Language Service Composition Based on Higher Order Functions

Takao Nakaguchi, Yohei Murakami, Donghui Lin and Toru Ishida

Abstract To support multi-language activities, various composite services are created by a service composition that combines existing services or changes the combination of services composed by existing composite services. Multi-language activities have a wide variety of domains and their needs may change with the participants or situations, so service composition must be able to freely create various services to suit the languages of the participants and/or domains of the activity targets. Since existing service composition technologies relies on the deployment process of created composite services toward service infrastructure for users to find and execute them, delay and costs are expensive. To solve this problem, we propose a method that introduces the concept of higher order functions. In concrete, we regard services as functions and pass the functions invoked from composite services as runtime parameters of composite services to compose services without any deployment processes, this yields service composition that can efficiently support multi-language activities. We apply the proposals to Language Grid, designed to gather and provide language services, and evaluate the results. They show that our proposals can create various composite services at runtime with quite practical overheads.

T. Nakaguchi (✉) · D. Lin · T. Ishida
Department of Social Informatics, Kyoto University,
Kyoto 606-8501, Japan
e-mail: ta_nakaguchi@kcg.edu

D. Lin
e-mail: lindh@i.kyoto-u.ac.jp

T. Ishida
e-mail: ishida@i.kyoto-u.ac.jp

T. Nakaguchi
Department of Web Business Technology,
The Kyoto College of Graduate Studies for Informatics, Kyoto, Japan

Y. Murakami
Unit of Design, Kyoto University,
91 Chudoji Awata-cho, Kyoto 600-8815, Japan
e-mail: yohei@i.kyoto-u.ac.jp

Y. Murakami et al. (eds.), *Services Computing for Language Resources*,
Cognitive Technologies, https://doi.org/10.1007/978-981-10-7793-7_3

Keywords Service composition · Higher order functions · Aspect oriented programming

1 Introduction

As the internationalization of society advances, demands for multi-language support have grown. Nowadays, composite service technology for the language processing domain is being used to support international activities by easing language barriers. Composite services can be created by combining existing services through the use of inter-service dependencies among component services in SOA-based service infrastructures [3]. Services that don't offer such dependencies are called atomic services.

A service infrastructure itself is a generic system but when used in a specific domain it becomes specialized for that domain. Taverna [15], one such service infrastructure, has a unique workflow editor that allows web services or functions to be executed on a local machine in various combinations that are controlled by workflows. HELIO [1] is a specialized version of Taverna for the solar physics domain while BioVeL Project [2] provides services specific to the biodiversity domain. Natural Language Processing is another of the targets that tends to specialization. Service Grid Server Software [13] is a service infrastructure that provides web services and adopts WS-BPEL[1] as its workflow description language and ActiveBPEL[2] as its workflow execution system. Specialized versions of this software for the language-processing domain include LAPPS Grid [5] and Language Grid [6].

The language-processing domain can be the one of the most effective specializations of service infrastructures. While there are many data or processing programs in this domain, that don't have well-standardized interfaces and are used most often in isolation. Murakami et al. proposed a domain model and service architecture to introduce SOA to this domain, which yield the Language Grid [12]. For the Language Grid, 27 key interfaces were standardized and they offer access to 227 atomic services. To date, 22 composite services have been established. Since services that are accessed through the same invocation interface can be switched, the number of composite services that can be based on those services is potentially extremely large. Unfortunately, complexity explodes if constituent services are nested in a circular manner. The difficulty of registering and managing all possible composite service variations statically makes it essential to create a mechanism that can identify the possible variations dynamically at runtime. Our solution is a description method that introduces higher-order functions into service composition and so can describe hierarchical service structures; we evaluate it by implementing and applying it to the Language Grid. The remainder of this chapter is organized as follows. Section 2 describes the problem of composite service variations. In Sect. 3, we propose a hierarchical

[1] https://www.oasis-open.org/committees/wsbpel/.

[2] https://sourceforge.net/projects/activebpel502/.

service composition description, and in Sect. 4, we apply it to an existing service execution system. In Sect. 5, we evaluate the efficiency of our proposed method. In Sects. 6 and 7, we mention works related to this chapter to position this study. Finally, we conclude the chapter in Sect. 8.

2 Composite Services and Their Variations

Service composition is a technology that creates composite services by combining existing services. A service wraps data or programs and runs independently [3]. To increase the interoperability of services or to make application development easier, invocation interfaces must be unified. In the Language Grid, *Juman*[3] service and *TreeTagger*[4] service are provided through the *MorphologicalAnalysis* interface. *LifeScienceDict*[5] service and *KyotoTourismDict* service are accessed through the *BilingualDictionary* interface. *JServer*[6] service and *GoogleTranslate*[7] are accessed through the *Translation* interface. Each service runs independently and can be accessed by multiple clients simultaneously.

We turn our attention to composite services. *BackupTrans* can combine several translation services and allows making another service to be invoked as a backup service when a main service fails. *DictTrans* combines a translation service with a bilingual dictionary service. *DictCrossSearch* combines several bilingual dictionaries and provides unified search across all of them.

These services can be constructed as shown in Fig. 1. Double-line rectangle denotes composite services and single-lined denotes atomic services. In the figure, *BackupTrans* service combines *DictTrans* and *GoogleTrans* service. *DictTrans* combines the morphological analysis service *Juman*, a bilingual dictionary service *DictCrossSearch* and a translation service *JServer*, and increases translation quality by using the bilingual dictionary to supply the special words that the translation service is unaware of. Unfortunately, depending on several services constitutes a greater risk than invoking a single service because a service may fail to execute depending on the situation such as high CPU loads or network problems. That is why *BackupTrans* combines a composite translation service and a single *GoogleTranslate* service. *DictCrossSearch* service invokes *LifeScienceDict* service and *KyotoTourismDict* service through the *BilingualDictionary* interface and is accessed through the same interface. This is but one example of the services construction possible. Another user may need to invoke *DictTrans* directly, to use single bilingual dictionary service instead of *DictCrossSearch* or to combine other translation service with *JServer*. Because user needs may change depending on the situation,

[3]http://nlp.ist.i.kyoto-u.ac.jp/?JUMAN.

[4]http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/.

[5]http://lsd-project.jp/ja/index.html.

[6]http://www.kodensha.jp/platform/.

[7]https://cloud.google.com/translate/.

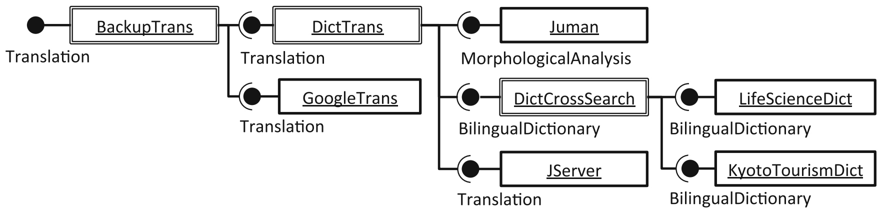
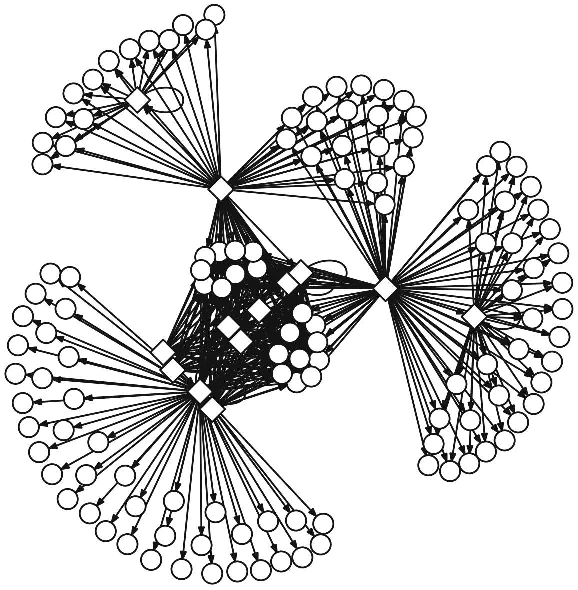


Fig. 1 Example construction of composite services

Fig. 2 Service invocation graph



such like the languages used by the participants or the domains targeted by the user, a technology to create composite services must have a wide flexibility and a rapidity to create various combinations of services.

Figure 2 graphs the relationship between composite services and services that can be invoked from these composite services on Language Grid. Diamond nodes denote composite services and circle nodes denote atomic services. Edges denote a relation between services and services that share the same interface. The graph does not show atomic services that are not invoked from composite services. The number of nodes is 138 and the number of edges is 451. As the graph shows, the service-relationships form a complex graph structure. Since variations exist in each combination of edges yielding an explosion in variation number, it is not realistic to register and manage all of them. In this chapter, we focus on the hierarchical structuring of services and propose a method to describe service composition by introducing higher-order functions. In addition, we implement a function that realizes a hierarchical service composition, which is described by the method, as an extension of an existing service composition execution engine.

3 Description Language for Hierarchical Service Composition

A higher-order function receives functions as its parameters or returns functions as its result. Huges showed their importance and examples of usage in detail [4]. By introducing higher-order functions, we can modularize processes and express an entire process as a combination of functions. In this chapter, we regard services as functions and express hierarchical service composition by using higher-order functions.

To invoke services, we must send information about method names and its arguments. Executing a composite service invokes other services. We assume services, which described by proposed method, are atomic services or a variation of composite service consisting of ids of a composite service and services invoked from it. For the latter, we introduce the higher-order function *bind*. *bind* can create a variation of a composite service which can be invoked by another composite service, so *bind* can create hierarchical graph structures. Figure 3 shows the grammar of the description.

description is a root element and expresses hierarchical service composition and its invocation. *service* is *service_id* or *composition* created by *bind* function. *composition* consists of the identifier of a composite service and the list of *invocation*. *invocation* consists of *invocation_id*, which identifies all invocations in composite service, *conds*, which identifies conditions that must be satisfied to invoke the *service*. *?*, + and * denote the occurrence zero or one, one or more and zero or more of preceding element. *LETTER* denotes normal characters other than white spaces.

Figure 4 shows an example description of the composition shown in Fig. 1. *bind* is used to specify the variation of *DictCrossSearch* and *BackupTrans* that *DictTrans* will invoke, and also *DictTrans* itself. As shown in Fig. 4, we can express the call graph structure of hierarchical service composition by introducing higher-order functions and invoke it. '清水寺は、京都にある寺院です' (KIYOMIZUDERA-HA, KYOTO-NI-ARU-JIIN-DESU) is a Japanese sentence meaning 'Kiyomizudera is a temple in Kyoto' in English.

```
description :== service "." method "(" args? ")"
service :== service_id | composition
method :== symbol
args :== ("′" symbol "′" ("," "′" symbol "′")* )
composition :== "bind(" service_id invocation+ ")"
invocation :== "," invocation_id ":" ("[" conds "]")? service
conds :== (cond ("," cond)* )
cond :== name ("==" | "<" | "<=" | ">=" | "!=") value
service_id :== symbol
invocation_id :== symbol
name :== symbol
value :== symbol
symbol :== LETTER+
```

Fig. 3 Grammar of hierarchical service composition description

```
bind(DictTrans,
 MorphologicalAnalysis: ['language'='ja'] Mecab,
 MorphologicalAnalysis: ['language'='en'] TreeTagger,
 BilingualDictionary: bind(DictCrossSearch,
  BilingualDictionary1: LifeScienceDict,
  BilingualDictionary2: KyotoTourismDict),
 Translation: bind(BackupTrans,
   MainTranslation: JServer, BackupTranslation: GoogleTrans)
 ).translate('ja','en', '清水寺は、京都にある寺院です。');
```

Fig. 4 Example of hierarchical service composition description

4 Implementation

The target of the implementation of the method proposed in this chapter is Language Grid. As we mentioned before, Language Grid adopts WS-BPEL as its description language of composite services and adopts ActiveBPEL as its composite service execution engine. To implement the proposed method, the following steps are needed: (1) Generate SOAP requests to invoke composite services from the description introduced in the preceding section and (2) to implement functions for determining services that the composite service actually invokes and generating the SOAP requests the composite service will send based on the hierarchical composite service description by extending the source code of ActiveBPEL.

4.1 Building SOAP Request

ActiveBPEL adopts SOAP[8] over HTTP[9] as its service invocation protocol. ActiveBPEL sends SOAP compliant requests to the services that will be executed. A SOAP request consists of *Header* and *Body*. *Header* can have any information while *Body* carries a method name and parameters. Thus, to implement the proposed method, we must generate the URL of services invoked and call graph information inserted into *Head* from the direct child element of the root element specified by the description. Moreover, we must generate information about the method and its parameters from the description that will be inserted into *Body*. Figure 5 shows the SOAP message generated from the description shown in Fig. 4. We omit namespaces and tag attributes for simplification.

Call graph information that will be used when the service is actually executed is generated in JSON[10] format, and added to the *Header* part of the SOAP messages, so that the service execution engine does not need to implement a description parser. *Body* carries a method name and parameters generated from the description. Though

[8]http://www.w3.org/TR/soap/.
[9]https://www.ietf.org/rfc/rfc2616.txt.
[10]http://json.org/.

```
<?xml version="1.0" encoding="UTF-8"?>
<Envelope>
<Header>
 <binding>[
  {"invocationId":"MorphologicalAnalysis",
   "conditions":[{"name":"language","op":"==","value":"ja"}],
   "serviceId":"Mecab"},
  {"invocationId":"MorphologicalAnalysis",
   "conditions":[{"name":"language","op":"==","value":"en"}],
   "serviceId":"TreeTagger"},
  {"invocationId":"BilingualDictionary",
   "serviceId":"DictCrossSearch", "children": [
    {"invocationId":"BilingualDictionary1", "serviceId":"LifeScienceDict"},
    {"invocationId":"BilingualDictionary2", "serviceId":"KyotoTourismDict"}
   ]},
  {"invocationId":"Translation",
   "serviceId":"BackupTrans", "children":[
    {"invocationId":"MainTranslation", "serviceId":"JServer"},
    {"invocationId":"BackupTranslation","serviceId":"GoogleTrans"}
   ]}</binding>
</Header>
<Body>
 <translate>
  <sourceLang>ja</sourceLang>
  <targetLang>en</targetLang>
  <source>清水寺は、京都にある寺院です。</source>
 </translate>
</Body>
</Envelope>
```

Fig. 5 Example of SOAP message

parameter names and the URL of service are not shown in Fig. 4, they can be extracted from the information of service definition information (WSDL[11]) of the service. Invoking the search API of Language Grid, returns the WSDL of services.

4.2 *Intercepting and Replacing Service Invocation*

Next, we consider extending the service invocation process in the composite service execution engine. To realize the proposed method, the extension of composite service execution engine must extract information from the call graph, shown in the *Head* part in Fig. 5, sent by the client and use said information to trigger the service invoked. If the client ascribes a sub call graph to the invoked service, we have to insert a sub call graph into the request sent to the service. For example, in executing the *DictTrans* composite service based on the description shown in Fig. 4, the engine must switch the service to *DictCrossSearch* service and insert a sub call graph that specifies *BilingualDictionary1* and *BilingualDictionary2* into the SOAP request when the invocation identified by *BilingualDictionary* is executed. The service identifiers

[11]https://www.w3.org/TR/wsdl.html.

inside a composite service correspond to a partner link name inside the corresponding BPEL description.

To implement the proposed method, we could extend ActiveBPEL to reduce the cost. Direct modification of the source code of ActiveBPEL directly complicates subsequent source code revisions. Worse, when the source code is modified by version up, we must confirm the consistency of modification before fixing the change. For that reason, we use Aspect Oriented Programming (AOP) [8] to extend ActiveBPEL. By using AOP, we can isolate the extension codes from the original software. In implementing our proposed method, we used AspectJ [7] to apply AOP and inserted our extension codes into ActiveBPEL. ActiveBPEL receives the execution request for the composite service, then executes the composite service by executing the activities written in BPEL and returning the result. Because the call graph information inserted into the execution request and the extending service invocation based on that information are needed in order to realize our method, we extend the processes of *Receive* activity and *Invoke* activity. Figure 6 shows the class diagram that denotes classes related to this extension.

The package *org.activebpel.*** denotes all packages and classes under the package *org.activebpel* that represent modules associated with ActiveBPEL. *ActiveBPEL-Aspect* and *AspectBase* are the aspect and the base class that we implemented. The package *org.apache.axis.***, *java.*** and *javax.xml.soap.*** denote a third party library and Java standard libraries. Pointcuts in *ActiveBPELAspect* catch the executions of methods in ActiveBPEL that are needed to modify behavior of composite services and use methods in *AspectBase* to switch services as specified in the description. As shown in Fig. 6, the ActiveBPEL dependency is enclosed in *ActiveBPELAspect*, so *AspectBase* is independent from it.

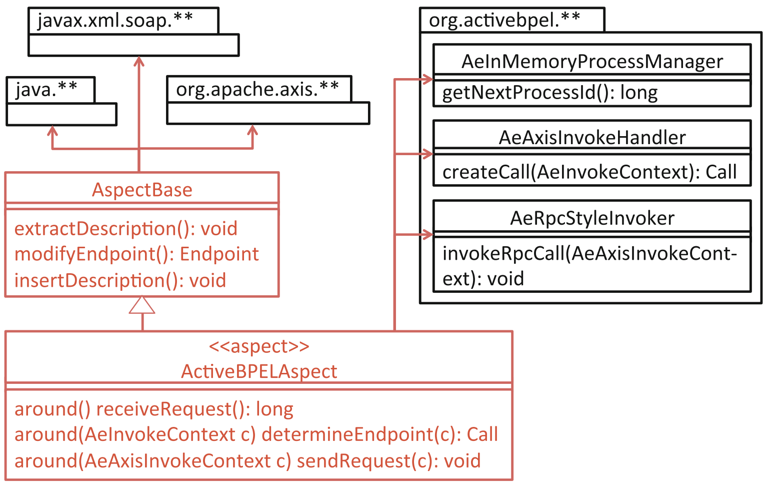

javax.xml.soap.**
java.**
org.apache.axis.**
org.activebpel.**
AeInMemoryProcessManager
getNextProcessId(): long
AeAxisInvokeHandler
createCall(AeInvokeContext): Call
AeRpcStyleInvoker
invokeRpcCall(AeAxisInvokeCont-ext): void
AspectBase
extractDescription(): void
modifyEndpoint(): Endpoint
insertDescription(): void
<<aspect>>
ActiveBPELAspect
around() receiveRequest(): long
around(AeInvokeContext c) determineEndpoint(c): Call
around(AeAxisInvokeContext c) sendRequest(c): void

Fig. 6 A module structure of extension and related classes of ActiveBPEL

5 Evaluation

We evaluate the proposed method by applying it to the Language Grid. In more detail, we estimate the number of composite services that our method can realize by calculating the number of all variations of composite services using real services registered on the Language Grid and measure the execution overhead of our implementation by comparing the execution time of composite services with and without our method.

5.1 Number of Service Composition that Can Be Realized

The purpose of the proposed method is to realize various service compositions at runtime without deployment process to service execution engine. We measure the actual number of service compositions as the measure used to rate of the effectiveness of our proposed method. It is calculated as the product of the number of services conforming to the service invocation interfaces specified in a composite service. The services can be atomic services or composite services. Though the service invocations can be executed as long as the interface and the network protocol of the services match the specification of the service invocations, they could fail. To successfully complete the invocations, we should consider domain-specific constraints. In the case of the Language Grid, the constraints are that invoked services must support the languages specified as parameters from their invokers (composite services or client programs). For example, the *DictTrans* composite service invokes services conforming to *MorphologicalAnalysis* interface, *BilingualDictionary* interface and *Translation* interfaces, respectively. To invoke the *DictTrans*, the client sends the source language and target language as part of parameters. Then, the *DictTrans* passes the source language to the *MorphologicalAnalysis* service, and a pair of source language and target language to the *BilingualDictionary* service and the *Translation* service. These services must support the language(s) passed as parameters. We can determine whether a service supports certain language(s) or not by invoking the search API of the Language Grid. We count only those variations that have services that satisfy this constraint.

Further, because composite services can be nested like a layer and a composite service can be used in different layers, it is impossible to calculate the number in a simple brute-force manner. Therefore, we calculate the variations in each layer step-by-step. In the first calculation stage, we calculate the number of variations of all composite services from just the atomic services assigned to invocations and create variations of composite services. In the stages after the first stage, we calculate the number of variations according to atomic services and the variations created in the previous stage. In this way, we can calculate the summation of the variations at the calculation stage s: $ALLV_s$ by using the following formula. The number of first calculation stage is 1.

$$ALLV_s = \sum_{k=1}^{n} v(k, s)$$

n is the number of all services. $v(k, s)$ is the number of the variations of service k at stage s and can be calculated by the following formula.

$$v(k, s) = \begin{cases} 0, & s = 0 \text{ or service } k \text{ is ATOMIC} \\ \prod_{l=1}^{m_k} vi(k, l, s), & s \geq 1 \end{cases}$$

$v(k, s)$ becomes 0 when s is 0 or the service k is an atomic service, otherwise it becomes the product of $vi(k, l, s)$, which is the number of services that can be assigned to each service invocation l of the composite service k. m_k is the number of invocations of service k. Though the first calculation stage is 1, we define the number of variations at the stage 0 as 0 because $vi(k, l, s)$, described below, refers previous stage. $vi(k, l, s)$ can be calculated by the following formula.

$$vi(k, l, s) = \sum_{j=1}^{n} \begin{cases} 0, & \text{service } j \text{ does not satisfy } C_{kl} \\ 1, & \text{service } j \text{ is ATOMIC} \\ v(j, s-1), & \text{service } j \text{ is COMPOSITE} \end{cases}$$

The calculation of $vi(k, l, s)$ considers only the those services that satisfy condition C_{kl}, which is that service j must conform to the interface of service invocation l and service j must support the language(s) passed from the composite service k. When the service j satisfies the condition and is an atomic service, $vi(k, l, s)$ is incremented by 1. If service j satisfies the condition and is a composite service, $vi(k, l, s)$ is incremented by the number of variations of service j at the former stage $s - 1$. At the calculation stage 1, $vi(k, l, s)$ becomes the number of atomic services that satisfy the condition C_{kl} because $s - 1$ is 0 so $v(j, s - 1)$ becomes 0.

Using the above formulas, we calculated the number of variations of composite services registered to the Language Grid that can translate Japanese (ja) to English (en). The number of atomic services that translate from ja to en is 65, the largest number of translation services among all the supported language pairs. Table 1 shows the results.

As shown in Table 1, the number of variations slightly increases with the stage number, and can easily explode when composite services are nested. We calculated the number of variations up to the 5th stage. In the Language Grid, 5-times nested service composition is practical as in nesting *BackTranslation* service which combines two *Translation* services to translate a translation result into source language, *BestTranslationSelection* service which combines several *Translation* services and *SimilarityCalculation* services to select a translation service whose result can be back-translated to yield a result most similar to the source text, *TranslationWith-Backup* service which combines two or more *Translation* services to realize a fallback function for translation, *DictTrans* service, and *DictCrossSearch* service. The vari-

Table 1 Calculation result

Calculation stage	ja → en
1	4.09×10^{6}
2	1.14×10^{33}
3	1.89×10^{165}
4	2.44×10^{826}
5	8.64×10^{4131}

ations of 5-times nested services which translate from ja to en were 8.64×10^{4131}. To deploy and use these variations, we must register and manage them in the service infrastructure. However, we can realize these compositions without paying this cost by using the proposed hierarchical service binding technique.

5.2 Execution Overhead

The proposed method is an extension to the existing workflow execution engines. It analyzes the *Head* part of SOAP request sent to a composite service, adds service binding information to the SOAP request from the composite service to component services, and modifies target URL of the SOAP request. These modifications might increase the execution time of composite services. However, each modification is not complex, and so overhead is insignificant. To confirm this, we prepare two configurations of ActiveBPEL, one is the existing system and the other is the system with our method. We use the same atomic services to execute a variation of *DictTrans* service which uses *DictCrossSearch* service in the both configurations, and compare the results. The call graph that is inserted into the SOAP request in the latter configuration is the same as that shown in Fig. 4. Figure 7 shows the sequence diagram of the composition we use for the evaluation. At each method invocation, a SOAP request is sent to target component. *Client* ran on a local machine and measured execution time, *DictTrans* and *DictCrossSearch* ran on ActiveBPEL in Language Grid and remaining services ran on a Language Grid server.

We extracted texts from two pages of Japanese Wikipedia, and translated them from Japanese to English using the both configurations of ActiveBPEL. One is the page for Kiyomizu Temple[12] and the other is the page for Life Science.[13] Twelve lines (A-L) were extracted from Kiyomizu Temple text and 7 (M-S) from Life Science. We translated each sentence 100 times. Figure 8 shows the result of execution. The left axis denotes the average execution time of 100 translations run for each sentence (A-S), while the right axis denotes the number of characters in each sentence. The filled and hollow rectangles denote the execution time by the existing method and the proposed method, respectively. The triangles denote the number of characters in

[12]https://ja.wikipedia.org/wiki/%E6%B8%85%E6%B0%B4%E5%AF%BA.

[13]https://ja.wikipedia.org/wiki/%E7%94%9F%E5%91%BD%E7%A7%91%E5%AD%A6.

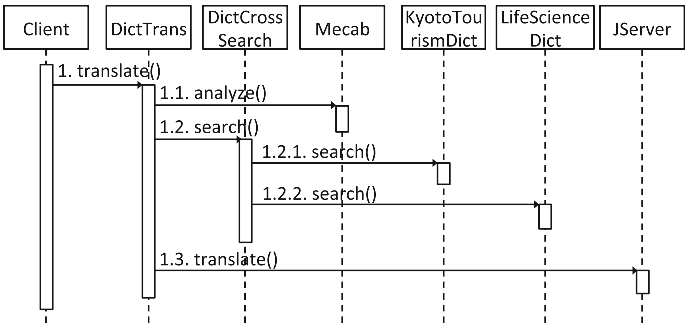


Fig. 7 Call sequence

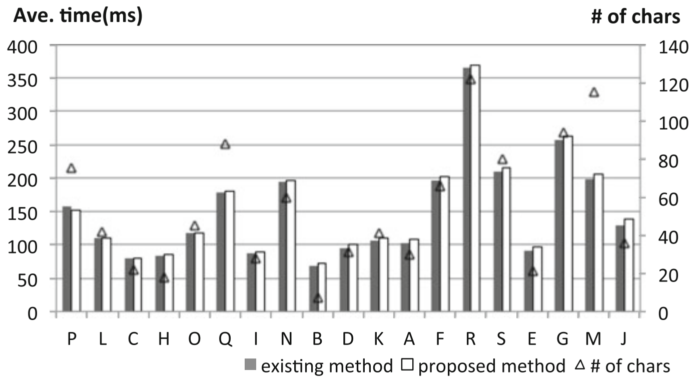


Fig. 8 Execution time (sorted by ascending order of overhead)

each sentence. These execution results are sorted in ascending order of the difference in execution time between the existing method and the proposed method. Although the execution time varies among invocations, which is caused by network delay and the different sentences length, the differences in execution times were small for all sentences and the average of the overhead is 2.12%. Thus the overhead penalty of the proposed method is limited.

6 Related Work

In this research, we use higher-order functions to express service composition consisting of hierarchical service combinations. The proposed method can be regarded as a novel service composition method, where a new composite service is executed by changing the call-relationship of existing composite services based on a given description. Moreover, the proposed method can be considered as a service reuse method since it does not produce new services, and simply uses existing atomic or composite services.

Some previous studies focused on automatic service composition methods. In the service composition methods that extend Golog [11] and McDermott [10], a user can describe his/her requirements in original languages (extended Golog language for former and extended PDDL for latter) and the service composition will be executed by selecting the appropriate services based on the description. However, when a user wants to conduct hierarchical service composition where other composite services are bound as a part of the overall composite service, these methods demand that concrete variations of composite services must be registered beforehand. In contrast, our method supports the description of hierarchical service composition without registering concrete composite service variations. Therefore, a user can invoke significantly many service composition variations without registering them beforehand and thus avoid paying the costs of managing them. QoS-based service selection [9] is one way to select services used in composite services; it involves the calculation of qualities of services that users prefer. The need of the user may change depends on the situation such as the domain of use or languages of other participants. The method proposed herein allows language service users to realize own service composition at runtime by calculating QoS and selecting services based on the results by using such selection technologies in combination.

The effort needed to improve the reusability of service composition has also been reported in some previous studies. For example, Fragmento [14] focuses on the reuse of process descriptions by extracting and sharing fragments of process descriptions that realize certain functions so that users can reuse them by searching a repository. In order to deploy composite services that use process fragments, the user must include the fragment of process description and related definitions such as WSDLs or XML Schemas into the services. If the fragment has some bugs, user must redo inclusion and deployment for all services that use the fragment. Our method uses existing composite services themselves as the unit of reuse and no modification is needed it, so we can receive the benefit of bug fix or other improvements of services without any additional works.

7 Discussion

As the proposed method provides for hierarchical service composition descriptions at runtime, we can now execute various variations of composite services without deploying new composite services. Moreover, developers can improve the reusability of composite services by providing fine granularity services that focus on simple problems. For example, developers can design the fallback function as a composite service that invokes alternate services when the main service fails where the invocation interface is the same for the main service and for the alternate services. When the invocation of the main service fails, this fallback composite service invokes alternate services until it succeeds or no more alternate service exists, and returns the result of successful invocation or error when no services succeed. Users can use this composite service by specifying a service with high quality and low availability to be the main service and services with relatively low quality and high availability to be the alternate services. Furthermore, as more services are created that offer fine granularity and high reusability, the more variations created by the combinations of these fine granularity services can be generated, and then other service composition methods, such like automated service composition, can become more robust.

Moreover, we can easily customize the service composition based on user requirements. Because services run independently from each other and provide certain functions to all users equally, it is difficult to change the behavior of existing services based on the requirements of each user or propagate information across services that is used in just one service composition. By using the proposed method, service composition can be customized flexibly by changing its construction based on the hierarchical description from different users. Since the customization of service composition is realized for each execution request, it does not affect the execution of requests from other users.

8 Conclusion

To well support the changing situations in which multi-language activities are used, service infrastructure must create various new services quickly by aggregating various services based on standardized invocation interfaces of them. In this chapter, we proposed a hierarchical service composition description and applied it to a real service infrastructure to dealt with this issue. The main contributions of this chapter are:

1. Propose a hierarchical service composition description by introducing higher-order functions.
2. Implement it by extending an existing service infrastructure.

For hierarchical service composition description, we introduced the *bind* function which can assign an atomic service or a variation of composite services to the service invocation in a composite service. This allows us to create service composition variations at runtime. For implementing the proposed method, we introduced AOP for modularizing codes by extending existing service infrastructures so as to reduce the dependency between workflow execution engine and our implementation. This extension enables composite service execution to recognize the hierarchical service composition description and transport the description as a part of the service invocation message.

Moreover, we applied the proposed method to an existing service infrastructure: the Language Grid, which is agglomeration of language resources and services, and evaluated the effect of the number of variations our method can realize under domain dependent constraints. We further evaluated the overhead of our method in terms of execution time. The results showed that various service compositions can be realized at runtime and the overhead of our method does not exceed 2.12%.

Acknowledgements This research was supported by a Grant-in-Aid for Scientific Research (S) (24220002, 2012–2016) and a Grant-in-Aid for Young Scientists (A) (17H04706, 2017–2020) from Japan Society for the Promotion of Science (JSPS). Most of this work was done while the first author was a researcher at Department of Social Informatics, Kyoto University.

References

1. Bentley, R., Csillaghy, A., Aboudarham, J., Jacquey, C., Hapgood, M., Bocchialini, K., Messerotti, M., Brooke, J., Gallagher, P., Fox, P., Hurlburt, N., Roberts, D., Duarte, L.S.: Helio: the heliophysics integrated observatory. Adv. Space Res. **47**, 2235–2239 (2011)
2. Donvito, G., Vicario, S., Notarangelo, P., Balec, B.: The Biovel Project: robust phylogenetic workflows running on the grid. In: EGICF12-EMITC2, pp. 26–30 (2012)
3. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education India, Delhi (2005)
4. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989)
5. Ide, N., Pustejovsky, J., Cieri, C., Nyberg, E., Wang, D., Suderman, K., Verhagen, M., Wright, J.: The language application grid. In: Chair, N.C.C., Choukri, K., Declerck, T., Loftsson, H., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., Piperidis, S. (eds.) Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14), pp. 22–30. European Language Resources Association (ELRA), Reykjavik, Iceland (2014)
6. Ishida, T. (ed.): The Language Grid: Service-Oriented Collective Intelligence for Language Resource Interoperability. Springer Science & Business Media, Berlin (2011)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of Aspectj. In: European Conference on Object-Oriented Programming, pp. 327–354. Springer, Berlin (2001)
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming, pp. 220–242. Springer, Berlin (1997)
9. Lin, D., Shi, C., Ishida, T.: Dynamic service selection based on context-aware QoS. In: 2012 IEEE Ninth International Conference on Services Computing, pp. 641–648 (2012)
10. McDermott, D.: Estimated-regression planning for interactions with web services. In: Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, pp. 204–211 (2002)
11. McIlraith, S., Son, T.C.: Adapting golog for composition of semantic web services. In: Proceedings of the 8th International Conference on Knowledge Representation and Reasoning, pp. 482–493 (2002)
12. Murakami, Y., Lin, D., Ishida, T.: Service-oriented architecture for interoperability of multilanguage services. In: Buitelaar, P., Cimiano, P. (eds.) Towards the Multilingual Semantic Web, pp. 313–328. Springer, Berlin (2014)
13. Murakami, Y., Lin, D., Tanaka, M., Nakaguchi, T., Ishida, T.: Service grid architecture. In: Ishida, T. (ed.) The Language Grid: Service-Oriented Collective Intelligence for Language Resource Interoperability, pp. 19–34. Springer, Berlin (2011)

14. Schumm, D., Dentsas, D., Hahn, M., Karastoyanova, D., Leymann, F., Sonntag, M.: Web service composition reuse through shared process fragment libraries. In: International Conference on Web Engineering, pp. 498–501. Springer, Berlin (2012)
15. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva de la Hidalga, A., Balcazar Vargas, M.P., Sufi, S., Goble, C.: The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic Acids Res. **41**(W1), W557–W561 (2013)