Service Supervision Patterns: Reusable Adaption of Composite Services

Masahiro Tanaka[1], Toru Ishida[1,2], Yohei Murakami[1], and Donghui Lin[1]

[1] Language Grid Project,
National Institute of Information and Communications Technology (NICT)
3-5 Hikaridai, Seika-cho, Kyoto, Japan
{mtnk,yohei,lindh}@nict.go.jp
[2] Department of Social Informatics, Kyoto University
Yoshida-Honmachi, Sakyo-ku, Kyoto, 606-8501, Japan
ishida@i.kyoto-u.ac.jp

Abstract. A composite Web service provided as a "cloud" service should make its constituent Web services transparent to users. However, existing frameworks for composite Web services cannot realize such transparency because they lack capability of adapting changes of behaviors of constituents Web services and business rules of service providers. Service Supervision, proposed in the previous work, allows us to flexibly adapt a composite Web service by combining control execution functions which control behavior of running instances of composite Web services. However, much flexibility of the execution control functions sometimes makes it difficult to design adaptation processes due to absence of accumulated know-how such as guidelines. Moreover, it often costs a lot to port adaptation processes to the model of composite Web service to be adapted. To solve the problems, we first organized various adaptation processes based on some previous works. Then we proposed Service Supervision patterns, which consist of typical requirements for adaptation and WS-BPEL processes satisfying the requirements by using execution control functions. The patterns are easy for designers of composite Web services to understand and make it possible to reduce cost to port them to the model of a composite service.

1 Introduction

In Cloud Computing, servers which provide Web services are transparent to users and users do not need to care numbers or locations of the servers. As for a composite Web service, which combines multiple Web services, the constituent Web services of the composite Web service should also be transparent to users when it is provided as a "cloud service". However, it is often difficult to realize the transparency because the constituent Web services can be provided by various service providers and the behaviors of the services can unexpectedly change. Therefore a composite Web service has to be capable of adapting to the changes.

For example, there are still many services deployed outside cloud and throughput of the services may decline in an environment where too many requests can

D.R. Avresky et al. (Eds.): Cloudcomp 2009, LNICST 34, pp. 147–163, 2010.

be given during a certain period. In that case, a composite Web service which combines such services needs to replace the constituent Web service with an alternative one in order to keep overall performance of the composite Web service. Another example is changes of business rules of service providers. If a service provider which provides one of the constituent Web services changes their business rules and becomes to require some preprocesses before execution of its service, the business logic of the composite Web service must be changed.

However, WS-BPEL[1], a standard language for a composite Web service, is not flexible enough to realize adaptation to frequent changes of the environment or business rules. In the existing framework for WS-BPEL, a model of a composite Web service (a definition of a WS-BPEL process) deployed on the execution engine cannot be modified. Therefore we need to modify the model first and then deploy it on the execution engine in order to adapt a composite Web service to an environment or business rules. This has often prevented flexible and rapid adaptation.

To make up the lack of flexibility, in [2], we proposed Service Supervision, which changes the behavior of a composite Web service without modifying its model using execution control functions such as step execution or changing an execution point. By providing the execution control functions as Web services, we make it possible to define a composite Web service which controls other composite Web service for adaptation. One of the major advantages of Service Supervision is reusability of the composite Web service which implements adaptation. Moreover, the execution control functions realizes more flexible control than that by some previous works on runtime adaptation([3,4,5,6]).

In the environment which frequently changes, however, we still have the following problems even if we introduce Service Supervision.

- **Difficulty in designing adaptation**
  Much flexibility of execution control functions sometimes makes it difficult to design adaptation processes due to the absence of accumulated know-how such as guidelines.
- **Cost of updating model**
  When permanent demand of an adaptation becomes apparent, it is better to update the model of the composite service. But it often costs a lot to port an adaptation process using execution control functions to the model of the composite Web service to be adapted.

Therefore we proposed Service Supervision Patterns, which guide designing adaptation processes for composite Web services. Software patterns including design patterns[7] have achieved a great success in design and analysis of software. Also in the area of workflows, workflow patterns[8] have been widely accepted.

In this paper, we organized various adaptations of composite Web services and extracted typical execution controls as Service Supervision patterns. The Service Supervision patterns consist of requirements for adaptation and WS-BPEL processes which implement the adaptation using execution control functions. Therefore it is easy for designers of composite Web services to reuse the patterns. The

patterns also show how to port the WS-BPEL processes for adaptation to the model of Web service to be adapted.

The rest of this paper is organized as follows. In Section 2, first we describe Service Supervision used to realize adaptation of a composite Web service and explain the prototype we implemented. Next we organize typical adaptation processes of composite Web services and show how to realize the adaptation process using execution control functions in Section 3. Then we propose Service Supervision patterns by extracting processes frequently appear in the previous section. After introducing some related works in Section 5, we conclude this paper in Section 6.

2 Service Supervision

In [2], the authors proposed Service Supervision, which changes the behavior of a running instance of a composite Web service without changing the model of the composite Web service. We show the overview of Service Supervision and explain the prototype that we developed in this section.

Several researches have tried to change behaviors of a composite Web service without modifying the model of a composite Web service. For example, Language Grid[9] provides dynamic binding, which allows a user to specify endpoints (addresses for accessing Web services) when invoking the composite Web service. In this work, a composite Web service is designed based on only the interfaces of the constituent Web services. AO4BPEL[6] and Dynamo[5] allow a user to add processes at certain points in a composite Web service based on the concept of AOP (aspect-oriented programming). However, some functions for adaptation, such as changing an execution point, cannot be achieved by adding a process by AOP.

On the other hand, Service Supervision monitors and changes the state of running instances and controls execution of the instances. This makes it possible not only to add a process to an existing composite Web service, but also to control execution state, including changing an execution point. Using Service Supervision, we can adapt a composite Web service to changes of the environment and business rules without modifying the model and deploying it.

2.1 Execution Control Functions

We implemented execution control functions shown in Table 1 to realize Service Supervision. The functions get/set the state of a running instance of composite Web service or control execution of a composite Web service itself.

The functions are provided as Web services. Therefore we can define a composite Web service which controls the behavior of an instance of other composite Web service by combining the execution control functions.

Although the execution control functions do not change the model of the composite Web service, they realize various processes required for adaptation.

Take an example to clarify the necessity of the execution control functions. In an environment where many Web services are published by various providers,

Table 1. Execution control functions

API	Effect
step	Execute the next activity in a composite Web service.
suspend, resume	Suspend/Resume execution of a composite Web service.
getVariable, setVariable	Get/Set variable defined in a composite Web service.
getState, setState	Get/Set states of activities, such as ready, running, finished and suspended.
setAddress	Set an endpoint address of an invocation in a composite Web service.
setEP	Set the activities which is executed next.
setBP	Set a breakpoint at an activity in a composite Web service and a callback Web service invoked when the the execution stops at the breakpoint.

such as the Language Grid[9], a Web service can be shared by some composite Web services in an unexpected way. For example, execution of the composite Web service in Fig. 1(a) may fail in such an environment. This composite Web service translates a long document. It first splits the given document into sentences (`split`) and then translates the sentences by the machine translation service (`translate`) in the loop. Next, it merges the results of translation (`merge`).

Assume that the provider of the machine translation service newly introduced a limit on number of invocations of its service because too many requests were given during a certain period. In such case, execution by a user may unexpectedly cause a failure of exectuion by another user. Thus, when the number of invocations approaches the limit, we need to switch the service to different one by other provider. To implement this solution, we need to modify the document translation service as shown in Fig. 1(b). Before invoking the machine translation service, the composite Web service invokes the external service to increment the recorded number of invocations (`count`).

However, the change of the model is not efficient when many service providers are involved and policies of the service providers frequently change.

Our solution based on Service Supervision is to introduce a composite Web service shown in the upper part of Fig. 2. This composite Web service counts

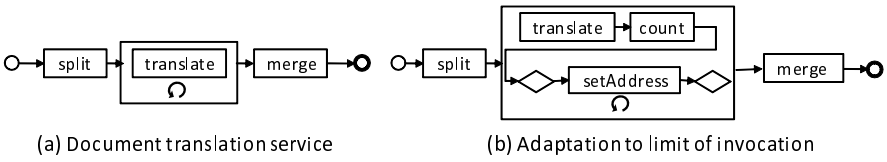


Fig. 1. Modification of a composite Web service for adaptation

the number of invocations of the machine translation service and changes the endpoint address to that of another machine translation service when needed.

The composite Web service first sets a breakpoint (`setBP`) before the invocation of the machine translation service `translate` in the document translation service. It also sets invocation of `count` as the callback Web service for the breakpoint. When `count` is invoked, it increments the recorded number of invocations of the machine translation service (`increment`). If the number of invocations of the machine translation service exceeds the limit, the endpoint address of the machine translation service is changed (`setAddress`).

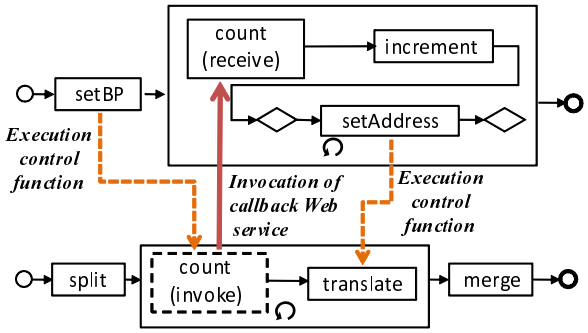


Fig. 2. Composite Web service which controls other composite Web service

One of the major advantages of our solution is reusability of the composite Web service for the adaptation. The composite service in the upper part of Fig. 2 can be applied to various composite Web services in which the number of invocation of a constituent Web service is limited just by setting the breakpoint.

2.2 Prototype

We developed a prototype of Service Supervision by extending an existing WS-BPEL engine, ActiveBPEL[1] as shown in Fig. 3.

The architecture consists of two parts: Composite Web service execution engine and interaction control engine. On the Composite Web service execution engine, both a composite Web service to be controlled and a composite Web service which controls it using execution control functions are executed.

The interaction control engine is responsible for coordination among more than one instances of composite Web services based on a given choreography because some adaptation processes require the instances to be synchronized. Assume that two instances of the document translation service try to invoke `count (invoke)` in Fig. 2 at almost the same time. The composite Web service in the upper part of Fig. 2 receives the request for `count (receive)` that arrives first and starts to increment the number of invocations. If the composite Web service

[1] `http://www.activevos.com/community-open-source.php`

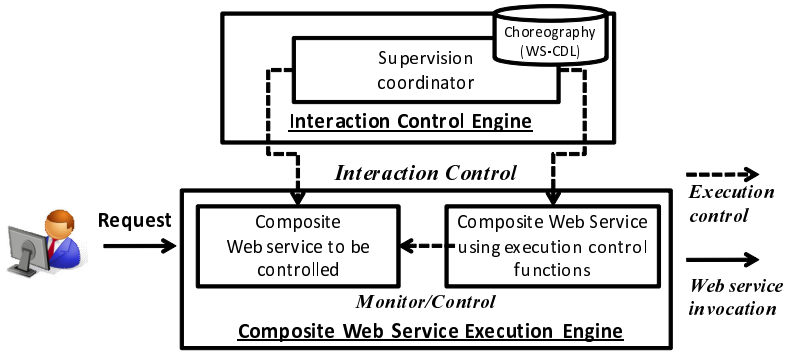


Fig. 3. The implemented prototype

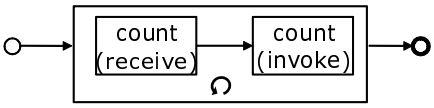


Fig. 4. Choreography for definition of control protocol

receives the request from another instance of the document translation service while incrementing the number of invocations, `count (invoke)` fails because it is not waiting for the request at `count (receive)`.

To solve this problem, we introduce choreography, which defines the protocol of interactions between a composite Web service which controls other composite services and the composite Web service being controlled. We adopt WS-CDL (Web Service Choreography Description Language)[10], a standard language for choreography of Web services. We show an example of choreography in Fig. 4, which defines protocol of interactions between the two composite Web services shown in Fig. 2.

In Figure 4, a rectancle which has a word inside represents an interaction between the two composite Web services. This protocol ensures that the execution of `count (receive)` in the composite Web service which controls the document translation composite Web service and `count (invoke)` in the document translation composite Web service are processed in this order.

3 Adaptation of Composite Service Using Execution Control

In this section, we organize various adaptation of composite Web service explained in some previous works[11,12,13,14]. The aim is to extract reusable processes for various adaptation like the composite Web service shown in the upper part of Fig. 2.

Some adaptation processes described in this section can be realized by the existing framework, such as WS-BPEL. But it is not flexible enough to adapt WS-BPEL process to frequent change of environment or business rule by changing the model of a composite Web service. Therefore we assume that an adaptation process is temporarily realized by Service Supervision, and that it is ported to the model when the adaptation process is permanently required.

3.1 Exception Handling

WS-BPEL provides exception handling mechanism. In a dynamic or open environment, however, the exception handling of WS-BPEL is not flexible enough.

Using execution control functions, we can realize more flexible adaptations as follows:

- **Recovery**
  Exceptions which are unexpected at the design time can be recovered by dynamically adding processes for montoring and recovering.
- **Alert**
  Continuous check of consistency of data enables us to detect symptoms of exceptions and to show an alert.
- **Avoid exception**
  We can often avoid exceptions by adding a preprocess of an input to a service or replacing a task which may cause an exception with a human task,
- **Enforcement by humans**
  In case that execution of a composite Web service cannot recover from an exception by an existing recovering process, humans often need to set states of tasks manually.

3.2 Dynamic Change

We show major adaptation processes which cover the change of requirements of users or state of services below.

- **Dynamic binding**
  In dynamic environment, we often need to select services at runtime. This is achieved by getting a list of available services and setting an endpoint address.
- **On-the-fly composition**
  According to the operator's request, the system is often required to generate a new process and temporarily add it into the composite Web service.

3.3 Human Involvement

BPEL4People[15] is an extension of WS-BPEL and realizes combination of human tasks and Web services. Using the extension, we can define an invocation of a human task in the manner similar to that of a Web service. When a human

task is invoked, the task is sent to a person who is responsible for the task. The human task is finished when the person inputs the result of the task.

However, human tasks often cause an unexpected problem due to the much flexibility of human behavior. We show adaptation processes required to handle the problems with human tasks below.

- **Negotiation**
  When the result of a human task is not good enough, the task needs to be executed again. This process often includes negotiation between the person who performs the task and the evaluator because the evaluation can be subjective and the evaluator must give a concrete instruction for re-execution.
- **Flexibility control**
  When the granularity of a human task is coarse, a person who is responsible for the task can efficiently perform his task. But deviation from the requirements of the task is prone to occur due to the flexibility. On the other hand, we can reduce deviation by defining fine tasks. In that case, the efficiency often declines. Therefore we need to control flexibility by configuring granularity of tasks.
- **Guideline**
  When the detail of the procedure of a task is not defined, showing guidelines can be a help for reducing deviation from the implicit requirements.
- **Clarify responsibility**
  More than one person or organization often involve in a task. If the task sometimes causes an exception, it is required to decompose the task in order to clarify the responsibility of people or organizations involved.
- **Reassignment** Based on the performance record of a person who is responsible for a task or changes of business rules, we often need to change the assignment of people to tasks. Therefore the operator needs to dynamically configure the assingment or invoke a composite Web service which decides the assignment.

3.4 Monitoring

An operator often needs to obtain and aggregate information of instances of a composite Web service. However, the existing standard framework, such as WS-BPEL, does not provide enough functions for monitoring. Therefore Service Supervision can help the operator monitor execution states from the following aspects:

- **Aggregate state information**
  By aggregating information of states of tasks (e.g. assigned, running, suspended, etc.) over multiple running instances, operators can know load on each Web service or a person who is responsible for the tasks.
- **Macro** An operator often needs to perform a complex procedure which collects and aggregate information of running instances. Therefore we need allow the operator to define his/her own procedure.

3.5 Migration

Migrating to a new SOA system often confuses users because procedures and operations for the users sometimes completely change. The load on the users can be reduced by incremental migration as shown below:

- **Plug-in**
  When a user interface for humans which is used before the migration, plugging it into a composite Web service which are newly introduced allows people work in a practiced manner.
- **Partial reuse**
  People who work following a business process can be confused if the whole business process is update at once. Therefore, we sometimes need to begin with replacing a part of the current business process with that of new one.
- **Transfer**
  When the model of a composite Web service is updated, a running instance which is created from the old model is sometimes required to migrate to the new model. Therefore we have to be able to create a new instance from the new model and migrate the execution state of the instance of the old model to new one keeping consistency.

4 Service Supervision Patterns

The adaptation processes described in the previous section can be realized by combining execution control functions shown in Section 2. However, the much flexibility of the execution control functions sometimes makes it difficult to implement the adaptation processes because a designer usually does not have experience on design using execution control functions. Therefore we propose Service Supervision patterns, which consists of typical requirements and WS-BPEL processes using execution control functions as solutions.

Software patterns, including design patterns, have achieved a great success in design and analysis of software. Also in the area of workflow, workflow patterns

Table 2. Comparison among software patterns, workflow patterns, and Service Sueprvision patterns

	Software patterns	Workflow patterns	Service Supervision patterns
Problem	Requirements for analysis, development and optimization of software	Requirements for construction of business flow	Requirements for adaptation
Solution	Direction of design and development	Activity diagram	Composite service using execution control functions
Focus	Abstraction of system architecture and design	Analysis of business	Operation and lifecycle of composite services

have been proposed and they show the design of workflows which satisfy various requirements[8]. On the other hand, Service Supervision patterns give requirements for adaptation process as problems and composite Web services which satisfy the requirements by combining execution control functions as solutions. For example, the composite Web service which is shown in Fig. 2 and controls the document translation service can be seen as a pattern which monitors the execution and adds some processes by generalizing "count" and "setAddress".

Table 2 shows the comparison among software patterns, workflow patterns, and Service Sueprvision patterns we propose in this paper.

Service Supervision patterns are easy for designers of composite Web services to understand because the solutions are described in WS-BPEL processes. Moreover, we need little change to port them to the model of a composite Web service to be adapted.

A composite Web service defined in a Service Supervision pattern consists of the following elements:

- Control constructs and activities of WS-BPEL
- Execution control functions
- Template task

A composite Web service provided as a solution of Service Supervision patterns runs on the same execution engine as composite Web services to be adapted. The execution control functions are ones that introduced in Section 2. A template task is defined according to the required adaptation processes.

We describe each Service Supervision pattern below. Tasks labeled as T represent template tasks. We omit activities which define dataflow for the simplicity.

4.1 Trigger Patterns

Runtime adaptations of a composite Web service are triggered when some changes or events which require adaptation are detected. Such detection is performed (a) at a certain point in a composite Web service, (b) continuously, (c) on operator's request, or (d) when time-out of a task happens. The following patterns realize the triggers for adaptations.

Pattern 1: Synchronous Watch

- **Description.** The task set to the template task is executed at a certain point of the composite Web service to be adapted.
- **Implementation.** Set a breakpoint at the point to which some processes should be added and set the composite Web service of this pattern as a callback Web service.
- **Example.** Adding a process for validation of the result of a constituent service and an exception handling process.
- **Porting to model.** Insert tasks set to the template task into the point where the breakpoint is set.

Pattern 2: Continuous Watch

- **Description.** The task set to the template task is continuously executed during the execution of composite Web service to be adapted.
- **Implementation.** Execute all tasks of composite Web service to be adapted by *step* execution and execute the template task after each *step*.
- **Example.** Checking consistency of data handled by the composite Web services.
- **Porting to model.** To add a monitoring process to many points in a composite Web service seriously declines the performance. Therefore this pattern should be used to find the point where some monitoring is required before the model is changed.

Pattern 3: Asynchronous Watch

- **Description.** The task set to the template task is executed on request.
- **Implementation.** Start execution of the template task after receiving a request.
- **Example.** Reporting execution state of a composite Web service on the request by operator's request.
- **Porting to model.** Add an asynchronous *Receive*, the task set to the template task and *Reply*.

Pattern 4: Timeout

- **Description.** The task set to the template task is executed when a task does not finish in a certain period of time.
- **Implementation.** Execute the target task by *step* and finish the instance of this pattern by *terminate*, which is a WS-BPEL activity. If the specified period of time elapses before the target task finishes, *suspend* execution of the composite Web service and recover the target task by the task set to the template task.
- **Example.** When a service is temporarily available or a human task is taking too long, this pattern makes it possible to dynamically change services or assignment of people.
- **Porting to model.** Replace *step* with the target task as asynchronous invocation and put the composite Web service of this pattern instead of the target task.

4.2 Evaluation and Retry Patterns

When the result of a task is invalid or the quality of the result is not good enough, we need to retry the task until an appropriate result is obtained. We show the two following patterns for the validation/evaluation of the result and retry.

Pattern 5: Automatic Retry

- **Description.** This pattern assumes that validation and retry are automatically performed. After validating the result of a task, this pattern retries the task if needed. The composite Web service which changes the conditions of execution of the task is set to template task.

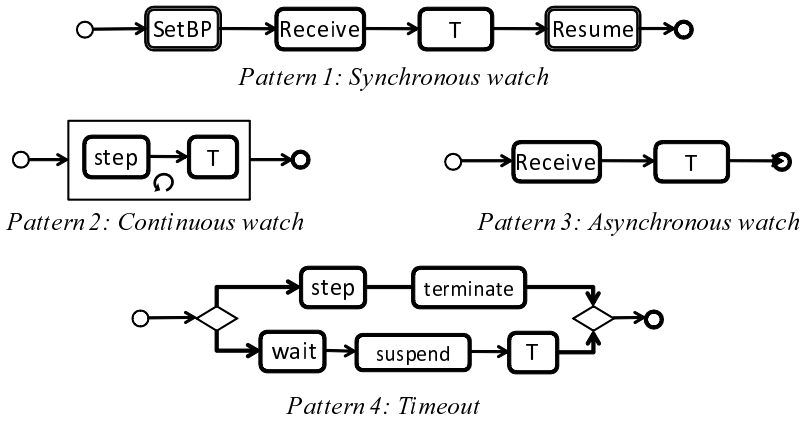


Fig. 5. Trigger patterns

- **Implementation.** Set execution point by *setEP*, retry the task by *step*, and change of the condition of execution at the template task in loop. To restore the execution state before retry, we introduced *getState* and *setState*.
- **Example.** This pattern enables us to switch a service to an alternative when execution of the service fails. This pattern also realizes the cycle of evaluation and change of parameters, which is shown as Program Supervision[16].
- **Porting to model.** Put the task to be retried and template task in loop and add activities which set states before retry.

Pattern 6: Human Evaluation

- **Description.** This pattern retries a task when the quality of the result of the task is not good enough. This pattern assumes that both the target task and the evaluation are performed by humans. Therefore this pattern allows people who are responsible for the tasks to communicate with each other by introducing a task for evaluation as a template task.
- **Implementation.** Instead of the task for changing conditions of execution in Automatic Retry pattern, put the task for evaluation and communication after *step* of the target task.
- **Example.** This pattern allows an evaluator to show the guideline for the task to a person who is responsible for the task even if the guideline was not defined when the model of composite Web service is designed.
- **Porting to model.** Put the task to be retried and the task set to template task in loop.

4.3 Patch Patterns

The following patterns are used to make up small defect keeping the most of initial behaviors.

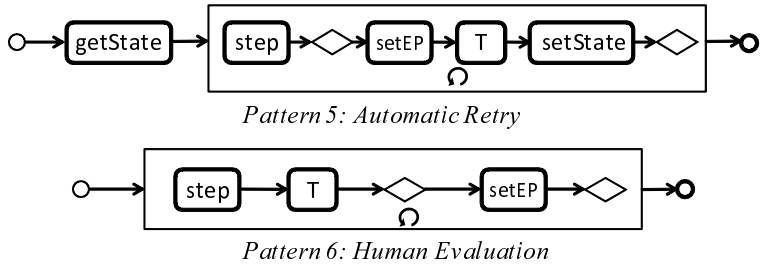


Fig. 6. Evaluation and retry patterns

Pattern 7: Add Alternative

- **Description.** This pattern adds a task which is an alternative of a task in a composite Web service when a given condition is satisfied.
- **Implementation.** Put the template task and the target task in conditional branches.
- **Example.** When a Web service often causes an exception under a certain condition, this pattern can be applied to temporarily delegate the task to humans.
- **Porting to model.** Replace the target task with the conditional branches defined in this pattern.

Pattern 8: Partial Execution

- **Description.** This pattern executes a part of an existing composite Web service.
- **Implementation.** *step* the tasks to be executed and *skip* other tasks.
- **Example.** This pattern realizes an incremental migration to a new composite Web service.
- **Porting to model.** Remove the tasks which are skipped by this pattern from the model of composite Web service.

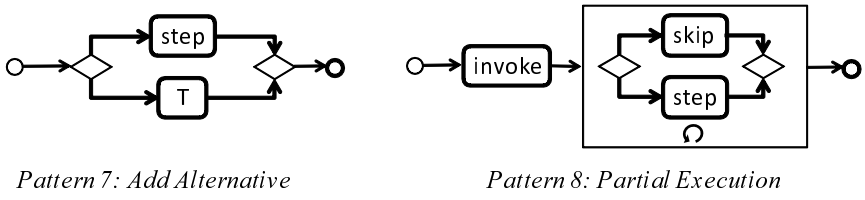


Fig. 7. Patch patterns

4.4 Granularity Control Patterns

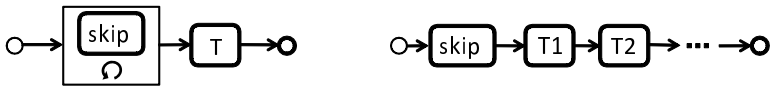
The following patterns compose or decompose tasks to control flexibility of human tasks.

Pattern 9: Compose

- **Description.** This pattern replaces consecutive tasks with one task which is equivalent to the consecutive tasks.
- **Implementation.** *skip* tasks defined in the model of a composite Web service and execute the task set to the template task.
- **Example.** This pattern is applied if the efficiency for a human task declines due to lack of flexibility.
- **Porting to model.** Replace consecutive tasks with the task set to the template task.

Pattern 10: Decompose

- **Description.** This pattern decomposes a task into some subtasks.
- **Implementation.** Execute the predefined subtasks and *skip* the task to be decomposed.
- **Example.** When a task is virtually executed by some people, this pattern is applied to clarify the responsibility of each person.
- **Porting to model.** Replace the target task with the subtasks set to the template tasks.



Pattern 9: Compose *Pattern 10: Decompose*

Fig. 8. Granularity control patterns

Table 3. Possible adaptations by Service Supervision patterns

Adaptation \ Patterns		Synchronous watch	Continuous watch	Asynchronous watch	Time out	Auto retry	Human evaluation	Add alternative	Partial execution	compose	decompose
Exception handling	Recovery	●	●		●	●					
	Alert	●	●	●	●						
	Avoid	●	●					●	●		●
	Enforcement			●	●		●				
Dynamic Change	Dynamic binding			●	●	●					
	On-the-fly composition			●			●	●			
Human Involvement	Negotiation	●			●		●				
	Control flexibility			●	●		●			●	●
	Guideline	●		●			●				
	Clarify responsibility			●	●		●				●
	Reassignment					●	●				
Monitoring	Aggregate state info	●	●	●							
	Macro	●	●	●							
Migration	Plug-in	●					●	●	●	●	
	Reuse	●						●	●		
	Transfer			●					●		

Table 3 shows adaptations described in Section 3 and Service Supervision patterns which can be used for each adaptation.

All adaptations are triggered by one of Trigger patterns. Using Trigger patterns, the operator can easily start or stop the adaptation processes. However, the adaptation processes have to be defined before they are applied. This is the reason the patterns do not work well for adaptations which require us to define an extreamly wide range of processes, such as on-the-fly composition and transfer, although the patterns can be frequently reused for rather simple adaptations such as exception handling.

5 Related Works

Software patterns, which describe typical problems and solutions in software development, have been expanded against the background of complexity of recent software development. The most well-known software patterns are design patterns[7] and they show means for system design based on object-oriented programming. On the other hand, van der Aalst et al. proposed workflow patterns[8], which show requirements for constructing business flows and activity diagrams as the solutions. The workflow patterns focus on analysis of business, excluding perspective of system implementation.

Similarly, Service Supervision patterns proposed in this paper also aims at reusing know-how about design. But Service Supervision patterns focus on adaptation processes which can be realized by execution control functions and there is no previous work on reuse related to composite Web service for adaptation as far as we know.

Several previous works have tried to change behaviors of a composite Web service without modifying the composite Web service. Most of them have adopted the concept of AOP (Aspect-oriented Programming).

Some works monitor the messages exchanged between services and modify them[3,4,5]. However, the works depend on their own descriptions. This leads to the cost of design when adaptation is ported to the model of the composite Web service.

AO4BPEL[6] enables us to insert processes described in BPEL into before or after an activity in an existing composite Web service as a pointcut. Therefore the processes defined for adaptation using AO4BPEL can easily be inserted into the model of a composite Web service. But some adaptation processes cannot be realized by the method because it does not provide execution control functions such as setting execution point. The authors also introduced some applications, but they are not comprehensively organized.

6 Conclusion

Service Supervision, which controls the behavior of running instances of composite Web services using execution control functions, allows us to flexibly adapt composite Web service to changes of the environment or business rules. This

makes constituent Web services of a composite Web services transparent to users and allows us to provide the composite Web services as a "cloud" service. However, the much flexibility of Service Supervision sometimes makes it difficult for the designer of composite Web services to design adaptation processes due to the absence of accumulated know-how. Moreover, it often costs to port the adaptation processes to the model of composite Web service to be adapted.

Therefore we proposed Service Supervision patterns, which provide typical requirements for adaptation and reusable WS-BPEL processes which implements the adaptation. The contributions of this work are as follows:

- We organized various adaptation processes based on some previous works and explained how they can be implemented using control execution functions.
- We extracted typical execution controls for adaptation processes and showed how to port them to the model of a composite Web service.

The Service Supervision patterns can reduce the load on the designer who implements adaptation processes or ports them to the model.

In future work, it is required to investigate the effect on the performance of each pattern. We expect that the temporary adaptation is achieved by Service Supervision, and then it is ported to the model when the permanent demand of the adaptation becomes apparent. Therefore the investigation on the performance helps the operator decide when and how the adaptation should be ported to the model.

Acknowledgment

This work was supported by Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Ministry of Internal Affairs and Communications of Japan.

References

1. Business process execution language for web services (BPEL), version 1.1 (2003), `http://www.ibm.com/developerworks/library/ws-bpel/`
2. Tanaka, M., Ishida, T., Murakami, Y., Morimoto, S.: Service supervision: Coordinating web services in open environment. In: IEEE International Conference on Web Services, ICWS 2009 (2009)
3. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: 17th International World Wide Web Conference (WWW 2008), pp. 815–824 (2008)
4. Mosincat, A., Binder, W.: Transparent runtime adaptability for BPEL processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 241–255. Springer, Heidelberg (2008)
5. Baresi, L., Guinea, S., Plebani, P.: Policies and aspects for the supervision of BPEL processes. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 340–354. Springer, Heidelberg (2007)

6. Charfi, A., Mezini, M.: AO4BPEL: An aspect-oriented extension to BPEL. World Wide Web 10(3), 309–344 (2007)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading (1995)
8. van der Aalst, W.M.P., Hofstede, A.t., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases 14(3), 5–51 (2003)
9. Ishida, T.: Language Grid: An infrastructure for intercultural collaboration. In: IEEE/IPSJ Symposium on Applications and the Internet (SAINT 2006), pp. 96–100 (2006)
10. Web services choreography description language version 1.0 (2005), http://www.w3.org/TR/ws-cdl-10/
11. Kammer, P.J., Bolcer, G.A., Taylor, R.N., Hitomi, A.S., Bergman, M.: Techniques for supporting dynamic and adaptive workflow. Computer Supported Cooperative Work (CSCW) 9(3), 269–292 (2000)
12. Müller, R., Greiner, U., Rahm, E.: Agentwork: a workflow system supporting rule-based workflow adaptation. Data and Knowledge Engineering 51(2), 223–256 (2004)
13. van der Aalst, W.M.P., Basten, T., Verbeek, H.M.W., Verkoulen, P.A.C., Voorhoeve, M.: Adaptive workflow. on the interplay between flexibility and support. In: Proceedings of the first International Conference on Enterprise Information Systems, pp. 353–360 (1999)
14. Han, Y., Sheth, A., Bussler, C.: A taxonomy of adaptive workflow management. In: ACM Conference on Computer Supported Cooperative Work, CSCW 1998 (1998)
15. WS-BPEL extension for people (bpel4people), version 1.0 (2007), http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/
16. Thonnat, M., Clement, V., Elst, J.v.d.: Supervision of perception tasks for autonomous systems: The OCAPI approach. In: 3rd Annual Conference of AI, Simulation, and Planning in High Autonomy Systems, pp. 210–217 (1992)