Patterns Support for Automatic Resource Management in Cloud Computing

Guangjun Cai
Information Engineering College, Henan University of Science and Technology, Luoyang 471003, China
Email: caiguangj@hotmail.com

Lei Zhang, Bin Zhao and Yong Liu
Beijing Smartdot Tech. Co., Ltd, Beijing 100192, China
The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
Information Engineering College, Henan University of Science and Technology, Luoyang 471003, China
Email: {zhanglei1980, zhaobingmail,luoyangliuyong}@163.com

***Abstract*—Cloud computing has become a popular topic in many areas. It can publish various resources offered by a number of vendors as services. However, there is no valid technology to manage the resources or services. In this paper, the relations among the resources are classified into four classes: the relations among the resources having the same character, the relations among the variable part and invariable part of a service, the relations among the resources having the same interaction manner, and the relations among the resources that are incompatible but have to interoperate. Then four patterns are proposed to manage the resources in cloud. The motivation, condition, structure, generation and consequence are presented for each pattern. They can be used to improve the availability, scalability and reusability of the resources provided.**

***Index Terms*—Automatic resource management, Design pattern, Reusing, Semantic, Cloud computing, Service composition**

I. Introduction

Cloud computing [1,2], a method of computing as a utility, has the potential to transform a large part of the IT industry, make software or service and shape the way that system is designed and purchased. With the advantages such as cost savings, high availability and easy scalability, it has emerged as a commercial reality and many vendors and industry observers predict it a bright future. 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 cloud computing. But for service composition to suit the future cloud computing is usually used to satisfy an on-demand request in an open, dynamic environment and need to discovery suitable services from a large number of existing services with a high complexity degree [3], there are few methods solving it lonely. Meanwhile, the resources in cloud often are provided in a virtual manner and come from various providers. Through using the domain control knowledge in some methods such as Ref. [4] can improve service composition efficiency and support more complex composition goal, they cannot replace the role of reuse. However, most of the composition results can only be reused in a low level for they are rather unstructured and flat process [5].

To address these limitations, some patterns are proposed in this paper. Design patterns can make software more flexible and reusable in object oriented programming by providing tested, proven development paradigms [6]. They present a better way to divide or organize the object and class. But most of them can only be used in a manual way, we cannot directly use them in cloud computing. In this paper, through considering design patterns as methods to simplify the relations among different objects, we propose four kinds patterns to manage resources.

The rest of this paper is shown as follows. In section II, we introduce the patterns and a specification language to describe the resources. Then the aggregation pattern, decomposition pattern, interaction pattern and mediation pattern are presented in section III. We analysis their effects and discuss some related works in section IV. Finally, the results are concluded and some advices for the future direction are given in section V.

II. Preliminary

Patterns as a general reuse strategy received wide acceptance in the field of software [7]. They can be categorizes into three categories: a template, a solution used to resolve the recurring problem and a rule depicting a relation between a context, a problem, and a solution. The resource manager pattern in this paper is derived from the design pattern in Ref. [6] used to keep objects separated and flexible. This may be helpful to manage resources in cloud. However, most of the patterns are given in a inform manner, we cannot automatically use.

The patterns in Ref. [6] can be thought of as knowledge to simplify the relations among objects or different parts in an object. Each pattern can reduce the many-to-many relationship, one-to-many relationship or many-to-one relationship by dividing or introducing

doi:10.4304/jsw.9.10.2741-2748

additional levels of indirection. For example, bridge pattern decouples an abstraction from its implementation allowing them to vary independently; build pattern simplifies the relation between the construction process and its representation. Based on this, this paper uses them as a method to regulate recourses, to provide high-availability and scalability services.

Semantic is a prerequisite for automatic discovery, automatic composition and automatic management. Consequently, we need a specification semantic language to describe the resources for automatic management. OWL-S [8] is a standard service ontology recommended by W3C. It consists of the profile, the process model and the grounding. The profile advertises the service in a concise manner. It includes the service name, contacts description, input, output, precondition, effect, parameter, category and type. The process model gives a detailed perspective on how to interact with a service. An atomic process is a description of a service that expects one message and returns one message. A composite process is composed of one or more processes. The control constructs in a composite process includes: sequence, split, split+join, choice, any-order, condition, if-then-else, iterate, repeat-while, and repeat-until. A simple process may provide a view of some atomic process or a simplified representation of some composite process. The grounding specifies the details of how to access the service. It can be seen as a mapping from an abstract specification to a concrete specification of a service. In this paper, we concentrate more on process model.

III. Resource Management Patterns

In order to use patterns in service composition and resource management, we in Ref. [9, 10] firstly classified the resources to three classes: simple service, composite service and atomic service according to process model in OWL-S. Secondly, we conceive the patterns as methods to divide or simplify the relations. Thirdly, we determine the relations we need to simplify. The relations can be classified into four categories. Ones are the relations among the resources with the similar character; the seconds are the relations in one resource or service; the thirds are the relations between the resources with the similar interactions or structure; the finals are the relations between the two or more resources needing to interact or compose. Finally, we present the motivation, condition, structure, generation and consequence for each pattern.

A. Aggregation Pattern

Aggregation pattern manages a set of resources which have the similar or same character. It generates a new aggregation service in a bottom to up manner. This results a more flexibility relation between the resources and the clients. The idea of this pattern comes from the compositor, decorator, façade pattern, and so on in Ref. [6]. It is applicable to the situation that more than one resource can provide the same function or be used in similar manner.

a. Motivation

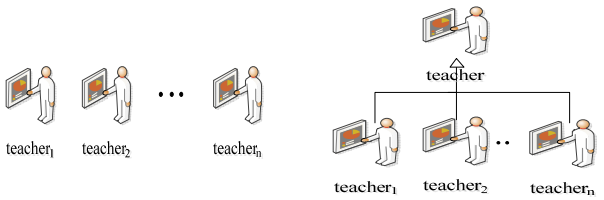


Figure 1. Motivation example of aggregation pattern

Consider the situation library management or that several teachers teach the same content. In the latter, it may be difficult to let student select teacher. And if binding the request to a specific teacher, it may be fail if the teacher has any trouble. A solution is to create a school to manage these teachers to provide the education services. The change is shown in Fig.1.

b. Condition

Aggregation pattern is used for organizing the resources owning similar characters. In OWL-S, some categories of equivalence are listed below.

- name equivalence: the name of one resource is same with another resource name;
- content equivalence: the content of one resource is equal to the content of another resource;
- signature equivalence: the inputs and outputs of one resource are equivalent to those of another resource;
- functional equivalence: beside with the equivalent inputs and outputs, the preconditions and effects of one resource are equivalent to those of another resource;
- behavior equivalence: one resource not only has the equivalent inputs, outputs, preconditions and effects with another resource, 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.

c. Structure

This pattern is a unary pattern, as listed in Fig. 2. *Resource*$_1$ and *Resource*$_2$ represent the existing resources which have similar characters. *FuncManager* is a newly introduced service for invoking and managing the existing resources.

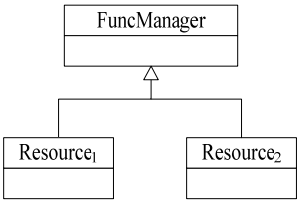


Figure 2. The structure of aggregation pattern

In OWL-S, *Resource*$_1$ and *Resource*$_2$ can be any type service; *FuncManager* can be a simple or composite service. *Resource*$_1$ and *Resource*$_2$ inherit the resource FuncManager.

d. Generation

The task to construct this pattern is to create an aggregation service for managing the resources having the similar characters. The algorithm is listed as follow. It needs some knowledge such as listed in section b to determine how to select the suitable services.

Algorithm1 GenAggPattern
Input: ResSet, $Rule_1$, newSer;//$Rule_1$ represents the knowledge, *newSer* represents the new resources
Output: FuncManager
Initialization: //used for a group resources in *ResSet*
for(each *subResSet*⊆*ResSet* have the same character *f* according to the $Rule_1$)
Create a service *FuncManager* according to the $Rule_1$ and the character *f;*
Set each resources in *SubResSet* as a resource needing to be managed by *FuncManager*;
endfor
updation: // used when adding a new resource into *ResSet*
if(there existing resource *r* in *ResSet* have the same character *f* with *newSer*)
if(there is *FuncManager* for *r*)
add *newRes* as a resource needing to be managed by *FuncManager;*
else
Create an service *FuncManager* according to the $Rule_1$ and the character *f;*
add *newRes* and *r* as resources needing to be managed by *FuncManager;*
endif

Its complexity depends on the type of resources equivalence and the rule to create aggregation service. In OWL-S, when improving the availability, the aggregation service can be implemented as a simple service; when improving the scalability, the aggregation service can be implemented as a composite service.

e. *Consequence*

Aggregation pattern introduces a new kind of resources for managing, locating and utilizing the existing resources to provide more value-added service to the user. It can let cloud users treat various resources uniformly or organize them into a hierarchical structure.

Through aggregating the resources with the same character, this pattern reduces the relations between the cloud resources and the cloud user from many-to-one to one-to-one between the user and the *FuncManager* and some one-to-one relations between the cloud resources and the *FuncManager*. Thus, this pattern can make it easy for you to introduce new kinds of resources or replace the existing resources.

Aggregation pattern let the resource availability unless all the resources it aggregates are out of date. When there are n resources with availability equal m, the availability can be improved from *m* to *1-(1-m)*n*. And we can scale up or scale down the efficient by selecting the appreciate resource or using more or less resources.

But aggregation pattern introduces a new kind of services. In some cases, we need some new methods to manage and invoke the aggregation services.

B. Decomposition Pattern

Decomposition pattern separates variable parts from a resource to satisfy more requests. This results in more flexible relationship between the variable parts and the invariable parts. The idea of this pattern comes from the factory pattern, state pattern and strategy pattern in Ref. [6]. It is applicable to the situation that the user needs multi-variants of the service.

a. Motivation

Consider the situation that a student only needs partial function of a service or partial content of a resource at a time. If each part of the service was static binding, it could be only partial match or mismatch with the request in most times. Meanwhile, it is difficult to change or extend the service. A solution is to create a separate service to manage the variable parts in the service in order to provide various functions. The change is shown in Fig.3.

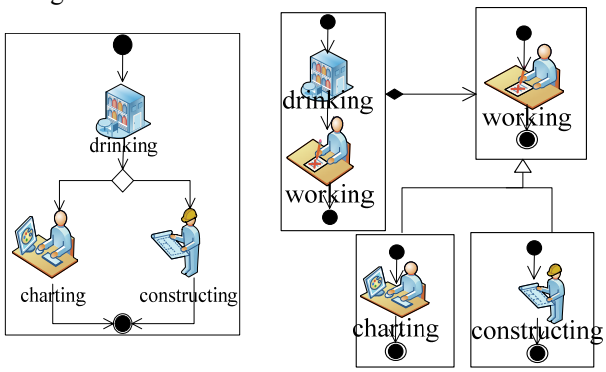


Figure 3. Motivation example of decomposition pattern

b. *Condition*

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

- choice change: there is one or more *Choice, Condition* or *If-Then-Else* element in the process model;
- times change: there is one or more *Iterate, Repeat-Until* or *Repeat-While* element in the process model;
- parallel change: there is one or more *Split* or *Split+Join* in the process model.

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

c. *Structure*

This pattern is a binary pattern, as shown in Fig. 4. $Variant_1$ and $Variant_2$ represent the variant of the variable parts of an existing resource. *VarManager* is a new service which provides an interface for each variable part. *Constant* is a new service consisting of invariable parts.

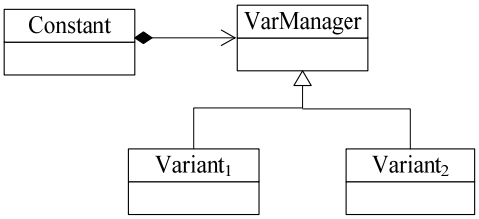


Figure 4. The structure of decomposition pattern

In OWL-S, *Variant*$_1$ and *Variant*$_2$ can be any type services; *VarManager* can be a simple or composite service; *Constant* is a composite service. *Variant*$_1$ and *Variant*$_2$ inherit *VarManager*, and *Constant* invokes *VarManager*.

d. Generation

The task generating this pattern is to separate the variable parts from a resource and to construct some variable hierarchies. It needs some knowledge to determine how to determine and to manage the variable parts. The algorithm is shown as follow.

Algorithm2 GenDecPattern
Input: res,newVar,rule$_2$ //*Rule*$_2$ is the knowledge to manage the variable parts in a resource *res*
Output: VarManager, Constant, VarSet
Initialization: //used for a resource
if(there is a variable part *in* a resource *res* according to the *Rule*$_2$)
VarSet←{*var* | *var* is a variant of the variable part };
Encapsulate each element in *VarSet* as a new service or resource and add it into *VarSet*;
endif
create a service *VarManager* according to the *Rule*$_2$ and *VarSet;*
set each services in *VarSet* as a service needing to be managed by *VarManager*;
modify *ser* to construct service *Constant* by replacing the variable part with the invocation to *VarManager*;
updation: // used when a new resource can be as a variant of the variable part
if(there existing a service *VarManager* that can manage *newVar*)
add *newVar* as a resource needing to be managed by *VarManager;*
endif

The complexity mainly depends on the complexity of the existing service and the rules to create service managing the variable its parts. In OWL-S, the *VarManager* service can be implemented as a simple service.

e. Consequence

Decomposition pattern introduces three new kinds of services for managing an existing service with the variable parts. The variable parts are separated from the service in order to satisfy more requests.

Through separating the variable parts from a resource, decomposition pattern reduces the relationship between the variable parts and the constant part from one-to-many to one-to-one and some one-to-many relationships. Thus, the same parts can be connected with more variants.

Decomposition pattern can help to use each part in more situations. When there are *n* variable parts with *m* variants in a service, this pattern can improve the number of requests that the service equal matches from *1* to *m*n*. And we can scale up or scale down the efficient by changing or replacing the variant with the appreciate service or by increasing or decreasing the number of services. And it can help to improve the flexibility of the variable parts of the resource.

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

C. Interaction Pattern

Interaction pattern decouples the composing (invoking) resources and the composed (invoked) resources so that they can vary independently. This results in high-reusability of the organization structures, composition methods or interaction manners and the relative elements. The idea of this pattern comes from the bridge pattern, the iterator pattern, the abstract factory pattern and so on in Ref. [6]. It is applicable when a group of resources can be interoperated or invoked in different manners or a group of structures or processes can be used to organize the different resources.

a. Motivation

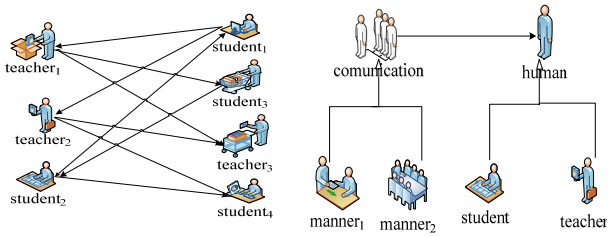


Figure 5. Motivation example of interaction pattern

Consider the situation that several interaction manners can be used both among the teachers and among the students. Two class of services need to create for each interaction manner. And when the other staffs come, new services as many as the number of interaction manners need to be created. A solution is to provide a mechanism to manage the relationships between the interaction manners and the entities. The change is shown in Fig. 5.

b. Condition

Interaction pattern is used when some resources have the same or similar interaction or some structure and process can achieve the equal effect. In OWL-S, some categories of interaction equivalence are listed below.

- interaction type equivalence: the type of messages two or more resources receiving and sending is same;
- interaction content equivalence: the content of messages two or more resources receiving and sending is same;
- interaction order equivalence: the order that two or more resources receive and send messages is same;
- interaction structure equivalence: the effect that two or more structures is equivalent;

- interaction process equivalence: two or more process is equivalent.

Besides these, there are some other type interaction equivalences. For example, two or more resources are described at the same language or need be used on the same security level.

c. Structure

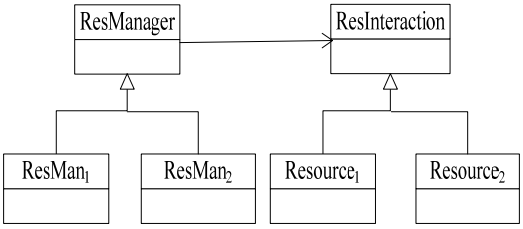


Figure 6. The structure of interaction pattern

This pattern is a binary pattern, as shown in Fig. 6. $Resource_1$ and $Resource_2$ represent the existing resources having the same invocation manner. *ResInteraction* provides an interface encapsulating the invocation manner of $Resource_1$ and $Resource_2$. $ResMan_1$ and $ResMan_2$, standing for the resource can be interacted in a specific manner. *ResManager* provides an interface encapsulating the resources.

In OWL-S, $Resource_1$ and $Resource_2$ can be any type services; $ResMan_1$ and $ResMan_2$ can also be any type services; *ResInteraction* and *ResManager* are simple services. In this pattern, $Resource_1$ and $Resource_2$ inherit *ResInteraction*, $ResMan_1$ and $ResMan_2$ inherit *ResManager*, and *ResManager* invokes *ResInteraction*.

d. Generation

The task generating this pattern is to create high level services to manage the interaction, the composition or invocation of the resources with the same interaction manner. Some knowledge to determine how to invoke the suitable resources and their composition are needed. To create a specific composition service, we can use the methods proposed in Ref. [4, 5, 11] or some others. The algorithm to use the pattern is shown as follow.

Algorithm3 GenIntPattern
Input: ResSet, newRes, newSer, $rule_3$, $rule_4$;//$Rule_3$ is the knowledge manage the resources with the same interaction manner; $Rule_4$ is the knowledge how to compose or invoke one or more resources from a group of resources with same interaction manner
Output: ResManager, ResInteraction, ResManSet
Initialization: //used for a group resources in *ResSet*
if(each $subResSet \subseteq ResSet$ have the same interaction *int* according to the $Rule_3$)
  Create a service *ResInteraction* according to the $Rule_1$ and the interaction *int*;
  Set each resource in *SubResSet* as a resource needing to be managed by *ResInteraction*;
  for(each composition manner responding to *ResInteraction* according to $Rule_4$)
    create a service $ResMan_i$ and add it into *ResManSet*;
  endfor
  create a service *ResManager* according to the $Rule_4$ and *ResManSet*;
endif
updationResInteraction: // used when a new interaction manner are introduced
if(there existing a resource *r* in *ResSet* have the same interaction *int* with *newRes*)
  if(there is *ResInteraction* for *r*)
    add *newRes* as a resource needing to be managed by *ResInteraction;*
  else
    create a service *ResInteraction* according to the $Rule_3$ and the interaction *int;*
    add *newRes* and *r* as resources needing to be managed by *ResInteraction;*
endif
updationResManager: // used when a new manager manner is introduced
if(there existing a *resMan* in *ResManSet* have the same manner with *newResMan*)
  if(there is *ResManager* for *resMan*)
    add *newResMan* as a subserivce needing to be managed by *ResManager;*
  else
    create a service *ResManager* according to the $Rule_4$;
    add *newResMan* and *resMan* as subserivce needing to be managed by *ResManager;*
endif

The complexity mainly depends on the type of resources interaction and the composition methods. In OWL-S, when improving the availability, the abstract interaction or composition service can be implemented as a simple service; when improving the scalability, they need be implemented as composite services.

e. Consequence

Interaction pattern introduces three new kinds of services responsible for the cooperation among some resources or the composition of some resources with the same interaction manner to provide value-added services. It decouples the cooperation process or composition structure from the specific resources so that they can vary independently.

Through constructing the abstract interaction service and the invocation or composition service, this pattern reduces the relation between an invoker (composer) and a service from many-to-many to some many-to-one relations between the invoker (composer) and the specific composite services, one-to-one relation between the *ResInteraction* and *ResManager* and one-to-many relation between the *ResInteraction* and the resources. Thus, this pattern can make it easy for us to add new kinds of resources as long as they can support a similar interaction or composition interface.

Interaction pattern lets the resources, interaction manner and composition logic to be used more situations. When there are n services can be invoked in m manner, this pattern can improve the number of satisfied requests from $m+n$ to $m \times n$. And we can scale up or scale down the efficient by changing or replacing composition manner. And it can help to improve the flexibility among a group of resources.

The disadvantage of this pattern is that it introduces a very complexity interaction hierarchy and three new

kinds of services. That may lead to new problem in some application.

D. Mediation Pattern

Mediation pattern introduces mediator hierarchies as a manager to decouple the relation among two or more related resources. This results in a more flexibility collaboration and lower couple among the resources. The idea of this pattern comes from the adapter pattern, memento pattern, and proxy pattern in Ref. [6]. It is applicable when a set of services need to collaborate but with some mismatch.

a. Motivation

Consider the situation that one teacher commonly need to teaching a lot of students. It is inconvenient to do in a one to one teaching model in most times for its low effectiveness. We usually do that in a classroom. The change is shown in Fig. 7.

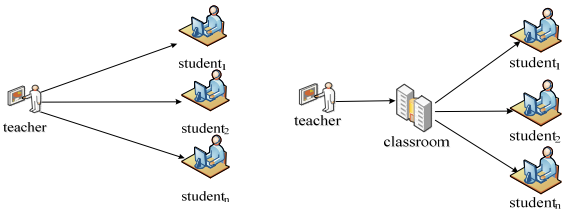


Figure 7. Motivation example of mediation pattern

b. Condition

Mediation pattern is used when there are two or more different type resources needing to collaborate or provide value-added but cannot be seen as a whole. It is often with some kind mismatches among them. For more detail of the mismatch, please see the work in Ref. [12], etc. In OWL-S, some categories of mismatches are listed below.

- number of messages mismatch: one resource needs to send messages to two or more resources, or vice versa;
- message type mismatch: the type of the message one resource sending is the subtype of the message another resource needing to receive, or vice versa;
- message content mismatch: the content of the message one resource sending is the subset or part of the message another resource needing to receive, or vice versa;
- message order mismatch: the order one resource sending messages is different from the order another resource receiving messages, or vice versa.

Besides these, there are some other type mismatches. For example, two or more resources are performed at the different location, used at the different time, and so on.

c. Structure

This pattern is a ternary pattern, as shown in Fig. 8. $Resource_1$ and $Resource_2$ represent the resources needing to collaborate but with some mismatch. $ResInteract_1$ and $ResInteract_2$ are the same with *ResInteraction* in interaction pattern. $Mediator_1$ and $Mediator_2$, standing for the service being responsible for collaboration of the $ResInteract_1$ and $ResInteract_2$. *MedManager* provides an interface for $Mediator_1$ and $Mediator_2$. The tasks of $Mediator_1$, $Mediator_2$ and *MedManager* may be to receive messages from the resouces which sending messages, to send messages to the resources which receiving messages or to adapt the difference between the messages and their order.

In OWL-S, $Resource_1$, $Resource_2$, $Mediator_1$ and $Mediator_2$ can be any type service; $ResInteract_1$, $ResInteract_2$ and *MedManager* can be simple or composite services. In this pattern, $Resource_1$ and $Resource_2$, inherit $ResInteract_1$ and $ResInteract_2$ respectively; $Mediator_1$ and $Mediator_2$ inherit *MedManager*; $ResInteract_1$ and $ResInteract_2$ can interact with each other through *MedManager*.

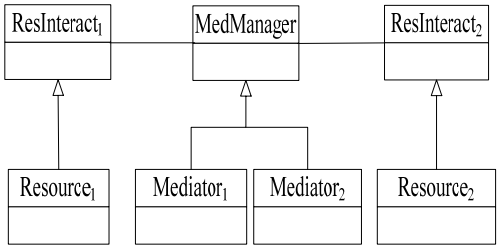


Figure 8. The structure of mediation pattern

d. Generation

The task of this pattern is to create high level services to manage the interaction and mediation of the resources to manage their mediation among the resources. To create a specific mediator service, we can use the methods proposed in Ref. [13]. The algorithm to use this pattern is shown as follow.

Algorithm4 GenMedPattern
Input: ResSet, $Rule_3$, $Rule_5$ //$Rule_5$ is the knowledge how to coordinate the resources cannot collaborate directly
Output: MedManager, MedSerSet, ResInteract
Initialization: //used for a group resources in *ResSet*
  create *ResInteract* for each resource *r* in *ResSet* according to the $Rule_3$, and set *r* as a resource needing to be managed by *ResInteract*;
  for(each mediation manner among the resources in *ResSet*)
    create a service $Mediator_i$ and add it into *MedSerSet* according to the $Rule_5$;
  endfor
  create a service *MedManager* according to the $Rule_5$ and *MedSerSet*;
updationResInteract: //same with the updationResInteraction in Algorithm 3
updationMedManager: // used when a new mediation manner are introduced
if(there existing a service *med* in *MedSerSet* have the same mediator logic with *newMed*)
  if(there is *MedManager* for *med*)
    add *newMed* as a service needing to be managed by *MedManager;*
  else
    create a service *MedManager* according to the $Rule_5$ and the mediator logic *med;*
    add *newMed* and *med* as the services needing to be managed by *MedManager;*
endif

The complexity mainly depends on the type of the mediation logic. In OWL-S, the mediation services need be implemented as composite services.

e. Consequence

Mediation pattern introduces a new kind of service for the collaboration between the services or composition of the resources to provide value-added services. The mediation logic is separated from the existing resources or services, allowing the same service to mediate or organize various resources.

Through constructing a mediation hierarchy, this pattern reduces the relations among the existing resources. It can reduce the relationship from many-to-many to one-to-one between the $ResInteract_i$ and the *MedManager* and one-to-many relation between the $ResInteract_i$ and the resources.

Mediation pattern allows the resources and the mediator to be used in more situations. When there are *n* manners the services can interact with each other, this pattern can improve the availability of mediator from *1* to *n*, the availability of service from *1* to *n*. Meanwhile, we can scale up or scale down the efficient by changing or replacing the mediator service.

The disadvantage of this pattern is that it may lead to a very complexity mediation hierarchy and three new kinds of services. That may result in new problem in some application. Meanwhile, there is still no method to automatically create the mediator service in some cases.

IV. Result Analysis And Comparison

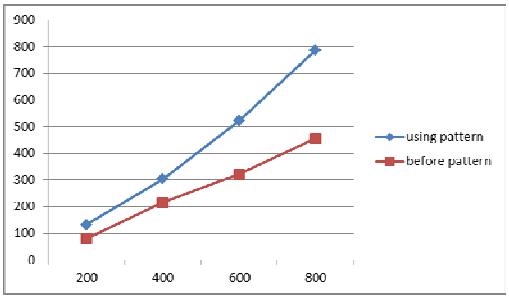
In this section, we analysis the effect of our method and compare it with the existing works.

A. Result Analysis

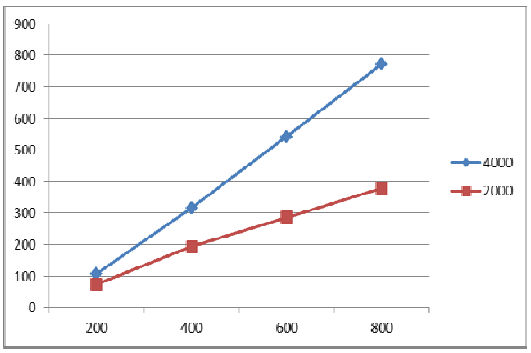
This method has been tested in a prototype platform with some randomly generated resources and requests. The number of satisfied requests before using pattern and that after using pattern based on *1000* resources is compared in fig. 5(a). And the number of satisfied request using pattern based on *2000* and *4000* resources is compared in fig. 5(b). The results show that our method can improve the availability of the resources.

B. Relate Works

Automatic service composition is crucial to cloud computing. Some composition results such as that in Ref. [4] cannot be reused or only be reused as a whole. We need to compose again from scratch once any one constituent service changes or the request change. Only few works have considered reuse problem by decoupling the relations among services or raising the abstract level. For example, a loose coupling operator is introduced in Ref. [11] to handle the sequence composition. It considers nothing about the reuse of the condition structure, loop structure, and the rest structure. The work in Ref. [5] develops a framework for analyzing composite service reuse and specialization. Aiming to raise the level of abstraction in services composition, it introduces the concept of service component class. However, it can only be done in a manual manner. Reference [7] creates a high level semantic model with aggregation and specialization relationships which can be used to automatic generate a composite service instance. However, how to determine the relationship at the concept level is remained nothing.



(a) The number of satisfied request before and using pattern



(b) The number of satisfied request based on 2000 and 4000 resources

Figure 9. Performance comparisons

Patterns have been used in many aspects of service composition. The first type of them is used as the summary of the applications knowledge. For example, patterns in Ref. [14] are proposed by extracting reusable business logic. Secondly, patterns are used to classify the relationships among services. The single-transmission bilateral interaction patterns, single-transmission non-routed patterns and so on are presented in Ref. [13]. The third type patterns in Ref. [15] are dedicated to improve service quality. Facing to the mismatch among services, Reference [12] uses patterns as template to construct adapter. Finally, some patterns may relate to two or more aspects, such as that in Ref. [16, 17].

Different from the existing methods, our work improve service reusability by raise the abstract level of the resources with the four kinds of patterns. Different from Ref. [7], our method concentrate more on the abstraction and separation of the services or resources than the aggregation and the specialization relationship, construct the service or organize the resources from bottom to top rather than from top to down, and be done in an automatic manner rather than a manual manner. Furthermore, our method can construct the new services from scratch based on the semantic description of the existing resources than to generate a specific service by transforming from a high level model. Compared with our previous work, this paper firstly presents four kinds

of patterns and uses them in a new area with some new contents different from the pattern in the former.

V. Conclusion And Future Work

Four kinds of patterns are proposed in this paper for managing resources in cloud computing. The first is used to aggregate the service with the same functionalities; the second is used to separate the changes from the composing result; the third is used to separate the interactions from the function parts and the final pay attention to two different roles. For each of them, we present its name, condition, structure, generation, consequence and example. These patterns can be used in higher automation degree.

Subsequent work will extend the pattern for supporting more effective manage strategy 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.

Acknowledgment

This research was supported in part by the Key Project of National Natural Science Foundation of China under Grant No. 90818026, the National 973 Fundamental Research and Development Program of China under Grant (No. 2009CB320701) and No. 2011CB302704.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, and et al, “Above the clouds: a Berkeley view of cloud computing,” Tech. Rep., U.C. Berkeley, February 2009.

[2] Y. Wei, and M. B. Blake, “Service-oriented computing and cloud computing: challenges and opportunities,” IEEE Internet Computing, vol. 14, no. 6, pp. 72–75, November 2010.

[3] O. Seogchan, L. Dongwone, and R. T. Soundar, “A comparative illustration of AI planning based web services composition,” ACM SIGecom Exchanges, vol. 5, pp.1–10. 2005.

[4] G. Cai, and B. Zhao, “An approach for composing services based on Environment Ontology,” Mathematical Problems in Engineering, in press.

[5] J. Yang, “Web service componentization,” Communications of the ACM, vol. 46, no. 10, pp. 35–40, October 2003.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, USA, 1994.

[7] R. Quintero and V. Pelechano, Conceptual modeling of service composition using aggregation and specialization Relationships, 44th annual Southeast regional conference, 2006, pp. 452–457.

[8] D. Martin and M. Burstein, “OWL-S: Semantic Markup for Web Services,” 2004, http://www.w3.org/ Submission/ 2004/SUBM-OWL-S-20041122/.

[9] G. Cai, L Liang, X Zhao, and R Zheng, Patterns support for automatic service composition, 6th Int. Conf. Automation and Logistics, 2012, pp.101–104.

[10] G. Cai, and B. Zhao, “Interaction and mediation patterns for service composition,” Applied Mechanics and Materials, Vols. 121–126, 2012, pp. 3988–3992.

[11] S. McIlraith and T. C. Son, Adapting golog for composition of semantic web services, 8th Int. Conf. Knowledge Representation and Reasoning, 2002, pp. 482–493.

[12] B. Benatallah, F. Casati, D. Grigori, H. M. Nezhad, and F. Toumani, Developing adapters for web services integration, 17th Int. Conf. Advanced Information Systems Engineering, 2005, pp.415–429.

[13] A. Barros, M. Dumas, and H. M. Hofstede, “Service interaction patterns: towards a reference framework for service-based business process interconnection,” 2005, pp. 1–26, http://citeseerx.ist.psu.edu/viewdoc/summary? doi=10.1.1.93.7328.

[14] M. Tut and D. Edmond, The use of patterns in service composition, 6th Int. Workshop Web Services, Business, and the Semantic Web, 2002, pp. 28–40.

[15] J. Joseph, “Patterns for high availability, scalability, and computing power with windows azure,” 2009, http://msdn.microsoft.com/enus/magazine/dd727504.aspx .

[16] O. F. Rana and D. W. Walker, “Service design patterns for computational grids,” Patterns and skeletons for parallel and distributed computing, UK: London, Springer-Verlag, 2003, pp.237–264.

[17] U. Zdun, C. Hentrich, and S. Dustdar, “Modeling process-driven and service-oriented architectures using patterns and pattern primitives,” ACM Transactions on the Web, vol.1, No.3, pp. 1–44, September 2007.

Guangjun Cai received the MS degree in computer science from the Xinjiang Technical Institute of Physics & Chemistry, Chinese Academy of Sciences, in 2006 and received the phD degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, in 2011. He is a lecturer in Information Engineering College, Henan University of Science and Technology. His research interests include service computing and software engineer.

Lei Zhang is a lecturer in Information Engineering College, Henan University of Science and Technology. Her research interests include software engineer and biocomputing.

Bin Zhao received the phD degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, in 2012. He is a post doctorate in Beijing Smartdot Tech. Co., Ltd. His research interests include requirement engineering, service computing and software engineer.

Yong Liu is a professor in Information Engineering College, Henan University of Science and Technology. His research interests include software architecture and software engineer.