grows, also the set of web services which are available in repositories is rising. We deal with the problem of performance and scalability of the composition process considering large number of services to be searched and composed.
- *Composition system*. Our aim is not only to develop a composition method, but also to design a composition system realizing it.

- *Continuing composition query arrival*. In real scenarios, the composition system must dynamically compose services based on the actual composition goal. The composition queries may arrive from multiple users. To handle multiple queries, our composition system stands as a querying system able to process continuing arrival of composition queries, while reacting also to possible frequent changes in the service environment.

Our composition system is built according to the requirements defined in the rules of *Web Services Challenge 2009*. However, it provides more functionality than required in the challenge. The additional requirements, which are not stated in the challenge rules, are: consideration of the pre-/post-conditions of the web services, processing multiple composition queries arriving independently, and reaction to the changes in the service environment.

Based on the defined requirements, we designed software architecture of the composition system as presented in Fig. 1.1. The system is divided into two main subsystems. The first includes components responsible for the preprocessing (bootstrap) phase. The second is responsible for the user querying phase.

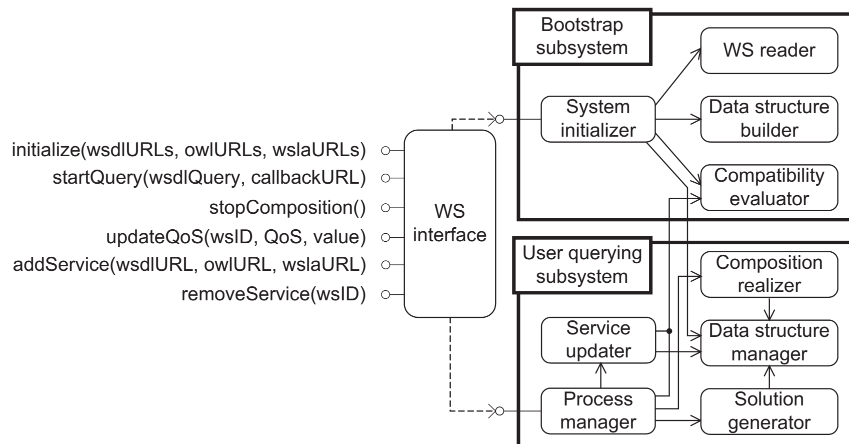

Fig. 1.1 Composition system architecture.

The *Bootstrap subsystem* is coordinated by the *System initializer* realizing the preprocessing based on the web service set available at the startup time. The *User querying subsystem* is managed by the *Process manager* coordinating the composition. *Composition realizer* is responsible to run the composition method. It operates over the data structures managed by the *Data structure manager*. In the case of a change in a service environment, the *Process manager* initializes the *Service updater* to process the request.

1.3 Solution: Realizing QoS aware service composition

1.3.1 Basic concepts

In general, several operations required during service composition have a complexity exponentially rising based on certain parameters. The main issues making our solution good performing are preprocessing, effective data structures, and algorithms. The preprocessing is crucial to quickly respond to composition queries. In our approach, we perform preprocessing before responding to composition queries, see top of Fig. 1.2. During it, we analyze the actual service set and build effective data structures. All the important computations, which can be done without the knowledge of the composition goal, are realized during preprocessing.

The most important is that we evaluate which services are compatible and can be chained. Based on the compatibility check a directed acyclic graph (DAG) of services is built. Each service is represented by a node. If two services can be chained, there is a directed edge from the ancestor to the successor service.

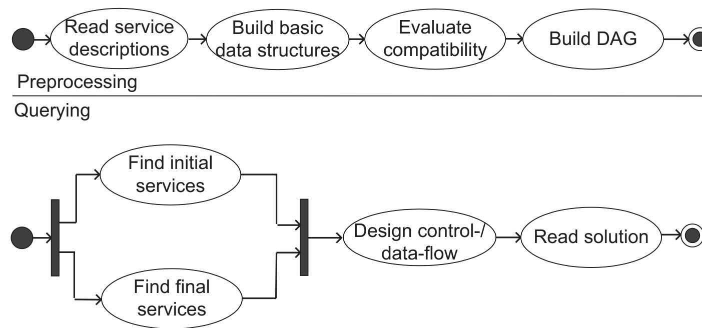

Fig. 1.2 Overview of the composition process.

The remaining task during composition is to find the initial/final services, and the design of the data-/control-flow, see bottom of Fig. 1.2. The initial services are those having provided all inputs in the composition query. The final services are those directly producing the outputs defined in the composition goal. The search for the initial, final services, and the design of the data-/control-flow can be completely realized only when the goal and the provided inputs are known. However, the effective data structures, built during preprocessing, are used here to realize them quickly.

Our composition method is based on two processes. The first selects services which have provided all inputs and thus can be used in the composition. The second selects services which cannot be used because they do not have provided all inputs. The inputs are provided in the composition query, or as an output of another usable service. The second process is not necessary to find the composition. It is used only to faster the selection of the usable services, which is a necessary process.

To select usable services, forward chaining is performed, starting with the initial services. During it, we realize also the QoS optimization. After, the solution is read backward, starting with the final services. The reading continues through each input, which is not provided in the composition query, over all the services already included in the composition. Considering QoS, we select those providers of the inputs which have the best aggregated value of a particular QoS attribute. We do not deal with calculation of the overall unified quality measure of the composite service.

The selection of the usable services 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 process, we propose *service space restriction* selecting the *unusable services*, i.e. a service with at least one unprovided input. 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 when the respective input is provided in the composition query. These services are identified during preprocessing. We call them *user data dependent services*.

1.3.2 Data structures

In our solution, we focused on a design of such data structures, which can be built during preprocessing and then effectively used during the querying phase. To do this, it was necessary to analyze which calculations, required during the composition, can be realized without the knowledge of the composition goal. The most important of them are: the processing of the semantic meta-data associated with the web services, the evaluation of which services can be chained (i.e. which services provide/consume inputs/outputs from the other services), and the search for services having an input/output associated with a defined concept or its sub-/super-class. To support these calculations, the following data structures are built:

- *Service*: represents a service including its attributes and reference to other *Services* providing/consuming its inputs/outputs (i.e. the collection of *Services* represents the directed acyclic graph of services). The *Service* data structure is the basic item in the other data structures.
- *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* or a concept subsuming *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 the *concept* or a concept subsumed by the *concept*.
- *InputDependents*: the same as *ConceptConsumers*, but contains only user data dependant services.
- *UserDataDependents*: a collection of user data dependant services.

The data structures are built to support fast execution of operations. The most important are finding service(s) with a given characteristic in a collection (e.g. having a defined output), iteration over a collection, deciding if a given service has a defined characteristic (e.g. if it is user data dependant). To support the fast finding and decision, hash tables are used. The iteration is supported by collecting the objects in arrays. However, this is true only if we do not deal with the dynamic changes in the service environment. If we do, arrays are not suitable, since increasing/decreasing its size is an expensive operation. Thus, a linked list is more appropriate.

The *ConceptProviders* data structure is used to quickly find the services directly providing the outputs defined in the goal. Quickly here means in constant time for each output. Analogically to the *ConceptProviders*, the *ConceptConsumers* is used to find the services consuming the inputs provided in the composition query. Notice that during the look for the initial, or final service, it is not necessary to deal with evaluation of the subsumption relations between concepts. The elements in the *ConceptProviders* and *ConceptConsumers* do already consider this.

The *AllServices*, *InputDependents*, and *UserDataDependents* data structures are used during the design of the control-/data-flow. During the selection of the usable services, the directed acyclic graph is traversed, starting with the initial services. Its creation during preprocessing showed to be crucial. The actual composition can be seen as a selection of a subgraph. It is already not required to find services providing, or consuming a defined input/output (i.e. the interconnections).

1.4 Lessons learned

1.4.1 Evaluation results

Our experiments were realized using data sets generated by a generator tool used at *Web services Challenge 2009*. We used service sets consisting from 10 000 up to 100 000 services. For each set, the solution requires at least 50 services to compose. The ontology consists from 30 000 to 190 000 concepts. The test sets are available at http://semco.fiit.stuba.sk/compositiontestsets/. Our experiments were realized using a Java implementation of our composition system on a machine with two 64-bit Quad-Core AMD Opteron(tm) 8354 2.2Ghz processors with 64GB RAM.

Control-/data-flow design

The aim of evaluating the control-/data-flow design phase is to show the: efficiency and scalability of our composition approach, and a dramatic improvement of the composition time due to the service space restriction.

During experiments we tested three configurations of the composition system:

1. the service space restriction runs in parallel with selection of the usable services, denoted as *Par*,
2. the selection of the usable services starts after the service space restriction finishes, denoted as *Seq*,
3. the service space restriction is not applied, denoted as *NoUnusab*.

The results are summarized in Tab. 1.1, Fig. 1.3, and Fig. 1.4. To be able to clearly state the difference between the composition with/without service space restriction, we provide also measurements depicting the number of crosses through two steps of the selection of usable services process. These two steps, denoted as A and B, are the most critical steps considering the performance (appear in the most inner loop). For other details, see [4].

Table 1.1 Experimental results.

Services	Composition time (msec)			Number of code line crosses					
	Par	Seq	NoUnusab	Par A	Seq A	NoUnusab A	Par B	Seq B	NoUnusab B
10 000	6	7	97	991	976	30 767	552	149	5 079
20 000	11	19	336	1 728	1 611	53 686	831	263	9 249
30 000	42	49	718	3 041	3 018	72 825	539	319	12 325
40 000	29	44	932	1 144	1 136	52 368	606	204	8 438
50 000	22	49	1 022	1 674	1 661	55 542	376	248	12 023
60 000	60	94	1 454	2 613	2 581	62 142	1 361	199	11 645
70 000	82	106	2 070	1 577	1 413	76 288	751	254	12 713
80 000	76	75	2 806	2 194	2 174	76 390	602	290	11 230
90 000	173	222	2 613	3 299	3 262	50 183	471	329	11 025
100 000	121	179	3 009	2 711	2 667	75 202	895	256	14 589

Dynamic changes in the service environment

The evaluation of our approach, in the context of dynamic changes in the service environment, focuses on the time required to add/remove services, compose services, and reinitialize the system, which is required after each composition (note that the QoS changes are managed in constant time). The results are presented in Tab. 1.2 and Fig. 1.5. During the experiment, we measured the time of adding 1 000 new services into a repository and permanently removing the same services. The added/removed services are selected randomly.

As we see, removing a service is more time demanding as adding. 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 between the services and their QoS parameters. Based on this, the hardest test set is the set with 90 000 services. 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.

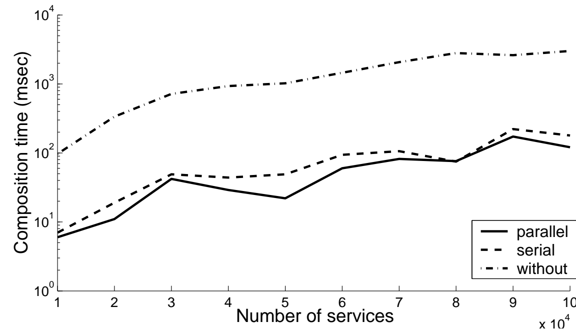

Fig. 1.3 Composition time.

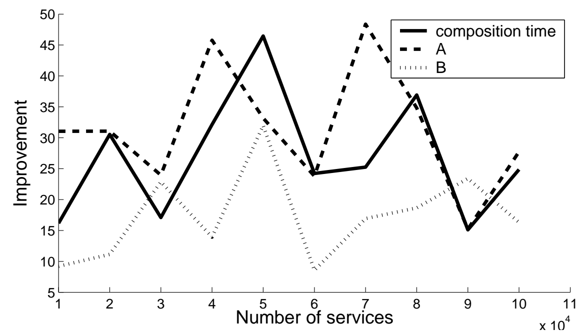

Fig. 1.4 Improvements.

Table 1.2 Operation times (in msecs).

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

Continuing query arrival

The evaluation of the composition system focuses on its behavior due to continuing query arrival, without or with the dynamic changes in the service environment. We measured the sojourn time and the queue sizes. The sojourn time is the time between the generation of the update, or composition request and the end of its processing.

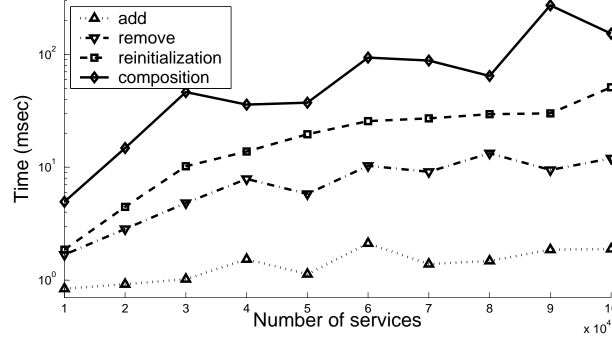

Fig. 1.5 Add/remove, reinitialization, and composition time.

We assume that the arrivals of both the update requests and composition 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 verify these assumptions. Due to this, we present also results where the interarrival times follow uniform distribution, to see the effect of the different distribution on the measured parameters.

Fig. 1.6 presents the dependency between the mean interarrival time and the sojourn time, for data sets with 20 000 up to 100 000 services. Note that the sojourn time and the mean queue size are linearly dependent. As we see, there is a significant difference between the results when different distributions are used. In the exponential case, 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. The system tends to be in a stable state if the mean interarrival time is more than a double of the composition time.

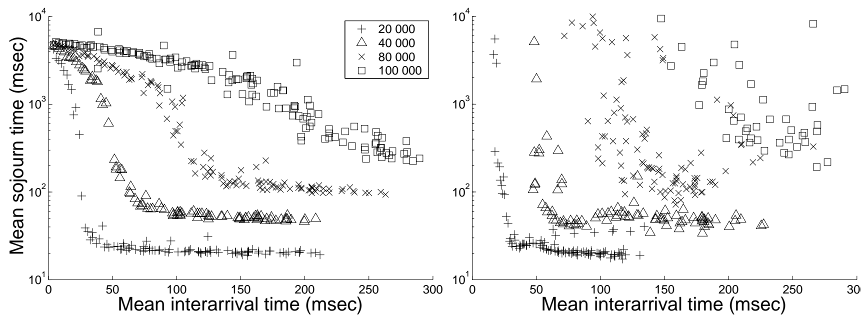

Fig. 1.6 Sojourn time: uniform distribution at left, exponential distribution at right.

Fig. 1.7 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

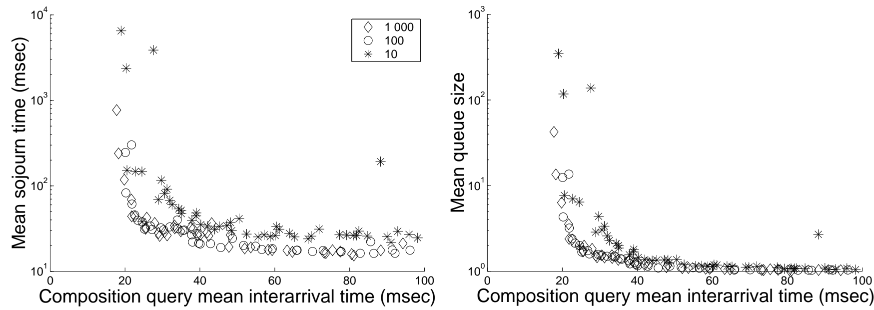

Fig. 1.7 Effect of dynamic changes in the service environment.

service, with 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 the update requests, from 1 000 to 10 msecs. Even in this case, we observed only a little delay in the composition. The mean composition query queue size remained low and the stability of the system was not upset.

1.4.2 Important observations

The issues affecting the effectiveness of the composition system can be divided into two categories: technical and conceptual solution. The technical issues include the chosen technologies and the way how they are used. Our experience is that these cannot be neglected. The performance of the system is significantly affected by the chose of the programming language and platform, relational database, external libraries, and other technical issues. On the other side, the performance characteristics of the standard technologies are usually known and a skilled developer should be able to choose the right technology and use it in an effective manner. Hence, however it can be a time and knowledge demanding task, the proper technical solution can be designed straightforwardly. This is not true for the conceptual solution whose bases are the proper methods and algorithms. These must be designed specifically for the web service composition problem. The most important issues, which had been shown to be important in our composition system, are: data structures, preprocessing, state space restriction, and parallel execution.

Data structures and preprocessing

The proper selection of the built-in data structures, or those available in external libraries, is important if the performance is an issue. Beside this, if we build our

own data structure, it is crucial to know which operations, performed over it, can become a potential performance bottle-neck. In this context it is very important to realize what can be done before we actually respond to composition queries. We believe that preprocessing is necessary to make the actual composition fast and do not waste time by realizing calculations which could be done already before.

A preliminary version of our composition system used relational databases to store some data and perform certain operations over them. During the composition, the relational database was queried to find relevant services (for example services consuming/requiring some data). Since the mentioned version of our composition system considers also the pre-/post-conditions of the services, the search for these services is a quite complex task. It requires evaluating logical implication between two formulae. The relational database showed to be a useful tool to realize this task. The declarative approach helped to relatively easily develop a method evaluating if a post-condition of some service implicates the pre-condition of another one (i.e. if they can be chained). Although the approach was convenient and feasible to realize the required task, the performance was poor when more complex logical formulae were processed. Due to this, we left the idea of using relational databases during service composition. Our new approach is based on encoding of the formulae and is realized purely programmatically [5].

Restricting the service space

One of the most important conceptual solutions in our composition system is the application of the service space restriction. It removes services which cannot be used in the composition, because they have not provided all inputs. The reduced service space is much easier to traverse and thus we can find the services required in the composition much faster.

We had been experimenting with different configurations of the composition system (as introduced in section 1.4.1). The best results, considering the composition time, are achieved with the *Par* configuration. In this case the selection of the usable services runs in parallel with the service space restriction. The *Ser* configuration performs slightly slower. According to *NoUnusab* configuration, both *Par* and *Ser* perform faster in more than one order of magnitude. The results considering the *Par* configuration show that even with the hardest test set, the composition time is below 200 msecs, which can be still considered as acceptable from the user point of view.

Fig. 1.3 shows the significant difference between the cases when service space restriction is applied, and if not. The measurements of how much times did the execution of the *select usable* services process cross those steps which appear in the most inner loops (denoted as A and B) clarifies the reason of the improvement, see Tab. 1.1. The decrease of the crosses, when service space restriction is applied, explains the improvement of the composition time. The *Par* configuration presents an improvement from 15 to 46 times in terms of composition time and adequate improvement in terms of the number of crossing steps A, B, see Fig. 1.4.

Parallel execution

The difference in performance between the *Par* and *Seq* configurations is minor (in contrast to the situation when no service space restriction is applied). During our experiments the *Seq* configuration performed in average 27 percent slower than the *Par* configuration. The differences between the composition times showed high deviation, from which we can conclude that the difference is significantly affected by the service set (e.g. the interconnections between the services based on their inputs/outputs). The experiments showed that, to achieve good results, it is not necessary to realize full service space restriction (remove all unusable services). The full service space restriction is realized during the *Seq* configuration. In this case, the selection of the usable services executes the fastest. However, since it must wait until the restriction is done, the overall composition time is not the lowest.

In the *Par* configuration, the selection of the usable services runs in parallel with service space restriction. The composition finishes when the usable services are selected. It is not necessary to wait until the service space restriction finishes too (since it is not a necessary process). This causes that the *Par* configuration performs better than *Seq* also on single-core computers. On multi-core computers, the benefits of parallel execution are even more significant.

Despite the fact that, the selection of the usable services and the service space restriction operate over the same data structure (the directed acyclic graph of services and the services themselves), no synchronization is required. In our implementation, due to the guaranteed atomicity of the access and assignment of the program variables, no conflicts can occur. The variable, to which both processes perform assignments, is the variable holding the information about the usability of the service. In any case, for a particular service, a value to this variable is assigned only once and only by one of the processes (it is once set as usable or unusable). Hence, the overall composition process does not suffer from synchronization overhead.

As we mentioned earlier, to realize the composition, we use forward chaining selecting the usable services. Note that in general, the solution for a particular composition goal does not require using all the usable services. This means that it is not necessary to find all the usable services. The selection of them can stop if the actual usable service set provides a solution for a composition goal and it is the optimal one considering the QoS. Hence, we can stop the selection of the usable services earlier and thus potentially save execution time. However, in this case, there is an additional overhead in the process because of the check if we already found a solution. Our experiments with the service sets, generated by the WS-Challenge generator tool, showed that it is not useful to perform a check and stop the selection of the usable services earlier. The reason is that, to find a solution, it was required to select almost all the usable services. We believe that this is only a special case of the data sets generated by the WS-Challenge generator tool. We expect that, in real scenarios with real services, it would be better to check whether we already have a solution for a composition goal and stop the selection of the usable services in this case and thus save execution time.

In the context of improving the performance of our composition system, we have been experimenting with executing the service space restriction in multiple, parallel threads. If the composition system operates on a computer with more than two threads, this could potentially lead to faster composition. It is important that in this case, the synchronization is already an issue. Conflicts may occur if multiple threads try to access the same object at the same time. Hence, the threads must be synchronized during those steps, when they may potentially manipulate the same object.

During our experiments, we tried to run the selection of usable service in 2 up to 6 threads, on a computer with 8 cores. The results showed that the synchronization overhead is too high. Comparing with the situation when the usable services selection runs in one thread, i.e. no synchronization was required, the results with multiple threads were worse.

1.4.3 Conclusions for future work

The automation of the service composition is crucial to be able to supply varying composition goals. Several problems must be addressed and solved to achieve fully automatic service composition. In the last years, the research of service composition automation tended to be focused on the issues related to the QoS, performance, and semantic matching problem. There is a tremendous amount of work concerning these. On the other side, there are plenty of problems without interest.

The performance in the context of service composition aware of the QoS showed to be a challenging task. Even though, there already are approaches showing promising results. Several papers showed that it is possible to realize a service composition even if the number of services considered during the composition rises to thousands, ten-thousands, or even some hundred-thousands. To make an imagination about the number of currently available web services, we made an overview of some public repositories. The number of services which can be found in these repositories varies from some thousands up to some ten thousands. The largest one, called *Seekda* (http://webservices.seekda.com/), included in August 2010 about 28 000 service descriptions, as claimed by the repository provider. Based on this, we believe that the performance of the current solutions is sufficient to handle the real situation.

The research regarding QoS aware service composition is noticeably farther as for example the research dealing with the pre-/post-conditions. We suppose that the research attention in future should move to other problems than the efficiency of QoS aware composition. Since there is still no practical application of service composition, we should analyze what are the real scenarios where it could be applied and which problems have to be solved before this can be achieved.

More attention should be given to those issues which are not related to the actual arrangement of services into a composite one. The whole life-cycle of web services, and their composition should be covered. More interest is required to the design of web services. The design decisions significantly affect the further possibility to compose the developed services. Semantic annotation of web services is also a sub-

ject of further research. Methodologies and tool should be developed to support this process. Finally, it should be analyzed if the description focusing on the I/O and pre-/post-conditions is sufficient.

1.5 Summary

In this chapter we presented a QoS and pre-/post-condition aware web service composition system able to handle changes in the service repository and continuing arrival of the composition queries. The composition system showed to be effective and scalable. We have presented experimental results showing good performance even if the number of services, which have to be considered during the composition, rises to 100 000 services. Even during the hardest test set, the system was able to perform the composition below 200 msecs.

The main sources of the effectiveness of our solution are: effective data structures built during preprocessing and service space restriction. We have analyzed what calculations are required to be performed during service composition and which ones of them do not rely on the knowledge of any composition query. These calculations should be realized as preprocessing. Moreover, we have precisely designed data structures which supply fast operations performed during the composition.

One of the most important issues of our approach is a service space restriction. Its aim is to speed up the composition process by restricting the number of services which must be considered when looking for a solution. Our experiments showed that the restriction is very beneficial. It improved the composition time in more than one order of magnitude.

Another important issue of our approach is the parallel execution of processes realizing the composition. Our composition approach is based on two processes, the selection of the usable services and the service space restriction. These can run in parallel without additional synchronization overhead. The parallel execution showed to be useful on single-core and also multi-core computers. We have been experimenting also with running the selection of the usable services in multiple threads. This approach did not show usefulness, since in this case the synchronization is required and the caused overhead leads to slower execution.

Our approach showed to be effective and scalable even in large-scale situations. We believe that our and also other composition systems proved to be effective enough to be able to operate over the amount of services which are available nowadays. Moreover, they show good scalability, thus, even if we expect that the number of services will raise, the performance will not be a problem.

We believe that the web services composition research should move its focus from performance to other important issues. The existing approaches to automatic service composition base on semantic meta-data. It is known that the idea of semantic web services and the Semantic Web in general is not practically applied. There is a lack of methodologies and tools supporting the creation, processing, and management of semantic metadata. Our future work is to move our focus from performance

issues and deal with other aspect of the service composition, which are important to achieve practical applicability.

Acknowledgment. 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 26240120029, co-funded by ERDF. We would like to thank to the organizers of the WS-Challenge 2009 to provide a great forum progressing the research of web services composition.

References

1. Alrifai, M., Risse, T., Dolog, P., Nejdl, W.: A scalable approach for qos-based web service selection. In: Service-Oriented Computing Workshops, pp. 190–199. Springer-Verlag (2009)
2. Bartalos, P., Bieliková, M.: 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, pp. 414–421. IEEE CS (2009)
3. Bartalos, P., Bieliková, M.: Semantic web service composition framework based on parallel processing. In: Int. Conf. on E-Commerce Technology 2009, pp. 495–498. IEEE CS (2009)
4. Bartalos, P., Bieliková, M.: Effective qos aware web service composition in dynamic environment. In: Int. Conf. on Information Systems Development 2010. Springer (Accepted) (2010)
5. Bartalos, P., Bieliková, M.: Qos aware semantic web service composition approach considering pre/postconditions. In: Int. Conf. on Web Services 2010, pp. 345–352. IEEE CS (2010)
6. Dustdar, S., Schreiner, W.: A survey on web services composition. IJWGS **1**(1), 1–30 (2005)
7. Haddad, J.E., Manouvrier, M., Rukoz, M.: Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. IEEE Transactions on Services Computing **99**(PrePrints), 73–85 (2010)
8. Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., Liu, Z.: Qsynth: A tool for qos-aware automatic service composition. In: Int. Conf. on Web Services 2010, pp. 42–49. IEEE CS (2010)
9. Kona, S., Bansal, A., Blake, M.B., Gupta, G.: Generalized semantics-based service composition. In: ICWS '08: Proc. of the 2008 IEEE Int. Conf. on Web Services, pp. 219–227. IEEE CS (2008)
10. Lécué, F., Mehandjiev, N.: Towards scalability of quality driven semantic web service composition. In: Int. Conf. on Web Services 2009, pp. 469–476. IEEE CS (2009)
11. Mrissa, M., Ghedira, C., Benslimane, D., Maamar, Z., Rosenberg, F., Dustdar, S.: A context-based mediation approach to compose semantic web services. ACM Trans. Internet Techn. **8**(1) (2007)
12. Papazoglou, M.P.: Web services and business transactions. WWW'03 **6**(1), 49–91 (2003)
13. Rosenberg, F., Muller, M.B., Leitner, P., Michlmayr, A., Bouguettaya, A., Dustdar, S.: Metaheuristic optimization of large-scale qos-aware service compositions. IEEE International Conference on Services Computing pp. 97–104 (2010)
14. Rozinajova, V., Kasan, M., Navrat, P.: Towards more effective support of service composition: Utilizing ai-planning, semantics and user's assistance. Int. Conf. on Next Generation Web Services Practices pp. 50–55 (2009)
15. Yu, H.Q., Reiff-Marganiec, S.: A backwards composition context based service selection approach for service composition. In: Int. Conf. on Services Computing 2009, pp. 419–426. IEEE CS (2009)