Web Semantics: Science, Services and Agents on the World Wide Web 7 (2009) 332–343

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

Automated generation of composite web services based on functional semantics

Dong-Hoon Shin[a], Kyong-Ho Lee[a,*], Tatsuya Suda[b]

[a] Department of Computer Science, Yonsei University, Seoul, Republic of Korea
[b] School of Information and Computer Science, University of California, Irvine, CA, USA

ARTICLE INFO

Article history:
Received 3 November 2008
Received in revised form 11 February 2009
Accepted 13 May 2009
Available online 6 June 2009

Keywords:
Web services
Composition
Ontology
Functional semantics
Graph model

ABSTRACT

Most of studies on the automated generation of composite Web services create composite services by chaining available services' inputs and outputs, but do not consider their functional semantics. Therefore, unsatisfied results may be generated against users' intentions. Furthermore, the time complexity is very high since every possible combination of available services should be considered. To resolve these problems, we propose a composition method that explicitly specifies and uses the functional semantics of Web services. Specifically, the proposed method is based on a graph model, which represents the functional semantics of Web services.

1. Introduction

With the spread of Web services in various fields, there is a growing interest in building a composite Web service, which supports high-level business processes. A composite Web service provides more complicated functions than can be provided by a single service by combining multiple services. However, as the number of Web services increases and user requests are becoming more and more complex, it is difficult and time consuming for a user to find the desired services. Furthermore, it is becoming harder and harder to manually build a composite Web service.

To resolve the difficulty of manually building a composite Web service, there have been many studies on the automated composition of Web services [12]. Previous works specify the semantic description of a service using service specification languages such as OWL-S [14], WSMO [21], and SAWSDL [13] based on a domain ontology, which is built by an ontological language such as OWL [20]. Likewise, a user's request is specified in terms of inputs, outputs, preconditions and effects. From these specifications, they automatically build composite Web services using various AI (Artificial Intelligence) techniques.

Previous methods construct composite Web services, which take the inputs entered by a user and return the outputs requested, by repeatedly finding and chaining appropriate services. When two services are chained, the preceding service should satisfy the precondition of the following service. Generally, two services with identical inputs, outputs, preconditions, and effects are regarded as identical services. Therefore, if a composite Web service satisfies the inputs, outputs, pre-conditions, and effects requested by a user, it is regarded as satisfying the user requirement. However, when component services or user requirements do not have pre-conditions and effects, the conventional methods may generate composite Web services, which differ from a user's intention, as shown in Fig. 1.

Fig. 1 illustrates how inappropriate composite services can be generated. A user is looking for a travel service, which takes duration, destination, and the number of passengers as input, and returns travel packages, which include transportation, accommodation, itinerary, and travel expense as output. Since the service is requested without specifying its precondition and effect, a service composition system may construct a conference search service, which finds the information about conference schedules such as duration, place, and registration fee. This does not match the user intention.

In general, two services with the same IOPE (input, output, precondition and effect) are considered to be identical each other. However, information services, which provide certain information to users, may not need any preconditions and have no effects after execution. If the precondition and effect of services are not specified, their functionalities may be different although their inputs and outputs match each other, as shown in the example of Fig. 1. Previous works based on IOPE cannot compose services correctly, which do not specify their preconditions and effects. Furthermore, previous works have exponential time complexity in proportion to the number of available services, since they have to consider every possible combination of available services.

* Corresponding author. Tel.: +82 2 2123 5712.
E-mail addresses: dhshin@icl.yonsei.ac.kr (D.-H. Shin), khlee@cs.yonsei.ac.kr (K.-H. Lee), suda@ics.uci.edu (T. Suda).

1570-8268/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.websem.2009.05.001

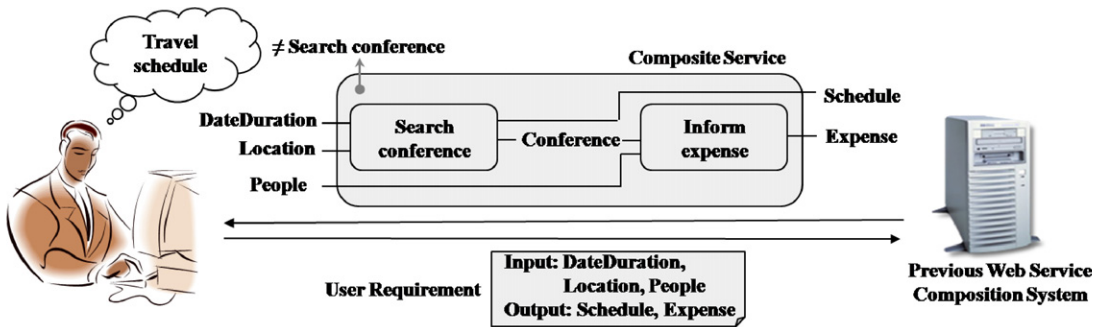

Fig. 1. An example of Web service composition without considering functional semantics.

To improve the functional correctness and time complexity of service composition, this paper proposes an efficient method for composing Web services. The proposed method explicitly specifies the functional semantics of a Web service based on a domain ontology. WSMO [21] defines the functional semantics of a Web service as the formal description of the service functionality, describing what a service can offer to its clients when it is invoked. By the definition, the description includes the capability and categorization of a service. Compared with this, in this paper, we define the functional semantics of a service as describing what a service actually does. In our approach, the service functionality of a service is represented by a pair of its action and the object of the action. The information about services is organized and stored in a proposed two-layer graph model. Given a user request, we search for composition paths in the graph model and construct a composite service from the paths discovered.

To evaluate the performance of the proposed method, we have conducted several experiments in terms of the speed and accuracy of composition. The experimental results show that the proposed method is faster and more accurate than the conventional graph-search based method. Our method improves the functional correctness of composite Web services by considering the functionality of the Web service itself and reduces the time complexity of the composition process by considering only combinations of functionally related services.

The remainder of this paper is organized as follows. Section 2 describes the features and drawbacks of previous works concerning automated composition of Web services. Section 3 explains how we specify the functional semantics of Web services and organize the specifications in the proposed two-layer graph. Section 4 gives a detailed explanation of the proposed composition method. In Section 5, the performance of the proposed method is evaluated and analyzed. Finally, Section 6 summarizes the conclusions and discusses opportunities for future works.

2. Related works

In general, a composite Web service corresponds to a state transition system, which has multiple states, arcs and available actions in certain states and represents transitions from an initial state accepting user inputs to a final state providing requested outputs and effects as shown in Fig. 2. In the transition system, applicable actions correspond to available services. If pre-conditions and required inputs of an action are satisfied in a certain state, transition from that state to another state can be made by applying the action. Most studies on automated composition of Web services try to resolve the composition problem by converting it into the problem of finding adequate transition systems. For this reason, various AI techniques such as HTN (Hyper Task Network) planning, contingency planning, constraint logic programming and linear logic theorem proof are used [3,6,7,9,11,15,19].

Rao et al. [6] convert service specifications and user requirements into axioms and theorems of linear logic, respectively, and then find an appropriate composite service through theorem proving. Agarwal et al. [19] classify Web services as their interfaces and build a composite service through logical and physical composition. In the logical composition step, they support building a composite service, including conditional branches, by applying contingency planning. Akkiraju et al. [9] use semantic matching, which considers domain dependant information as well as domain independent information, and improves semantic precision of generated composite services by considering semantic ambiguity during the planning process. Kona et al. [15] use constraint logic programming to discover a Web service and build a composite service. Their method checks whether requested outputs and effects are reachable from the provided inputs and preconditions using available services. Then they build an appropriate composite service based on the reachability.

However, the methods mentioned above assume that every available service has pre-conditions and effects. Therefore, if information services are necessary to build a composite Web service, their method must build the composite service using only I/O parameter matching. There may be multiple services with quite different functionality, although they have the same type of input and output parameters. The resulting composition may not satisfy the user's intention. In addition, since all possible combinations of available services must be considered in their composition processes, they have high time complexity.

Meanwhile, Alkamari et al. [1] note that semantic Web service composition (SWC) approaches based on AI techniques have limita-

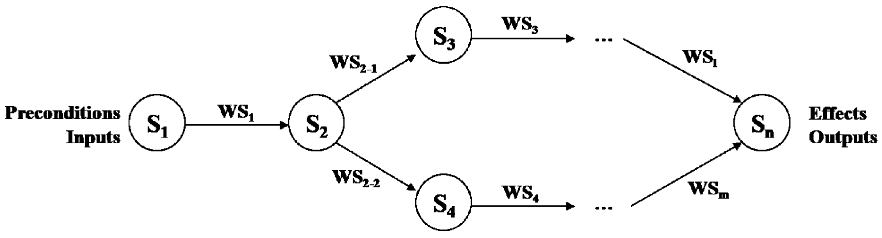

Fig. 2. A composite Web service and its state transition representation.

tions in practical environments with large scale search space due to high computational costs. They propose a signature-based composition of services utilizing WSDL specification and UDDI. Alkamari et al. claim that their method reduces the computational cost of the composition process, without excessive loss of correctness, by narrowing the search space to services which are within a particular business domain. Although their claim is persuasive from a practical point of view, it seems that most approaches based on AI techniques tend to create more accurate composite services for the user requirement. To resolve these problems of correctness and complexity, there have been many other efforts [2–5,8,10,11,16,22,23] in the SWC fields.

To improve the correctness of a Web service search, Ye and Zhang [8] proposed a method that explicitly specifies the functional semantics of services. They specify a service and a user requirement using object, action and constraints as well as input and output parameters. Utilizing this information, they find a service to satisfy the user requirement. However, they don't consider how the specification of functional semantics can be applied to service composition.

Liu et al. [23] proposes a Web service model in which inputs and outputs of service are expressed using RDF graph pattern, as well as a domain ontology. They improve the correctness of composite services without preconditions and effects using semantic propagation based on graph substitution. Thakker et al. [2] try to improve the accuracy of automatic matchmaking of Web services by taking into account the knowledge of past matchmaking experiences for the requested task. In their method, service execution experiences are modeled using case based reasoning. These two methods are helpful for improving the correctness of composite service, but their method is still not free from the complexity problems of the composition process.

Unlike previously mentioned works, some studies [4,5,10,16,22] explicitly or implicitly notice the time complexity problem of the composition process. Hoffmann et al. [5] mention that the time complexity problem of SWC stems from the combinatorial explosion of available services, and that the problem can be resolved by using their polynomial computation of a successor state and heuristic function. However, they fail to suggest an appropriate heuristic function, leaving it for future work.

Gamha et al. [22] and Li et al. [4] classify available services according to their functionalities, where each set of services with the same functionality is called a community service or meta-service, respectively. They build abstract composite services which consist of community services (or meta-services), and then substitute each community service with concrete services. Although the goal of their method is to improve the dynamism of composition, it is also helpful to improve the time complexity of the composition process by reducing the search space.

Hashemian and Mavaddat [16] store I/O dependencies between available Web services in their dependency graph, and then build composite services by applying a graph search algorithm to the graph. In their graph, each service and I/O parameter is represented as a vertex. Also, input to a service and output from a service are represented as incoming and outgoing edges, respectively. Using their dependency graph, Hashemian and Mavaddat can only search connectable services, and this makes it possible to have low time complexity, compared with previous AI based approaches. However, they cannot guarantee that generated composite services correctly provide the requested functionality, since they only consider matching and dependencies between input and output parameters without considering functional semantics.

Meanwhile, Sirin et al. [3] and Thiagarajan [11] propose a Web service composition method which considers functional semantics of services and improves time complexity of the composition process based on HTN planning. HTN predefines how every service should be chained, composed or decomposed. Generally, it is impossible to find out the functionality of every service and the composition or decomposition relations between whole services and then use that information to construct a HTN. Their approach is appropriate to find an execution path, which processes a given input in a large composite service. However, it is not suitable for general Web service composition, which provides new functionality by composing multiple services from a user's request.

This paper proposes a Web service composition method which guarantees functional correctness of Web service composition, including information services, and also improves the time complexity of the composition process by considering only functionally related services during the composition process, similar to the method of Hashemian and Mavaddat. For these purposes, the proposed method specifies the functional semantics of a service with an object and an action similar to the method of Ye and Zhang, and extends the dependency graph of Hashemian and Mavaddat so that the graph includes functional semantics as well as I/O dependencies between available services.

3. Specification and organization of services

As shown in Fig. 3, our method explicitly specifies functionalities of available services as well as I/O types, and stores this information in the proposed graph model, a service relation graph. Composite services which meet a user requirement are automatically built by searching the graph.

In this section, we present a method to specify the functionality of a Web service using a domain ontology, which consists of data ontology and action ontology. We describe how a service relation graph is constructed from a domain ontology and service specifications using functional semantics.

3.1. Specifying functional semantics

Many Web services that provide useful information are specified by I/O without pre-conditions and effects. However, I/O cannot sufficiently describe the functionality of the service itself. Function-

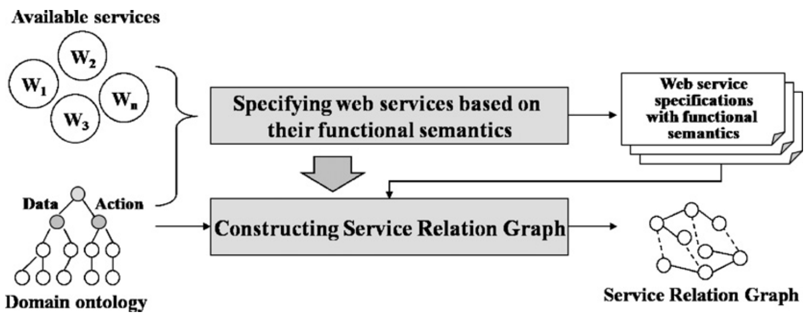
Available services
W1
W2
W3
Wn
Specifying web services based on their functional semantics
Web service specifications with functional semantics
Data
Action
Domain ontology
Constructing Service Relation Graph
Service Relation Graph

Fig. 3. Specifying services and constructing a service relation graph.

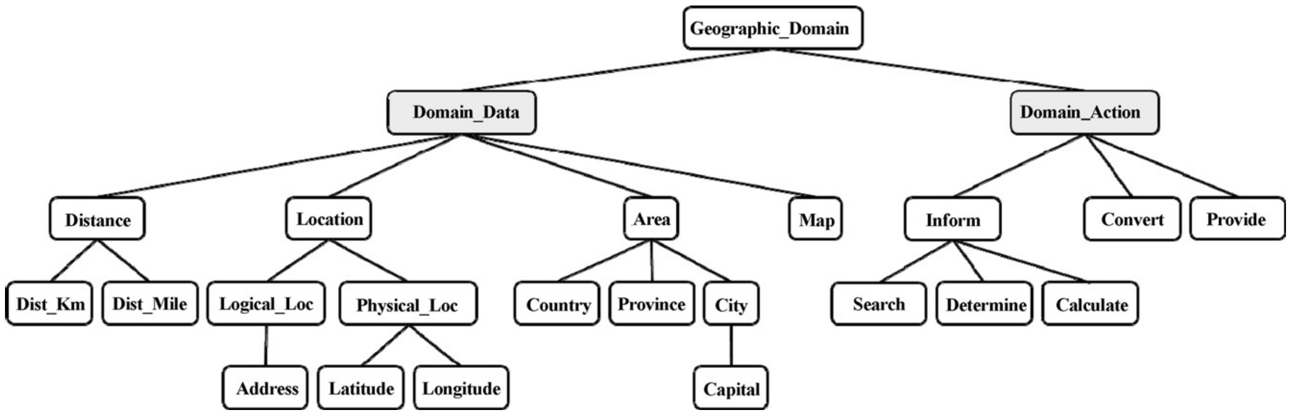

Fig. 4. An example of a domain ontology.

alities which are provided by two services may differ from each other, although the I/O types of the services are the same. Therefore, when a requested service requires to one or more information services, previous works cannot guarantee that the composed service will provide the requested functionality. As shown in Fig. 1, if a service is composed by chaining input and output parameters of available services without considering the functional semantics of the service itself, the composed service may not provide the expected functionality, even if it provides the requested outputs.

To resolve this problem, this paper uses a method which explicitly specifies the functionality of Web services. The method, similar to Ye and Zhang's, describes the functionality of a service as a performed action and an object of that action. For example, the functionality of a Web service[1] which calculates distance can be specified as {*Calculate, Distance*}.

- *Service functionality* = {*action, object*}
  A service is specified by its functionality and the sets of I/O as follows. For example, a service, which calculates the distance between two cities in km, can be specified as {{*Calculate, Distance*}, {*City, City*}, {*Dist_km*}}.
- *Web service* = (*service functionality, input set, output set*)

  The service specification of the proposed method may not correctly specify a service or a user's intention. There are some cases where two services have an identical description by the proposed specification method but provide different functions in reality. For example, one service may determine the geographic coordinates of a city while the other determines the geographic coordinates of a post office in the city. These services may not be distinguished by the proposed specification method. Although our method has a limitation, it is simple and useful means to directly express what a service actually does.

  To specify a Web service, this paper also uses a domain ontology which separately defines domain data and action. Domain data defines concepts which are used to specify I/O and objects of services within a domain. Domain action defines concepts which are used to specify actions performed by services within a domain. In other words, I/O and an object of a service are mapped to concepts of domain data, and an action of the service is mapped to a concept of domain action.

Generally, there are many actions in a domain which can be defined in a variety of manners. However, in order to effectively reflect functional semantics into a composition process and facilitate and generalize the process, it is essential to define domain action in a consistent manner for every domain. To do that, this paper proposes and uses the following four constraints for defining domain action.

- A verb is used to define the concept of an action.
- When there are multiple verbs describing an action, only one of them must be used to define the concept of the action.
- An action described as a concept can only be provided by combinations of one or more its children.
- An action described by a concept cannot be provided by combinations of any other concepts that are not its children.

Fig. 4 shows an example of a domain ontology, which consists of data and action ontologies. Unlike general domain ontology, the proposed ontology contains sub-ontology for specifying the action of a service. This action ontology defines action concepts, which explain a semantic hierarchy of functions and are used to specify the functionalities of services in a specific domain. The ontology should be defined by an ontology expert who has a good knowledge of a domain. Using the action ontology, service providers can explicitly specify the functionally of services and users can also effectively express their requirement. Meanwhile, the North American Industry Classification System (NAICS) [17] and United Nations Standard Products Services Code (UNSPSC) [18] are standards for the classification of products and services. They are similar to the proposed ontology in terms that a lot of their classifications of services are defined using verbs and nouns. They can be used as domain ontology for the proposed composition method after appropriate modifications.

3.2. *Organizing a service relation graph*

To facilitate building a composite Web service, service specifications are stored in the proposed graph model, which is called the SRG (Service Relation Graph). The SRG consists of three components: a data dependency graph, which represents data dependencies between services, an action graph, which represents relations between domain actions, and mappings between the two graphs. Table 1 shows an example of available services which are specified by the proposed method using the geographic ontology of Fig. 4. We borrowed this example from Hashemian and Mavaddat and slightly modified it into a suitable form. Fig. 5 shows the SRG which is constructed with services from Table 1.

A data dependency graph is constructed by chaining available services and their I/O concepts mapped to the domain data on the domain ontology. It is similar to the dependency graph which was

[1] Generally, a Web service can have multiple operations with different functionalities, so it should be specified by a set of operations. However, this paper assumes every service has a single operation for convenience of explanation.

Table 1
An example of services in a geographic domain.

Service name	Specification	Functionality
C_1	{{Inform, Capital}, {Country}, {Capital}}	Outputs the capital of a country
C_2	{{Calculate, Distance}, {City, City}, {Dist_km}}	Calculates a distance between two cities in km
C_3	{{Determine, Physical_Loc}, {City, Province}, {Latitude, Longitude}}	Determines the latitude and longitude of a city
C_4	{{Convert, Distance}, {Dist_Mile}, {Dist_km}}	Converts a unit of distance from miles into km
C_5	{{Convert, Distance}, {Dist_km}, {Dist_Mile}}	Converts a unit of distance from km into miles
C_6	{{Search, Logical_Loc}, {PhNo}, {Address}}	Searches an address from a phone number
C_7	{{Provide, Map}, {Address}, {Map}}	Provides a map of a city

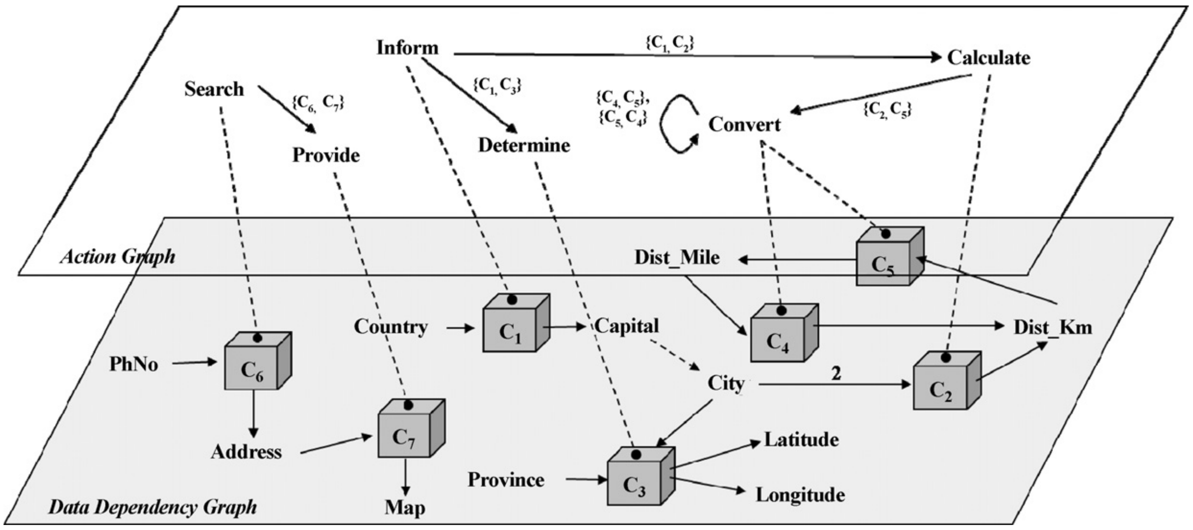

Fig. 5. An example SRG constructed from Table 1.

proposed by Hashemian and Mavaddat. However, our data dependency graph is different from their graph in that every service becomes a node, as well as an I/O concept. This makes it easy to find a specific service that accepts a particular set of I/O. In our graph model, each service node is connected to its I/O concepts with incoming and outgoing edges. Like C_2 of Fig. 5, if a service has more than one input (or output) that is mapped to the same concept, their number becomes the label of an incoming (or outgoing) edge between the service and the concept node. Also, like City and Capital in Fig. 5, if there is a parent–child relation between two concept nodes in the domain ontology, an edge from the child concept node to the parent node is created on the graph.

An action graph is constructed with action concepts of domain action. In the graph, every action concept becomes a node. Every service node of the data dependency graph is mapped to an action concept which the service provides. If there is an edge between two adjacent service nodes on the data dependency graph, a new edge is created on the action graph between the nodes which are mapped to the service nodes. The new edge has the same direction as the edge between the service nodes, and a set which consists of the services becomes its label.

4. Composition of services

The proposed Web service composition method consists of three steps: constructing sets of candidate Web services, path finding on the service graph, and building composite services as shown in Fig. 6.

This section explains each step in detail with appropriate examples. Four kinds of candidate service sets are introduced, and the method used to construct these service sets from user requirements is explained in Section 4.1. A method which finds composition paths to build composite services on the SRG is presented in Section 4.2. In Section 4.3, a method which builds composite services from composition paths found in earlier steps is explained. Finally, our composition algorithm is introduced.

4.1. Finding candidate services

A composite service which meets a user requirement has general constraints which have to be satisfied. The service has to be executable without any other inputs except user provided ones, and has to return requested outputs. Most importantly, the service has to provide requested functionality. Our approach finds candidate services as a starting point to generate composite services which satisfy the constraints mentioned above. Possible candidate services of a composition can be discovered from a user requirement. We classify these candidate services into core service and auxiliary service. This is a conceptual classification. In a practical composition process, core services and auxiliary services are classified again into four types of sets according to their characteristics: CWS_{ia} (Input Acceptable Candidate Web Services), CWS_{oa} (Output Acceptable Candidate Web Services), CWS_{sc} (Single Core Candidate Web Services), and CWS_{cc} (Composite Core Candidate Web Services). In this classification, CWS_{sc} and CWS_{cc} come under core services and CWS_{ia} and CWS_{oa} come under auxiliary services.

A core service is an essential service for satisfying the user-requested functionality. Unlike core service, auxiliary service supports a transformation between I/O types of connected services, rather than directly implementing a requested functionality. In our approach, core and auxiliary services are dynamically determined during runtime based on requested functionality and separately searched on the SRG. This makes our method focus on searching only functionally related services and also helps to reduce the time required to discover available composition paths on the SRG.

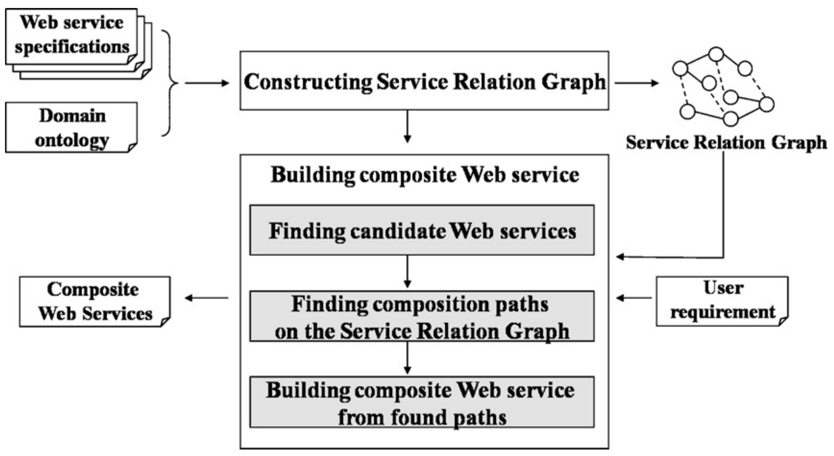
Web service specifications
Domain ontology
Constructing Service Relation Graph
Service Relation Graph
Building composite Web service
Finding candidate Web services
Finding composition paths on the Service Relation Graph
Building composite Web service from found paths
Composite Web Services
User requirement

Fig. 6. The proposed Web service composition method.

For example, assume that a user requests a service which accepts two cities and returns the distance between two cities in mile. The user requirement is specified as {{*Calculate*, *Distance*}, {*City, City*}, {*Dist_Mile*}}. In Fig. 7, there is no single service satisfying the requirement, but the combination of C_2 and C_5 can provide the requested functionality. In this example, C_2, which provides the requested functionality specified by {*Calculate, Distance*}, is a core service and C_5, which supports the transformation between *Dist_km* and *Dist_Mile*, is an auxiliary service.

In our composition process, CWS_{ia} consists of services which accept at least one user input, while CWS_{oa} consists of services which provide at least one output requested. In order to guarantee that a composite service is executable with the user-provided inputs, the composite service has to include at least one of the CWS_{ia} elements. Similarly, a composite service has to include at least one CWS_{oa} element in order to provide the requested outputs. CWS_{sc} is constructed with services which have the same type object and action concepts as those requested. In other words, CWS_{sc} is a set of core services which provide the requested functionality by themselves. If a composite service with an element of CWS_{sc} is executable without any other inputs except the user provided ones and the service provides requested outputs, it can be a solution for the user requirement.

Fig. 7 is an example of candidate service sets that are constructed from a user request ({*Calculate, Distance*}, {*Country, Country*}, {*Dist_Mile*}). Service C_1, which accepts user input '*Country*', is selected as an element of CWS_{ia} and service C_5, which returns the requested output '*Dist_Mile*', is selected as an element of CWS_{oa}. In a similar way, service C_2 provides the requested functionality {*Calculate, Distance*}, thus it is selected as an element of CWS_{sc}.

Meanwhile, CWS_{cc} consists of services whose action concepts are child concepts of those requested and whose object concepts are the same concepts or descendent concepts of those requested. In Section 3.1, we have proposed four constraints for defining domain action which can be performed within a domain. According to the constraints, if an action with a required functionality has children in the domain ontology, the functionality may be provided by a composition of one or more child concepts from the action. This means that if a user requests complex functionality represented as a high-level action concept, the functionality can be provided by a composition of services which have a child concept of the requested action.

Fig. 8 is another example that constructs candidate service sets from a user requirement ({*inform, Location*}, {*Country, Province, PhNo*}, {*Address, Latitude, Longitude*}). In this example, there is no service which has the same action and object concept as that requested. However, there are services which have child concepts of *Inform (Search, Determine,* and *Calculate)* and they become elements of CWS_{cc}.

4.2. Finding composition paths

After finding candidate service sets, every path between action concepts of CWS_{ia} and CWS_{oa} elements, which is called a composition path, is searched on the action graph of the SRG. On the action graph, if there is an edge between two nodes, it means that two service nodes on the data dependency graph are adjacent, and there is an edge between the nodes. Additionally, it is also revealed that the two services can be sequentially composed. On the other hand, if there is no edge between two nodes on the action graph, two services which have those concepts as an action can never be com-

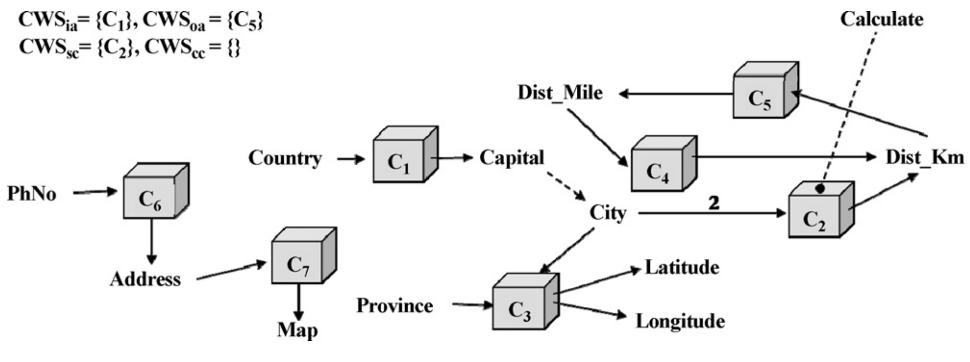

Fig. 7. An example of constructing candidate service sets.

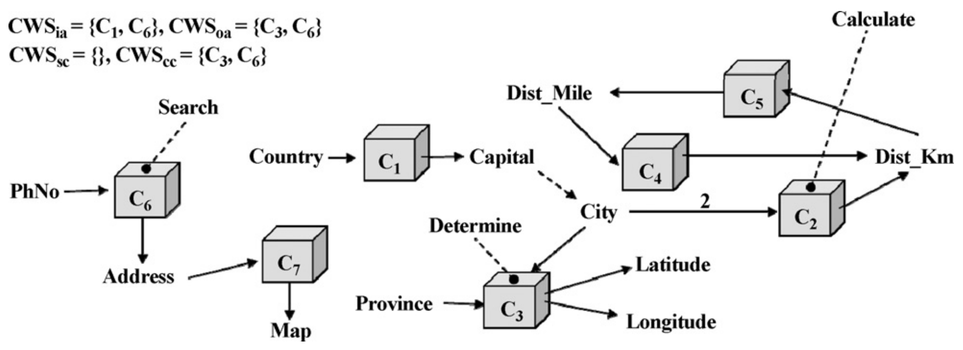

Fig. 8. An example of constructing candidate service sets.

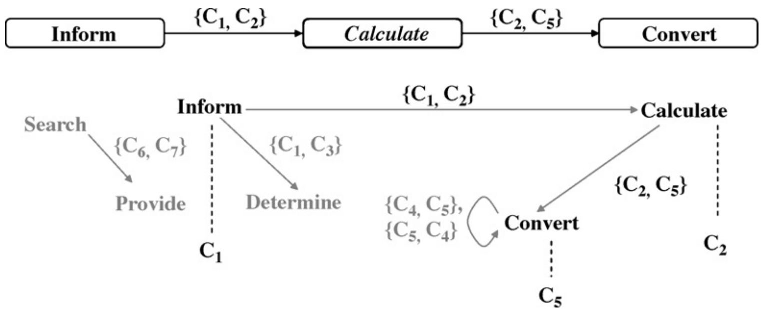

Fig. 9. An example of finding composition paths on the action graph.

posed sequentially. Therefore, if there is no path between action concepts of CWS_{ia} and CWS_{oa} elements on the action graph, any composite Web service which meets a user requirement cannot be built.

As mentioned before, core service is an essential service which provides required functionality for a user. To guarantee that a composite service provides a required functionality, the service must contain a core service. In our approach, CWS_{sc} and CWS_{cc} consist of candidate core services. CWS_{sc} includes services which directly provide requested functionality and CWS_{cc} includes services which may provide requested functionality in combination. Therefore, paths which are searched in the second step should contain at least one action concept from the CWS_{sc} or CWS_{cc} elements.

Fig. 9 is an example of a composition path, which is discovered on the action graph for given candidate service sets in Fig. 7. In Fig. 9, element C_1 of CWS_{ia}, element C_2 of CWS_{oa}, and element C_5 of CWS_{sc} have *Inform*, *Convert,* and *Calculate*, respectively, as their action concepts. The proposed method finds a path which starts from *Inform*, passes through *calculate*, and ends in *Convert* on the action graph.

4.3. Building composite services

This section describes how to build composite services from the paths found. At first, for each composition path, all of the possible service chains are extracted from the labels of the composition path. A service chain is a sequential connection of services and is constructed from the sets that are selected in order from each label of the composition path. A set is selected from each label and its first element should be the same as the last element of the set selected just before. If the composition path consists of one action concept node, each service that provides the action constructs a service chain for itself. Fig. 10 illustrates the extraction of service chains from a composition path. Note that this example uses a composition path, which is not taken from the SRG of Fig. 5. Here, AC_n and C_n within the given composition path represent action concepts and available services, respectively.

After extracting every possible service chains, a set of candidate composite services (CCS) is built by combining the service chains and are optimized by merging common services. A CCS is a composite service, which does not require inputs other than those provided

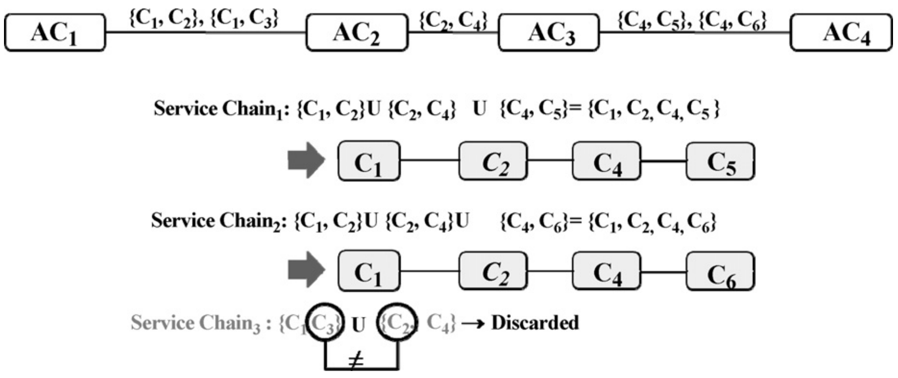
AC1
{C1, C2}, {C1, C3}
AC2
{C2, C4}
AC3
{C4, C5}, {C4, C6}
AC4
Service Chain1: {C1, C2}U {C2, C4} U {C4, C5}= {C1, C2, C4, C5}
C1
C2
C4
C5
Service Chain2: {C1, C2}U {C2, C4}U {C4, C6}= {C1, C2, C4, C6}
C1
C2
C4
C6
Service Chain3 : {C1, C3} U {C2, C4} → Discarded
≠

Fig. 10. An example of extracting service chains from the path found.

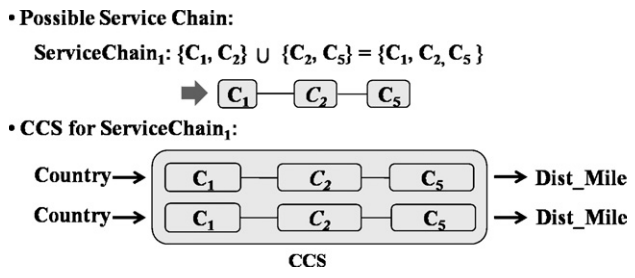
• Possible Service Chain:
ServiceChain$_1$: {C$_1$, C$_2$} ∪ {C$_2$, C$_5$} = {C$_1$, C$_2$, C$_5$}
C$_1$
C$_2$
C$_5$
• CCS for ServiceChain$_1$:
Country
Country
Dist_Mile
Dist_Mile
CCS

Fig. 11. An example of constructing CCS from service chains.

by a user request and satisfy all the outputs of the user request. Consequently, a CCS is returned as a solution after optimizing. A set of CCS is constructed as follows:

- For each input provided by a user, if there is a service chain, which starts with a service taking the input, the service chain is selected as a CCS. If the cardinality of an input is more than one, the input selects service chains as many as the cardinality.
- If a CCS requires inputs other than those provided by a user request, it is discarded.
- If a CCS does not cover all the outputs requested, it is discarded.
- The CCS found is added to the CCS set.
- The above steps are repeated until there are no more combinations of service chains.

Fig. 11 illustrates how a service chain is constructed from the composition path of Fig. 9 and a CCS is constructed from the service chain. In this example, with the user input, *City*, of cardinality 2, the CCS constructed includes two identical service chains.

As the final step of the composition, the proposed method optimizes all of CCSs included in the CCS set by merging common services, which do not need to be executed more than once. This optimization is applied to every pair of the service chains contained in a CCS. The merging of two common services can be allowed only when the CCS still does not require other inputs than those provided by its user after merging. Finally, the optimized CCS set is returned, where the CCS set contains all possible solutions for the user request. Fig. 12(a) is the result after the CCS of Fig. 11 has been optimized. Fig. 12(a) and (b) corresponds to the final solutions to the user requirements of Figs. 8 and 9, respectively.

Meanwhile, in Fig. 12(b), a user might want a service, which returns an address and its latitude and longitude. However, the proposed method may return a composite service, which returns an address and the latitude and longitude of the city, where the address is located. It is approximately what the user is searching for and may be an incorrect answer for the user intention. As mentioned in Section 3.1, this is because a user's intention may not be wholly expressed by the proposed specification method. Given a service request, the proposed method finds all the possible answers. However, they may contain a service, whose functionality does not coincide with the user intention. This problem can be solved by incorporating users' feedback or a more sophisticated service specification. We leave this problem as a future improvement.

5. Experimental results and analysis

Similar to previous works, the proposed method builds a composite Web service by chaining services with the same type of I/O. However, the method also considers the functional semantics of services as well as I/O type. Based on this, functionally unrelated paths are removed in the composition process. Through this, our method builds composite services faster than previous works, and the generated services have high precision on average. To verify this, we have conducted two types of experiments for composition speed and precision of generated composite services, and have compared the results with previous methods. Also, the time complexity of our method has been analyzed.

5.1. Composition speed

Most previous works for the automated composition of Web services have exponential time complexity since they have to consider every combination of available services for creating required composite service. Unlike the previous works, Hashemian and Mavaddat chain services with the same type of I/O and store that information in the form of a graph model when the services are published. Through this, their method can chain possible services and has relatively lower time complexity than the previous methods.

Meanwhile, Sirin et al. built a composite service rapidly using a pre-defined HTN for a certain domain. However, their method is not suitable for building a composite service for providing functionality which did not exist at the design time. This paper examines the composition speed of our method and compares the result with the method of Hashemian and Mavaddat, which has the lowest time complexity among previous works (except the method of Sirin et al.).

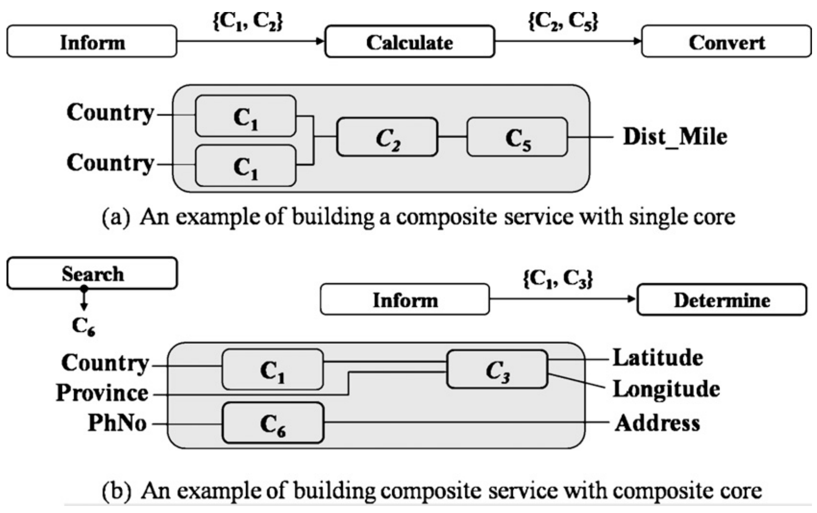

Fig. 12. An example of the composite services constructed from the paths found.

Table 2
Experimental settings for the comparison of composition speed.

	The number of available services	Ontology size		The number of I/O included in a user requirement
		The number of data concepts	The number of action concepts	
Experiment 1	50–250	100	50	5
Experiment 2	150	25–225	13–118	3
Experiment 3	100	50	25	2–10

Generally, a precise evaluation for a service composition method needs to analyze composite services built by the method and measure the execution time of the whole composition process. However, Hashemian and Mavaddat failed to clearly provide a method to combine the composition paths found on their graph, resulting in difficulty building a composite service from the paths. Because of this, we have compared the elapsed time for the path finding of each method, and indirectly evaluated the performance of each method based on the results. Since both methods build a composite service from composition paths found in earlier steps and these processes are very similar, we confirm that the processes are performed with the same time complexity. Consequently our experiment is meaningful for evaluating the composition speed of the two methods, although the experiment does not consider the whole composition process.

Elapsed time for path finding on a graph depends on the graph size and the number of paths to be discovered. In both our method and the method of Hashemian and Mavaddat, graph size is decided by the number of services and the size of the domain ontology. Also, the number of paths that should be discovered is decided by the number of inputs and outputs in the user requirements. Based on these observations, we have conducted three experiments, as shown in Table 2, and compared the experimental results of the two methods.

We have implemented our method and the method of Hashemian and Mavaddat using Java, and then measured the elapsed time to find the composition paths using the time stamp of the system. For fair evaluation, the original implementations of both of the approaches should be compared. However, we could not help implementing the approach of Hashemian and Mavaddata by ourselves for our experiments since we could not obtain the implementation. To conduct fair experiments, we need a sufficient number of services and ontologies with a variety of sizes. However, it is very hard to collect or manually construct appropriate data. For this reason, we randomly generated experimental data which conforms to the conditions of Table 2.

In Experiment 1, we measured the time elapsed to find every composition path on each graph model by increasing the number of services from 50 to 250. In this experiment, a domain ontology which contains 100 data, 50 action concepts and 20 user requirements, where each requirement contains 5 input and output parameters, has been used. Similarly, we have conducted Experiment 2 and Experiment 3 after changing the size of the domain ontology and the number of I/O requirements, respectively. Figs. 13–15 show the experimental results.

Fig. 13 shows that the more available services there are, the more time it takes to find composition paths on the graph. The size of the data dependency graph grows larger as the number of available services increases in both methods. However, the proposed method finds paths not on the data dependency graph but on the action graph, whose size is determined by the number of action concepts. Although the size of candidate service sets grows larger and the number of paths discovered increases as the number of services increases, the elapsed time to find path does not increase rapidly. This is since the size of the action graph is relatively tiny compared with the dependency graph of Hashemian and Mavaddat.

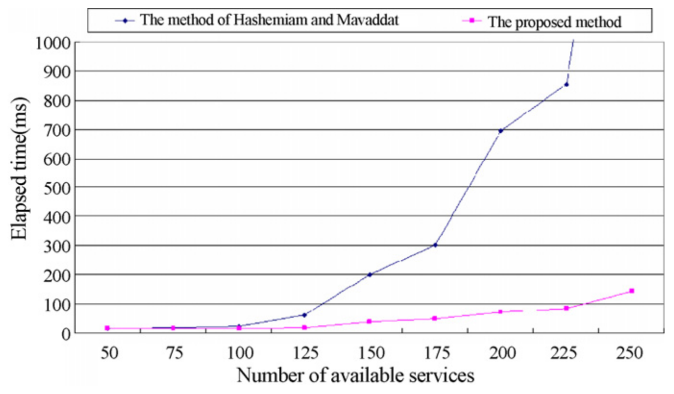

Fig. 13. Experiment 1: Elapsed time to find paths in terms of the number of available services.

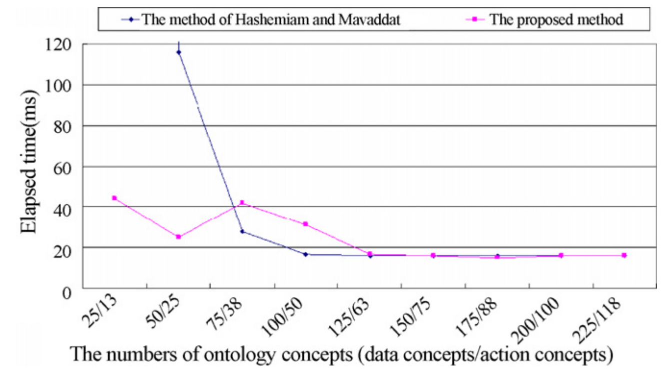

Fig. 14. Experiment 2: Elapsed time to find paths in terms of the size of a domain ontology.

Fig. 14 shows that the elapsed time to find composition paths decreases as the size of the domain ontology grows larger. This result is caused by fixing the number of services to 150. In the method of Hashemian and Mavaddat, the size of the dependency graph is decided based on the number of available services and

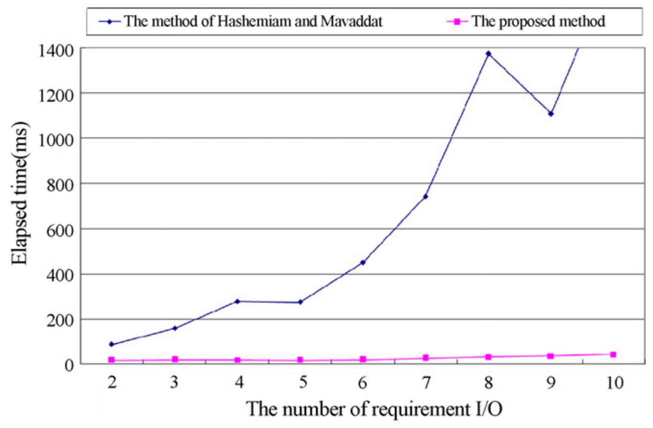

Fig. 15. Experiment 3: Elapsed time to find paths in terms of the number of I/O requirements.

Table 3
Experimental setting to compare precision of generated composite services.

	The number of available services	Ontology size		The number of I/O included in a user requirement
		The number of data concepts	The number of action concepts	
Test data	50–250	100	50	2–5

the I/O concepts of those services. Although the size of the domain ontology increases, the size of the dependency graph is unchanged.

Similarly, in the proposed method, the size of the data dependency graph has nothing to do with the size of the domain ontology. Even if the size of the action graph grows larger as the number of concepts in the domain action increases, it does not affect the elapsed time to find paths, since the number of edges on the action graph is unchanged. Edges on the action graph can be changed only when the number of available services is changed.

However, if the size of domain ontology grows, the connectivity of a service relation graph may decrease since the number of data and action concepts for service specification also increases. The decrease in connectivity means that the probability of finding a solution to a user request decreases. By this reason, it frequently occurs that a composite service which meets a user requirement cannot be built as the number of data and action concepts increases. In this case, path finding is terminated in an early stage, resulting in a decrease in time. Consequently, Experiment 2 is to show how the size of domain ontology affects the failure of service composition in terms of the numbers of available services.

Fig. 15 shows that the elapsed time to find paths increases as the number of input and output parameters increases. The method of Hashemian and Mavaddat searches every path between the input and output concepts of a user requirement on the dependency graph. Therefore, as the number of parameters of a user requirement increases, the number of paths to be discovered increases, resulting in the increment of elapsed time. Similarly, the number of paths to be discovered also increases in the proposed method as the number of parameters increases. However, elapsed time increases only slightly in our method, since the size of the action graph where the searching is performed is relatively tiny.

5.2. Precision of generated composite services

The proposed method uses action ontology which is used to define the functionality of a Web service, as well as data ontology. Using this action ontology, our method does not consider composition paths which consist of functionally unrelated services with a user request in the composition process. Therefore, the proposed method can build more precise composite services for the user's intention compared with previous methods, which build composite services by simply chaining input and output parameters of services. To verify this, we have conducted an experiment to compare the precision of generated composite services by our method and by previous methods.

The experimental data of Table 3 was prepared as follows. Domain ontology was randomly generated and nine sets of available services were generated based on the ontology. Each available service set contains different numbers of services from 50 to 250. We also constructed 15 composite services manually by referring to the ontology and the services. The 15 specifications were used as user requirements. Consequently, for a user request, there is at least a composite service as a solution while the ontology and the available services were randomly generated.

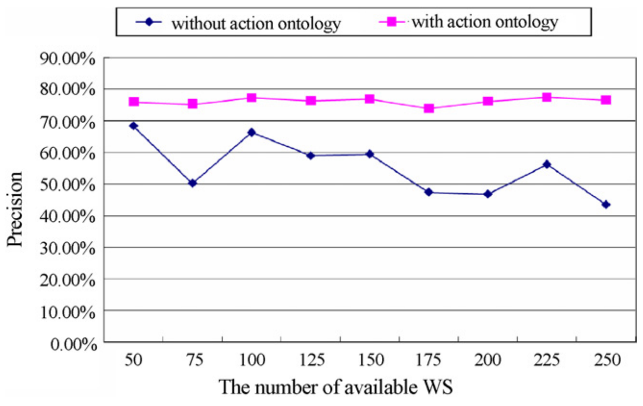

Fig. 16. Performance comparison in terms of precision.

Generally, we use accuracy, which consists of precision and recall, to measure the performance of a composition method. However, it is hard to know how many solutions there are in our experiments since domain ontology and available services were randomly generated. Since we are able to find how close a generated composite service is to a user requirement, we use precision to evaluate the accuracy of the proposed method.

A composite service, which accepts as many user inputs as possible and includes services that provide the similar functionality to a user request, can be considered as a more suitable service for the user intention. Therefore, we calculate the precision of a composite service using Eq. (1), where the weight w_1 and w_2 were set to 0.2 and 0.8, respectively. Fig. 16 presents the result of this experiment.

$$\textit{The precision of a composite services} \\ = (\textit{usage rate of user input}) \times w_1 + (\textit{functional similarity}) \times w_2 \tag{1}$$

The usage rate of Eq. (1) is calculated by Eq. (2), in which inputs with the same type are counted once. Our method and the method of Hashemian and Mavaddat do not construct composite services, which require other more inputs than a user does. As a result, the number of inputs required by a composite service is not more than the number of the inputs provided by a user.

$$\textit{Usage rate of user input} \\ = \frac{\textit{The number of the inputs required by a composite service}}{\textit{The number of user inputs}} \tag{2}$$

The functional similarity of Eq. (1) is calculated by Eq. (3), where *semDist* denotes a minimum distance between the action concepts, only for the descendant concepts of the requested action concept, of component services and the requested action concept on the action graph.

$$\textit{Functional similarity} \\ = \begin{cases} \textit{if a composite service contains a service whose action and object concept is equal to the requirement}' & : 1 \\ \textit{else if a composite service contains a service whose action concept is a descendant concept of the requested} & : \dfrac{0.8}{2^{semDsit-1}} \\ \textit{Else} & : 0 \end{cases} \tag{3}$$

Fig. 16 shows that the precision of the method of Hashemian and Mavaddat generally decreases as the number of services increases. Previous composition methods, which do not use action ontology, generate all possible combinations of available services that are executable by user inputs and provide the requested outputs. Therefore, the result of previous methods includes composite services, which provide quite different functionality from the user's intention, resulting in a low precision.

In addition, the number of services that can be chained increases as the number of available services increases, and the number of possible combinations of available services also increases. However, the number of services functionally unrelated to a user's intention increases more than the number of functionally related services. The result of previous methods includes more and more composite services which do not correspond to the user's intention. Consequently, precision in previous works decreases as the number of available services increases.

Compared with this, the proposed method explicitly specifies functionalities of user requirements and Web services, and then the information is organized and stored in the SRG. By using the SRG, our method can only search composition paths which necessarily include core services to provide requested functionality in the path-finding step. Therefore, the result of our method does not include composite services which consist of functionally unrelated services. This makes our method maintain high precision, regardless of the number of available services.

Meanwhile, although the precision of our method is high on average, it cannot be perfect since composite services with multiple core services are included in the result. Correctness of a composite service with multiple core services can only be judged by the user. We only know that certain combinations of multiple core services can provide the requested functionality since a user describes his/her requirement using high-level ontological conceptualization. This is a drawback, and will be the next research issue for our method. Nevertheless, a composite service with multiple core services is meaningful from a viewpoint that it raises the possibility of finding a required composite service while maintaining high precision in every case.

Since the data sets used in our experiments are randomly generated, the results may not completely guarantee that the proposed method is superior to previous methods. However, the experimental results show that our method is more able to be scaled and has faster composition speed than previous methods in the domain that has fewer action concepts than data concepts. Also, the proposed method builds more correct composite services than previous methods regardless of the number of available services within a domain.

5.3. Time complexity

Many challenges remain to apply automated composition of Web services to the real world. Of these challenges, one of the greatest is the time complexity of a composition algorithm. Generally, for a given request, finding a valid composition that provides the expected functionality is a very complex problem. Most studies on automated composition create a composition by simply chaining inputs and outputs of services. This results in high time complexity, since every possible combination of available services must be considered. In this section, we analyze the time complexity of the proposed composition method and compare it with the complexity of Hashemian and Mavaddat.

As mentioned before, the proposed composition algorithm consists of three steps: constructing candidate service sets, discovering composition paths on the SRG and building composite services. In this section, we calculate the time complexity of each step and then determine the total time complexity for the entire process. To calculate and analyze a time complexity, we assume an upper bound for some parameters according to the analysis of Hashemian and Mavaddat: i and o for the numbers of inputs and outputs in the user requirements, t and a for the numbers of objects and actions in a user requirement. A time complexity for each step is described in Eqs. (4)–(6), and the time complexity of the entire process is calculated with Eq. (7).

When constructing candidate service sets, the four sets CWS_{ia}, CWS_{oa}, CWS_{sc} and CWS_{cc}, should be found based on a user requirement. This process can be done in a constant amount of time by using a hash table of ontological concepts. Therefore, to find all elements of CWS_{ia}, there are i hash table lookups. Similarly, constructing CWS_{oa}, CWS_{sc} and CWS_{cc} requires o, $t+a$, and $t+a$ lookups for the hash table, respectively. As a result, the complexity of this step has constant order.

$$\begin{aligned}&\textit{Complexity of constructing candidate service sets} = O(c),\\ &\textit{where c denotes a constant}.\end{aligned} \tag{4}$$

In the second step, every path on the action graph between action concept nodes implemented by elements of CWS_{ia} and CWS_{oa} is searched for, where all paths should pass through at least one action concept node implemented by elements of CWS_{sc} or CWS_{cc}. Path finding on a graph is a linear process based on the size of the graph and the time complexity $O(v+e)$, where v and e are the number of nodes and edges in the graph. Meanwhile, our action graph may have cycles, and these can disturb termination of path finding. To ensure that path finding terminates in all cases, we can use a length limit for the paths found. By using a constant limit of l for the length of the paths, we only have to check $O(v^l)$ paths in the worst case. Generally, to find every path on a graph requires a time complexity $O(v^l(v+e))$, where the length limit of each path is l. However, our algorithm searches for paths which pass through at least one specific action concept node, which are implemented by elements of CWS_{sc} or CWS_{cc}. Therefore, the complexity of the second step is $O(v^{l-1}(v+e))$, as in Eq. (5).

$$\begin{aligned}&\textit{Complexity of discovering composition paths on the SRG}\\ &\quad = O(v^{l-1}(v+e)),\\ &\textit{where v and e are the number of nodes and edges, respectively, in}\\ &\quad\textit{the action graph of SRG, and l is a limit on the length of the paths}.\end{aligned} \tag{5}$$

In the last step, for each path found, possible sequential service chains are sought and then possible compositions which provide requested output from the user input are built from combinations of these service chains. Since the number of paths found in earlier steps is v^{l-1} and the length limit of each path is l, the process of searching for sequential service chains can be done with complexity $O(v^{l-1})$. Building valid compositions from sequential service chains requires more time. To find every possible composition, every combination of a service chain which ends with a service that returns each requested output should be checked. For each requested output, there are v^{l-1} service chains which end with a service that returns said output in the worst case. Therefore, building valid compositions from service chains has a complexity of $O(v^{o(l-1)})$.

$$\begin{aligned}\textit{Complexity of building composite services} &= O(v^{l-1}) + O(v^{o(l-1)})\\ &= O(v^{o(l-1)}).\end{aligned} \tag{6}$$

Using Eqs. (4)–(6), the time complexity of the whole composition process is calculated in Eq. (7) and polynomial in the graph size and the number of user inputs.

$$\begin{aligned}\textit{Complexity of the entire process} &= O(c) + O(v^{l-1}(v+e)) + O(v^{o(l-1)}) \\ &= O(v^{o(l-1)}) \quad (\textit{where } o \geq 2)\end{aligned} \tag{7}$$

In their paper, Hashemian and Mavaddat determine that the time complexity of their method is $O(v'^{l}(v'+e'))$, where v' and e' are the number of nodes and edges in the graph, respectively. The complexity of their method is also polynomial in the graph size and seems to have lower complexity than that of the method proposed here. However, their complexity was not calculated from the whole composition process, but from the process of path finding. They discuss the process of path finding to determine the complexity of the whole process, since building the composition setup based on valid paths is very straightforward. However, if there is more than one valid path for each input–output pair, every combination of paths found should be checked to find all the possible compositions. It is reasonable that their complexity is $O(v'^{ol})$ for the whole composition process.

Moreover, their dependency graph consists of nodes with the same number of data concepts on domain ontology, while the action graph of the proposed method consists of nodes with the same number of action concepts. Generally, domain ontology has fewer action concepts than data concepts, as shown in Fig. 2. Therefore, it is reasonable that v and e are smaller than v' and e' in most cases. Consequently, it is expected that our proposed method is able to find possible composite services faster than the method of Hashemian and Mavaddat. This also suggests that the proposal has a lower time complexity than most previous works, which do not consider the functional semantics of the Web service itself.

6. Conclusions and future works

Most previous composition methods based on AI techniques do not consider functional semantics, and consequently, cannot sufficiently capture the functionalities of information services without preconditions and effects. They may generate composite services which do not satisfy users' intention when the generated composite service requires information services as its component service. To resolve this problem, we proposed a composition method that explicitly specifies the functional semantics of a service itself.

In our method, Web service specifications are stored in the proposed graph model. A composite service which meets a user requirement is automatically built by searching the graph. Based on the user requirement, available services are dynamically classified into core services and auxiliary services at run-time. Consequently, a composite service contains a core service, which provides the requested functionality. The proposed method improves the correctness of the composite services generated. Additionally, the proposed method has lower time complexity than previous works since our composition process is mostly performed on the action graph, which includes a relatively small number of nodes. Functionally unrelated services are automatically excluded from the searching process.

To evaluate performance, we conducted several experiments in terms of composition speed and precision. The experimental results for composition speed shows that our method generates composite services faster than previous methods when the number of action concepts is smaller than the number of data concepts. Also, the composite service generated by our method has higher precision than those of previous methods, which do not consider functional semantics of Web services.

The proposed method specifies the functionality of a Web service as a one-to-one relationship of action and object. However, there may be cases where two services have an identical description by the proposed specification method, but may function differently in reality. In addition, a user may have to use a combination of multiple actions and objects to specify certain functionality. To resolve this problem, we will develop a more sophisticated method of specifying services. We have a plan to enhance the proposed method in order to improve the time complexity and precision of the composition process. We will also consider non-functional attributes of services such as QoS and user-defined constraints in the composition process.

Acknowledgement

This research was supported by the Seoul R&BD Program (10561), Korea.

References

[1] A. Alkamari, H. Mili, A. Obaid, Signature-based composition of Web services, in: Proceedings of MCETECH'08, IEEE Computer Society, Washington, DC, 2008, pp. 104–115.
[2] D. Thakker, T. Osman, D. Al-Dabass, Knowledge-intensive semantic web services composition, in: Proceedings of UKSIM'08, IEEE Computer Society, Washington, DC, 2008, pp. 673–678.
[3] E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau, HTN Planning for Web Service Composition Using SHOP2, Journal of Web Semantics, vol. 1, no. 4, Elsevier B.V., 2004, pp. 377–396.
[4] H. Li, H. Wang, L. Cui, Automatic composition of web services based on rules and meta-services, in: Proceedings of CSCWD'07, IEEE, 2007, pp. 496–501.
[5] J. Hoffmann, J. Scicluna, T. Kaczmarek, I. Weber, Polynomial-time reasoning for semantic web service composition, in: Proceedings of SERVICES'07, IEEE Computer Society, Washington, DC, 2007, pp. 229–236.
[6] J. Rao, K. üngas, M. Matskin, Logic-based web services composition: from Service Description to Process Model, in: Proceedings of ICWS'04, IEEE Computer Society, Washington, DC, 2004, pp. 446–453.
[7] L.A.G. da Costa, P.F. Pires, M. Mattoso, Automatic composition of web services with contingency plans, in: Proceedings of ICWS'04, IEEE Computer Society, Washington, DC, 2004, pp. 454–461.
[8] L. Ye, B. Zhang, Discovering web services based on functional semantics, in: Proceedings of APSCC'06, IEEE Computer Society, Washington, DC, 2006, pp. 348–355.
[9] R. Akkiraju, A. Ivan, R. Goodwin, B. Srivastava, T. Syeda-Mahmood, Semantic matching to achieve web service discovery and composition, in: Proceedings of CEC/EEE'06, IEEE Computer Society, Washington, DC, 2006, p. 70.
[10] R. Aydoğan, H. Zirtiloğlu, A graph-based web service composition technique using ontological information, in: Proceedings of ICWS'07, IEEE Computer Society, Washington, DC, 2007, pp. 1154–1155.
[11] R. Thiagarajan, M. Stumptner, Service composition with consistency-based matchmaking: a CSP-based approach, in: Proceedings of ECWOS'07, IEEE Computer Society, Washington, DC, 2007, pp. 23–32.
[12] S.C. Oh, D.W. Lee, S.R.T. Kumara, A Comparative Illustration of AI Planning-based Web Services Composition ACM SIGecom Exchanges, vol. 5, ACM, New York, 2006, pp. 1–10, 5.
[13] Semantic Annotations for WSDL and XML Schema (SAWSDL), W3C Recommendation 29 August 2007, Available on http://www.w3.org/TR/2007/REC-sawsdl-20070828.
[14] Semantic Markup for Web Services (OWL-S), W3C Member Submission 22 November 2004, Available on http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122.
[15] S. Kona, A. Bansel, G. Gupta, Automatic composition of semantic web services, in: Proceedings of ICWS'07, IEEE Computer Society, Washington, DC, 2007, pp. 150–158.
[16] S.V. Hashemian, F. Mavaddat, A graph-based framework for composition of stateless web services, in: Proceedings of ECOWS'06, IEEE Computer Society, Washington, DC, 2006, pp. 75–86.
[17] The North American Industry Classification System (NAICS), Available on http://www.census.gov/eos/www/naics.
[18] The United Nations Standard Products and Service Code (UNSPSC), Available on http://www.unspsc.org.
[19] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A service creation environment based on end to end composition of web services, in: Proceedings of WWW'05, ACM, New York, 2005, pp. 128–137.
[20] Web Ontology Language (OWL), W3C Recommendation 10 February 2004, Available on http://www.w3.org/TR/2004/REC-owl-features-20040210.
[21] Web Service Modeling Ontology (WSMO), W3C Member Submission 3 June 2005, Available on http://www.w3.org/Submission/2005/SUBM-WSMO-20050603.
[22] Y. Gamha, N. Bennacer, L. Ben Romdhane, G. Vidal-Naquet, B. Ayeb, A Statechart-based model for the semantic composition of web services, in: Proceedings of SERVICES'07, IEEE Computer Society, Washington, DC, 2007, pp. 49–56.
[23] Z. Liu, A. Ranganathan, A. Riabov, Modeling web services using semantic graph transformations to aid automatic composition, in: Proceedings of ICWS'07, IEEE Computer Society, Washington, DC, 2007, pp. 78–85.