A Scalable Architecture for Automatic Service Composition

Incheon Paik, Wuhui Chen, *Member*, *IEEE*, and Michael N. Huhns, *Fellow*, *IEEE*

Abstract—This paper addresses automatic service composition (ASC) as a means to create new value-added services dynamically and automatically from existing services in service-oriented architecture and cloud computing environments. Manually composing services for relatively static applications has been successful, but automatically composing services requires advances in the semantics of processes and an architectural framework that can capture all stages of an application's lifecycle. A framework for ASC involves four stages: planning an execution workflow, discovering services from a registry, selecting the best candidate services, and executing the selected services. This four-stage architecture is the most widely used to describe ASC, but it is still abstract and incomplete in terms of scalable goal composition, property transformation for seamless automatic composition, and integration architecture. We present a workflow orchestration to enable nested multilevel composition for achieving scalability. We add to the four-stage composition framework a transformation method for abstract composition properties. A general model for the composition architecture is described herein and a complete and detailed composition framework is introduced using our model. Our ASC architecture achieves improved seamlessness and scalability in the integrated framework. The ASC architecture is analyzed and evaluated to show its efficacy.

Index Terms—Automatic service composition architecture, four-stage composition, functional scalability, nested composition, composition property transformation

1 Introduction

Service-oriented computing (SOC) enables new kinds of flexible and scalable business applications and can improve the productivity of programming and administering applications in open distributed systems [1]. Web services are already providing useful APIs for open systems on the Internet, and, thanks to the Semantic Web, are evolving into the rudiments of an automatic development environment for agent-based applications [1], [2]. To further this environment, the goal of automatic service composition (ASC) is to create new value-added services from existing services, resulting in more capable and novel services for users [3], [4].

Automatic service composition usually involves four stages [5], [6]:

1. planning a workflow of individual service types;
2. locating services from a service registry, i.e., finding service instances;
3. selecting the best candidate services based on nonfunctional properties (NFPs) for deployment and execution; and
4. executing the selected services. When an exception occurs during execution, services might have to be retried or the planning and selection stages might have to be redone [5].

- *I. Paik and W. Chen are with the Division of Information Systems, School of Computer Science and Engineering, University of Aizu, Tsuruga, Ikki-machi, Aizu Wakamatsu, Fukushima 965-8580, Japan. E-mail: paikic@u-aizu.ac.jp, chenwuhui21@gmail.com.*
- *M.N. Huhns is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208. E-mail: huhns@sc.edu.*

Manuscript received 19 Feb. 2012; revised 3 Sept. 2012; accepted 5 Nov. 2012; published online 16 Nov. 2012.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2012-02-0025.
Digital Object Identifier no. 10.1109/TSC.2012.33.

Some stages can be merged according to the domain, problem, and composition conditions [7], [8], [9].

The four stages for ASC provide a good basis for analyzing automatic service composition; however, previous studies have considered mainly individual stages or integrating the stages for more realistic composition. There is still no wholly integrated framework for ASC and there is significant complexity when attempting to apply it in real scenarios. Furthermore, in many of the studies of service composition, there has been confusion about the roles of the composition stages, because functions belonging to one stage in some studies have been part of different stages in other studies. We believe that a consistent framework will help to illuminate the problems of automated composition and give a starting point for approaches to overcome the complexity. Also, our analysis of existing composition approaches has revealed two other issues important for a more complete ASC framework: making functional goals scalable and making the composition seamless. We present a more complete framework for ASC, which addresses these additional issues, by incorporating the following features:

1. *Scalable functional goals: nested workflow management.* Service compositions in the literature usually assume completion of the e-composition at one time. However, this is not always possible, because of the distributed and dynamic nature of compositions that might occur across enterprise boundaries. To meet functional goals that are broad and comprehensive, ASC can include nested dynamic compositions at sublevels. ASC must also consider dynamically changing workflows to fulfill new goals introduced at higher levels of abstraction. If the architecture is nested, the workflow manager can control replanning and reselection from exceptions and also orchestrate nested composition flows.

2. *Seamless composition: identification of composition properties (NFPs).* Services have both functional and nonfunctional properties. In general, functional properties (FPs) concern requirements within the domain of a service request, whereas nonfunctional properties (NFPs) concern requirements on the services themselves. NFPs subsume quality-of-service (QoS) parameters by including preferences and similar "soft" constraints. FPs must be satisfied, but NFPs do not. Also, there is not a crisp distinction between FPs and NFPs, but we have found that a description of goals and services is clearer when we separate them.

   A user or developer specifies a composition goal that includes FPs or constraints consisting of their attributes (for instance, "Arrange a trip from Aizu to Los Angeles" and "Total Cost is less than 100,000") and NFPs (for instance, "Cost for the composed services to arrange the trip is less than 5,000" and "Time to execute the composed services must be less than 30 s"). These are specified at the business (abstract) level to a service composer. Usually, service instances that are matched with a functional operation signature are located during the discovery stage, but composition properties, i.e., the NFPs and some of the FP's attributes in the user's request, which are abstract as in the examples above, are not available as explicit operations with detailed parameters, so that they can be considered during the selection process. However, the identification and consideration of the composition properties are necessary for seamless ASC.
3. *Framework for ASC: modified four-stage process.* Previous studies on service composition have focused on one or two of the four stages. Some studies deal with abstract figures of ASC or a comparison of composition approaches [4]; however, it is essential to consider all the functional blocks for composition, together with the entire structure and interactions of the ASC. To make this manageable, structural and behavioral object analysis can be carried out hierarchically. A top-level analysis would deal with more abstract, general concepts. A lower level analysis would enable detailed choices for algorithms and methods for ASC. The entire UML-based framework we provide for integrating the modified four-stage architecture with the above two additional issues can contribute to explaining the organic structure and behavior of ASC and its further development.

In the remainder of the paper, Section 2 overviews the state of the art of ASC. Section 3 describes the dynamic orchestration model and nested composition architecture. Section 4 introduces the linkage between requests and services, especially the transformation of nonfunctional properties for the selection stage. Section 5 suggests a novel architecture for ASC, considering the issues of Sections 3 and 4, and describes a case study. Section 6 evaluates our architecture and the case studied. Section 7 compares work related to our approach. Finally, Section 8 concludes the paper and suggests future work.

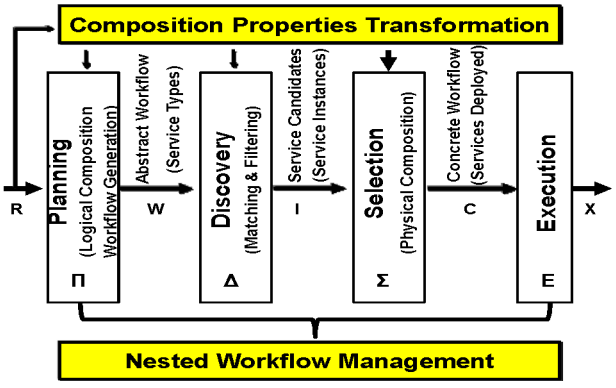

Fig. 1. Four stages of service composition and our extensions for nested ASC.

2 Preliminaries

2.1 The Four Stages of Composition

The four stages for automatic service composition are depicted in Fig. 1, which has been augmented with more information from that presented in [5]. Formally, we denote the following:

- R: Set of user's requests at the service level.
- $\mathrm{W} = \{\mathrm{t}_1, \mathrm{t}_2, \mathrm{t}_3, \ldots \mathrm{t}_l\}$: Set of l abstract tasks in an abstract workflow W.
- Planning Π: $\mathrm{R} \rightarrow \mathrm{W}$.
- $\mathrm{I}_i = \{\mathrm{i}_{i1}, \mathrm{i}_{i2}, \mathrm{i}_{i3}, \ldots, \mathrm{i}_{im}\}$: Set of m service instances advertised in a service registry for an abstract task t_i. I is the set of I_i, for $1 \leq i \leq l$. If each task in a workflow has m instances, then the total number of service instances available for the workflow is $l \times m$.
- Discovery $\Delta : W \rightarrow \mathrm{I}$.
- $\mathrm{C}_j = \{\mathrm{c}_{j1}, \mathrm{c}_{j2}, \mathrm{c}_{j3}, \ldots, \mathrm{c}_{jp}\}$: Set of p selected service instances to be executed from the service instance set I_j. C is the set of C_j where $1 \leq j \leq l$.
- Selection $\Sigma : \mathrm{I} \rightarrow C$.
- $\mathrm{X} = \{\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3, \ldots, \mathrm{x}_q\}$: Set of q executed service traces.
- Execution $\mathrm{E} : \mathrm{C} \rightarrow \mathrm{X}$.

There are several viewpoints about the service composition process. Usually the composition work is considered to be divided among the four stages. The planning stage is described as logical composition, while selection is described as physical composition [6], [10]. We chose this four-stage process as a basis because

1. It is considered to be fundamental, and most existing approaches can be mapped or related to it easily.
2. It is straightforward to show how our approach augments it to make service composition more flexible and easier for humans.
3. Our augmentation of it can deal with failures via exception handling and backtracking. When a failure occurs, a return to the planning stage is not required.

2.2 Motivating Scenario and Our Extended Framework for Seamless ASC

In automated service composition, a plan—a sequence of services to fulfill a functional goal—is generated during the

planning stage. However, the one-time plan may be a subprocedure of a higher level goal.

For a real example, consider the scenario of a tour group traveling from Aizu to Los Angeles. To create the tour group package (the top goal), there must be a composition of three subprocesses: 1) scheduling the trip, 2) making group reservations, and 3) repeating the trip scheduling and reservations for each participant.

In the scenario, the first composite service for finding a trip schedule tries to find the best workflow for the trip and candidate services for the workflow. If the group wants to go by train to Narita International Airport near Tokyo and then fly to Los Angeles, there are three segments: by local train from Aizu to Koriyama train station; by bullet train from Koriyama to Tokyo; and by JR Express from Tokyo to Narita, from where a series of flights can complete the journey to Los Angeles. This composite service can be developed dynamically by the ASC.

After deciding the trip schedule, the service must book all travel resources, such as transportation and lodging, according to the schedule from the previous step. If the booking fails because of unavailable transportation or no rooms in a hotel, the composition manager must return to the planning or selecting stages. Some reservations request payment when booking and some after booking. After making a reservation, a user may confirm it by making a payment, or cancel it by not paying. The payment process can be activated by the user or can be automatic. Finally, the composed service for the group tour using the above two services will be activated. The three composite services can be generated by the planner of ASC at the outer level, and the composition manager orchestrates recursively the nested composition process to make the group tour.

The four-stage process characterizes composition from an abstract workflow to a concrete one, but it does not address seamless ASC. Therefore, we extend it with the components *nested workflow management* and *composition properties transformation*, as shown in Fig. 1. The nested workflow management block orchestrates nested workflows for each stage of ASC to handle composite goals.

3 Workflow Orchestration in a Nested Composition for Scalability

3.1 Orchestration in Service Composition

There have been many studies of ASC, but they have only considered it as a one-step composition. When one-step composition does not achieve the goal requested by a user, additional processes must be orchestrated dynamically to reach the final goal. This procedure can be viewed as a multistep composition via orchestration of the workflows in the nested composition structure.

In the scenario, the trip scheduling service can be constructed by an ASC. Here, the planner of the ASC generates an abstract workflow (using staged composition and execution) for traffic routes and hotels between Aizu and LA, and selects an optimal workflow using a metric of preconditions. Then, the ASC discovers service candidates and selects optimal instances of services using QoS and user constraints on the workflow, which are normal steps in an ASC activity [11].

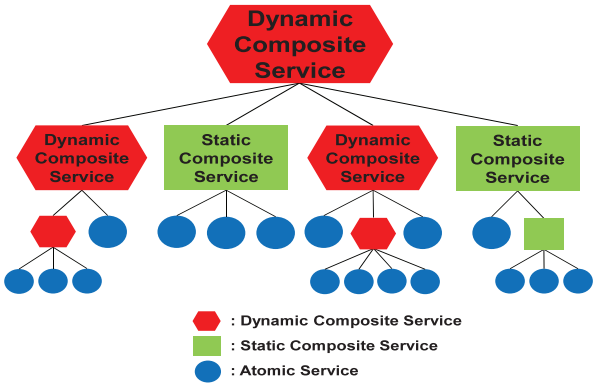

Fig. 2. General composition model with nested composition.

To achieve the final goal, the selected trip schedule would be passed to the reservation process, and the results of these two processes would be combined to create the group tour. Thus, the results of the subprocesses must be orchestrated to achieve the final goal by an outer ASC.

3.2 Conceptual Model of Nested Composition

There are two approaches to managing workflow in service composition: centralized orchestration and distributed choreography [12]. ASC usually adopts the centralized management of services for one specific goal and uses the orchestration paradigm.

Workflow management at a higher level mirrors orchestration. The orchestrator at a higher level integrates three types of services: dynamic composite services, static composite services, and atomic services, as shown in Fig. 2. A dynamic composite service is a service created dynamically by an ASC. A static composite service is a predefined service that may have been produced manually or by extraction tools. An atomic service may have been published in a service repository, and we consider such services to be static ones. These services are used by a nested ASC (NASC) to achieve a goal at an outer level. In general, a hierarchy of compositions can be created to achieve the final goal. These will be located by matching their Input-Output-Precondition-Effects (IOPEs) during the composer's discovery function.

3.3 Workflow Orchestration in Nested Composition

3.3.1 Top-Down Approach

The orchestration strategy for services composition using a top-down approach can be considered as an extension of the conventional ASC approaches. By considering characteristics of dynamic services, the discovery stage and the execution stage are modified in the top-down approach. The important characteristic of dynamic services is that one dynamic service potentially conforms to more than one static service. In the ASC described here, several sequences of service types can be created in the planning stage, and the optimal service instances that have the required operators are selected for the service types after the invocation of the discovery stage. Finally, the ASC outputs the composite services, combining the operators of the selected services. Procedure 1 shows the top-down approach for nested

composition, where the essential methods and parameters are shown to fulfill a desired composition.

Procedure 1: Managing nested workflow by a top-down approach to ASC.

Require: *PlanningDomain pd, UserConstraint* ***uc***
Ensure: ***ExecutableWorkflow*** ← *ASC(PlanningDomain pd,* ***UserConstraint uc****)*

1: *Let* ***ServiceType*** *be service types for an AbstractWorkflow*
2: *Let* ***ServiceInstance*** *be service instances for a ServiceType*
3: *Let* ***AbstractWorkflow*** *be workflows, consisting of a set of ServiceTypes*
4: *Let* ***ConcreteWorkflow*** *be workflows, consisting of a set of ServiceTypes*
5: *Let* ***ExecutableWorkflow*** *be executable workflow languages*
6: *// Planning Stage*
7: ***AbstractWorkflow*** ← *generateAbstractWorkflow(pd)*
8: *// Discovery Stage*
9: **for** $i = 0$ *to* ***AbstractWorkflow****.length* **do**
10: ***ServiceType*** ← *AbstractWorkflow[i].get ServiceTypes*()
11: **for** $j = 0$ *to* ***ServiceType****.length* **do**
12: ***ServiceInstance*** ← *discoverServices(ServiceType[j])*
13: **if** ***ServiceInstance****.length is 0* **or** *ServiceType[j] calls a dynamic service* **then**
14: *pd2* ← *generatePlanningDomain(ServiceType[j])*
15: ***ServiceInstance*** ← *(ServiceInstance) ASC(pd2,* ***uc****)*
16: **end if**
17: *ServiceType[j].setServiceInstances(****ServiceInstance****)*
18: **end for**
19: *AbstractWorkflow[i].setServiceTypes(****ServiceType****)*
20: **end for**
21: *// Selection Stage*
22: ***ConcreteWorkflow*** ← *doCPSelection(pd,* ***uc****,* ***AbstractWorkflow****)*
23: *// Prepared for Execution Stage*
24: ***ExecutableWorkflow*** ← *generateExecutableServices (pd,* ***ConcreteWorkflow****)*
25: **for** *i = 0 to* ***ExecutableWorkflow****.length* **do**
26: *ExecutableWorkflow[i].temporalpublish()*
27: **end for**
28: **if** *Process is in Inner ASC* **then**
29: **return** ***ExecutableWorkflow***
30: **else**
31: **return** *Invocation Results of* ***ExecutableWorkflow***
32: **end if**

The flow of the procedure is as follows: First, the ASC function takes the *PlanningDomain* variables and a set of *UserConstraints*. *PlanningDomain* has a problem space and a planning space that are used to invoke the planning for one domain. The *UserConstraint* set contains information about the user constraints. The user constraints and the functional goal can be varied according to the user's requirement. Note that *AbstractWorkflow* is used for storing the state of the results derived from the planning stage and the discovery stage, while *ConcreteWorkflow* is used for storing the state of the results derived from the selection stage and the execution stage. In line 7, the *generateAbstractWorkflow* method receives the *PlanningDomain* variable, and generates an *AbstractWorkflow* set. The size of *AbstractWorkflow* set is determined by the number of generated sequences from *ServiceTypes*. This process corresponds to the planning stage introduced in Section 2.1.

The discovery stage of top-down composition is shown between lines 9 and 20. This process discovers the service instances for each service type. If the *ServiceInstance* set cannot be discovered for a *ServiceType* from the service repositories, or the *ServiceType* is regarded as a dynamic service, the internal ASC is invoked recursively to generate a *ServiceInstance* set for the *ServiceType*. Following the call of the inner ASC, a variable *PlanningDomain* is generated by the *generatePlanningDomain* method. The method creates the needed information (the required planning and problem spaces) for a planner, which can then derive the desired workflows for the domain. Using the generated *PlanningDomain*, the inner ASC can create services that fulfill the functional goal in the domain. The methods *setServiceInstances* and *setServiceTypes* (lines 17 and 19) are used to write the obtained parameters to the corresponding variables.

In line 22, the *doCPSelection* method is called, and the method generates a set of *ConcreteWorkflow*s. The *doCPSelection* method selects the optimal *ServiceInstance* from the discovered *ServiceInstances* for each *ServiceType* by using the NFPs information and the user constraint satisfaction measures. So the generated *ConcreteWorkflow* set has optimal service instances for the service types. By the processes of line 22, the results (such as the selection stage in Fig. 2) can be obtained.

The execution stage of this approach corresponds to lines 25 and 32 in Procedure 1. In line 24, the *generateExecutableServices* method receives the *PlanningDomain* and *ConcreteWorkflow* sets, and generates a set of *ExecutableWorkflows*. After the generation of the *ExecutableWorkflow* set, it is published temporarily as services that can be accessed by a client application. If the process is in the inner ASC, it returns the generated *ExecutableWorkflow* set denoted in line 29. In this process, the parent ASC can obtain the *ServiceInstance* set from the Inner ASC shown in line 15. Also in line 15, the *ExecutableWorkflow* set is converted to a *ServiceInstance* set. This implies that *ExecutableWorkflow* is substantially similar to *ServiceInstance*. In line 31, the invocation results of the executable workflow are returned. This process is invoked when the highest services located on the nested structure are generated. Consequently, by using Procedure 1, the inner ASC can be called recursively and the nested dynamic composition structure can be generated.

3.3.2 Bottom-Up Approach

A bottom-up approach for managing composition utilizes human inputs for fulfilling the nested dynamic service composition. In practical applications, manual service selections by clients are important, because there are cases where some of the services selected automatically by ASC cannot be satisfied for a client. The bottom-up approach to address this problem can be realized by changing the selection stage of the top-down orchestration of a composition. Instead of a selection based on composition properties, such as QoS information and user constraints, human selections are utilized. Many dialog forms can be considered to obtain the human selections. For example, the

following question forms are appropriate: *What sequence of service types is preferred from the planner? What service instance is preferred for the service type?* Details of the procedure for this approach are at [13].

3.4 Investigation of Functional Scalability

The proposed nested multilevel composition provides functional scalability as a composition parameter. Two approaches to scalability are possible: bottom-up and top-down. A bottom-up approach is suitable for user-driven composition, while a top-down approach appears more suitable for machine-planned composition without user intervention (only the top-down approach is introduced here; moreover, for readability purposes, we limit the scalability to a single independent domain to simplify the complex composition problem). An example to show functional scalable composition in the travel domain can be found at our demonstration site [14]. An example about orchestration of scalable goal composition is illustrated in Section 5.2.6 and an evaluation of its computational cost is given in Sections 6.2 and 6.3.

4 Linking Requests with Services: Identifying Composition Properties (NFPs)

User requests are expressed in an abstract form based on the user's semantics in the goal domain. For a request given to the composer by the user, a goal consists of the functionality to be achieved, desirable nonfunctional properties, and other related information [15]. In addition, there may be other nonfunctional properties that are not related to the requests. There are two types of goal: one is understood by requestors only and the other is registered, so that it can be understood by the ASC system. Registered goals can help the discovery service to locate the corresponding services in the service domain. The abstract requests must link to the corresponding services. And, it is important to refine the generic and abstract goals to concrete goals and to discover services from the abstract goal [15], [16].

The purpose of service discovery in a composition system is to find services in a registry that can achieve the functional part of a goal, and many studies have tackled this problem. The nonfunctional part has been considered for composition stages, especially service selection. NFPs are usually described in abstract form at the abstract goal level, and cannot be adapted during the selection stage. Therefore, the abstract NFPs must be converted to concrete NFPs that have their identification information in real services for seamless automatic composition.

The services in the selection stage must be identified for execution at the next stage, so service identity, operator, and parameters are required. As abstract NFP requirements from the user or the planner do not have this information, it must be obtained through transformations before beginning the selection stage.

4.1 Definition of a Nonfunctional Property

Descriptions of a service in terms of its operation signatures are classified as FPs, and descriptions of the service's characteristics that are not directly related to the functionality, such as QoS attributes, are classified as NFPs [17]. It is sometimes difficult to classify constraints or user preferences as FP or NFP (for instance, constraints with attributes of functional properties from multiple services, or mixed properties with NFPs in some situation). In our research, constraints are classified as NFPs if they are not directly related to descriptions of the functionality that fulfills a user's goal. The attributes of each property are of resource type or value type. User preference constraints describe basic desire, priority, and choice and are expressed in first-order logic or description logic. Many studies of service composition use the term "constraints" for the NFPs of a service. In this paper, we use the term *constraint* as a representative term and the attribute-value clause as the basic form for NFPs thereafter.

Constraints have two kinds of sources at three levels. Internal constraints are defined by the composer and external constraints are imposed by the user. We divide the characteristics of the constraints into three levels: abstract, intermediate, and concrete.

Abstract constraints. These constraints are at a level near humans' natural concepts. All terms are abstract and the constraints may be defined informally.

Intermediate constraints. These consist of a relation, two terms, context information, and an operator. They are generated by extracting abstract relations, terms, and context information from abstract terms (which may include context information) in natural language or compound terms at an abstract level. All the terms are terminal (not compound) and have not been bound to concrete terms.

Concrete constraints. These have relations, terms as binding information, and indexes to specific tasks in an abstract workflow.

4.2 Data Model of Properties

The data model of composition properties in our ASC is based on the above constraints and other properties for all composition stages. A complete transformation requires transformation of abstract constraints to intermediate constraints that consist of terms registered in an ontology registry, and of intermediate constraints to concrete ones that can be used in the selection stage. The concepts of transformation are depicted in Fig. 3. Here we suppose that the transformation from abstract to intermediate has been completed by natural language processing methods or text mining techniques.

We introduce next the definitions of intermediate constraints that relate to the transformation directly. These are defined in BNF in the following, where "*" means a list of anterior properties.

Definition 1. ***FP and NFP for Planner.***

Request := ***pDomain*** *Parameter**
Parameter := ***name value***

A *Request* consists of functional properties (FPs) and nonfunctional properties (NFPs) for the planner. It is described in simple first-order logic (FOL). The planner requires a problem domain ***pDomain*** as FP and parameters *Parameter** as FPs and NFPs. Each parameter has a **name** and **value**.

An example of a *Request* is:

(trip ((departurePlace Aizu)
(arrivalPlace San Francisco)
(priority earliness) (stay 3)))

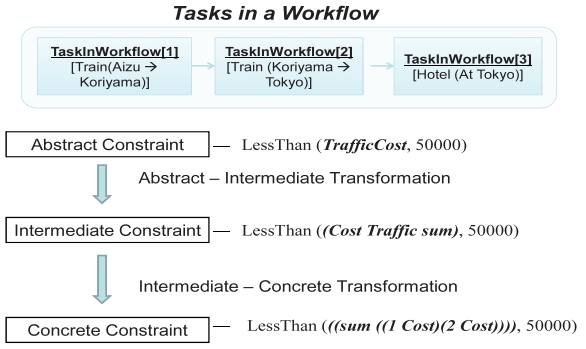

Fig. 3. Transformation of composition properties.

Definition 2. ***FP of Abstract Task.***

AbstractTask := ***sDomain*** *Parameter**

An *AbstractTask* is an FP of an abstract task and has service domain **sDomain** and some parameters *Parameter**. The **sDomain** is a class in the service domain ontology described in the following section.

An example of an *AbstractTask* is:

(TrainService ((departureStation Tokyo) (arrivalStation Narita) (line NEX)))

Definition 3. ***Intermediate Constraint.***

IntermediateConstraint := $IntermediateTerm_l$ *Relation* $IntermediateTerm_r$

IntermediateTerm := (***vDomain*** *Context Operator*) | ***value***

Relation := = | ? | > | < | ? | ?

Operator := ***sum*** | ***average*** | ***max*** | ***min*** | ***additional-function***

Context := ***index*** | ***sDomain***

An intermediate constraint, *IntermediateConstraint*, has two intermediate terms, *IntermediateTerm*, and one relation, *Relation*. An *IntermediateTerm* has a variable domain ***vDomain***, a context term, *Context*, and a term operator, *Operator*, or there may be an intermediate term with a constant value, ***value***. The ***vDomain*** is a class of variable in the domain ontology. The *Context* has two types of index: one is a pure index for pointing, ***index***, and the other is a service domain, ***sDomain***, that will give index information through inference on the service domain ontology. The ***index*** points to an *AbstractTask*. The *Operator* is an aggregating operator such as ***sum***, ***average***, ***max***, ***min***, or some ***additional-function***. This operator calculates using variables pointed to by *vDomain* and *Context*. The *sDomain* of the *Context* points to some abstract task belonging to this domain.

An example of *IntermediateConstraint* is:

((Cost AllService sum) < 300,000 yen).

Definition 4. ***Concrete Constraint.***

ConcreteConstraint := $ConcreteTerm_l$ *Relation* $ConcreteTerm_r$

ConcreteTerm := (*Variable** *Operator*) | ***value***

Variable := ***index vDomain***

Intermediate constraints, *IntermediateConstraint**, are transformed to concrete constraints, *ConcreteConstraint**, by the transformer for the selection stage. A *ConcreteConstraint* has two concrete terms: *ConcreteTerm* and a *Relation*. A *ConcreteTerm* has an index for a task in the abstract workflow ***index*** and a variable domain in the ontology ***vDomain***, or there may be a concrete term with constant value, value.

An example of *ConcreteConstraint* is:

(((2 Seat)) nil) = economy)

This means that the *Seat* variable of the second abstract task is economy class. This concrete constraint is made from the following *IntermediateConstraint:*

((Seat Airplane nil) = economy)

This means that the seat in an airplane must be economy class.

Definition 5. ***Identification Information.***

Variable := ***index vDomain***

Candidate := *Service**

Service := ***sReference*** *Attribute**

Attribute := ***vDomain value aReference*** | ***pReference***

The candidate generator uses some service candidates *Candidate** as identification information. The number of *Candidate** is the same as the number of *AbstractTask**. An *AbstractTask* corresponds to a *Candidate*. A Service is selected by CSP from Candidate to satisfy *ConcreteConstraint**. A Service has identification information. The ***sReference*** is a reference to a service such as the URL of WSDL and some attributes. Attributes, *Attribute**, are input parameters and the output operator of a service. An attribute has ***vDomain***, value, and reference to a process ***pReference*** or parameter ***aReference***. When a CSP solver receives a value from a service, if the value of attribute is input, this value is used, but if the value of attribute is not input (the input is nil), ***pReference*** is used as an operator of a service, and ***aReference*** is used as the parameter.

4.3 Transformation of Composition Properties (NFPs) on Domain Ontologies

There are many ways to map terms from one domain to another, including using tables, special data structures, database schemas, or ontologies. Ontologies are chosen for the transformation of composition properties because they can describe the terms of services in natural language and allow logical reasoning and extension. We put each service in a class in the service domain ontology, and each parameter and operator of a service in a class in the variable domain ontology. A child class in the service domain includes all attributes of its parent class. The classes, relations, and their hierarchy in the variable domain ontology can be used for inference operations to extract knowledge about service variables. In this algorithm, we selected exactly matched classes as links for intermediate-level terms to concrete-level terms after the taxonomy classification. This approach can be extended to more general cases such as plug-in match, subsumption match, nearest-neighbor match, and fail.

The algorithm uses the service domain and service variable ontologies to infer the match between the abstract and the concrete. According to the characteristics of the various service domains, the ontologies for the domains can be changed. If new services and conditions are added to the domain, the ontology can be updated accordingly.

To transform an abstract constraint into a concrete one, we must find binding information from the abstract terms. In detail, the procedure must find a service identification (or index) in the abstract workflow (here, the workflow is presumed sequential) and information about the operation and parameters of the service. Translating a highly abstract

constraint to the intermediate level requires natural language parsing and semantic analysis, so we focus on the transformation of intermediate constraints into concrete ones in this research. Algorithm 1 transforms intermediate terms of constraints to concrete terms that have binding information. The transformation flow of the algorithm diverges according to context and operator. In the first two cases, the intermediate term has constant or direct index information, and the transformation can be done simply. If an intermediate term has a constant value, it becomes a concrete term that has the same constant value (lines 3 and 4). If an intermediate term has an index, it becomes the service identification index of a concrete term (lines 5-8).

Algorithm 1. Transformation of Constraints.

```
Let CC be ConcreteConstraint*
Let ic be an IntermediateConstraint
Let it be an IntermediateTerm
Let r be a Relation
  CC ← transform(itl) × r ×  transform(itr)
 1: function transform(it)
 2:   Let workflow be AbstractTask*
 3:   if it is a value
 4:     return it
 5:   else if it.context is an index
 6:     Let ct be a ConcreteTerm
 7:     ct.add(getVariable(it.context, it.vDomain))
 8:     return ct
 9:   else if it.operator ≠ nil
10:     Let ct be a ConcreteTerm
11:     foreach t in workflow
12:     if t.sDomain ∈ it.context
13:         ct.add(getVariable(index of t, it.vDomain))
14:       end
15:     end
16:     ct.operator ← it.operator
17:     return ct
18:   else
19:     Let cts be new ConcreteTerm*
20:     foreach t in workflow
21:       if t.sDomain ∈ it.context
22:         Let ct be a new concrete term
23:         ct.add(getVariable(index of t, it.vDomain))
24:         cts.add(ct)
25:       end
26:     end
27:     return cts
28:   end
29: end
Function getVariable(index, vDomain)
Let v be new Variable
v.index ← index
v.vDomain ← vDomain
return v
end
```

In the next two cases, there is no direct information for service identification, so inference on the service and variable domain ontologies is required. Currently, our implementation matches the corresponding class by querying the class hierarchy of the ontology. If an intermediate term has no operator (this means that just one service is involved) and a service domain, it becomes a concrete term that has a variable. The variable has an index of an abstract task in the service domain of the intermediate term (lines 9-17). If an intermediate term has an operator and a service domain, the transformer collects all the services related to the operator in the workflow and applies the operator to the related services (lines 18-28).

5 Architecture for Scalable ASC

In this section, we present our entire architecture considering workflow orchestration with nested composition and NFP transformations for ASC in UML. Fig. 4 shows the entire composition architecture.

5.1 Top-Level Architecture

While the existing four-stage architecture finds a set of services to fulfill a goal, the two added parts deal with the orchestration of nested workflows for more general goals. This contributes to functional scalability and helps link NFPs from the abstract level to concrete ones. Because a description of the entire structure and behavior of the architecture of ASC is complex, we use top-level class and sequence diagrams to explain the abstract concepts of the architecture in this section. More detailed design by middle-level diagrams is illustrated in the following section using an instance of our motivating scenario. Fig. 4 provides top-level class diagrams for the structure of the ASC. The following are mainly explanations of the functionality of the classes. The first four classes are for the functions found in the existing four stages.

The *WorkflowGenerator* creates an abstract workflow to satisfy the *FunctionalProperty* of a user's request. The methods may include algorithms for planning, FSM, workflow generation, or a dedicated application. The *ServiceDiscoverer* provides discovery services. The primary goal of the class is to provide candidate services (service instances) to fulfill functionalities in the tasks from the *WorkflowGenerator*.

The *ServiceSelector* selects service instances to fulfill the NFPs from the user or the *WorkflowGenerator*. All the NFPs in the abstract level should be transformed to the concrete level before selection. The *ServiceSelector* can use any kind of *SelectionMethod,* such as planning, integer linear programming, or CSP-constraint optimizing. The *ServiceExecutor* executes the selected service instances from the *ServiceSelector*. It has the *ExecutionMonitor* class that keeps track of the execution of the services in the *ExecutionEngine* to maintain quality of performance.

The next two classes allow scalable composition with seamless processing of ASC. The Transformer converts abstract NFPs into concrete NFPs with binding information, so the NFPs can be understood by the *ServiceSelector*. In detail, the transformer captures the meaning of the terms in the abstract NFPs to link to terms of intermediate NFPs that are composed of terminal terms. It then transforms the intermediate NFPs into concrete NFPs. As described in Section 4, the transformation is processed using ontology

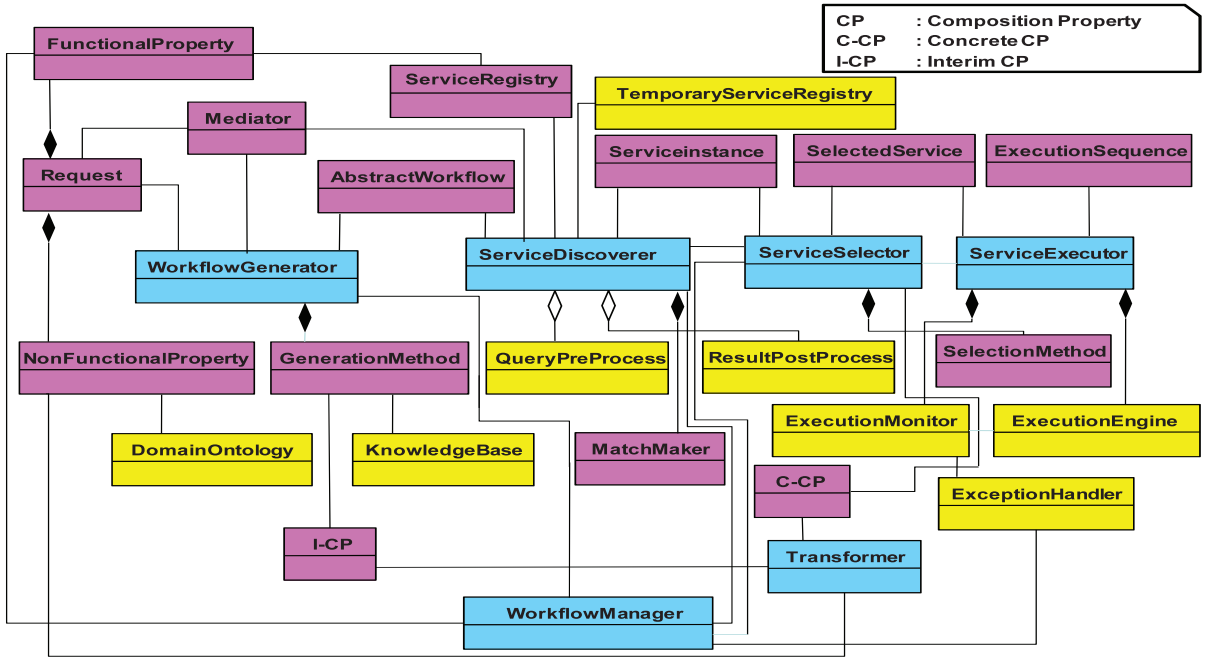

Fig. 4. Top-level UML class diagram of scalable ASC.

matching between terminal terms of the intermediate NFPs and the service domain ontology.

The *OrchestrationManager* is responsible for orchestrating the entire composition of services. It refines the user's generic goals into concrete goals found in the registry and analyzes the goals to identify services to satisfy the goals as required. The manager orchestrates all the composition steps in the nested structure of the ASC to reach the final goal. It allows the ASC to create new composite services for ones not found in the service registry, as described in Section 3. The management of nested workflow orchestration gives our system multilevel functionally scalable composition with dynamically derived goal parameters.

5.2 Lower-Level Architecture for the Example Scenario

While the top-level architecture describes the conceptual framework of the scalable ASC, the lower level architecture deals with details of implementation. In our research, we do not cover a complete transformation of the design specification, such as in a model-driven architecture, but we illustrate in detail how our upper-level architecture can be designed and implemented. We apply a divide-and-conquer approach based on details of the domain instead of directly tackling the large task of implementing a complete domain-independent ASC. The two core composition parts, the planning stage, and the selection stage, and the two new parts of transformation and orchestration are illustrated.

5.2.1 User Request for the Group Tour Scenario

We use the scenario introduced in Section 2.3 for details of request and composition. In the scenario, a user calls an agent to create a tour for a group to travel to Los Angeles. For this, the user inputs this goal as a functional requirement, along with nonfunctional requirements, such as the departure date and location and the arrival date and location. The user may provide additional constraints and preferences as nonfunctional requirements.

User request (input). Users may describe their requests in several forms: natural language, logic languages, a GUI, or dedicated goal modeling languages. Deriving real services from abstract goals or requests by inferencing is necessary for automatic composition. In our scenario, the requirements of functionality and nonfunctionality are described in first-order logic. However, the terms may not be terminal, which means that they require transformation.

5.2.2 Planning with HTN

Composition of predefined processes in OWL-S, BPEL, or WSMO to describe the workflow for reaching a goal statically can be used in the planning stage. We adopt an hierarchical task network (HTN) planner [18] to develop the workflow dynamically. It provides a strong foundation for planning an abstract workflow in this stage. We can encode an OWL-S process model or any kind of formation of the user's requests in the HTN planner. The HTN planner then develops a plan that describes a workflow to reach the goal state. Definition 6 describes HTN formally; HTN has been drawn in UML to combine it with the upper-level architecture of ASC.

Definition 6. (HTN Planning Problem and Plan). A planning problem is a tuple Plan = <*State, Task, Domain*>, and Domain = <*Axiom, Operator, Method*> *where* a plan, $\Pi = (\pi_1, \pi_2, \ldots, \pi_n)$, *is a sequence of instantiated operators that will achieve task from state in domain, which will be the abstract workflow in our composition scheme.*

In the composition system, a π_i is to be mapped to a task of the abstract workflow directly. For example, in the scenario, the planner generates an abstract workflow,

(moveByVehicle(Aizu, Koriyama), moveByVehicle(Koriyama, Tokyo), moveByVehicle(Tokyo, LA) ..., stayAtHotel(LA)).

5.2.3 Property Transformation

An original nonfunctional property from the user may have unregistered complex terms that require the transformation. The terms in the domain ontology for transformation are presumed to have clear and distinct meanings for real services. Finding concrete terms (intermediate constraints) from compound terms with many meanings (abstract constraints) ultimately requires human intelligence. Our detailed design for the transformation focuses on a transformation algorithm that accepts intermediate constraints and outputs concrete constraints. As explained in Section 4.2, the intermediate constraints have relations and intermediate terms. An intermediate term has context that explains the operation of the constraint, and the context is described in the ontology. The transformation algorithm should know not only intermediate terms, but also attributes of real services. The attributes of services have variable domains that relate to the domain ontology and references.

The proposed transformation algorithm uses the ontologies to include all classes of the services and service variables being transformed. According to the characteristics of the various service domains, the ontologies for the domains can be changed if new services and conditions are added. To allow this, and to be able to extract the ontology from the Web, we divided the ontology into two parts: unchangeable concepts and variable concepts.

We construct a basic ontology with unchangeable concepts using existing knowledge and ontologies. Variable concepts are then added by searching for existing websites and services. When a new class is added for variables in the ontology, we merge synonymous terms, and the process continues until no more new classes are found in the websites selected for our target.

5.2.4 Service Selection for Execution Using CSP

Service selection, which is also called physical composition, must find a concrete sequence of service instances for execution that matches those that are discovered for the abstract service sequence. The service selector chooses services that best satisfy nonfunctional properties. Our implementation uses constraint satisfaction problem (CSP) to find such service instances.

Definition 7. (NFP-Based Service Discovery and Service Selection Using CSP). *We separate the NFP from the usual nonfunctional properties. The NFP includes properties related to system characteristics. Other properties, user's preferences, and constraints are considered in the selection stage after NFP in the discovery stage.*

The function of extracting candidate services from abstract workflow by NFP is

- *GenerateCandidates:* $\Pi \times NFP \rightarrow X$, *where*
  - Π *is a series of abstract tasks generated by the planning stage, and NFP represents attributes for NFP.*
  - *X is a series of candidate services generated by "GenerateCandidates" that were filtered by NFP attributes.*

We describe the service selection based on CSP, $CSP, = <X, D, C>$, *where*

- *X is the same as in Definition 7;*
- *D is a set of instances of the ontology of input or output for the real process, for example, the ontology of* <IOPE> *in OWL-S; and*
- *C is a set of constraints, which can be user's constraints or preferences in the form of relations and terms.*

Constraints can change dynamically by system effects or by the user. As a result of selection in our scenario, a sequence of service instances: (*Banetsu101Service, Shinkansen-Yamabiko205Service, JRExpress101Service, ANA255Service, LAHotelService*) can be passed to the next stage for a combination of service operators. Other service instance sequences can also be selected.

5.2.5 Orchestration for Scalable Composition

The top goal of our scenario, "*MakeATourGroup*," develops a sequence of subtasks *("TripSchedule," "Reservation," "MakeTourGroup")* as a plan for achieving the goal. The inner ASC recursively develops another plan for the *"TripSchedule"* task, as illustrated in this section. We can also imagine situations that request more compositions upward or downward. For example, in the "*TripSchedule*," when a flight departs in the early morning, the passenger must stay in a hotel near Narita. This requires another plan at a lower level of composition. The orchestration manager controls the multiple compositions to provide functional scalability as a composition parameter.

6 Evaluating Feasibility of the Approach

The proposed architecture consists of the existing four-stages augmented with *the orchestration manager* for nested workflow to deal with scalable composition goals and *the composition property transformer*. As it is difficult to conduct an architectural level comparison and evaluation, we show feasibility by implementing ASC using HTN [18] for the planning and CSP [34] for lower level selection. We show the efficacy of our approach by measuring the computation cost not only for scalable composition and property transformation, but also for the entire four-stage ASC process, because they work organically in the same framework. Because functional scalability by depth of nesting is measured only for a travel scenario, we explain a more general measure in Section 6.3.

6.1 Case Implementation of the Architecture

As shown in the complete process for automatic service composition in Fig. 4, there are six principal interfaces for the *OrchestrationManager, WorkflowGenerator, ServiceDiscoverer, ServiceSelector, ServiceExecutor, and Transformer* at the top level. There are other interfaces and classes with close relationships with the six principal ones to describe the most general concepts of service composition at the top level. The classes in the top level have methods for

construction, communication to other functional blocks, and implementation of the interfaces of the upper level.

All the interfaces and classes in this level are provided as Java APIs of our NASC development kit (NASCDK). Developers can construct services using the top-level interfaces and classes, and customize their own composer by extending or implementing the classes or interfaces. Our implementation example covers the entire composition construction using NASCDK.

6.2 Computation Cost of the NASC System

To evaluate the implemented NASC system, we used the group tour scenario. The NASC system has six main measurable entities from workflow generation to service execution for which computation costs can be assessed. For evaluating complete service composition, we focused on: nested workflow management for scalable composition, workflow generation, property transformation, and service selection. These components comprise the service composition to create a concrete workflow for execution. We considered five variables for evaluating computation cost: depth of nested workflows, tasks, constraints, service instances, and classes of ontology. The depth of nesting affects the entire performance of the composition system, because it creates tasks for workflow; the number of tasks relates to the planning; the constraints, and service instances to selecting and transforming; and the classes to transforming. From the experiment, we plotted the principal entities in composition against the five variables, showing the factors that influence system performance and scalability.

Fig. 5a presents the computation cost (in milliseconds) of service composition (workflow generation, transformation, and selection) while we vary the depth of nesting. A cycle of new nesting occurs when there is a dynamic composite service (i.e., it is not found in the GSR, so it requires dynamic composition by ASC) in the current sequence of abstract tasks. In this experiment, the depth of nesting varied from one to five with a step of one. We set each abstract workflow in one nest level as one dynamic composite service, and with four dynamic composite service abstract tasks. We set the value of the other variables as follows: 1,000 classes, and 10 service instances per task. The computation cost for composition increases quickly with the number of nests, and the selection time is 76 percent of the total time in this case. The principal source of the increased time is service selection, because there are more service instances that the composer must select when more tasks are generated in further levels of nesting. Because our experimental system uses a basic backtracking algorithm in solving CSP that uses exhaustive search, it has high-computational overhead. Heuristic algorithms and optimization techniques could improve it.

In Fig. 5b, we can extract similar results. This figure shows computation cost as the number of tasks (W in Section 2.1) varies. We set the values of the variables to be 1,000 classes, five service instances per task, 2,000 constraints, and nesting depth of one. As in the former case, although the rate of increase is less than that shown in Fig. 5a, the cost increases quickly with the number of tasks for the same reason. The selection time is 39 percent of the total time in this case

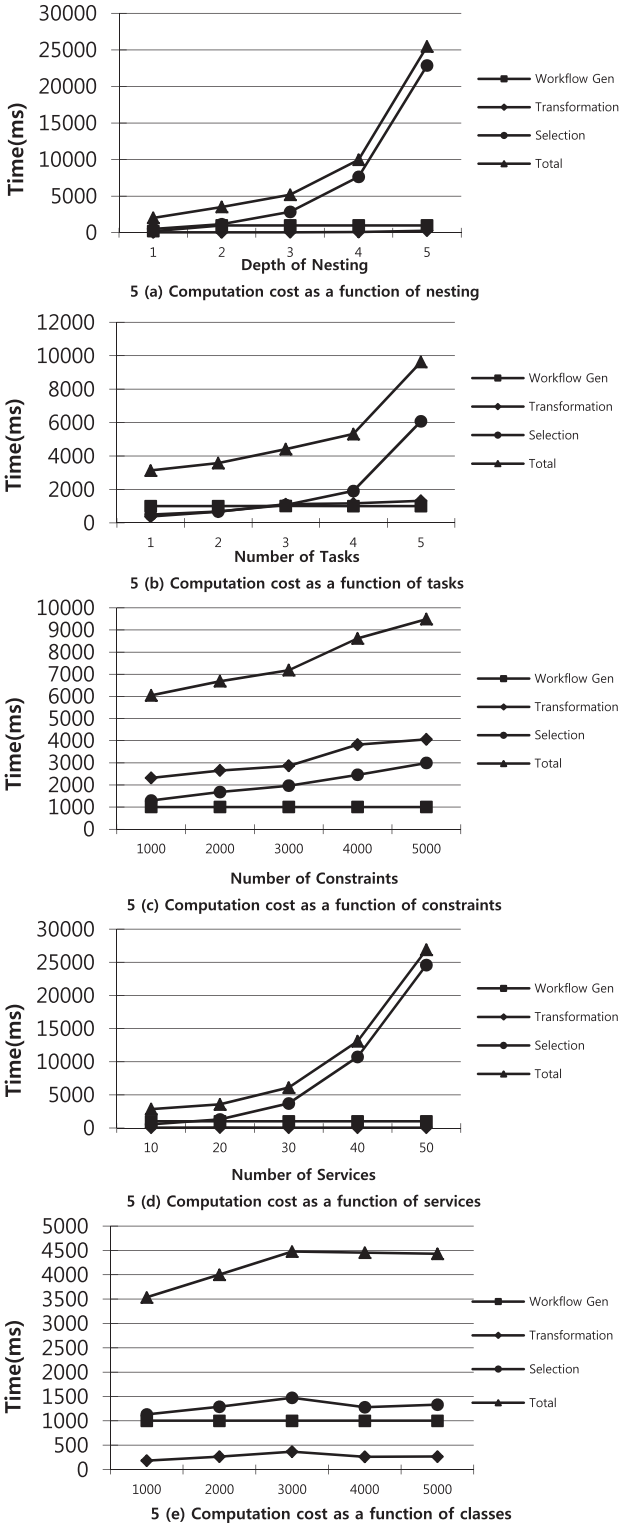

Fig. 5. Experimental result of computation cost for composition.

(by task). The cost of service selection is lower than that of workflow generation when the number of tasks is small, because the planner requires a steady amount of computation cost independent of the number of tasks, but that of service selection is affected more.

Fig. 5c shows the composition time cost as a function of the number of constraints. We varied the number of constraints from 1,000 to 5,000 in steps of 1,000 constraints.

The values of the other variables were set as follows: 1,000 classes, four service instances per task, four tasks in a workflow, and a nesting depth of one. The result shows that the transformation time increases linearly according to the number of constraints under these conditions, and the total composition cost increases linearly accordingly. This case is different from the earlier ones in that the cost of selection is relatively lower than that of transformation. The selection cost also increases linearly.

Fig. 5d shows the computation cost according to the number of service instances (I_i in section 2.1) per task. We varied the number of services from 10 to 50 in steps of 10 services. The values of the other variables were set to be: 1,000 classes, 10 constraints, four service instances per task, four tasks in a workflow, and a nesting depth of one. This plot is similar to that of Fig. 5a, and the selection time is a similar proportion, 78 percent, of the total time. The computation cost in this case is dependent on that of service selection, although the scale on the x-axis is different.

Finally, Fig. 5e depicts the computation cost when we vary the number of classes in the domain ontology used for transformation. The figure shows that the number of classes does not affect computation cost; i.e., the principal factor is service selection. Thus, the hashing search algorithm for ontology in Jena (a Semantic Web framework for Java) makes it faster for some ontology instances.

Our experimentation results can be summarized as:

- The computation cost for workflow generation and transformation is relatively small and stable under fixed planning space for a workflow, while that of service selection increases exponentially according to the number of tasks (and further depth of nesting with dynamic composite service) in the workflow.
- Service selection has the greatest effect on the total composition cost. The influence on the selection cost as a portion of the total computation cost is in the number of services, nests, and tasks.
- The number of constraints affects the computation cost for composition linearly.
- The fixed-size number of classes has no effect on the computation cost for composition.

Oh et al. [7] proved that Web service composition is an NP-complete problem. The planning algorithm for service composition and service network with the algorithm described in this paper can be considered as the simple integration of the service selection process in our architecture. The problem of workflow generation is the same as that of service selection, but the rate of increase is far smaller than that for selection.

6.3 Cost for Functional Scalability

As the results in the previous section were based on a travel scenario, another experiment has been carried out to simulate a more natural situation. We used a simple mathematical expression that consists of nested arithmetic calculations to model nested service composition for a goal requiring scalability. To create the nested structure of dynamic services to simulate a practical situation, the following settings were adopted: one ASC generates 3 to 5 workflows randomly (sequence of service types), and 1 or 2 ASCs occur in the generated workflows. The number of the service types for a workflow was set between 3 and 5, inclusive. We checked the composition times compared with the depth of the nested structure, and observed that the composition time increases linearly with the depth of the nested structure. When it is assumed that one ASC calls the inner ASC once or twice, the composition time grows according to O(2N) in the worst case, where N is the depth of the composition.

6.4 Composition Architecture

Our model provides an improved architecture for automatic service composition with multiple levels. The top level describes the topmost view of the concepts of ASC, and it works as an unchangeable framework of the architecture. The lower levels describe the upper-level blocks in more detail. One contribution of the model for ASC is that it gives a standard for the four-stage automatic composition architecture. Developers can therefore inherit classes or implement classes and interfaces in the top level as templates and for reuse, and extend them by customization for their own usage. The architecture clarifies current definitions and concepts of service composition roles and is expected to provide a referential perspective for development of service composition. Its other contribution is to give suggestions for integration or extension of the composition stages. The ASC is too complex and variable to provide a complete and unchangeable architecture. Our architecture contributes to taming the complexity of ASC. A model-driven approach would be helpful to provide a more flexible architecture for ASC.

7 ASC Illumination and Related Work

Recent efforts to improve service computing have focused on service composition and its automation. Our architecture aims at integrating the efforts consistently and providing a blue print. We first illuminate the works in terms of the four stage process, and then compare it with related works.

7.1 Illumination of ASC

7.1.1 Composition Approaches

Several studies have investigated methods and standards for service composition. One summarized service composition frameworks and compared them according to the main issues for composition [4]. Service composition has traditionally been divided into planning, discovery, selection, and execution, as shown in Fig. 1 [5]. Agarwal et al. [10] saw the composition task as logical composition that generates an abstract workflow, and physical composition to produce a workflow for execution.

7.1.2 Workflow Generation

In the planning stage of ASC, a classical planner generates an abstract workflow to satisfy the functional properties [2], [7]. McIlraith et al. used Golog, an implementation of situation calculus, to generate a workflow for service composition [2]. Hierarchical task network [18] is another approach used to generate sequential workflows. The HTN

planner can treat NFPs without binding information in terms of preconditions in the planning semantics, and this was extended for interaction during HTN planning [19]. Sohrabi et al. [20] presented a composition technique for user preferences in Golog based on the situation calculus. A state-transition system was combined with the planner to develop an abstract workflow in which NFPs could be considered [21]. Berardi et al. [22] suggested generation of the workflow using a finite state machine, which supports sequential workflow, but workflows are generally described as DAGs. The workflow language supports DAGs, and the Semantic Web service composition frameworks such as OWL-S [23], web service modeling ontology (WSMO) [15], web service modeling language (WSML) [24], and workflow language [25] all support DAGs.

7.1.3 Workflow Orchestration and Choreography

A workflow can be described as a DAG using Petri nets [26] or state charts [27]. Orchestration and choreography in service composition are also important concepts in deciding composition flow [15], [24]. ASC usually adopts the centralized management of services as one specific goal, and uses the same paradigm of orchestration. One study modeled orchestration and choreography of multiple services, and evaluated the performance [28]. Another study investigated fundamental modeling of orchestration and choreography [29]. Models and analysis for scalability in service [30], [31], [32] have been investigated.

7.1.4 Discovery and Selection with Nonfunctionality

Service discovery finds a desired functionality in registries for the services generated in the planning stage. In detail, the service discovery attempts to find a bundle of IOPEs of an operation in a service matching a query, but considers the semantics of the functionality because of their ambiguity and overlapping semantics [33]. One study suggests calculating the composability of services participating in a composition [34].

Service selection finds optimal service instances to satisfy nonfunctionality while fulfilling a goal [35]. The nonfunctionality can be QoS issues, user preferences, or user constraints [36], [37], [38]. Services are selected to satisfy overall NFPs with binding information. The selection problem is to find a set of services that satisfy the given constraints or boundary conditions among the candidates. Several approaches for selecting services have been investigated, such as CSP [39], constraint optimization [37], and linear programming [38].

7.2 Comparison with Related Work

The three categories for classifying service composition approaches—manual, semiautomated, and automated composition—are discussed in [40]. Many approaches for service composition have aimed at better automation. Fully automated composition requires understanding of the context, semantics, and problem space of the general composition domain. As the complexity of the service composition domain is so great, proposed approaches have dealt with small parts of fully automated composition with restrictions. We see that all the approaches to composing services automatically use basically the four-stage composition architecture wholly or partially with variants. Our architecture gives a standard for metrics to classify the approaches in the literature. We categorize the approaches according to the main stage(s) in the architecture to which they contribute for composition.

The first category clarifies approaches based on planning for workflow in ASC. They generate sequences of abstract tasks using AI planning techniques such as the situation calculus [2], HTN planner [18], [41], or a rule-based approach [42].

The second category is based on the combination of planning and discovery for composition. The planning stage produces an abstract workflow using rule-based techniques and the discovery stage produces a concrete workflow using a mapping function. Some discoverers include workflows that have been generated previously by the composer. Deelman et al. [43], Majithia et al. [40], and Agarwal et al. [10] contribute mainly to this category. Run et al. [44] suggested planning on heterogeneous ontology.

The third category sees discovery as a main role for composition and selection for execution as the next most important. It focuses on various service instances through semantic matching of services. The concept of discovery can include other service compositions, such as selection to locate suitable services. Sycara et al. [33] control composition by discovery, Zaremba et al. [45] by selection in discovery.

The fourth category focuses on the selection for whole composition. In the composition system, when an abstract workflow and service candidates of the workflow are given, these methods select a set of service instances to satisfy conditions such as QoS or user constraints. Approaches by Hassine et al. [37] using constraint optimization, Zeng et al. [35] using local and global optimization with integer programming, METEOR-S [38] using linear programming, Wagner et al. [46] using clustering techniques contribute to this category.

The fifth category is based on unifying planning, discovery, and selection. It models service composition as the problem space of planners or problem solvers, which provides discovery service implicitly and planning and selection at the same time. The approaches of Kona et al. [8] using constraint problem and Oh et al. [7] using planning are included in this category. Lecue and Mehandjiev [47] studied on optimal composition considering semantic causal links and selection.

All the composition approaches were designed from different viewpoints, with different targets and environments for composition; each approach satisfies the goals for which it was motivated. Research about service composition in the literature has mostly concerned one-step composition that fulfills a final goal once in a fixed ASC framework. There has been research about orchestration or choreography of workflows for service composition too, but they are not under a certain ASC framework, but general and naturally distributed service environment.

Our approach is about orchestration of nested workflow for service composition in a consistent ASC framework. Prior research on service selection relies on the premise that the properties have been identified already. Our framework

TABLE 1
Comparing ASC Approaches

Approaches / Quality Issues	Category 1	Category 2	Category 3	Category 4	Category 5	Our Approach
Main Role for ASC	Planning	Planning & Discovery	Discovery	Selection	Planning & Selection	All in Architecture Level
Functional Scalability	Medium	Medium	Low	Low	Medium	High
Seamlessness	Low	Medium	Low	Low	Medium	High
Adaptability	Low	Low	Medium	High	High	High

considers identification of abstract composition properties needed for seamless ASC. Table 1 compares the approaches using high-level metrics. The metrics assessing the quality of composition are main role, functional scalability, seamlessness, adaptability, and user interaction. Directly related works describe a simple and conceptual explanation of service composition with four stages [5] and conceptual comparisons of some composition approaches [6]. They are not concrete at an early stage and deal with just approximate composition. Our approach is more complete, together with functional scalability, seamlessness, and implementation framework.

8 Conclusion

In this paper, we have presented a functional scalable architecture for automatic service composition based on four stages by adding orchestration of nested workflows and composition property transformations to the existing process. We suggested two new blocks for a more complete outcome. The workflow orchestration manager deals with nested automatic service compositions to reach a final goal that cannot be completed in one composition step. The nested composition functions let us move away from one-step composition for a fixed goal. A method of capturing concrete binding information for the terms of NFPs from the goal-level description was also developed. It covers the classification of NFP terms in services by abstractness and a transformation algorithm for the terms using domain ontology.

Our multilevel description of the entire composition architecture gives procedural design information to developers, with flexibility in composition algorithms and approaches. It can also work as a standard for a general composition architecture that can provide clues for comparison. A design example shows how our architecture can be applied to any specific choice for ASC. The evaluation of the experiment of the example design shows the relationships between functional entities for composition and composition variables, and it shows enough scalability for medium sizes of services and ontology.

Future work includes proving superiority of our approach in each block and generalizing models to cover any number of staged architectures to extract transformation rules for model-driven approaches. A unified approach will be explored to address issues about practicality. Finally, combining our architecture with discovery of service links on natural service environment will be explored.

References

[1] M.P. Singh and M.N. Huhns, *Service Oriented Computing*. Wiley, 2005.

[2] S.A. McIlraith, T.C. Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46-53, Mar./Apr. 2001.

[3] B. Srivastava and J. Koehler, "Web Service Composition—Current Solutions and Open Problems," *Proc. ICAPS Workshop Planning for Web Services*, pp. 28-35, 2003.

[4] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51-59, Nov./Dec. 2004.

[5] D.B. Claro, P. Albers, and J.K. Hao, *Web Services Composition in Semantic Web Service, Processes and Application*, J. Cardoso and A. Sheth, eds., pp. 195-225, Springer, 2006.

[6] V. Agarwal, G. Chafle, S. Mittal, and B. Srivastava, "Understanding Approaches for Web Service Composition and Execution," *Proc. First Bangalore Ann. Compute Conf.*, pp. 18-20, 2008.

[7] S. Oh, D. Lee, and S.R.T. Kumara, "Effective Web Service Composition in Diverse and Large-Scale Service Networks," *IEEE Trans. Services Computing*, vol. 1, no. 1, pp. 15-32, Jan.-Mar. 2008.

[8] S. Kona, A. Bansal, M.B. Blake, and G. Gupta, "Generalized Semantics-Based Service Composition," *Proc. IEEE Int'l Conf. Web Services*, pp. 219-227, 2008.

[9] F. Lecue and A. Delteil, "Making the Difference in Semantic Web Service Composition," *Proc. 22nd Nat'l Conf. Artificial Intelligence*, pp. 1383-1388, 2007.

[10] V. Agarwal et al., "A Service Creation Environment Based on End to End Composition of Web Services," *Proc. 14th Int'l Conf. World Wide Web (WWW '05)*, pp. 128-137, 2005.

[11] E.M. Maximilien and M.P. Singh, "A Framework and Ontology for Dynamic Web Services Selection," *IEEE Internet Computing*, vol. 8, no. 5, pp. 84-93, Sept./Oct. 2004.

[12] C. Peltz, "Web Services Orchestration and Choreography," *Computer*, vol. 36, no. 10, pp. 46-52, Oct. 2003.

[13] H. Mizugai, I. Paik, and W. Chen, "Scalable Orchestration Strategy for Automatic Service Composition," *Proc. IEEE Int'l Conf. Computer Information Technology*, June 2010.

[14] "Nested AWSC Applet," http://semweb.u-aizu.ac.jp/NestedAWSCApplet/index.html, 2013.

[15] WSMO, "The Web Service Modeling Ontology (WSMO) Primer," http://www.wsmo.org/TR/d3/d3.1/v0.1/, 2005.

[16] M. Riemsdijk, M. Dastani, and M. Winikoff, "Goals in Agent Systems: A Unifying Framework," *Proc. Seventh Int'l Conf. Autonomous Agents and Multiagent Systems (AAMAS '08)*, pp. 713-720, 2008.

[17] F. De Paoli, M. Palmonari, M. Comerio, and A. Maurino, "A Meta-Model for Non-Functional Property Descriptions of Web Services," *Proc. IEEE Int'l Conf. Web Services*, pp. 393-400, 2008.

[18] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN Planning for Web Service Composition Using SHOP2," *J. Web Semantics*, vol. 1, pp. 377-396, 2004.

[19] U. Kuter, E. Sirin, B. Parsia, D. Nau, and J. Hendler, "Information Gathering during Planning for Web Service Composition," *Proc. Int'l Semantic Web Conf. (ISWC '04)*, pp. 335-349, 2004.

[20] S. Sohrabi, N. Prokoshyna, and S.A. McIlraith, "Web Service Composition via Generic Procedures and Customizing User Preferences," *Proc. Int'l Semantic Web Conf. (ISWC '06)*, pp. 597-611, 2006.

[21] P. Traverso and M. Pistore, "Automated Composition of Semantic Web Services into Executable Process," *Proc. Int'l Semantic Web Conf. (ISWC '04),* pp. 380-394, 2004.
[22] D. Berardi, D. Calvanese, G.D. Giacomo, M. Lenzerini, and M. Mecella, "Automatic Composition of E-Services that Export Their Behavior," *Proc. Int'l Conf. Service-Oriented Computing (ICSOC '03),* pp. 43-58, 2003.
[23] OWL Services Coalition, "OWL-S: Semantic Markup for Web Services," http://www.daml.org/services/owl-s/1.0/owl-s.pdf, 2003.
[24] WSML, "The Web Service Modeling Language (WSML)," http://www.wsmo.org/TR/d16/d16.1/v1.0/, 2008.
[25] YAWL Foundation, "Yet Another Workflow Language," http://www.yawl-system. com/index.html, 2013.
[26] R. Hamadi and B. Benatallah, "A Petri Net-Based Model for Web Service Composition," *Proc. Australasian Database Conf. (ADC '03),* pp. 191-200, 2003.
[27] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 4, pp. 293-333, 1996.
[28] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, "Choreography and Orchestration: A Synergistic Approach for System Design," *Proc. Third Int'l Conf. Service Oriented Computing,* pp. 228-240, 2005.
[29] A. Barros, M. Dumas, and P. Oaks, "Standards for Web Service Choreography and Orchestration: Status and Perspective," *Proc. Third Int'l Conf. Business Process Management (BPM '05),* pp. 61-74, 2006.
[30] A. Clark, S. Gilmore, and M. Tribastone, "Quantitative Analysis of Web Services Using SRMC," *Proc. Ninth Int'l School on Formal Methods for the Design of Computer, Communication and Software Systems,* pp. 296-339, June 2009.
[31] M. Bravetti, S. Gilmore, C. Guidi, and M. Tribastone, "Replicating Web Services for Scalability," *Proc. Third Conf. Trustworthy Global Computing,* pp. 204-221, Nov. 2007.
[32] J. Wu, Q. Liang, and E. Bertino, "Improving Scalability of Software Cloud for Composite Web Services," *Proc. IEEE Int'l Conf. Cloud Computing,* pp. 143-146, 2009.
[33] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated Discovery, Interaction and Composition of Semantic Web Services," *J. Web Semantics,* vol. 1, no. 1, pp. 27-46, Dec. 2003.
[34] B. Medjahed and A. Bouguettaya, "A Multilevel Composability Model for Semantic Web Services," *IEEE Trans. Knowledge and Data Eng.,* vol. 17, no. 7, pp. 954-968, July 2005.
[35] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Service Composition," *IEEE Trans. Software Eng.,* vol. 30, no. 5, pp. 311-327, May 2004.
[36] I. Paik, H. Takada, and M. Huhns, "Transforming Abstract QoS Requirements, Preferences, and Logic Constraints for Automatic Web Service Composition," *Proc. IEEE Int'l Conf. Web Services (ICWS),* pp. 764-765, Sept. 2008.
[37] B. Hassine, S. Matsubara, and T. Ishida, "A Constraint-Based Approach to Horizontal Web Service Composition," *Proc Fifth Int'l Conf. Semantic Web,* pp. 130-143, 2006.
[38] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Constraint Driven Web Service Composition in METEOR-S," *Proc. IEEE Int'l Conf. Services Computing,* pp. 23-30, 2004.
[39] I. Paik, D. Maruyama, and M. Huhns, "A Framework for Intelligent Web Services: Combined HTN and CSP Approach," *Proc. IEEE Int'l. Conf. Web Services (ICWS '06),* pp. 959-962, 2006.
[40] S. Majithia, D. Walker, and W.A. Gray, "A Framework for Automated Service Composition in Service-Oriented Architectures," *Proc. First European Semantic Web Symp.,* pp. 269-283, 2004.
[41] P. Rodriguez-Mier, M. Muciente, and M. Lama, "Automatic Web Service Composition with a Heuristic-Based Search Algorithm" *Proc. Int'l Conf. Web Service (ICWS '11),* pp. 81-88, 2011.
[42] E. Motta, J. Domingue, L. Cabral, and M. Gaspari, "IRS-II: A Framework and Infrastructure for Semantic Web Services," *Proc. Int'l Semantic Web Conf.,* pp. 306-318, 2003.
[43] E. Deelman et al., "Mapping Abstract Complex Workflows onto Grid Environments," *J. Grid Computing,* vol. 1, no. 1, pp. 25-39, 2003.
[44] K. Ren, N. Xiao, and J. Chen, "Building Quick Service Query List Using WordNet and Multiple Heterogeneous Ontologies toward More Realistic Service Composition," *IEEE Trans. Services Computing,* vol. 4, no. 3, pp. 216-229, July 2011.
[45] M. Zaremba, J. Migdal, and M. Hauswirth, "Discovery of Optimized Web Service Configurations Using a Hybrid Semantic and Statistical Approach," *Proc. Int'l Conf. Web Service (ICWS '09),* pp. 149-156, 2009.
[46] F. Wagner, F. Ishikawa, and S. Honiden, "QOS-Aware Automatic Service Composition by Applying Functional Clustering," *Proc. Int'l Conf. Web Services (ICWS '11),* pp. 89-96, 2011.
[47] F. Lecue and N. Mehandjiev, "Seeking Quality of Web Service Composition in a Semantic Dimension," *IEEE Trans. Services Computing,* vol. 23, no. 6, pp. 942-959, June 2011.

Incheon Paik received the ME and PhD degrees in electronics engineering from Korea University in 1987 and 1992, respectively. During 1993-2000, he worked as an associate professor at Soonchunhyang University, Korea. From 1996 to 1998, he was a visiting researcher at the State Key Laboratory, Beihang University, Beijing, China. He is currently an associate professor at the University of Aizu, Japan. His research interests include Semantic Web, web services and their composition, web data mining, awareness computing, security for e-business, and agents on Semantic Web. He served several conferences as a program chair and program committee member for numerous international conferences. He is a member of the ACM, the IEEE, and the IPSJ.

Wuhui Chen received the bachelor's degree from the Software College, Northeast University, China, in 2008 and the master's degree from the School of Computer Science and Engineering, University of Aizu, Japan, in 2011. He is currently working toward the PhD degree at the School of Computer Science and Engineering, University of Aizu, Japan. His research interests include services computing, cloud computing, mobile services, and social computing.

Michael N. Huhns received the BS degree in electrical engineering from the University of Michigan and the MS and PhD degrees from University of Southern California. He holds the NCR Professorship and is the chair of the Department of Computer Science and Engineering, University of South Carolina. He also directs the Center for Information Technology. He is the author of nine books and more than 200 papers in machine intelligence, including the coauthored textbook *Service-Oriented Computing: Semantics, Processes, Agents*. He serves on the editorial boards for 12 journals. He is a senior member of the ACM and a fellow of the IEEE.