2012 IEEE Ninth International Conference on Services Computing

A Service Binding Framework for Open Environment

Masahiro Tanaka, Yohei Murakami
National Institute of Information and Communications Technology (NICT)
3-5 Hikaridai, Seika-cho, Kyoto, Japan
{mtnk, yohei}@nict.go.jp

Donghui Lin, Toru Ishida
Department of Social Informatics, Kyoto University
Kyoto 606-8501 JAPAN
{lindh, ishida}@i.kyoto-u.ac.jp

***Abstract*—A service invocation in a composite service is usually defined by its interface for abstraction. Once the interface is standardized and many services which have the same interface become available, a user can bind preferable ones to the composite service only by setting endpoints. Since new services become available day by day in open environment, it is crucial to automatically construct the bindings according to user's requirements. However, runtime binding construction often degrades the performance because searching a vast amount of possible bindings is time-consuming. Moreover, bindings need to be modified even during execution of a composite service when some adaptation process is applied. In this paper, we proposed a framework for constructing bindings, which is based on ATMS (Assumption-based Truth Maintenance System). The framework effectively caches possibilities of bindings and can manage changes of bindings when a runtime adaptation process is applied. We implemented a prototype of the framework and confirmed that it works fast enough for a real scale service composition.**

***Keywords*-runtime adaptation; service composition; assumption-based trust maintenance system**

I. Introduction

With the mature of services computing technology, many organizations have published their services in open environment. One of the major advantages of services computing is that it allows us to easily combine services provided by various organizations. For example, the Language Grid[1] is an infrastructure for combining a wide range of language resources such as machine translators and dictionaries as services. Currently about 140 organizations in 18 countries including universities, research institutes and companies have joined and more than 170 services are available on the Language Grid. As interfaces of services on the Language Grid are standardized based on service types, a constituent service of a composite service can be defined as an *abstract service*, which defines only a service interface. A user determines *bindings* from the abstract services to *concrete services*, which are executable and provided by various organizations, only by setting endpoints. The composition style allows users to configure non-functional properties of a composite service by selecting concrete services according to the user's requirements.

On the other hand, it is often tough for a user to define bindings. A user just wants to designate concrete services which are appropriate for his purpose. But it is sometimes required to construct complex hierarchical bindings to combine the necessary concrete services because a composite service can be bound to an abstract service in another composite service. Moreover, a user may not be interested in selection of all services because non-functional properties of some services do not matter for the user. Take a composite service for translation as an example. Assume this service consists of a machine translator service, a technical term dictionary service, and a morphological analyzer service. Selection of concrete services for the machine translation and the dictionary can be crucial for a user because they affect translation quality. But non-functional properties of a morphological analyzer such as throughput can be less important when the user uses the service for a non real-time process. As this example shows, the system for service composition is required to automatically construct the whole bindings while containing concrete services which are explicitly designated by a user.

However, automatic binding construction at runtime declines the performance due to the following reasons.

- Searching for consistent bindings at runtime takes a long time due to a vast amount of possible bindings.
- Bindings need to be modified even during execution of a composite service when a runtime adaptation process is applied.

Many previous works have proposed methods for constructing bindings such as [2], [3]. The methods show how to find a combination of concrete services which satisfy constraints given by service providers, such as prohibition of combination with competitors' services. But some of them do not consider construction of hierarchical bindings. Some work including [4] introduced hierarchical planning, but the methods which construct bindings from scratch for every service composition are not efficient enough when availability of services partially changes. Some previous works such as [5], [6] using AOP (Aspect-oriented Programming) focus on adaptation for dynamic changes. But they do not consider composite services which are defined based on abstract services to improve reusability.

In this paper, we propose a service binding framework in

978-0-7695-4753-4/12 $26.00 © 2012 IEEE
DOI 10.1109/SCC.2012.74

IEEE computer society

order to solve the above problems. The model and algorithms introduced in our framework are based on the idea of ATMS (Assumption-based Truth Maintenance System)[7]. ATMS was originally proposed for hypothetical reasoning, and it efficiently maintains environments where results of reasoning hold. Based on the idea of ATMS, our framework can judge whether a composite service can be realized using a certain set of services. Moreover, our framework modifies bindings reusing the existing binding information when a runtime adaptation such as Service Supervision[8] changes the set of constituent services of a composite service.

The rest of this paper is organized as follows. In Section 2, we take the Language Grid and Language Grid Toolbox as examples of an infrastructure for service composition and its client software in order to explain our motivation. Next we propose a model and an algorithm for binding construction in Section 3. We also explain that how we can modify the bindings when a runtime adaptation is applied in Section 4. Then, in Section 5, we show experimental results using our prototype. After introducing some related works in Section 6, we conclude this paper in Section 7.

II. Motivating Scenario

In this section, we take services used for language translation as an example to explain our motivation. Figure 1 shows some available services and possible bindings. A service cluster means a set of services of the same type. A dashed arrow represents a possible binding.

Suppose client software first sends a request to the *translation selection* service. This service executes back-translation for two machine translation services. Back-translation first translates an input sentence into other language, and then translates the result into the original language. The quality of the translation result is estimated by calculating similarity between the input sentence and the result of back-translation. The service estimates that translation quality is better when the similarity is higher than another one.

A constituent service of a composite service is called an *abstract service* because it defines only interface and a *concrete service* or other composite service needs to be bound to the abstract service. The translation selection service consists of three abstract services: Translator 1, Translator 2, and Similarity Calculator. In Fig. 1, the *translation with dictionary* service is bound to Translator 1, and the *translation with parallel text* service is bound to Translator 2.

The *translation with dictionary* service translates technical terms in an input sentence using a dictionary service. The other parts are translated by a machine translation service. This improves the translation quality of documents in a special domain. The service also uses a morphological analyzer service in order to recognize border of words and part of speech of each word. The *translation with parallel text* service translates an input sentence based on statistic processing of parallel texts given as a training set. This requires a dependency parser service. In the hierarchical structure of bindings, a leaf must be a concrete service.

Service interfaces are standardized based on service types. Thus, a service bound to an abstract service can be easily replaced with other service only by changing endpoint addresses (URLs) if their types are same. This allows a user to flexibly combine services which have various non-functional properties.

Since hierarchical bindings shown in Fig. 1 can be too complex for user to manually construct for every service composition, the system for service composition is required to automatically construct bindings. Especially in open environment where various organizations can provide their services and states of the services often change, the automatic binding construction needs to satisfy the following requirements.

First, bindings need to be constructed based on user's runtime request and constraints given by service providers. As for Language Grid Toolbox[1], which is a series of multilingual communication support tools using language services on the Language Grid, allows users to simply designate services via user-friendly interface (Fig. 2). Moreover, a user and service providers may have policies given as constraints on combination of services. A user wants to limit a certain combination of service types when the user knows the combination leads to low quality results. Service providers can also have constraints of combination of concrete services, such as prohibition of combination with competitors' services. In Language Grid Toolbox, a strategy for constructing bindings is hard-coded because it takes too much time to construct bindings which include services designated by the user and satisfy given constraints. But such a hard-coded strategy does not work well in open environment where various users and providers can join.

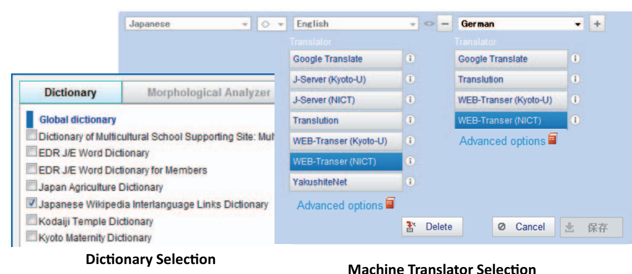

Figure 2. Service settings on Language Grid Toolbox.

Second, bindings often need to be modified even at runtime. Assume an input string to the translation with dictionary service is too long and number of morphemes generated by a morphological analyzer exceeds the limit of the dictionary service. In that case, a runtime adaptation such

[1]Trial Site: http://langrid.org/tools/toolbox/

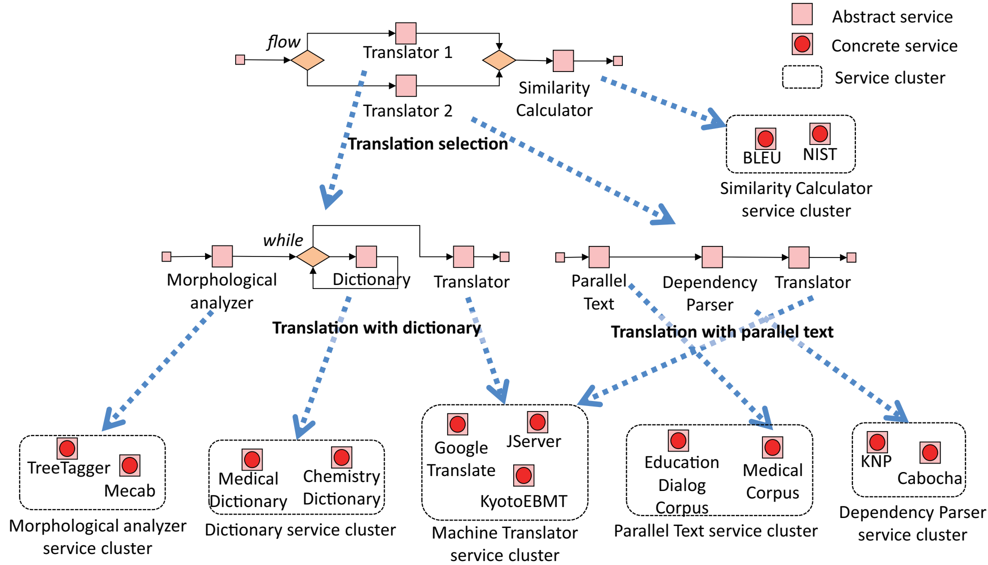

Figure 1. An example of service bindings.

as adding alternative services needs to be applied to the running composite service. According to the changes, bindings should be updated keeping designated services included and constraints given by a user and service providers satisfied.

Various approaches for automatic binding construction proposed in previous works can satisfy the above requirements. However, binding construction based on the approaches is time-consuming and the performance may decline because they construct bindings from scratch. In our assumption, availability of services does not drastically change in a short time. Thus, we need an approach to reuse bindings according to changes of user's requirements and constraints.

III. Construction of Bindings

Our approach to the problems described in the previous section is to process binding construction as *hypothetical reasoning*. Service composition can be processed as reasoning because it consumes some services and produces a new service like a rule in reasoning. Moreover, the reasoning is hypothetical because a set of services required for a composition may conflict with the other set for other composition due to constraints on combination of services like non-monotonic reasoning. Based on the relationship between service composition and hypothetical reasoning, we introduced the idea of ATMS (Assumption-based Truth Maintenance System)[7]. In this section, we show a model and an algorithm for binding construction based on ATMS.

A. Extension to ATMS

ATMS creates a node which corresponds to *data*. The node has a set of *environments*, where the data holds. An environment is represented as a set of symbols of data which correspond to premises or assumptions. A set of environments where data holds is called *label* of the node. Given justifications by a reasoning engine, ATMS creates a new node or updates labels of nodes according to dependencies among data.

The above framework works for our service composition. For service composition, data of a node corresponds to an abstract service or a composite service. A rule corresponds to a composite service because it consumes some services and produces a new service. An environment in a label of a node corresponds to a possible set of required abstract services. Once the graph is created, we can know a combination of abstract services which are required to realize the target composite service just by checking labels of the node of the target service. Then we can identify a combination of concrete services which are bound to the abstract services and satisfy constraints. A constraint solver can be used to solve the constraints.

The more intuitive approach to apply ATMS to service composition is to map a concrete service to a node. In this approach, we can determine complete bindings based on a label of a node without using a constraint solver. However, we assume many concrete services are available for each service cluster. Since the computational complexity of ATMS is NP-complete, the number of nodes can significantly affect the performance. This is the reason we introduced the two-step service composition, which consists of constructing a graph and solving constraints. We mention this again in Section 5.

To adapt ATMS to service composition, we introduced the following extensions.

- Reasoning based on service types

- Merging environments considering services explicitly designated by a user

In an inference engine used with ATMS, an antecedent of a rule is matched with data of a node. On the other hand, for our service composition, constituent abstract services of a composite service are defined based on service types. For example, any service which belongs to Machine Translation service, including both concrete services and composite services, can be bound to an abstract translation service in Translation Selection service in Fig. 1.

Moreover, ATMS removes an environment which subsumes other environments in the same label. This is because data which holds in an environment certainly holds in other environment which subsumes the environment. In our service composition, however, different environments correspond to different bindings. Although an environment is subsumed by other environments, the corresponding composition can be necessary as a part of the bindings which realize target composition. Thus, we remove an environment only when the environment contains all services specified by a user. This is based on the idea that the bindings which contain smaller number of services are better as long as services designated by a user are contained.

B. Formalization

We show the formalization of data structure following the idea of ATMS. First a node is defined as follows.

$$< name, type, label, justification >$$

$name$ is an identifier of a service and $type$ is the type of the service. $label$ represents sets of environments which are sets of abstract services required to realize this service. $justification$ is given by an inference engine and shows which nodes justify this node.

An abstract service which corresponds to a concrete service designated by a user is represented as a *premise*, which has empty sets as its label and justification. Other abstract services are represented as *assumptions*, which have themselves in their labels and justifications.

A composite service is defined as a rule which has the following form.

$$type_1, type_2, ..., type_n \rightarrow (name, type)$$

$type_i$ is the type of a service which is a component of the composite service. $name$ and $type$ are the identifier and the service type of the composite service. A new node is created by applying a composite service if a node which has the same name does not exist. Otherwise the label of the existing node is updated.

Take services shown in Section 2 as an example. Assume that the service types MT, Dic, MA, PT, DP, and Sim are available (they represent machine translator, dictionary, morphological analyzer, parallel texts, dependency parser, and similarity respectively) and that our target application is a translation tool. Since a machine translation service is certainly used for the application, the abstract service of translation is represented as a premise. When a user designated a concrete dictionary service, the correspoinding abstract service is also represented as a premise.

$$(AbstractTranslator, MT, \{\{\}\}, \{\{\}\})$$
$$(AbstractDictionary, Dic, \{\{\}\}, \{\{\}\})$$

Other abstract services are defined as assumptions. For example, a dictionary can be optionally used to be combined with a translator.

$$(AbstractMorphologicalAnalyzer, MA,$$
$$\{\{AbstractDictionary\}\}, \{(AbstractDictionary)\})$$

We also assume the following composite services are available. They represent the translation with dictionary service, the translation with parallel texts service, and the translation selection service in Fig .1 respectively.

$$\begin{aligned}(a) MT, MA, Dic &\rightarrow (TransWithDic, MT)\\ (b) MT, DP, PT &\rightarrow (TransWithPT, MT)\\ (c) MT, MT, Sim &\rightarrow (TransSelect, MT)\end{aligned}$$

Moreover, an application may have policies on combination of abstract services or composite services. Such a constraint is integrated into ATMS by defining it as a rule which produces contradiction. Assume the combination of AbstractTranslator and AbstractParser is prohibited. This constraint is represented as:

$$AbstractTranslator,$$
$$AbstractParser \rightarrow \bot$$

Following the above definition, we can construct a graph by applying composite services. We call the graph a *binding dependency graph*. Figure 3 shows a binding dependency graph after some rules are applied. Rectangles and ellipses are nodes of the binding cache graph. Rectangles on the left hand represent atomic services. Ellipses represent composite services and links represent dependency between nodes. A double-lined ellipse represents a premise.

A name of a node is shown in an ellipse. A label is shown besides the node which corresponds to a composite service. For example, TransWithDic is created by applying a composite service (a) to three abstract services, AbstractDictionary, AbstractMorphologicalAnalyzer, AbstractTranslator. Since availability of the composite service depends on that of these atomic services, the label contains the names of the services. Similarly, TransWithPT is created by applying (b).

TransSelect node can be created by applying (c), but some justifications can be given by an inference engine. Given three other translation services, AbstractTranslator, TransWithDic, and TransWithPT, TransSelect node can have three environments. Each of them shows a possibility of

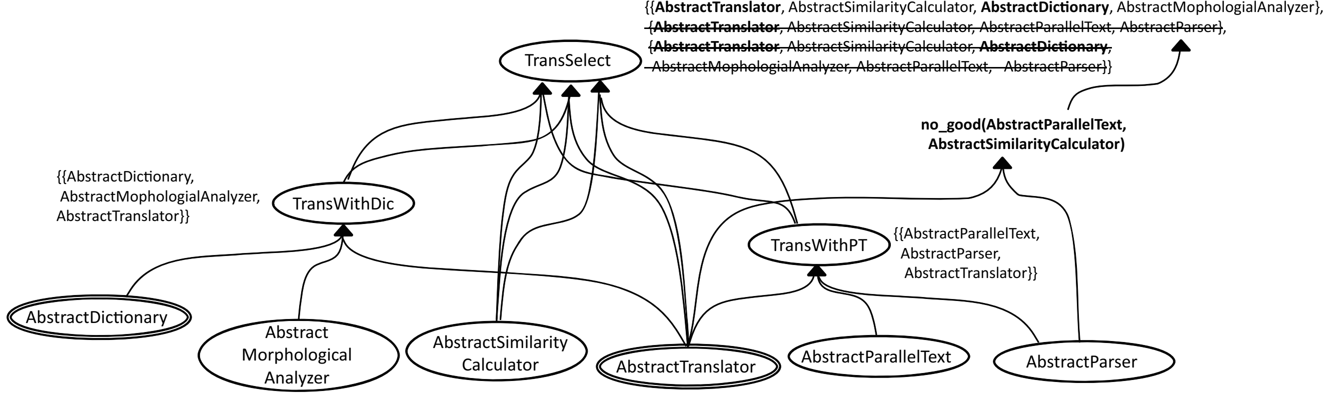

Figure 3. A binding dependency graph.

bindings using two of the three other translation services. In Fig. 3, the third environment subsumes the first one and the first one contains all abstract services corresponding to designated services. Thus, the third environment is removed as a strike-through is shown in Fig. 3.

When a justification of contradiction is inferred by AbstractTranslabor and AbstractParser, a node of $nogood$ is created. The second environment in the label of TransSelect is removed according to the nogood because the environment subsumes the nogood combination.

In the initial state, we have nodes of abstract services as a binding dependency graph. Following the style of ATMS, they are sent to an inference engine as premises or assumptions. Given the justification by the inference engine, a binding dependency graph is constructed by Algorithm 1. The algorithm takes a set of abstract services S_d which corresponds to designated services as premises and the justification $s \leftarrow s_1, ..., s_n$ given by the inference engine. The algorithm is an extension of the algorithm shown in [9]. The major extension to ATMS is judgment criteria of removing environments. As shown in line 7, we remove environments only when subsumed environment in a label contains types of all designated services.

Algorithm 1 Construct(S_d, $s \leftarrow s_1, ..., s_n$)

```
1:  for all justification j_k (s ← s_k1, ..., s_k_{n_k}) do
2:      L_k ← {∪ e|e ∈ ∀i (label of s_ki)}
3:  end for
4:  L_new = {∪ e|e ∈ ∀i (label of s_i)}
5:  L ← Remove no_goods, and supersets of others from
    {e ∈ ∀k(L_k) ∪ {L_new}}
6:  for all e ∈ L do
7:      if ∃e′(e ⊇ e′ ⊇ S_d) then
8:          Remove e ∈ L′
9:      end if
10: end for
11: if L = label of s then
12:     return
13: end if
14: if s = ⊥ then
15:     Mark ∀e ∈ L nogood
16:     Remove the nogoods from all node labels
17: else
18:     J ← A set of justifications having s in antecedents
19:     for all j_l ∈ J do
20:         Construct(S_d, j_l)
21:     end for
22: end if
```

IV. Adaptation for runtime change

Once a binding dependency graph is created, we can identify which abstract services are required. Thus we can find a combination of concrete services which satisfy constraints given by service providers. To find the combination of concrete services, we use a constraint solver.

However, since a composite service consists of services provided by various organizations, runtime errors are often caused due to unexpected changes of states or specification of services. This is the reason we often introduce some runtime adaptation mechanism to avoid the errors.

In this section, we first overview an execution control framework called *Service Supervision* as an adaptation mechanism. Next we show how to efficiently modify bindings when the business logic of a composite service is changed by the runtime adaptation.

Many previous works on runtime adaptation for composite services rely on AOP (Aspect-oriented Programming) in order to dynamically add some processes. Some other works transform a composite service according to requirements at runtime. One of the most flexible approaches is *Service Supervision* proposed in [8]. Service Supervision provides execution control APIs which change business logic of a composite service for runtime adaptation. Table 1 shows some of the execution control APIs. These APIs are applied to composite services which are defined as workflows. Since

these APIs are implemented as Web services, this enables us to define a composite service for controlling other composite services.

Figure 4 shows an example of adaptation using these execution control APIs. The translation with dictionary service is shown in the bottom part in Fig. 4 as the target of the adaptation. Assume the length of an input to translation service exceeds the limit. The *Supervision Service* shown in the upper left part in Fig. 4, which uses the APIs shown in Table 1, is activated. The composite service inserts invocation of another composite service *Adaptation Service*, which is shown on the upper right part of Fig. 4 as a parallel process. The adaptation service first splits an input document into sentences, and then executes a translation service for each sentence. Finally the results of the translation of each sentence are merged. The invocation of the translator service originally defined in the translation with dictionary service is skipped.

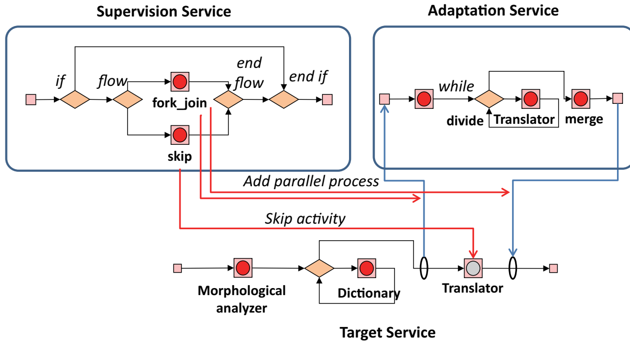

Figure 4. An example of adaptation using execution control APIs.

The adaptation mechanism can flexibly change behavior of a running instance of a composite service. Moreover, adaptation processes can be easily understood and reused because they are defined as workflows.

However, when an adaptation process adds or skips invocations of services in a composite service, bindings must be modified. Therefore, we need an efficient algorithm for modifying bindings in order to reduce decline of performance. In most of cases, the execution control APIs affect only some of abstract services. The idea of our approach is that we can reuse the binding dependency graph for modifying bindings during execution.

When a new service is added to the target service, only the label of the target service is calculated based on all the justifications for the target service. Note that services which are already executed are given to inference engine as premise in order to improve efficiency of reasoning. Algorithm 2 shows the above process.

$adapt$ corresponds to one of the functions in Table 4 which adds or removes antecedents of a composite service. cs_{target} is the composite service to be adapted. S_b is bindings which are already determined by the user and executed services. G is a binding dependency graph which is already constructed by Algorithm 1. D is available concrete services for abstract services. C is a set of constraints on combination of concrete services.

Algorithm 2 ModifyBinding($adapt, cs_{target}, S_b, G, D, C$)

1: $cs'_{target} \leftarrow$ adapt(cs_{target})
2: **repeat**
3: $(s \leftarrow s_1, ..., s_n) \leftarrow$ InferenceEngine(G, $cs'_{target} XS$)
4: $L_{new} = \{\cup\, e | e \in {}^{\forall}i$ (label of s_i)$\}$
5: $L \leftarrow$ Remove no_goods, environments which subsume other environment containing S_d from $\{e \in {}^{\forall}k(L_k) \cup \{L_{new}\}\}$
6: **for all** $env \in L$ **do**
7: $soluation \leftarrow$ solveCSP(env, D, C, S_b)
8: **end for**
9: **until** Solution found or no new justification found
10: **return** $solution$

First antecedents of cs_{target} are added or removed. Next the inference engine finds the justification using the adapted composite services. The label L of the target service is calculated in the same way as Algorithm 1. Then, for each environment in L, the algorithm tries to find a combination of concrete services by using a constraint solver. An environment env is given as a set of variables of CSP (constraint satisfaction problem). D and C are given as the domain and the constraints respectively. Since values of some variables are already determined, S_d is also given to the solver. The algorithm finishes when any of consistent combinations is found.

V. EXPERIMENTS

We conducted some experiments in order to evaluate how efficiently our method solves service composition. Since the computational complexity of our method can become large according to the number of service types, we first investigated time of constructing binding cache graphs with various numbers of service types.

In our experiment, given n service types, we defined n composite services. The number of antecedents of the composite services are all set to 3. Types of services and antecedents of composite services are selected at random from defined n service types. The maximum depth of the propagation in Algorithm 1 is set to 3.

Figure 5 shows time of construction of binding dependency graphs. The horizontal axis represents the number of service types n. The vertical axis represents the execution time. We conducted the construction of binding dependency graph 50 times for each n and plotted the point on the chart.

The figure shows that execution time can be longer as the number of service types increases. But the points are sparse in the right hand of the figure because some execution

Table I
EXAMPLES OF EXECUTION CONTROL APIs[10].

API	Effect
`skip(pid, start, end)`	Skip activities in the specified range. This API takes a process ID and activity locations which specify the range to be skipped.
`fork_join(pid, start, end, invocation)`	Add a parallel process to the specified instance. This takes a process ID, activity locations where the parallel process starts/finishes and invocation information (endpoint address, operation and input data) as parameters.
`fork(pid, start, invocation)`	Add a parallel process to the specified instance. This takes a process ID, an activity location where the parallel process starts and invocation information as parameters.
`insert(pid, start, pname, param)`	Invoke a service at the specified location. This takes a process ID, an activity location where the invoked process starts and invocation information as parameters.

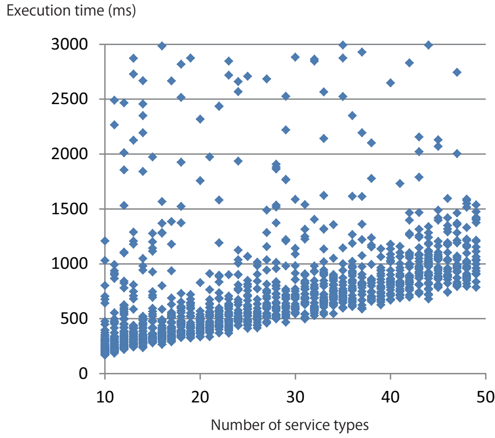

Figure 5. Execution time of binding dependency graph construction.

requires longer than 3,000 ms, which is the maximum number shown in the figure. The construction of the binding cache graph is based on ATMS, whose computational complexity is NP-complete. Therefore, some experimental settings which are randomly generated take long time. When service types of composite services and their antecedents are biased, a node can have many edges and computational complexity can be large.

While construction of a binding dependency graph is done by offline, updating a binding dependency graph and solving constraint satisfaction performed at runtime. Figure 6 shows the time of the processes assuming a new constituent service is added. We used an existing constraint problem solver[2]. In this experiment, we defined 20 concrete services for each service cluster and ${}_nC_2 \times r$ binary constraints, which prohibit combination of two concrete services. r represents the number of constraints which are defined for services in two service clusters. We set r to 10 in our experiment. The scale of the problem is large enough because the number is much bigger than the number of the services on a real service composition environment such as the Language Grid, which is introduced in Section 1.

As Fig. 6 shows, the number of types does not affect time of updating a graph because the update process shown in Algorithm 2 does not propagate unlike Algorithm 1. On the other hand, time of constraint satisfaction becomes long as the number of services types increases because constraint satisfaction is also NP-complete. When a bigger value is given for r, the time of constraint satisfaction becomes longer. This result clarifies that our method works well when there are many composite services and policies on abstract services but not so many constraints on combination of concrete services.

[2]http://www.emn.fr/z-info/choco-solver/

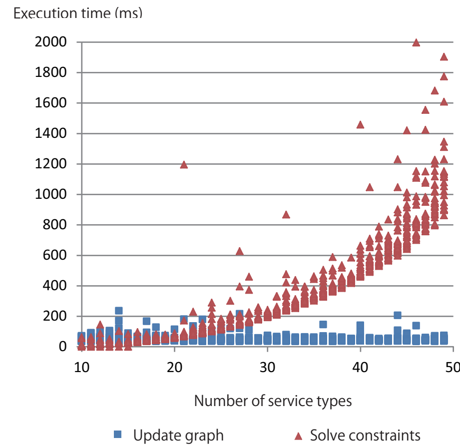

Figure 6. Execution time of updating graph and solving constraints.

As we briefly mentioned in Section 3, the model which defines concrete services as nodes can omit the process of constraint satisfaction. Based on the model, labels of nodes of composite services consist of concrete services. However, number of nodes can be much bigger when we define concrete services as nodes in a graph and computational costs of binding dependency graph construction can be very large. In our experiments, binding dependency graph construction based on the formalization was intractable and rarely finished in several hours given the same conditions used in experiments of Fig. 5. This is the reason we proposed the combination of ATMS based on service types and a CSP solver as a practical approach.

VI. Related Works

Many previous works have proposed methods for service composition. Approaches taken by most of the previous works can be classified into two types: vertical composition and horizontal composition. The former generates business logic of a composite service using various techniques including AI planning. The latter assumes that composite services which consist of abstract services are given and assigns appropriate concrete services to the abstract services. The framework proposed in this paper is classified into the horizontal composition.

The method proposed in [3] is one of the most popular works on horizontal composition and tries to find a combination of services which gives the best QoS. Similarly, the method proposed in [2] also obtains a combination of services which satisfies given constraints considering interfaces of services. But these previous works do not consider the hierarchical bindings such as those shown in Fig. 1. In [4], the authors proposed a method for hierarchically composing actions based on planning. But they do not consider the frequent changes of availability of services.

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)[11], using a proxy to monitor/change messages exchanged between a composite service and invoked services([5], [6]), and transforming the model of a composite service. An execution control framework proposed in [8] provides more flexible adaptation as described in Section 4.

However, flexibility of adaptation often makes coordination between services difficult and causes another runtime error. Therefore, a comprehensive approach of service composition, adaptation, and verification is required. In [12], the authors proposed an architecture which combines selection of concrete services and runtime adaptation given a set of target abstract services. The verification method based on model checking which is designed for runtime adaptation is also proposed in [10]. The framework proposed in this paper works as the basis of the previous works by constructing bindings which can be easily modified according to runtime adaptation.

VII. Conclusion

In this paper, we proposed a method which constructs bindings for open environment. The contribution of this work is as follows.

- We propose a model and algorithm based on ATMS which enables us to efficiently determine bindings which combines services designated by a user and satisfies constraints given by the user service providers.
- We showed an algorithm for modifying bindings during execution of a composite service according to runtime adaptation which changes constituents services of the composite service.

For future work, we will apply the framework to client software that uses services on the Language Grid in order to test our framework in practical use.

Acknowledgments

This works was supported by Strategic Information and Communications R&D Promotion Programme from Ministry of Internal Affairs and Communications.

References

[1] T. Ishida, *The Language Grid: Service-Oriented Collective Intelligence for Language Resource Interoperability*. Springer, 2011.

[2] 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.

[3] 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.

[4] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN planning for web service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377–396, 2004.

[5] 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.

[6] 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.

[7] J. de Kleer, "An assumption-based TMS," *Artificial Intelligence*, vol. 28, no. 2, pp. 127–162, 1986.

[8] 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.

[9] J. de Kleer, "A general labeling algorithm for assumption-based truth maintenance," in *the 7th National Conference on AI (AAAI-88)*, 1988, pp. 188–192.

[10] M. Tanaka, Y. Murakami, and D. Lin, "A service execution control framework for policy enforcement," in *the 8th International Conference on Service Oriented Computing (ICSOC 2010)*, 2010, pp. 154–161.

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

[12] M. Tanaka, Y. Murakami, D. Lin, and T. Ishida, "Service supervision for service-oriented collective intelligence," in *IEEE 7th International Conference on Services Computing(SCC 2010)*, 2010, pp. 154–161.