

A Method for Modeling Business Processes with OWL-T Language

Vuong Xuan TRAN¹ and Hidekazu TSUJI²

¹*Graduate School of Science and Technology, Tokai University, Japan, txvuong@yahoo.com*

²*School of Information Technology & Electronics, Tokai Univ., Japan, htsuji@keyaki.cc.u-tokai.ac.jp*

In order to satisfy incremental business demands, it is often required to combine functionalities of several services together. A number of approaches for service composition have been therefore proposed in both academic and industrial communities. These approaches, such as BPEL, WSCI, etc. can be categorized into static, manual service composition methods. In addition, by applying Semantic Web technologies, many research works have been investigated to support automatic service composition. Despite of the significant results being achieved, the task of service composition is still a challenging and complex issue. The main reason is that the former approaches require too much detail and technical interventions for defining business processes while the latter approaches are not much scalable for sophisticated applications. In this paper, we will introduce our approach for developing an ontology/language based on the OWL, called OWL-T (T stands for task), which can be used for users describing and specifying formally and semantically their needs at a high-level abstraction, which can be then transformed into executable business processes by underlying systems. The OWL-T aims at facilitating the modeling of complex demands or systems without regarding details of low-level and technical aspects of underlying infrastructure.

1. Introduction

SOA technologies, especially Web Services, have been recognized as promising approaches for distributed computing systems, such as enterprise and scientific applications. As business requirements grow more and more complex, it is essentially required to integrate available services in order to create new value added systems. This leads to the issue of service composition which both academic and industry researchers have spent a lot of efforts in its various aspects over recent years. Nevertheless, despite of significant and important achievement, service composition still remains as one of the most challenging tasks, especially on the Semantic Web. On the one hand, static composition methods like [1] require the understanding of detailed and technical knowledge in order to create a composite service. This is a highly complex task for dealing with the whole process manually. On the other hand, automatic approaches [2, 3] provide advantages for supporting automation but those methods are not scalable due to two main reasons. Firstly, they often base on a close world assumption in which detailed and complete knowledge of services must be available for operating. Secondly, it is very difficult to generate complete solutions for complex systems by relying on just generic requirements from users.

How to overcome the limitations and to take the advantages of these methods in order to support automatic service composition of complex systems? For this purpose, we developed an ontology and a language based on the OWL, called OWL-T, which can be used to express user demands or business processes at high-level abstraction without regarding technical details of underlying infrastructure. Such a task or business process template can be then transformed into executable processes by employing some automatic methods of service composition.

The remainder of this paper is structured as follows. The system architecture overview is presented in section 2. In section 3, we define and describe main concepts of the OWL-T, including task hierarchy, task structure, and task constructs, followed by some examples for illustrating how to use the OWL-T. Finally, we conclude the paper in section 4.

2. Overview of Our Approach

The service composition process in our system consists of five main steps: Business/Task Designing, Task Reasoning, Service Discovery and Selection, Planning, and Execution, as illustrated in Figure 1. The

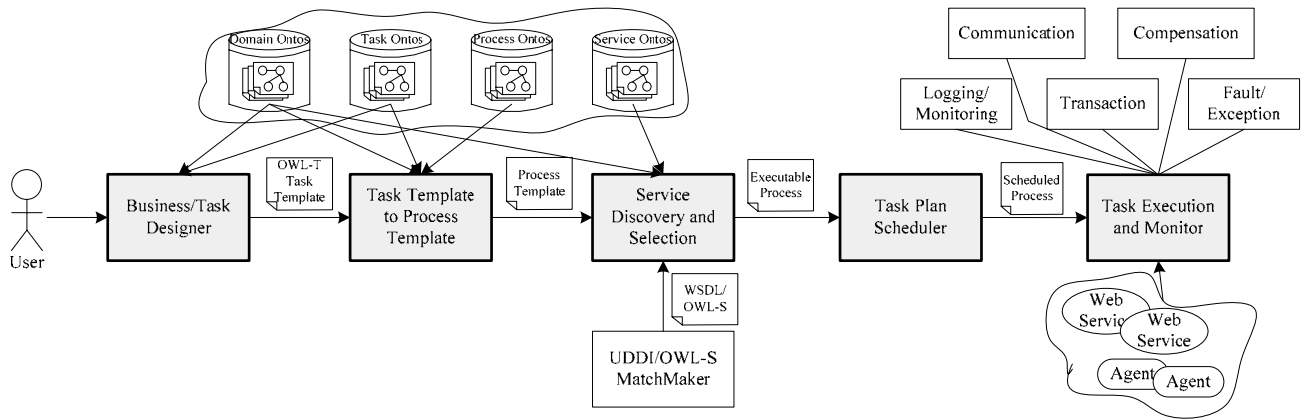


Figure 1. System overview

corresponding components of these steps are described in the following sections.

1) Business/Task Designer

This component mainly interacts with users and supports users specifying a task template or a business process template defined in the OWL-T. The resulting template reflects user demands at a high level abstraction. During the process of defining a template, a user may need to refer to some domain ontologies for essential concepts for describing desired tasks, such as information for inputs, outputs, preconditions, effects, and so forth. In addition, the user can reuse tasks defined in published task ontologies and modify them according to his/her needs. The user can also specify the order of component tasks combined together to satisfy the overall goal.

2) Task Template to Process Template

This module is responsible for realizing a corresponding process template for a task template defined in the OWL-T. In addition to an OWL-T template, domain ontologies and task ontologies are necessary to support the interpreting and reasoning process. Besides, we also introduce the concept of process ontology used to support the reuse of process templates for corresponding tasks. It, therefore, improves the performance of this component if some tasks are successfully realized before.

3) Service Discovery and Selection

In order to support for automatic and dynamic discovery and selection of services, this component requires semantic descriptions of available services defined in WSDL-S [4] or OWL-S [5]. It may also refer to some domain ontologies for necessary information, e.g. concepts used to define a service's parameters. Its main function is realized by a dynamic and automatic selection algorithm that receives a process template and semantic descriptions of services as inputs and returns an

executable process. This step also relies on UDDI registries and other complementary methods for semantic matching like WSDL-S and OWL-S matchmaker.

4) Task Plan Scheduler

The role of this component is to schedule the execution of an executable process, i.e. it makes a decision of which service and when the service will be executed. The scheduler also determines other aspects like synchronization, coordination and priorities of all component services.

5) Task Execution and Monitoring

The main task of this module is to invoke and to coordinate concrete services at runtime. Its functionalities are accomplished by various parts including communication, exception handling, transaction handling, logging/monitoring, and compensation handling.

3. OWL-T Specification

3.1. Task Hierarchy

A task specified in the OWL-T reflects a user's demand with necessary information. A task can be simple and able to be achieved by a service's operation or it can be complex and requires a composite service. Types of tasks are classified in a hierarchy in Figure 2.

- **Atomic Task:** An atomic task can be directly completed by an operation of a service. Some examples are tasks of converting currency or getting weather information.

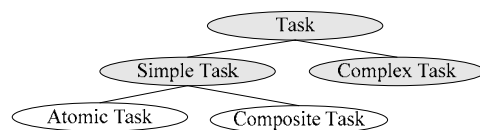


Figure 2. Task hierarchy

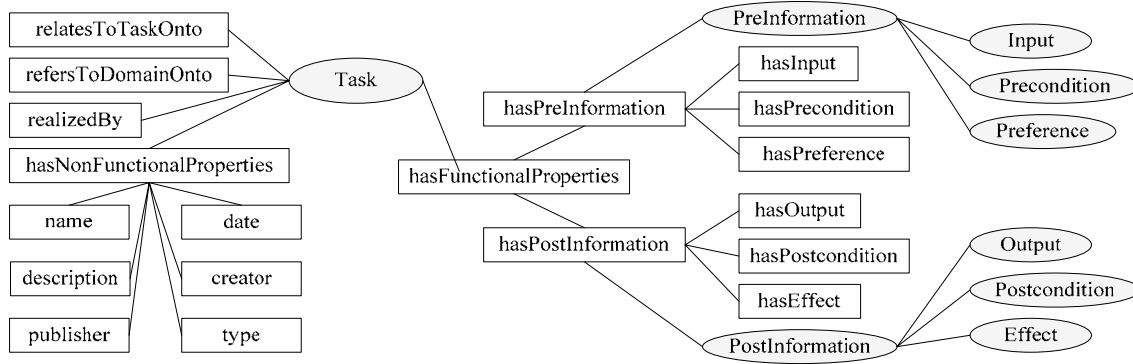


Figure 3. Task structure

- *Composite Task*: A composite task is necessarily completed by a composition of several services. In essence, a composite task consists of several atomic tasks. Some examples are tasks of booking a flight or booking a hotel.
- *Simple Task*: A simple task is either an atomic task or a composite task.
- *Complex task*: A complex task consists of one or more simple tasks. An example of a complex task is a task of planning a trip which consists of several tasks like flight booking, hotel booking, and car renting.

The task hierarchy model in the OWL-T plays two important roles. On the one hand, it helps a user easily specify his/her need by a simple task or a complex task without paying attention to underlying technical aspects. When selecting a complex task from a task ontology, the user can modify it by adding, removing, or ordering its simple tasks. On the other hand, by realizing a task is an atomic task or a composite (complex) task, the system can appropriately transform it into an executable process.

3.2. Task Structure

For a task, inputs, outputs, constraints (preconditions and postconditions), preferences, and effects are needed to realize an appropriate process for it. These are specified by using concepts and definitions in some domain ontologies. Besides that we also need information that helps users decide to select and to use a predefined task, such as name of the task, description of what the task can do, etc. A task may be also related to some processes of a process ontology.

In the following we describe in detail properties and components of the task structure. These properties and components are classified and grouped into five main parts which relate to task ontologies, domain ontologies, process ontologies, non-functional properties, and functional properties. Major properties and classes as well as their relationships are illustrated in Figure 3. The

central concept is the class *Task* which is described by five main properties connecting a task to other related elements, such as ontologies of domains, tasks, and processes, inputs, outputs, preconditions, and so forth.

- *relatesToTaskOnto*: This property specifies the relationship of a task with a task ontology. A task may relate to more than one task ontology.
- *refersToDomainOnto*: A task may refer to several domain ontologies which provide necessary concepts and definitions for describing its inputs, outputs, preconditions, preferences, and so forth.
- *realizedBy*: When a task is successfully realized by a process template, the template can be stored in a process ontology so that it can be reused later for the corresponding task to improve system's performance. A task can be realized by several processes.
- *hasNonFunctionalProperties*: This property has several sub-properties which are used to describe non-functional aspects of a task, such as *name* for naming the task, *description* for describing text information like what the task can perform under which condition, and other properties like *creator*, *publisher*, *date*, *type*.
- *hasFunctionalProperties*: In the OWL-T, we separate two types of functional properties: one is for describing information needed for a task before executing it and the other is for describing information after executing the task. The first one includes classes of *Input*, *Precondition*, and *Preference* which are subclasses of the class *PreInformation*. The second one includes *Output*, *Postcondition*, and *Effect* which are subclasses of the class *PostInformation*. We will give details of these classes in the following.

```

<simpleTask rdf:ID = "bookFlight">
  <relatesToTonto rdf:resource = "#TranTonto"/>
  <refersToDonto rdf:resource = "#FlightDonto"/>
  <name>BookFlight</name>
  <description>This task is for booking an
ordinary flight ticket...</description>
  ...
  <hasInput rdf:resource = "#CustomerInfo"/>
  <hasPrecondition>
    <Expression>(Price <= 500)</Expression>
  </hasPrecondition>
  <hasPreference>
    <Expression>(Flight in morning)</Expression>
  </hasPreference>
  <hasOutput rdf:resource = "#TConfirmation"/>
  <hasPostcondition>
    <Expression>(all account information of user
must be deleted)</Expression>
  </hasPostcondition>
  <hasEffect>
    <Expression>(ticket is delivered at user's
address)</Expression>
  </hasEffect>
</simpleTask>

```

Figure 4. A simple task bookFlight defined in the OWL-T

Input: An input represents information required for performing a task, e.g. customer information and ticket order.

Output: An output describes information returned after performing a task, e.g. a ticket confirmation.

When defining a task, we may need to refer to domain ontologies for concepts and definitions that are used to describe inputs and outputs of the task.

Precondition: A precondition represents conditions that must hold in order for a task to be performed successfully, e.g. price must be less than \$1000.

Preference: A preference describes properties of preferred solutions, e.g. preferring to fly at daytime rather than at night.

Postcondition: A postcondition represents conditions that must hold after performing a task, e.g. deleting user's private information after a task finishes.

Effect: An effect describes actual events that occur after performing a task, e.g. a ticket is delivered at a user's address.

An example of a simple task *bookFlight* defined in the OWL-T is illustrated in Figure 4. We simplified the example for illustration purpose.

3.3. Task Constructs

Task constructs are used to define a complex task from simple tasks or other complex tasks. These include *OrderedAll*, *NonOrderedAll*, *SynNonOrderedAll*, *Switch*, *Choice*, *Or*, *IfThenElse*, *RepeatUntil*, *WhileDo*.

- *OrderedAll*: An *OrderedAll* construct contains a list of tasks which are performed in their order of appearance, e.g. booking a flight ticket before booking a hotel.

```

<orderedAll>
  <simpleTask rdf:ID = "bookFlight"/>
  ...
</simpleTask>
<nonOrderedAll>
  <simpleTask rdf:ID = "bookHotel">
    ...
  </simpleTask>
  <simpleTask rdf:ID = "bookRestaurant">
    ...
  </simpleTask>
  <choice>
    <condition>
      <Expression>none of login</Expression>
    </condition>
    <bag>
      <simpleTask rdf:ID = "findMovies">
        ...
      </simpleTask>
      <simpleTask rdf:ID = "findOperas">
        ...
      </simpleTask>
    </bag>
  </choice>
</nonOrderedAll>
</orderedAll>

```

Figure 5. A complex task defined in the OWL-T

- *NonOrderedAll*: A *NonOrderedAll* construct contains a bag of tasks that can be performed without regarding of their order of appearance, e.g. booking a hotel and reserving a restaurant.
- *SynNonOrderedAll*: A *SynNonOrderedAll* construct contains a bag of tasks that can be performed without regarding of their order of appearance, but with a synchronization, e.g. booking a flight and transportation before booking a hotel (i.e. booking a hotel after both previous tasks completing and reaching a synchronized condition).
- *Switch*: A complex task defined by this construct consists of an ordered list of component tasks which each of them is connected to a condition. Component tasks will be checked with their condition, one by one in their order. The first task which its condition holds will be selected.
- *Choice*: A complex task defined by the *Choice* construct includes two parts: a condition and a bag of component tasks. According to the condition, none or more component task are selected.
- *Or*: This construct is similar to the *Choice* construct but only one component is selected.
- *IfThenElse*, *RepeatUntil*, *WhileDo*: These constructs, as in programming languages, are used for expressing conditions or iterations of tasks.

As an example, in Figure 5, the defined complex task states that the *bookFlight* task must be done before the others. Then, in any order, the tasks *bookHotel*,

bookRestaurant, and none or more of tasks *findMovies*, *findOperas* will be performed. The condition for selecting none or more tasks of *findMovies*, *findOperas* is that there is no login requirement from services used to realize those tasks. This reflects the fact that a user often does not want to provide his/her private information like email when doing such optional tasks.

4. Conclusions

We have proposed a semantic framework which supports for automatic and dynamic service composition. The main concern is the OWL-T, an ontology and a language used to express user demands in terms of business processes or task templates which can be used to facilitate an automatic process of service composition.

References

- [1] A. Alves, et al. Web Services Business Process Execution Language 2.0, www.oasis-open.org, 2006.
- [2] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN Planning for Web Service Composition using SHOP2. *Journal of Web Semantics* 1(4), 377–396, 2004.
- [3] M. Klusch, A. Gerber, and M. Schmidt. Semantic Web Service Composition Planning with OWLS-XPlan. In *Proc. of the AAAI Fall Symposium on Semantic Web and Agents*, Arlington VA, USA, AAAI Press, 2005.
- [4] R. Akkiraju et al. Web Service Semantics - WSDL-S, <http://www.w3.org/ Submission/WSDL-S/>, 2005.
- [5] A. Ankolekar. OWL-S: Semantic Markup for Web Services, <http://www.daml.org/services/owl-s/1.0/>, 2003.