A Service Composition Approach Based on Patterns

Guangjun Cai^{1,2, a} and Bin Zhao^{1, b}

¹ The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, CAS
Beijing 100190, P.R.China

² Graduate University of Chinese Academy of Sciences, Beijing 100049, P.R.China

^a caiguangj@hotmail.com (corresponding author), ^b zhao.ict@gmail.com

Keywords: Pattern; Service Composition; Reuse; Automatic

Abstract. Service composition is an emerging way to provide the value-added service. The existing automatic composition approaches pay little attention to service reuse. In this paper, we propose two kinds service composition patterns to improve the reusability of the composition result and propose an approach to compose services based on pattern. Meanwhile, the composition approach can be done in an automatic manner without any other knowledge. The experiment and analysis show it is effective.

Introduction

Service composition, which can create new value, to solve complex problems or to best suit the request by reusing the existing services, is crucial to the success of Service-Oriented Computing[1] and plays an important role in cloud computing [2]. A lot of researches have been done on it. Most of them focus on automatic service composition. But for the complexity of service composition, few automatic service compositions methods can completely solve it independently and we need pay more attention on service reuse. However, the composition result generated by automatic methods is rather unstructured and flat process because services have been composed in a rather ad hoc and opportunistic manner [3]. It is very difficult to reuse.

Design patterns have proven they can make software more flexible, modular and reusable in object oriented designing and programming [4]. They provide a better way to organize the objects. For the limitation of the patterns and the differences between services and objects, however, we cannot directly use them in service composition, especially cannot be used in an automatic way. In this paper, through conceiving design patterns as a method to simplify relations among different objects from complex relation to simple relations, we propose a service composition approach based on patterns.

The rest of this paper is structured as follows. First, we introduce the aggregation pattern and decomposition pattern. Then, an improved composition method is proposed based on the two patterns. Third, we compare the composition result and efficiency before and after using the pattern and some related works are discussed. In final section, we conclude the results and give some advices for the future direction.

Service Composition Patterns

Software patterns have gained wide acceptance in the field of software development since they provide a systematic reuse strategy for design knowledge [4]. We conceive the patterns in [4] as some rules to simplify the relations among objects [5] and use OWL-S [6] as a specification language to describe the services. Two kinds of patterns are presented for considering the relations among the services having the same functions, the relations between the constant part and variable parts in one composite service.

Aggregation Pattern. Aggregation pattern aggregates a set of services having similar or same function in a bottom to up manner and provides a unified interface for them. This results in a more flexibility relations between the users and these services. The idea of this pattern comes from the compositor, decorator, façade pattern, and so on in [4]. It is applicable to the situation that more than one service can provide the same function.

Motivation. In order to prevent a service from being bind directly by its user, separate service changes to its user, let service can provide more types functions and be easily replaced by simialr service without changing the custom services.

Condition. Aggregation pattern is used for composing the services which have similar characters. In OWL-S, some categories of functions equivalence are listed below.

- signature equivalence: the inputs and outputs of one service are equivalent to those of another service;
- functional equivalence: beside with the equivalent inputs and outputs, the preconditions and effects of one service are equivalent to those of another service;
- behavior equivalence: one service not only has the equivalent inputs, outputs, preconditions and effects with another service, but also has equivalent process model.

Besides these, there are some other type equivalences. For example, two or more services are provided by the same provider, can be used at the same time, and so on.

Structure. The structure of this pattern is shown in figure 1. *Service₁* and *Service₂* represent the existing services which have similar functions. *AggService* represents a service for invoking and managing the existing services.

Fig.1. The structure of aggregation pattern

Generation. The task generating this pattern is to create an aggregation service to organize the services having similar characters. Some knowledge to determine how to select the suitable services is needed. The algorithm is listed as follow.

Algorithm1 GenAggPattern

Input: SerSet, rule₁, newSer; //rule₁ represents the knowledge to determine the relation among services, newSer represents the new resources

Output: AggService

if(each service in SerSet have the same character *f* according to the rule₁)

 Create a service *AggService* according to the rule₁ and the character *f*;

 Set each services in *SubSerSet* as a service inheriting the service *AggService*;

endif

The complexity mainly depends on the type of services equivalence and the rules to create aggregation service.

Consequence. Aggregation pattern introduces a new kind of services for locating and utilizing the existing services to provide more value-added service to user. It can let users treat various services uniformly.

Through aggregating the services with the same functionality, this pattern reduces the relations between the services and the user from many-to-many to one-to-one between the user and the *AggService* and some one-to-many relations between the services and the *AggService*. It is easier for us to add new services.

But aggregation pattern introduces a new kind of services. Consequently, they need new approaches to manage and invoke. When there are only few services having similar functions, it may result to increase the total number of services. And, it has the disadvantage of making the services overly general. You might find it harder to restrict certain services in some time.

Decomposition pattern. Decomposition pattern separates variable parts from a service to satisfy more requests. This results in a more flexibility relations between the variable parts and other parts. The idea of this pattern comes from the factory, state and strategy pattern in [4]. It is applicable to the situation the user needs multi-variants of the service.

Motivation. In order to increase the flexibility and reusability of a composition service, let it provide more types functions and adapt the changes of the requirement.

Condition. Decomposition pattern is used when there are one or more variable parts in a composite service. In OWL-S, only the composite service has a process description and can be modified. Some categories of service changes are listed below.

- path change: there is a *Choice* or *If-Then-Else* element in the process model;
- num change: there is a *Iterate*, *Repeat-Until* or *Repeat-While* element in the process model;
- dependence change: there is a *Split* or *Split+Join* in the process model.

Besides these, there are some other changes in a service, such as the change of data type and the platform the service deploys on.

Structure. The structure of this pattern is shown in figure 2. $Variant_1$ and $Variant_2$ represent the variant of one variable parts of the existing service. *VarService* is a new introduced service providing an interface for the variable part. *Service Constant* is a new service consisting of invariable parts and the abstraction of the variable part of the existing service.

Fig.2. The structure of decomposition pattern

Generation. The task of generating this pattern is to separate the variable parts from a service and to construct a variable hierarchy. Some knowledge to determine how to manage the variable parts is needed. The algorithm is shown as follow.

Algorithm2 GenDecPattern

Input: $ser, newSer, rule_2$ // $rule_2$ is the knowledge to represent one variable part in ser

Output: *VarService*, *Constant*, *VarSerSet*

if (there is variable parts in ser according to the $rule_2$)

$VarSet \leftarrow \{var \mid var \text{ is a variant of the variable part } \}$;

Encapsulate each element in $VarSet$ as a new service and add it into $VarSerSet$;

endif

Create a service *VarService* according to the $rule_2$ and $VarSerSet$;

Set each services in $VarSerSet$ as a service implementing or specifying the service *VarService*;

Modify ser to construct service *Constant* by replacing the variable part with the invocation to *VarService*;

The complexity mainly depends on the complexity of the existing service and the rules to create the service responds for the variable parts.

Consequences. Decomposition pattern introduces three new kinds of services for managing an existing composite service with variable parts. The variable parts are separated from the services.

Through dividing the variable parts and the constant part, this pattern reduces the relations between them from many-to-many to one-to-one between the invariable parts and the *VarService* and some one-to-many relations between the *VarService* and each variant. This results that the constant part can be connected with more variants.

The disadvantage of this pattern is that it may produce more services than one without it. And its effects depend on the accuracy of the forecast to the changes.

Service Composition Approach

The two kinds of patterns can be used for any service composition method only when their composition result can be reused. The application of pattern in service composition is shown in algorithm3. The complexity of the part except *comMeth* depends on the number of the variable parts, the number of variant of each variable part, the number of aggregation pattern and the number of

selected component services. Of course, the pattern can be used in some other times of service composition, such as using the aggregation pattern in the process discovery the component service to improve the efficiency.

Algorithm3 SerComBasedPattern

Input: serSet, k, r, rule₁ and rule₂ //k is the knowledge to compose the service, r is the requirement needing to satisfy, rule₁ and rule₂ are same to the previous algorithm1 and algorithm2
Output: comSer
comSer=comMeth(serSet, k, r); //comMeth is an existing service composition approach
for(each variable parts *vp* in *comSer* according to the rule₂)
 if(there is decomposition pattern *dp* that corresponding to *vp* according to the rule₂)
 Replace the *vp* with the corresponding varService in *dp*;
 else
 create a pattern for the variable parts *vp* in the *comSer*;
 endif
endfor
for(each component service *s* in *comSer*)
 if(there is aggregation pattern *ap* that equivalent to *s* according to the rule₁)
 Replace the *s* in service *comSer* with the corresponding aggService in *ap*;
 endif
endfor

Example and Analysis

Consider an example in student managing. The composition result without using patterns is shown in figure 3(a). It can be only partial match or mismatch between the request and the service in most times. Meanwhile, it is difficult to change or extend the service. After using the decomposition patterns, the result is shown in figure 3(b) and we can extend the variable parts to support more changes. And after using the aggregation pattern, we can provide the service in various levels and use the service coming from different providers. The composition result using *eating* aggregation pattern is shown in figure3(c).

Fig.3. An illustration example before and after using patterns

We test this approach in a prototype platform with a wide set of randomly generated services and requests. The number of satisfied request before and using pattern are compared in figure 4(a). And the number of satisfied request using pattern based on 1000 and 2000 services are compared in figure 4(b). The results show that the method is effective.

Related Work

Only a few works have considered reuse problem by decoupling the relations among services or raising the abstract level. The work in [3] developed a framework for analyzing composite service reuse and specialization. Aiming to raise the level of abstraction in services composition, they

introduce the concept of service component class. The work in [8] suggests to automatic generate low couple service model by creating a high level semantic model. None of them can be used to create or determine the structure or relation among the component services.

Fig.4. Performance comparisons before and after using patterns

Besides these, patterns have been used in many areas of service composition. One type of them is used as the summary of the applications knowledge, such as the work in [9]. Second, patterns in [10] are used to classify the interaction relationships among services. The third type patterns are dedicated to improve service in various aspects. Different from them, we use the pattern as a method to improve the reusability of the service composition result.

Conclusion and Future Work

This paper proposes two kinds of patterns and one composition approach based on patterns. It not only can provide an automatic composition approach, but also it can be done in automatic manner to provide more reusable composition results without any other context knowledge.

Subsequent work will extend the patterns for supporting more composition method in more specific domain or with more specific manage knowledge. In other hands, the relation between the patterns and the other factors need further research.

Acknowledgements. This work is partially supported by the Key Project of National Natural Science Foundation of China under Grant No. 60736015 and 90818026, the National 973 Fundamental Research and Development Program of China under Grant No. 2009CB320701.

References

- [1] M.P. Papazoglou, D. Georgakopoulos, Service-oriented computing, Communications of the ACM, 2003, 46(10):25–29.
- [2] W. Yi, M.B. Blake, Service-oriented computing and cloud computing: challenges and opportunities, IEEE Internet Computing, 2010, 14(6): 72-75.
- [3] J. Yang, Web Service Componentization, Communications of the ACM, 2003, 46(10): 35-40.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, USA, 1994.
- [5] G. Cai, Design pattern projection based on structure, High Performance Computing Technology, 2010, 6:36-41.
- [6] D. Martin and M. Burstein(eds.), OWL-S: Semantic Markup for Web Services, 2004, <http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/>
- [7] B. Benatallah, F. Casati, D. Grigori, et al, Developing adapters for web services integration, in Proc. 17th Int. Conf. Advanced Information Systems Engineering, 2005, pp.415-429.
- [8] R. Quintero and V. Pelechano, Conceptual Modeling of Service Composition using Aggregation and Specialization Relationships, in Proc. 44th annual Southeast regional conference, ACM, 2006, pp. 452-457.
- [9] M. Tut and D. Edmond, The Use of Patterns in Service Composition. in Proc. 6th Int. Workshop Web Services, Business, and the Semantic Web, 2002, pp. 28-40.
- [10] A. Barros, M. Dumas, and H. M. ter Hofstede, Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection, 2005, 1-26, doi. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.7328>.