Transformation of Intermediate Nonfunctional Properties for Automatic Service Composition

Haruhiko Takada and Incheon Paik

School of Computer Science and Engineering
University of Aizu, Aizu-Wakamatsu, Fukushima, Japan
harulin3@gmail.com, paikic@u-aizu.ac.jp

Abstract. Service-oriented computing provides an evolving paradigm for flexible and scalable applications of open systems. Web services and their automatic composition are in the mainstream of the evolution of new value-added services. Functional and non-functional aspects are considered together for automatic service composition (ASC). After locating suitable functionality for the required composition, non-functionalities are considered to select the final set of services. Non-functional properties (NFPs) obtained from users or identified during planning or discovery usually have abstract concepts that cannot be identified at the selection stage. In this paper, we propose a transformation technique for automatic composition that identifies binding information in the selection stage from intermediate abstract NFPs. The classification of abstraction level in NFPs, a model to define abstract and concrete NFPs, and an algorithm for transformation from intermediate to concrete level are presented. The identification of the binding information is based on domain ontologies for services. Evaluation in our algorithm according to characteristics of NFPs is shown. Our work will contribute to modeling and identification of NFPs for ASC.

Keywords: Automatic service composition, Non-functional property, Constraint, Semantic Web, Transformation.

1 Introduction

Service-oriented computing (SOC) enables new kinds of flexible and scalable business applications of open systems and improves the productivity of programming and administering applications in open distributed systems. 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 agents. 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.

Services composition traditionally follows a four-stage procedure: planning, discovery, selection, and execution [1]. In the planning stage, an abstract workflow to satisfy the functionalities is created. In the discovery stage, service candidates are located for each task based on functional properties (FPs) or QoS requirements. In the selection

A. Dan, F. Gittler, and F. Toumani (Eds.): ICSOC/ServiceWave 2009, LNCS 6275, pp. 218–227, 2010.

stage, a set of services that fulfill the non-functional properties (NFPs) is selected. Finally, the selected services are executed.

Many efforts have focused on the four stages or integrating them to improve the performance of automatic composition in several aspects. When a user gives a composition goal that includes functional and non-functional requirements to a composer, a sequence of abstract tasks to satisfy the user's functional requirements is generated, together with additional interim non-functional requirements, in the planning stage. The non-functional requirements may be described in high-level abstractions such as natural language, i.e., they do not have binding information, that is input and output of service instance located, that can be used in the selection stage. The composition result, an ordered set of services to satisfy all the requirements, is obtained by selecting the services that fulfill a set of NFPs (requested by the user or generated in the planning stage) from those discovered in the selection stage. The services in the selection stage should be bound and grounded for execution at the next stage, so they require binding information such as identification of service, operator, and parameters. As abstract NFP requirements from the user or the planner do not have this information, they must be related through transformations to the information before entering into the selection stage.

We present an approach for transforming abstract requirements into concrete ones that can be used in the selection stage. We suggest a model for transformation with classification of the levels of abstraction. Extracting proper semantics for requirements from natural-language requests imposes another heavy load of translation. We supply an NFP definition for abstract-level and intermediate-level that our algorithm can transform to concrete ones. Matching operations that bind abstract properties to concrete processes and properties require sophisticated procedures, because the context of non-functionality is based on large dynamics and semantics. Therefore, we exploit a transformation algorithm that works on domain ontologies constructed from information about real service domains.

The rest of the paper is organized as follows. Section 2 describes related work in the area of automatic service composition. Section 3 presents the data model of an NFP in Backus–Naur form (BNF). Section 4 explains the algorithm to transform intermediate level NFPs to concrete ones. Section 5 gives conclusions and mentions future work.

2 Related Work

In the selection stage of the four-stage composition architecture [1], optimized services are selected to satisfy overall NFPs (the term constraint is used to describe NFPs in many papers) with binding information. The selection problem is to find a set of services that satisfy the given constraints or boundary conditions from the candidates. Several approaches for selecting services have been investigated, such as the constraint satisfaction problem (CSP), the constraint optimization problem (COP) [2], and linear programming [3]. In those approaches, attribute–value clauses were used for NFPs, where an attribute identifies the NFP involved, and the value is the physical

value of the property offered by the related service. The policy-centered metamodel (PCM) [4] for NFPs was suggested to describe more general characteristics of properties in the real world. PCM can deal with constraint operators including range and can offer clustering. It is based on policies described by a BNF in the ontology and can be applied to general NFPs.

NFPs are described as attribute–value in the form of an ontology in the frameworks of OWL-S [5] and the Web service modeling language (WSML) (in description logic or first-order logic). The Web service modeling ontology (WSMO) [6] tries to add enriched semantic descriptions of existing Web services and defines an explicit conceptual model for semantic Web services based on the Web Service Modeling Framework. WSML was proposed for modeling Web services (functionality and non-functionality), ontologies, and related aspects based on WSMO, together with the formal grounding of the language based on logical formalisms such as description logic (DL), first-order logic (FOL), and logic programming.

Definition of abstractness and terms of constraints has been studied [7]. Our approach is based on the attribute–value scheme, which is the most representative and reasonable description for NFPs. As we defined the essential concept of non-functional characteristics, it can be extended easily. Automatic transformation from abstract NFPs to concrete ones with binding information to support the selection stage is presented in this paper.

3 Definition of Non-Functional Property

NFPs of services include QoS requirements, user preferences, and various constraints from internal or external requests. The constraints for QoS requirements consider execution cost, time, reputation, success rate, availability, reliability, and security (encryption, confidentiality), and attributes of each property can be classified as being of types resource or value [7,8]. The user preference constraints describe basic desire, priority, and choice in the form of FOL or DL. They can be an extended general formula to combine the atomic preference formulas. Many studies of service composition use the term constraint for NFP for a service. The term constraint can cover NFPs, QoS, preferences, and other constraints. In our research, we use constraint as a representative term and the attribute–value clause as the basic form for NFPs.

3.1 Level of Constraints as NFPs

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. High-level constraints are highly abstract and visceral for users, while low-level constraints can be dealt with in the selection stage because they contain binding information.

Level 3: Abstract Constraints. The constraints are at a high abstraction level that is near humans' natural concepts. All terms are abstract, and the constraint may not be

defined in formal terms. They can be in natural language (NL) or may contain several complex meanings in a keyword.

Level 2: Intermediate Constraints. The intermediate constraints 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 NL or nonterminal terms at level 3. All the terms are terminal and have not yet been bound to concrete terms.

Level 1: Concrete Constraints. These have relations, terms as binding information, and indexes of abstract workflow.

3.2 Data Model of Property

The data model of composition properties in our ASC is based on the above constraints and other properties for all composition stages.

Definition 1: FP and NFP for Planner

Request ::= **pDomain** *Parameter**

Parameter ::= **name value**

A *Request* consists of FPs and NFPs for the planner. It is described in simple FOL style. 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 SanFrancisco*)

(*priority earliness*) (*stay* 3))

This means that a user wants to go from Aizu to San Francisco as soon as possible and stay there for three nights. The *departurePlace*, *arrivalPlace* and *stay* are used as FPs and *priority* as an NFP. If the user sets (*priority cheapness*), the planner makes another plan, i.e., not by Shinkansen (high-speed train) but by ordinary railways or highway buses.

Definition 2: FP of Abstract Task and Workflow

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

*Workflow ::= AbstractTask**

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*))

It means traveling by train from Tokyo to Narita using the Narita Express.

And a *Workflow* is sequence of *AbstractTask*(s).

Definition 3: Abstract Constraint

AbstractConstraint ::= Relation NonTerminalTerm$_1$ NonTerminalTerm$_2$

NonTerminalTerm ::= **w$_1$ w$_2$ w$_3$ … w$_n$**

Relation ::= $= \mid \neq \mid > \mid < \mid \geq \mid \leq$

In this paper, an abstract constraint *AbstractConstraint* has two nonterminal terms *NonTerminalTerm* and one relation *Relation*. *NonTerminalTerm* has possibility that term has compound meaning. For example, *TotalCost* has means of the summation and meaning of the price. An example of an AbstractConstraint is

(< TotalCost 300000 yen)

It means that a user wants total cost is less than 300,000 yen.

Definition 4: Intermediate Constraint

IntermediateConstraint ::= Relation IntermediateTerm$_1$ IntermediateTerm$_2$

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

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 ontology shown in Section 4.3. The *Context* has two types of index: one is a pure index **index** for pointing and the other is a service domain **sDomain** that will give 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 AllServices sum) 300,000yen)

It has same meaning to abstract constraint's ones. This means that the total cost of all services in the abstract workflow must be less than 300,000 yen.

Definition 5: Concrete Constraint

ConcreteConstraint ::= Relation ConcreteTerm$_1$ ConcreteTerm$_2$

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

Variable ::= **index vDomain**

*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

(= (nil (2 Seat)) economy)

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

(= (Seat Airplane nil) economy)

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

Definition 6: Binding Information

Variable ::= ***i*ndex vDomain**

*Candidate ::= Service**

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

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

The candidate generator makes some service candidates *Candidate** as binding 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 binding information. The **sReference** is a reference to a service such as the URL of WSDL and some attributes. *Attribute** are input parameters and the output operator of a Web service. An *Attribute* has **vDomain**, **value**, and reference to a process **pReference** or parameter **aReference**. When the 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 Web service, and **aReference** is used as the parameter to obtain a value.

3.3 Construction of Ontology

Ontologies for the service domain, variable domain, operator, and pointing method are used in a transformer. Each Web service must belong to a class in the service domain ontology. Each parameter and operator of a Web service must belong to 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 several inference operations to extract knowledge about service variables. The ontology for operators includes all aggregating operators, such as *Summation* and *Average* that the service selector can treat, together with their aliases. The ontology for pointing includes a relational index for a workflow, such as *First* and *Last*. Our current transformation algorithm uses the ontologies to include all classes of the services and the variables being transformed. According to the characteristics of the various services domains, the ontologies for the domains can be changed.

4 Discovering Binding Information : Transformation of NFPs

To transform an intermediate constraint into concrete constraints, we must find binding information from the intermediate 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. We presume that the intermediate constraint can be obtained from users directly, or by translating abstract constraints to the intermediate level.

The following algorithm transforms an intermediate constraint to some concrete constraints that have binding information. Let *CC* be a set of concrete constraints as a transformation result. Let *it* be an *IntermediateTerm* in this constraint. Let *r* be a *Relation* in the constraint. *CC* is constructed as:

$$CC = r \times transform(it_1) \times transform(it_2).$$

The function *transform* is shown in Algorithm 1 in Fig. 1 and transforms an intermediate term into concrete terms.

Algorithm 1. *transform*(*it*)
if *it* is a constant value
  return *it*
else if *it.context* is an *index*
  return *(nil, (it.context, it.vDomain))*
else if *it.operator = nil*
  return *{ct: ct = (nil, (*index of *t, it.vDomain)),*
    t ∈ workflow, t.Context ⊆ it.Context}
else
  return *{(it.operator, {v: v = (*index of *t,*
    it.vDomain), t ∈ workflow, t.Context ⊆ it.Context})}
end

Fig. 1. Algorithm for Transformation

The function *transform* diverges according to the context and the 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. If an intermediate term has an index, it becomes the service identification index of a concrete term. In the next two cases, there is no direct information for service identification, so an inference on the service and the 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 (only 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. 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.

5 Example of Transformation Flow

In this section, we show an example of transformation flow using the travel scenario. There is a station at departure location A, and it connects to a station B at an airport (for departure). There is another airport (for arrival) at a location C. There are three trains from A to B on railroad line AB, and three airplanes from B to C. There are two hotels in C.

In the train, the fare for the line AB is 3000 and its timetable is described in the table 1. Three airplanes departure from Airport B at 11:00 and the cost and seat are described in the table 2. Information of two hotels in C is described in the table 3.

Table 1. Service candidates for trains from station A to station B

	Station A	Station B
$Service_1$	7:10	8:10
$Service_2$	9:10	10:10
$Service_3$	11:10	12:10

Table 2. Service candidates for airplanes form the airport B to the airport C

	Seat	Cost
$Service_4$	Economy	100,000
$Service_5$	Business	200,000
$Service_6$	Economy	150,000

Table 3. Service candidates for hotels in location C

	Cost (for 3 nights)
$Service_7$	30,000
$Service_8$	60,000

First, a user requests as follows.

I want to make a trip from A to C after 2009/11/23 8:00 and stay 3 days in a hotel in C. Seat for airplane is economy. Total cost is less than 150,000.

Next, a natural language processor translates it to some structural properties as follows.

Request = (*Trip* ((*departurePlace A*)(*arrivalPlace C*) (*stay* 3)))
$IntermediateConstraint_1$ = ((*TimeFrom 1 nil*) >*2009/11/23 8:00*)
$AbstractConstraint_1$ = (*SeatAirplane* = *economy*)
$AbstractConstraint_2$ = (Total*Cost* < *150,000*)

Next, we presume that two abstract constraints were translated by a translation method. As an example, nonterminal term "TotalCost" is to be translated to intermediate

terms. All candidates of an intermediate term that is made to compound "TotalCost" are *(TotalCost AllServices nil), (Total AllServices Cost), (Cost AllServices Total), (Total Cost nil)* and *(Cost Total nil)*. The confided intermediate term is only *(Cost AllServices Total)* because "Cost" is in the variable domain ontology and "Total" is in operator ontology that describes "Total" is same to class "Summation". Therefore the nonterminal term is translated to *(Cost AllServices Summation)*. Finally, two abstract constraints are translated as follow.

$IntermediateConstraint_2$ = ((*Seat Airplane nil*) = *economy*)
$IntermediateConstraint_3$ = ((*Cost AllService Sumation*) < *150,000*).
Next, the planner generates an abstract workflow as follows.
$AbstractTask_1$ = (*train* (*departureStation stationA*)(*line lineAB*)
(*arrivalStation stationB*)))
$AbstractTask_2$ = (*airplane* ((*departureAirport airportB*) (*arrivalAirport airportC*)))
$AbstractTask_3$ = *(hotel (place C))*

Two interim intermediate constraints for temporal connection are generated by the planner.

$IntermediateConstraint_4$ = ((*TimeTo 1 nil*) > (*TimeFrom 2 nil*))
$IntermediateConstraint_5$ = ((*TimeTo 2 nil*) > (*TimeFrom 3 nil*))

The transformation algorithm creates concrete constraints from intermediate constraints according to the ontology.

$ConcreteConstraint_1$ = (((*1 TimeFrom*)) *nil*) > *2009/11/23 8:00*)
$ConcreteConstraint_2$ = (((*2 Seat*)) *nil*) = *economy*)
$ConcreteConstraint_3$ = (((*1 Cost*)(*2 Cost*)(*3 Cost*)) *sum*) ≤ *150,000*)
$ConcreteConstraint_4$ = (((*1 TimeTo*)) *nil*) < ((*2 TimeFrom*)) *nil*))
$ConcreteConstraint_5$ = (((*2 TimeTo*)) *nil*) < ((*3 TimeFrom*)) *nil*))

Let us illustrate the transformation of the intermediate constraint $IntermediateConstraint_2$ to a concrete one. The meaning of the constraint is that the user wants an economy class seat on the airplane. The relation '=' and constant *economy* are used in the concrete constraint without change. (*Seat Airplane nil*) is an intermediate term that will be transformed. *Airplane* is domain *sDomain*. The only abstract task pointed to by the domain is $AbstractTask_2$ because $AbstractTask_2$.sDomain belongs to *Airplane*. The index of the new concrete term is therefore set to 2. *Seat* is the class name of the *variable domain ontology*. It will be used in the concrete term without change. In this case, only one abstract task has the service domain *Airplane*.

Finally, service candidates described in the tables 1, 2 and 3 and concrete constraints are solved by CSP solver. $Service_2$ is selected to an $AbstractTask_1$ by CSP solver because $Service_1$ does not satisfy $ConcreteConstraint_4$ and $Service_3$ does not satisfy $ConcreteConstraint_1$. $Service_4$ is selected to an $AbsractTask_2$ because $Service_5$ is not economy class and if $Service_6$ is selected, summation of cost is over. $Service_7$ is selected to an $AbstractTask_3$ because if $Service_8$ is selected, summation of cost is over. Therefore, the concrete process is *(*$Service_2$*,* $Service_4$*,* $Service_7$*)* If user allow this plan, the composer books to use these concrete services.

6 Conclusions and Future Work

Planning, discovery, and selection, and also translation and transformation of NFPs, are all important phases for ASC. We defined the level of NFPs, a data model of the abstract-level, intermediate-level, and concrete-level NFPs. We also presented a novel transformation algorithm to generate concrete constraints from intermediate constraints. The algorithm works well for intermediate constraints that are defined on the service domain ontology. The accurate translation of constraints that have highly abstract terms to intermediate constraints is important issue to solve abstractness for ASC too.

Future work will address the following issues. First, algorithm to translate abstract-level NFPs to intermediate-level ones will be studied. Second, more semantically complex FPs and NFPs including inference and rules on domain ontology will be defined in the intermediate level for more intelligent transformation.

References

1. Claro, D.B., Albers, P., Hao, J.K.: Web Services Composition in Semantic Web Service, Processes and Application, pp. 195–225. Springer, New York (2006)
2. Hassine, A.B., Matsubara, S., Ishida, T.: A Constraint-based Approach to Horizontal Web Service Composition. In: Proc. of International Semantic Web Conference, Athens, U.S.A, pp. 130–143 (2006)
3. Aggarwal, R., Verma, K., Miller, J., Milnor, W.: Dynamic Web Service Composition in METEORS. In: Proc. IEEE Int. Conf. on Services Computing, Shanghai, China, pp. 23–30 (2004)
4. De Paoli, F., Palmonari, M., Comerio, M., Maurino, A.: A Meta-model for Non-functional Property Descriptions of Web Services. In: Proc. IEEE Int. Conf. on Web Services, Beijing, China, pp. 393–400 (2008)
5. OWL-S: Semantic Markup for Web Services (2004), `http://www.w3.org/Submission/OWL-S/`
6. WSML. The Web Service Modeling Language (WSML). Final Draft (2008), `http://www.wsmo.org/TR/d16/d16.1/v1.0/`
7. Paik, I., Takada, H.: Modeling and Transforming Abstract Constraints for Automatic Service Composition. In: Proc. of IEEE International Conference on Computer Information Technology, Xiamen, China, pp. 136–141 (2009)
8. Traverso, P., Pistore, M.: Automated Composition of Semantic Web Services into Executable Process. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 380–394. Springer, Heidelberg (2004)