Service Supervision for Service-oriented Collective Intelligence

Masahiro Tanaka*, Yohei Murakami*, Donghui Lin*, Toru Ishida†*

*Language Grid Project
National Institute of Information and Communications Technology,
3-5 Hikaridai, Seika-cho, Kyoto, Japan
{*mtnk, yohei, lindh*}@nict.go.jp
†*Department of Social Informatics, Kyoto University*
Yoshida-honmachi, Sakyo-ku, Kyoto 606-8501 Japan
ishida@i.kyoto-u.ac.jp

Abstract—**Service-oriented collective intelligence, which creates new value by combining services provided by various organizations via services computing technologies, has been gaining in importance with the development of services computing technologies. Because collective intelligence needs many participants, it is crucial to build a framework where a wide variety of policies of service providers are satisfied. In this paper, we propose an architecture which handles a comprehensive process of service selection, adaptation, and coordination to satisfy policies of service providers. First the system selects services, and then adapts the services to the given policies if any of available services cannot satisfy the policies. To achieve this, we formalized this problem as an extension of constraint satisfaction problem and showed a solution. Moreover, the system often needs to force a composite service to follow protocols given by service providers. Therefore we proposed a method which uses meta-level control functions for composite services in order to change order of service execution.**

Keywords-**Service Supervision; Web service; coordination; collective intelligence;**

I. INTRODUCTION

Various programs and data have become available as Web services with the development of services computing technologies. To realize service-oriented collective intelligence, which integrates the services provided by various organizations, it is required to build a new framework satisfying the service providers' policies. Because collective intelligence needs many participants and it is crucial to convince service providers that their policy will be certainly satisfied on a framework where their services are deployed.

Service providers may have a wide variety of policies, such as limitation of transferred data, constraints on combinations of services and so on. From this aspect, service-oriented collective intelligence is quite different with traditional collective knowledge which relies on single license e.g. Wikipedia. For example, the Language Grid[1] is a platform for service-oriented collective intelligence, which allows to combine various language services such as machine translators and dictionaries. On the Language Grid, the service providers can set permission to use their service for each user and limit amount of transferred data.

To give service providers more incentive to join a platform of service-oriented collective intelligence, a composite service which combines multiple services needs to be executed satisfying policies of all the service providers concerned.

Some previous works have proposed methods to find a combination of services which satisfy given requirements. They select appropriate services from a set of services which are functionally equivalent[2], [3]. However, the previous works are not enough from the following aspects because they focused on finding a combination of services which give the best QoS, not on satisfying policies of service providers.

- There is often no combination of services which satisfy policies of all service providers concerned because the number of services is limited in reality.
- Policies of service provider may include constraints on execution of a composite service, which cannot be resolved by service selection.

To solve the problems, we propose an architecture which performs a comprehensive process of service selection, adaptation and coordination. First it tries to find a combination of services which satisfy policies of service providers in the similar fashion to the previous works. But it adapts the services to given policies by changing attributes of the services if there is no service required to satisfy the policies. Moreover, using meta-level control functions for composite services, it monitors and changes execution state of composite services in order to satisfy constraints on execution.

Some dynamic adaptation methods for composite services have been proposed. Most of them adopt aspect-oriented programming (AOP)[4] or a proxy between the composite service execution engine and invoked services[5]. The method proposed in [6] uses meta-level control functions to control execution of composite service. However, these works focus on adaptation without changing model of composite services. Therefore they do not handle a comprehensive process including service selection and coordination. We call the architecture Service Supervision because it continuously monitors or manages execution of services, and controls the

behavior of services.

The rest of this paper is organized as follows. In Section II, we show a scenario which describes our research problems and goals of the architecture we propose. From Section III to Section V, we explain three layers in our architecture. Each of layer is responsible for service selection, adaptation, and coordination respectively. In Section VI, we briefly discuss the scalability of the proposed architecture. After we introduce related works in Section VII, we conclude this paper in Section VIII.

II. DESIGN GOAL

In this section, we first describe a scenario of a composite service which runs on a service-oriented collective intelligence platform. Then we show an overview of the architecture we propose.

A. Scenario

We take a composite service for translation deployed on the Language Grid[1] as an example. The composite service combines a morphological analyzer, a machine translator, and technical term dictionaries. This service improves translation quality of technical documents by translating technical terms in the given sentences using the technical term dictionaries, not the machine translator.

Figure 1 shows the overview of the composite service. A square which contains a circle represents a service invocation activity. First the given sentences are divided into morphemes by the morphological analyzer. Next dictionaries find technical terms which consist of the morphemes and return the translation of the technical terms. Finally the translator translates the whole sentences.
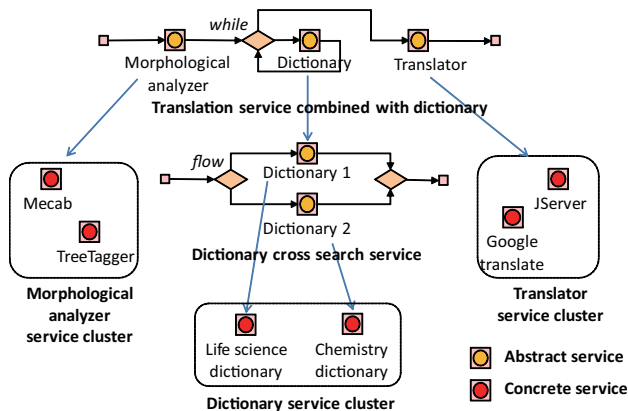

Figure 1. A composite service for translation

As shown in Fig. 1, we assume that a composite service is defined in a workflow description language such as WS-BPEL[7]. In a composite service, the constituent services define only the interface and are not bound to any endpoint. We refer to such a service as an *abstract service*. For example, on the Language Grid, an abstract service is defined for each service type such as translators and dictionaries. Endpoints for the services are determined when the composite service is invoked. A service to which an endpoint is bound is called a *concrete service*.

On the Language Grid, all the information on concrete services and composite services are managed by Language Grid Service Manager[1] shown in Fig. 2. Currently more than 70 concrete services and 12 composite services are registered and various information of the services including lisence and WSDL are managed on Language Grid Service Manager.

Service Name	Service Type	Languages (in Language Code)	Provider	Status
Google Translate	TRANSLATION	(sq<->ar), (sq<->bg), (sq<->zh-CN), (...	Google, Inc.	Run
Hamamatsu City Documents Related to Foreign Students	PARALLEL TEXT	(ja<->en), (ja<->pt), (ja<->es), (en<...	Language Grid Operation Ce...	Run
HTML Text Extractor	OTHER	(*)	Language Infrastructure Gr...	Run
ICTCLAS	MORPHOLOGICAL ANALYSIS	(zh)	NLP Group, Institute of Co...	Run
J-Server (Kyoto-U)	TRANSLATION	(ja<->en), (ja<->ko), (ja<->zh)	Ishida and Matsubara Labor...	Run

Figure 2. A list of services registered on Service Manager.

A set of concrete services which is bound to an abstract service is called a service cluster. In Fig. 1, there are two concrete services in each service cluster. In our example, we bind one of the two concrete services to the morphological analyzer and the machine translator. For the dictionary, we first bind a composite service for cross search and then bind two concrete services to abstract services in the cross search composite service. The Language Grid allows users to specify the hierarchical binding by describing the binding in SOAP header of the request message.

We show the process of execution of this composite service below. First we select concrete services which satisfy the user's request and service providers' policies. In our example, we assume that the request is translation from Japanese to English and that the user specifies the life science dictionary and the chemistry dictionary for the two abstract dictionary services.

When receiving the request, the system selects concrete services for the rest of the abstract services. Suppose it first tries Mecab, which is a morphological analyzer for Japanese, and Google Translate, which provides translation from Japanese to English. However, the combination may have the following three problems.

The first problem is constraints on a combination of services. Assume the provider of the chemistry dictionary prohibits use of its service with Google Translate. In such a case, the system needs to select JServer, which is another

---

[1]http://langrid.org/service_manager/

machine translator and supports Japanese-English translation.

The second problem is constraints on execution of a constituent service. The provider of the translator JServer may limits the length of the input to 1000 characters in order to reduce server load. In this case, we need to introduce an adaptation process which divides a long input into sentences before translation.

The third problem is constraints on execution of a composite service. Assume that both the life science dictionary and the chemistry dictionary are provided by the same provider and that the provider prohibits concurrent access to the two services to prevent a user from giving too much load. In this case, two dictionaries which are defined to be executed in parallel should be controlled to be executed sequentially.

B. Architecture Overview

We propose an architecture to solve the problems described in the previous scenario.

Our architecture consists of three layers which are connected to each other and solve the three types of constraints respectively. Figure 3 shows the overview of the architecture.

The roles of layers are as follows:

Selection Layer
This layer selects concrete services from service clusters to satisfy constraints on combination of services. The constraints are retrieved from service providers' policies and stored in the repository in this layer. If no combination which satisfies all constraints is found, this layer delegates control to Adaptation Layer.

Adaptation Layer
This layer adapts a constituent service in a composite service to the given constraints by changing attributes of the service. The adaptation is achieved by weaving processes before/after invocation of the target service based on AOP. More than one adaptation process can be combined by the planner in this layer. If the constraints are not satisfied in this layer, this layer delegates control to Coordination Layer.

Coordination Layer
This layer controls the order of services execution in a composite service. Using meta-level control functions for composite services proposed in [6], this layer forces execution of a composite service to follow a protocol defined by the given choreography in WS-CDL. If this layer cannot satisfy the constraints, execution of the composite service fails.

Based on this architecture, execution of a composite service proceeds as described below.

First a user sends a request to the composite service execution engine. The user can specify concrete services which are bound to abstract services if he/she needs.

The composite service execution engine invokes the service selection process before invocation of each service.

It continuously needs to confirm that the constraints are satisfied because the state of services is changing and the output of a service may violate the constraints.

Selection Layer retrieves information from Service Repository and Service Provider Directory and stores the information in it. According to the information, it also finds a combination of services which satisfy constraints in response to the request from composite service execution engine.

Adaptation Layer stores various adaptation processes, such as dividing an input / merging outputs, and adding an authentication process. It decides which adaptation processes should be applied and the order of adaptation processes. After it finds a set of adaptation processes to be applied, the adaptation processes are registered to AOP Manager. AOP Manager is an extension of composite service execution engine. It monitors execution of a composite service and "weaves" adaptation process.

Coordination Layer has a repository of choreography, which is created from the model of composite services and service providers' policies on execution. After performing an execution control function of meta-level controller, it immediately finishes and returns control to the composite service execution engine.

III. Selection Layer

Selection Layer finds a combination of concrete services which satisfy constraints by service providers' policies.

In this paper, we formalize selection of services as constraint satisfaction problem (CSP). Given service profiles, a set of available concrete services and policies of service providers, the selection of services are defined as follows:

Variable
A variable represents an abstract service in a composite service. If a composite service invokes other composite services, the set of variables contains variables of abstract services in all composite services.

Domain
A domain represents a set of concrete services in a service cluster. A domain is described as $D = \{x_1, x_2, ...\}$, where $x_1, x_2, ...$ are concrete services. Attributes of concrete services are described as $x_i.attr_1, x_i.attr_2, ...$.

Constraints
Constraints represent policies of service providers and requirements of a user. Constraints are described as predicates based on attributes of concrete services or input/output of services.

Domains are retrieved from Service Repository according to the service type of each abstract service. Constraints are obtained by transforming service providers' information.

In the initial state, the set of variable contains variables which correspond to all abstract services. However, if it is ensured that a service will not be execute due to a conditional branch during execution, the variable corresponding to the service is removed.
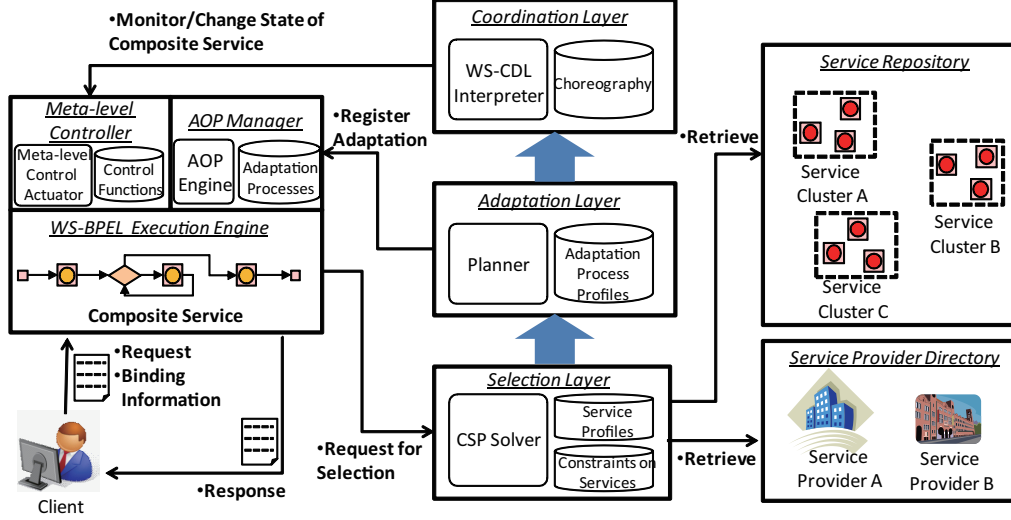
Figure 3. Layers of Service Supervision system.

We show an example of the definition for the translation composite service described in the previous section. As the composite services consist of a morphological analyzer, two dictionaries and a translator, variables X_{ma}, X_{dic1}, X_{dic2}, X_{trans} are defined. The variables have the corresponding domains as follows:

$$
\begin{aligned}
D_{ma} &= \{x_{ma\_mecab}, x_{ma\_treetagger}\} \\
D_{dic1} &= \{x_{dic\_life}, x_{dic\_chem}\} \\
D_{dic2} &= \{x_{dic\_life}, x_{dic\_chem}\} \\
D_{trans} &= \{x_{trans\_google}, x_{trans\_jserver}\}
\end{aligned}
$$

We also show some of attributes of concrete services below. The following definition represents that a morphological analyzer Mecab accepts only Japanese, and TreeTagger accepts English, German, French, Italian and so on. For a translator JServer, the supported language pairs and limitation of input sentences are defined.

$x_{ma\_mecab}.sourceLang = \{Japanese\}$
$x_{ma\_treetagger}.sourceLang =$
$\quad \{English, French, German, Italian, ...\}$
$x_{trans\_jserver}.langPair =$
$\quad \{(Japanese\ to\ English)(English\ to\ Japanese)...\}$
$x_{trans\_jserver}.maxInputLength = 1000 ...$

The constraints are shown below.

$C = \{$
$\quad C1 : X_{trans}.maxInputLength >$
$\quad\quad\quad\quad length(Request.input),$
$\quad C2 : exclusive(x_{trans\_goole}, x_{dic\_chem}),$
$\quad C3 : parallel\_prohibited(x_{dic\_life}, x_{dic\_chem}) ... \}$

$C1$ represents requirements based on the process of the composite service and shows that the limitation of input length of a translator must be longer than input sentences. $C2$ prohibits the combination of GoogleTranslate and the life science dictionary. $C3$ prohibits parallel execution of the chemistry dictionary and the chemistry dictionary.

In our example, we assume the following request from a user. The request consists of sentences to be translated, the source language, and the target language.

$Request = \{$
$\quad input$:"Sentence to be translated"
$\quad\quad\quad\quad$ (actually 1500 characters in Japanese),
$\quad source\_language : Japanese,$
$\quad target\_language : English\}$

User can specify the concrete services bound to any abstract services in a composite service when he/she invokes the composite service. For the abstract services which the user does not specify concrete services bound to, the process in Selection Layer selects concrete services. A user usually specifies concrete services when he/she knows how attributes of concrete services affect the result. Otherwise, a user delegates the selection to the system.

In our example, the user specifies the life science dictionary and the chemistry dictionary for two dictionaries in order to translate sentences in the area of biochemistry. On the other hand, the user does not specify concrete services for the morphological analyzer and the translator because he/she does not know which service can get a better result.

The CSP which formalizes the above conditions has no solution. Therefore it is impossible to satisfy the user's requirements and service providers' policies using the available services. The combinations which have the least number of violations are as follows. The former violates $C1$ and $C2$,

and the latter violates $C2$, $C3$.

$$(X_{ma}, X_{dic1}, X_{dic2}, X_{trans}) =$$
$$(x_{ma\_mecab}, x_{dic\_life}, x_{dic\_chem}, x_{trans\_jserver})$$
$$(x_{ma\_mecab}, x_{dic\_life}, x_{dic\_chem}, x_{trans\_google})\}$$

The attributes of services which cause violation are changed in Adaptation Layer in order to resolve the violation. To determine which service should be adapted, we extended CSP based on the ideas of Open CSP[8] and Partial CSP[9].

In Open CSP, a new value for a domain is obtained when any combination of existing values cannot satisfy constraints. The paper shows that the domain of a variable which is located in the deepest leaf of a search tree of backtrack search certainly needs to be extended to resolve violations. Therefore, we need to adapt one of concrete services which correspond to such a variable.

Moreover, to determine which concrete service should be adapted, we also need to know a combination of services which gives the least violation count. Therefore we applied the idea of Partial CSP, which finds a solution which gives the least violation when the problem is over-constrained.

Figure 4 shows the algorithm which applies the ideas to the problem of finding a combination of services and determining a service to be adapted.

The algorithm begins with depth-first search (line 1-22). The index of the deepest variable in the search tree is recorded as k. During backtrack search, the least violation count and the combination of services which gives the count are also record as M and $currentSelection$ respectively.

Then it extends the domain D_k by giving k and $currentSelection$ to Adaptation Layer and searches for solution again (line 23-30).

IV. ADAPTATION LAYER

I If Selection Layer cannot find a combination of services which satisfy all constraints, Adaptation Layer adapts a service to constraints by changing attributes of services. The adapted service can be considered as a new service in Selection Layer.

To find appropriate adaptation processes, profiles of adaptation processes are stored in Adaptation Layer. Figure 5 shows an example of the profile of an adaptation process. The adaptation process first divides an input string into sentences before a service is invoked, and then invokes the target service for each sentence. Finally it merges the results after the target services finishes. This process is used to adapt a translation service which has the limit of input length.

An adaptation process is identified by the name described in "adaptation" tag. Preconditions and effects are described in a profile. In the description of preconditions and effects, "Request" represents a request given by the user and "Service" represents a service to which this adaptation process is applied.

```
function: searchCombination(X, D, C)
Inputs:
    X: Variables ({X1, ..., Xn}), D: Domains ({D1, ..., Dn}),
    C: Constraints

 1:  i←1, k←1, M←∞
 2:  while(i > 0)
 3:    if all values in Di are checked
 4:      reset xi,  i←i-1
 5:    else
 6:      xi←next value of Di
 7:      if ({x1, ..., xi}) satisfies C
 8:        k← max(k, i+1)
 9:      end if
10:      i←i+1
11:      if i > n
12:        if ViolationCount({x1, ..., xn}) = 0
13:          return {x1, ..., xn}
14:        end if
15:        if N > ViolationCount({x1, ..., xn})
16:          N← ViolationCount({x1, ..., xn})
17:          currentSelection ←{x1, ..., xn}
18:        end if
19:        i←i+1
20:      end if
21:    end if
22:  end while
23:  xk'← findAdaptation(k, currentSelection)
24:  if xk' is failure
25:    return failure
26:  end if
27:  Dk ← Dk ∪ {xk'}
28:  Change order of variables (Move Xk
29:      to first  as X1)
30:  return searchCombination(X, D, C);
```

Figure 4.   Algorithm for finding a combination of services

The profile in Fig. 5 shows that the input must consist of more than one sentence and the longest sentence must be shorter than the limit of input length of the target service. This also shows the adaptation process removes the limit of input length if applied.

As shown in Fig. 3, profiles of adaptation processes are stored in Adaptation Layer, but the implementation is in AOP manager, which is an extension of the composite service execution engine. It is difficult for the standard framework to change the model of a composite service and to deploy the new model if adaptation processes applied are frequently changed. This is the reason the adaptation is realized by using AOP techniques.

Various adaptation methods for composite services using AOP have been proposed in previous works(e.g. [4], [5]). One of the most flexible methods[5] weaves a process described in WS-BPEL into the target composite service. We assume we adopt this method for our architecture and show the implementation of adaptation process in Fig. 6.

The adaptation process divides an input and merges the

```
<adaptation name="DivideAndLoop">
  <precondition>
    <expression language="javascript">
      numOfSentences(Request.input) > 1
      && Service.maxInputLength(
        length(getLongestSentence (
          Request.input)))
    </expression>
  </precondition>
  <effect>
    <expression language="javascript">
      Service.maxInputLength = INF
    </expression>
  </effect >
</adaptation>
```
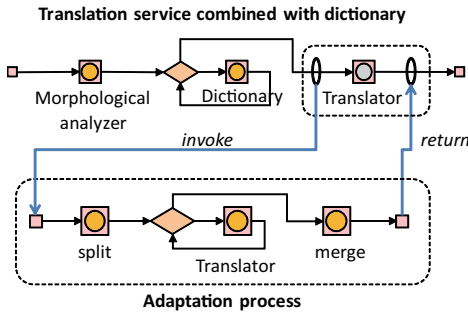
Figure 5.   Profile of an adaptation process.

Figure 6.   Adaptation process for division and merging

```
function: findAdaptation(k, currentSelection)
Inputs:
 k: Index of target service
 initialSelection: A combination of services which
   give the least violation count without adaptation

 1:  currentSelection, tempSelection ←initialSelection
 2:  leastViolation ←ViolationCount(currentSelection)
 3:  for MAX times
 4:    PossibleAdaptation = {a | a ∈ Set of adaptation
 5:     processes and all preconditions of a are satisfied}
 6:    for each a in PossibleAdaptation
 7:     nextSelection ←Apply a to k th service  of
 8:             currentSelection
 9:    if(leastViolation > ViolationCount(nextSelection))
10:       tempSelection ←nextSelection
11:    end if
12:   end for
13:   if tempSelection = initialSelection
14:     coordinateOrder()
15:     return failure
16:   end if
17:   if tempSelection = currentSelection
18:      return k th service of currentSelection
19:   end if
20:   currentSelection ←tempSelection
    end for
```

Figure 7.   Search algorithm for adaption.

output before/after the target service is invoked. The invocation of target service is located in loop in the adaptation process. The invocation which is originally defined in a composite service is skipped.

Using adaptation method described above, more than one adaptation process can be applied to one target service. We search for a sequence of adaptation processes by hill climbing taking the count of violation as the evaluation value. Figure 7 shows the algorithm for selecting adaptation processes. This algorithm is called by the algorithm in Fig. 4.

The algorithm takes the index of the target service and a combination of services which gives the least count of violation as an input. It tries to apply various adaptation processes in order to find a sequence of adaptation processes which can reduce the count of violations. If this algorithm cannot reduce the violation count, it invokes Coordination Layer and returns failure to Selection Layer (line 14-15). The service to which some adaptation processes are applied is returned to Selection Layer and added to the domain in CSP (line 18).

We refer to $x_{trans\_jserver}$ which the adaptation process in Fig. 7 is applied to as $a_{div\_merge}(x_{trans\_jserver})$. The domain of translation services is extended as follows:

$$D_{trans} = \{x_{trans\_google}, x_{trans\_jserver}, \\ a_{div\_merge}(x_{trans\_jserver})\}$$

We can obtain the following combination whose count of violations is reduced to 1 with the adapted service.

$$(X_{ma}, X_{dic1}, X_{dic2}, X_{trans}) = \{ \\ (x_{ma\_mecab}, x_{dic\_life}, x_{dic\_chem}, a_{div\_merge}(x_{trans\_jserver}))$$

V. Coordination Layer

Some service providers have policies about execution, e.g. constraints on order of service execution. Such constraints cannot be solved by adaptation of a service. Therefore we control the order of service execution in Coordination Layer.

The standard framework for composite services such as WS-BPEL does not provide a method for changing order of service execution at runtime. This is the reason we adopt meta-level control functions for composite services proposed in [6]. Table I shows some of the meta-level control functions. Using the functions, we can obtain and change state of a running composite service. The functions are implemented by modifying the composite service execution engine.

API	Effect
suspend	Suspend an activity whose state is 'Ready'. The state of the suspended activity will be changed to 'Suspended'
resume	Resume an activity whose state is 'Suspended'. The state of the suspended activity will be changed to 'Running'.
getProcessState	Get the current state of all activities in a process of the composite service.
terminateProcess	Terminate the process of the composite service.

The constraints on order of service execution can be defined by a choreography description language such as WS-CDL. We can check if an order of service execution satisfies the constraint or not using WS-CDL interpreter. Figure 8 shows the procedure for checking the order of service execution.

```
Function coordinateOrder()

 1:  choreography ← Protocol of service execution
 2:     based on model of composite service and
 3:     policies of service providers
 4:  currentState ← getProcessState()
 5:  queue ← getReadyActivity(currentState)
 6:         ⊔ getSuspendedActivity(currentState)
 7:  for each act in queue
 8:    if accept(choreography, currentState, act)
 9:      resume(act)
10:      return
11:    else
12:      suspend(act)
13:    end if
14:  end for
15:  if getRunningActivity(currentState) is empty
16:    terminateProcess()
17:  end if
```

Figure 8. Procedure for coordinating services.

We define states for an activity in a composite service. 'Ready' represents that service is ready to be executed. 'Suspended' represents that the service is suspended after it once becomes 'Ready'. 'Running' represents that the service is being executed and 'Finished' represents that the execution of the service finished. The composite service execution engine sets the state of activities as execution of a composite service proceeds.

The algorithm in Fig. 8 first gets states of all activities in the composite service. Next it puts activities whose states are 'Ready' or 'Suspended' into a queue. Then it checks each activity can be accepted by the given choreography using the WS-CDL interpreter. If an activity is accepted, the activity is invoked. Otherwise, the state of the activity is changed to/keeps being 'Suspended'.

If all activities in the queue are in 'Suspended' and there is no 'Running' activity, execution of the composite service is terminated because no activity can be executed and the state will not be changed.

In our example, concurrent access to the chemistry dictionary and the life science dictionary is prohibited. However, the composite service does not define the order of execution of the two services. Therefore the system monitors the state of execution and prohibits invocation of one of the dictionary services while another one is running.

VI. DISCUSSION

In this section, we discuss the characteristics and the operation of the architecture proposed in this paper.

The algorithm shown in Fig. 4 for Service Selection Layer performs simple depth first search for the simplicity of description. This leads the computational complexity of the process to $O(n^b)$ where n is the number of abstract services and b is the size of service clusters. This can be unacceptable for some applications.

In [9], however, more efficient algorithms for Partial CSP such as extensions of branch and bound method and back-marking method are proposed. The ideas of these methods can be easily applied to the algorithm 4. For example, we can introduce branch and bound method by back-tracking when the current number of violations exceeds the least recorded number of violations. This can generally make this algorithm more efficient.

The procedure of seaching for adaptaiton processes proposed in this paper is not complete. First a combination of concrete services which gives the least violation count cannot always reach the best solution. Other combination may give less violation count by applying adaptation processes. Secondly, search by hill-climbing can reach local minimum.

However, the proposed architecture and ideas of Open CSP and Partial CSP are not limited to the search methods we showed in this paper. We can easily introduce complete algorithms by revising algorithms in Fig. 4 and Fig. 7 in order to keep completeness. When adopting the architecture, we need to choose appropriate methods depending on the required realtime property of applications.

VII. RELATED WORKS

In the architecture proposed in this work, service selection, adaptation and coordination are performed during execution of composite service.

Many previous works have proposed methods for service selection. For example, the method proposed in [2] focuses on finding a combination of services which gives the best QoS. The method proposed in [3] select services considering interfaces of services in addition to QoS.

These works assume that vast amount of services are stored in a service cluster. In reality, however, the number of services which have equivalent functions is limited. That is

the reason the previous works often cannot find a combination of services. Moreover, to handle the policies of service providers, we need not only finding a static combination of services but also dynamic adaptation and meta-level control of composite services.

Also in the area of dynamic adaptation, there have been some previous works. Most of the works can be classified into three types: weaving a new process based on AOP (Aspect-oriented Programming), using a proxy to monitor/change messages exchanged between a composite service and invoked services, and transforming the model of a composite service based on definition of additional processes.

AO4BPEL[5] is one of the framework for realizing AOP of composite services. It allows a user to define a pointcut in a WS-BPEL process and weave a process described in WS-BPEL as an advice. This can add processes for adaptation without changing the model of a composite service.

For service-oriented collective intelligence, however, it is required to satisfy various policies of service providers. This needs a comprehensive process of service selection, adaptation and coordination. AOP is suitable for adaptation as described in the previous section, but it is not flexible enough to coordination such as controlling order of service execution.

The work proposed in [4] adopts a framework using a proxy. It checks if messages exchanged among a composite services and the constituent services satisfy the given conditions when the composite service execution engine invokes the constituent services. If any of conditions is not satisfied, it performs some recovering processes, retries invocation, or changes the service to an alternative. But this focuses on adaptation of single service and does not deal with policies of all service provider concerned.

The methods which transform the model of composite services such as [10] does not need to modify the composite service execution engine. It suits to adding exception handling processes, but the composite service needs to be deployed if the adaptation process changed.

Meta-level control functions for Web services are proposed in [6]. The work provides more flexible controls than other works described above. But the flexibility often gives too much load on operators of the platform. The architecture proposed in this work partially adopted the work focusing on satisfying constraints on execution.

VIII. Conclusion

In this paper, we proposed an architecture which performs a comprehensive process for service selection, adaptation and coordination. The architecture is designed to satisfy policies of service providers.

The contributions of this paper are as follows:

- We proposed a method for service selection and adaptation in an integrated fashion by formalizing the problem

as an extension of constraint satisfaction problem.
- We showed a framework which controls order of service execution using meta-level control functions for composite service.

To realize service-oriented collective intelligence, it is required to satisfy various policies of service providers. This paper is the first work which handles a comprehensive process of service selection, adaptation and coordination from the aspect.

In future work, we will apply the proposed architecture to a real environment, such as the Language Grid, where many service providers have joined.

References

[1] T. Ishida, "Language Grid: An infrastructure for intercultural collaboration," in *IEEE/IPSJ Symposium on Applications and the Internet (SAINT-06)*, 2006, pp. 96–100.

[2] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, pp. 311–327, 2004.

[3] A. B. Hassine, S. Matsubara, and T. Ishida, "A constraint-based approach to horizontal web service composition," in *the 5th International Semantic Web Conference (ISWC 2006)*, vol. 4273, 2006, pp. 130–143.

[4] L. Baresi, S. Guinea, and P. Plebani, "Policies and aspects for the supervision of BPEL processes," in *the 19th International Conference on Advanced Information Systems Engineering (CaiSE07)*, 2007, pp. 340–354.

[5] A. Charfi and M. Mezini, "AO4BPEL: An aspect-oriented extension to bpel," *World Wide Web*, vol. 10, no. 3, pp. 309–344, 2007.

[6] M. Tanaka, T. Ishida, Y. Murakami, and S. Morimoto, "Service supervision: Coordinating web services in open environment," in *IEEE International Conference on Web Services (ICWS-09)*, 2009, pp. 238–245.

[7] "Business process execution language for web services (BPEL), version 1.1." http://www.ibm.com/developerworks/library/ws-bpel/, 2003.

[8] B. Faltings and S. Macho-Gonzalez, "Open constraint programming," *Artificial Intelligence*, vol. 161, no. 1-2, pp. 181–208, 2005.

[9] E. C. Freuder and R. J. Wallace, "Partial constraint satisfaction," *Artificial Intelligence*, vol. 58, no. 1-3, pp. 21–70, 1992.

[10] A. Mosincat and W. Binder, "Transparent runtime adaptability for bpel processes," in *the 6th International Conference on Service Oriented Computing (ICSOC 08)*, 2008, pp. 241–255.