A Model-Driven Approach for Reusing Service Compositions

Carlos Granell, Michael Gould, Dolores María Llidó, Rafael Berlanga
Department of Information Systems
Universitat Jaume I, Castellón (Spain)
{carlos.granell, gould, doloresmaria.llido, berlanga}@uji.es

Abstract

The web service approach provides mechanisms for simplifying application integration. However, to meaningfully facilitate scalable development and maintenance of web service applications it is necessary to pay attention to the reuse of not only atomic web services but also existing compositions. In an effort to increase the level of service reusability, we present a model-driven approach providing a more flexible, structured service model to facilitate reuse of existing compositions to create new complex compositions. The goal of this approach is two-fold: provide a simple but expressive service model represented jointly in terms of reusable services, building blocks for constructing and reusing new compositions, and abstract patterns, for managing reusable service composition; and provide a model-driven approach for transforming such reusable building blocks into executable processes. The paper focuses on both how the abstract pattern set is derived and how the model-driven approach facilitates the creation of executable process.

1. Introduction

Recent advances in the development of web services technology has led to several approaches addressing the common problem of web service composition [13]. In [16] the authors state that this problem can be viewed as three basic tasks: the first one is to make a plan composed of activities that describes the needed functionality to provide a solution of the problem. The second task is to discover the web services that fit the activities in the plan. Finally, the third one establishes concrete interactions with those web services involved in the plan and executes the resulted service composition. Although all tasks are important (see [11] for details about how to combine service discovery, composition and execution), here we focus just on the last problem so-called concrete composition and execution in order to provide a way for maximizing service composition reuse and generating executable processes following a model-driven approach that facilitates the scalable development and maintenance of web service applications.

In this paper we also address service flexibility, feature that are closely related to service granularity and have a great impact on the level of reusability. It is necessary to pay a close attention to the granularity of a web service to be composed because modeling systems as small pieces offers great flexibility and reusability. Moreover, that leads to a better decoupling of the system.

Our research on providing the above features of reusability and flexibility for the web service composition has lead to model-driven approach, which defines the notion of reusable service as a building block with which to assemble and manage reusable services (building blocks) to construct new, more elaborated reusable services. Here, we present the abstract patterns, the second key piece in our model-driven approach that permits to put together reusable services. Our framework interprets the model, reusable services and abstract patterns, to semi-automatically generate executable processes by means of model-driven transformations. We then discuss its implementation in order to underscore the need for quick and flexible service composition.

2. Model

Our belief is that web service composition solutions require basic building blocks in terms of a simple and useful model for simplifying the design of complex applications and enabling high-level reuse. Then, we define our model that is the basis for the model-driven approach proposed in this work: a *reusable service model* to represent and manipulate service descriptions as building blocks; and *abstract patterns* for managing such reusable services and their orchestration. Once defined the model, the model-driven approach describes a methodology that allows us to manage and reuse reusable services and eventually transform them into executable processes in a semi-automated way according to abstract pattern's business logic. The rest of the section overviews the proposed model to later discuss in more detail the abstract patterns in section 3. Section 4 completes our approach presenting the model-driven framework and prototype implementation.

2.1. Reusable service model

Assuming the service-oriented architecture (SOA) paradigm, the core of our model-driven approach is essentially collections of reusable services that are easy to handle and use by users or programs. The key idea here is to include aspects that are needed to discover and compose services —simple, composite or WSBPEL process— but avoiding overloading the underlying model with uncommon features, making the service model ease to be reused later. To obtain a simple but representative reusable service model, this is described by an interface captured in an abstract description, which is characterized by a set of aspects expressing what functionality a reusable service offers and how to structure the data exchanged with it [8]. We then define the following aspects for defining a reusable service:

Descriptive aspects are metadata concerned with the context in which a reusable service performs. Some examples are textual description and service category.

Functional aspects detail, syntactic and semantically, service functionality in terms of the operation name and input-output parameters.

Structural aspects show how a reusable service is internally organised as a combination of other simpler reusable services (orchestration).

Binding aspects establish and filter (additional) data flow between reusable services permitting two services combine syntactically.

A reusable service encapsulates all the above aspects representing different but complementary views. For example, descriptive and functional aspects are useful for discovery because they express the functionality offered by a reusable service, while structural and binding aspects are critical for composition and execution. As the latter aspects are not necessary during the discovery process, they should be kept unknown to the user in this phase for simplicity and clarity reasons. To offer a suitable level of encapsulation, access to a reusable service is controlled by a public and private interface. The public interface openly expresses the service's descriptive and functional aspects. The private interface represents an internal view of the service, encapsulating structural features such as abstract patterns as well as the necessary binding aspects for data flow. From a practical view, a reusable service is a simple but expressive enough model to represent features needed in practice (see [9] for details about reusable service model in a real scenario).

2.2. Abstract patterns

Workflow patterns [2] are crucial to fully define our model because they express how a reusable service is internally organized, so workflow patterns are closely related to the reusable service's structural aspects and hence to the reusable service's private interface.

Since a composition process can be considered the application of a composition operator [17], in our model-driven approach, workflow patterns serve as composition operators for composing reusable services. However, as referenced in [1, 13], several modeling languages exist for describing web services compositions yet they support different and overlapping workflow patterns. It is then necessary to define a subset of consistent and non-redundant workflow patterns, named *abstract patterns* here, which let managing more easily reusable service orchestration, as we see in the next section.

Reusable services by definition address reusability. To complement the reusable service model, abstract patterns permit service flexibility. Users usually find service operations that have operation-level mismatches but perform the same functionality. For example two service operations have the same functionality but may differ in the operation name or type of parameters. Abstract patterns provide a solution to avoid mismatches of service operations by hiding them under the same reusable service operation due to abstract patterns. This lets us take advantage of the flexibility feature due to availability of multiple service operations for the same reusable service functionality.

3. Abstract patterns analysis

Before starting the abstract pattern analysis, we should explain why we have chosen the workflow pattern approach [2] as basis of our work and which requirements and criteria guide our pattern analysis.

Most existing languages for web service composition and for describing business processes come from the workflow area [3]. Various works compare such composition languages with workflow patterns [1, 18, 20] highlighting the strong link between them. Furthermore, our pattern analysis takes the workflow patterns provided by [2] as starting point to derive the target set of abstract patterns for our approach.

Our pattern analysis is driven by the following criteria. Simplicity is one goal in order to keep the number of redundant workflow patterns to a minimum, in contrast to overlapping and alternative patterns present in WSBPEL [15]. In addition, the derived abstract patterns should fit nicely with the reusable service model features such as independence (no data dependence), flexibility, and reusability. The remainder of this section details our pattern analysis through a three-step approach. First, patterns suitable for web service context in general are identified; then these are fully fitted for specific requirements of reusable services; finally, the target set of abstract patterns are grouped into two categories to underscore either reusability or flexibility support.

Id	Pattern name	WS	RS
Basic control flow patterns			
1	Sequence	√	√
2	Parallel split	√	√
3	Synchronization	√	√
4	Exclusive choice	√	√
5	Simple merge	√	√
Advanced branching and synchronization patterns			
6	Multi choice	√	√
7	Synchronizing merge	√	√
8	Multi merge	√	√
9	Discriminator	√	√*
Structural patterns			
10	Arbitrary cycles	√	√*
11	Implicit Termination	x	x
Patterns involving multiple instances			
12	M.I. without Synchronization	√	x
13	M.I. with design time knowledge	√	x
14	M.I. with Runtime Knowledge	x	x
15	M.I. without Runtime Knowledge	x	x
State-based patterns			
16	Deferred choice	√	x
17	Interleaved Parallel Routing	√	x
18	Milestone	x	x
Cancellation patterns			
19	Cancel activity	x	x
20	Cancel case	x	x

Table 1. Workflow patterns relevance for web services (WS) and reusable services (RS). Symbol * denotes pattern original meaning has changed.

3.1. Abstract patterns for web services

We begin with the workflow patterns listed in Table 1 that are grouped in five categories regarding their original functionality. Numbers in brackets throughout this section correspond to the "Id" column in Table 1.

Basic control flow patterns. All these patterns are considered relevant because they are simpler but also basic patterns: *sequence* (1), *parallel split* (2), *exclusive choice* (4), and their respective counterpart patterns regarding synchronization such as *synchronization* (3) and *simple merge* (5).

Advanced branching and synchronization patterns. These patterns are fully supported neither by some workflow engines [2] nor by most composition languages [1]. *Multi choice* (6) and *synchronization merge* (7) patterns are significant for web services because they provide the logical function OR. The next one, *multi merge* pattern (8), joins two or more branches without synchronization. For instance, a user wants to execute two gazetteer services –a service that takes a toponym, address or business name and returns a list of possible matching geographic locations, typically as street addresses or x,y coordinate pairs in a known coordinate reference system such as latitude/longitude–, returning a list of possible matching geographic locations. *Multi merge* may be suitable when such gazetteer services are being executed in parallel arriving at the joining service "post locations list to user", which is executed twice for each incoming gazetteer service. On the other hand, the *discriminator* pattern (9) is quite useful because can potentially improve service flexibility. Now both gazetteer services are executed in parallel yet, in contrast to *multi merge*, the first gazetteer service to complete its task will execute the following service in the chain ("post locations list to user") while the other gazetteer service's result will be ignored. Then, two alternative services with the same functionality are available for execution.

Structural patterns. This category groups *arbitrary cycles* (10) and *implicit termination* (11) patterns. For the former, loops are basic constructs for modeling web services composition. For the latter, the *implicit termination* pattern simply ends a workflow execution because there is nothing to do. From a web services viewpoint, there is no need to support explicitly a termination pattern.

Patterns involving multiple instances. This set of patterns generates several instances of one activity, normally at run-time. The first two patterns (12-13) are interesting in the web service context. The pattern 12 involves multiple instances running in parallel without synchronization. Suppose that users are now interested in retrieving locations for a places (toponym) list. A gazetteer service will be invoked several times depending on the number of requested places in the list. The pattern 13 is similar to the previous one but involving synchronization in this case. Regarding the last two patterns (14-15), they are not relevant for web services because the number of instances is unknown at design time, just when our model-driven approach is carried out.

State-based patterns. This category involves *deferred choice* (16), *interleaved parallel routing* (17), and *milestone* (18). *Deferred choice* (16) is identical to *exclusive choice* (4) except that the condition is given by an external input. Next pattern (17) is actually an unordered sequence of activities. Both patterns are attractive for the web services context. The *milestone* pattern (18) holds great importance at run-time yet it has no relevant impact at design-time. Moreover, taking into account the simplicity criterion, the *sequence* (1), *parallel split* (2), *choice* (4, 6), and *loops* (10) constructs can be jointly combined to offer the same meaning of *milestone* pattern [14].

Cancellation patterns. Lately, we do not consider necessary either an explicit *cancel activity* (19) or *cancel case* (20) patterns for the web services context, as in the case of *implicit termination* (11).

3.2. Abstract patterns for reusable services

Workflow patterns considered for web services (marked with "√" in column "WS" in Table 1) are the starting point for defining the abstract patterns for reusable services.

Beginning with the case in sequence, we consider only ordered *sequences* (1) of reusable services. Because *unordered sequences* (17) assume no data dependence among contained workflow activities [2] they can be modeled as sequential. Therefore, in our approach, a sequence is always modeled by the pattern (1).

Regarding parallel patterns (2-8, 16) we select the minimum pattern set for defining the basic logical functions AND, XOR and OR applied to reusable services. Every logical function is fully defined by a split and a join pattern. For instance, both *parallel split* (2) and *synchronization* (3) correspond to the logical function AND. The function XOR comprises *exclusive choice* (4) and *simple merge* (5). The same rationale is applied to OR function that is represented by *multi-choice* (6) and *synchronizing merge* (7).

The remaining parallel patterns (8, 16) are not considered. *Multi-merge* (8) can be represented as a combination of *parallel split* (2) and *synchronization* (3) patterns [10]. Then, the *multi-merge* pattern (8) is non relevant as our goal is to avoid alternative, overlapping patterns due to simplicity. *Deferred choice* pattern (16) is not an appropriated pattern either since it depends on external inputs. Obviously, we prefer reusable services without external data dependences to increase the reusability level among reusable services.

The following discussion is about *discriminator* pattern (9) that is not only useful for web services but also for modeling reusable services as it meets nicely with flexibility requirement. However this pattern cannot be directly transformed into WSBPEL constructs [18]. Then, we derive a *sequential discriminator*, renamed here to denote a change, that maintains the essential meaning of the *discriminator* (9) but now contained services are running in sequence. Returning to our gazetteer example, the main difference now is that gazetteer services are executed in sequence rather than in parallel. If the first gazetteer completes successfully it returns its locations list and the other gazetteer is ignored. If first one fails, the next gazetteer is invoked. The essential meaning is maintained: consider either the first service's result or the second service's result, but not both.

As in *discriminator* (9), we assume a *simple loop*, renamed here to denote a change, instead of *arbitrary cycles* (10) because the former consider a single entry and exit point. In addition, *arbitrary cycles* are not supported by WSBPEL [1, 18, 20]. Finally, patterns 12 and 13 are not relevant for reusable services applying the same simplicity rationale as in the case of *multi-merge* (9): both can be modeled with basic patterns [14] already contemplated in our analysis.

3.3. Selection and composition abstract patterns

Here we focus on how to combine the relevant set of abstract patterns (marked with "√" in column "RS" in Table 1) derived from the previous analysis.

Id	**Pattern name**	**Pattern combinations**
Composition patterns		
CP1	SEQ	Sequence (1)
CP2	AND	Parallel split (2) + synchronization (3)
CP3	XOR	Exclusive choice (4) + simple merge (5)
CP4	OR	Multi choice (6) + synchronizing merge (7)
CP5	LOOP-COND	Simple loop(10)
CP6	LOOP-ITER	Simple loop (10)
Selection patterns		
SP1	AND-DISC	Parallel split (2) + sequential discriminator (9)
SP2	OR-DISC	Multi choice (6) + sequential discriminator (9)

Table 2. Selection and composition abstract patterns.

The reason to combine such abstract patterns is that some of them cannot be individually applied to model reusable services. It is important to keep in mind that the ultimate goal of abstract patterns is to serve as composition operators and that reusable services are the unique building blocks in the composition process. It is obvious that a given reusable service will generate a hierarchical, tree-based structure in which each node represents the different reusable services involved in such a composition. Abstract patterns connect parent and direct children nodes. For this reason, it is necessary to pair the abstract pattern set considered in section 3.2 instead of treating them individually. Then, abstract patterns are grouped into two categories: *selection patterns* that place emphasis on flexibility feature whereas *composition pattern* on reusability and service orchestration. Table 2 summarizes the resulting *selection* (SP1-2) and *composition patterns* (CP1-6).

Composition patterns. Because the reusable service's hierarchical structure is traversed in a depth-first algorithm (children nodes are first reached than parent), we consider split patterns for the children nodes (that are first reached) and the join patterns for the parent nodes. Thus, three split patterns (2, 4 and 6 in Table 1) may be combined with three join patterns (3, 5 and 7 in Table 1). However, only some of these combinations match. In particular, the split and join patterns are compatible only if both express the same logical function (AND XOR, OR). Therefore, appropriate combinations of split and join patterns lead to three composition patterns (CP2-4 in Table 2).

The specification of the *sequence* pattern (1) is immediate because it can be applied individually to reusable services. Then *sequence* pattern (1) becomes the composition pattern SEQ (CP1 in Table 2).

Two kinds of simple loop-based patterns are derived from *simple loop* pattern (9) for reusable services. On one hand, a conditional loop or COND-LOOP (CP5 in Table 2) can be used to repeat a reusable service until a condition *cond* is fulfilled. It is important to point out that this condition has to be specified in terms of internal data values described within the reusable service's data flow (bindings aspects), preserving thus the autonomy and independence. On other hand, an *iterative loop* or ITER-LOOP (CP6 in Table 2) iterates *n* times (known at design-time) on a certain reusable service.

Selection patterns. The *sequential discriminator* pattern (9) acts as a join pattern because it allows us to select the result of one web service from a potential web service set. As in the case of CP-2-4 patterns, we need to add a split pattern due to the hierarchical structure defined by a reusable service. The split pattern deals with the candidate web services set whereas the join pattern refers to the parent reusable service. Furthermore, adding *parallel split* (2 in Table 1) or *multi choice* (6 in Table 1) as a split pattern to the *sequential discriminator* leads to two selection patterns named AND-DISC (SP1 in Table 2) and OR-DISC (SP2 in Table 2) respectively. The former is suitable when all candidate web services are considered initially. For the latter the user explicitly includes some conditions to discard some web services from the initial web service set.

Selection patterns strive to provide a solution for operation-level mismatching mentioned in section 2. For instance several web service operations under the same reusable service operation are modeled by the AND-DISC selection pattern, what means that if the first web service operation in the list fails, the selection pattern logic takes the second one and so on, until any of the web service operations is successfully completed.

4. Model-driven framework

At this stage, once the model is defined, it is desirable to provide a framework to facilitate managing the model presented here: reusable services and abstract patterns. This section describes the framework architecture and some key implementation features.

4.1. Architecture

Existing composition languages for web services like WSBPEL follow a two-level architecture split into the application level (processes) and the concrete services level (web services). In order to provide the required levels of flexibility and reusability, we rely on the abstraction and decomposition ideas to simplify and structure more easily the proposed architecture [19, 21]. Our reusable services reside in the abstract services level, an additional layer between the other two layers.

The key principle in the abstract services level is that a decomposition relationship is defined between different reusable services, where a certain reusable service is decomposed (or implemented) by the reusable services at lower level which in turn implement it. A reusable service is then implemented by simpler reusable services at lower levels. This relationship between adjacent levels produces a hierarchical structure in which, given a reusable service (parent node), direct children nodes are reusable services at lower level. By simply traversing the structure it eases to find out the functionality offered by a reusable service because children nodes are the functional decomposition of a parent node. Therefore, the role of the abstract services level is to convert web service compositions (lower level) into executable processes (higher level) but at the same time increasing the levels of reusability and flexibility in the target processes.

4.2. Implementation

The model-driven framework, which aligns with the earlier layered architecture, has been implemented in a Java-based prototype as a set of plug-ins on top of the Eclipse platform. The main components are described briefly as follows:

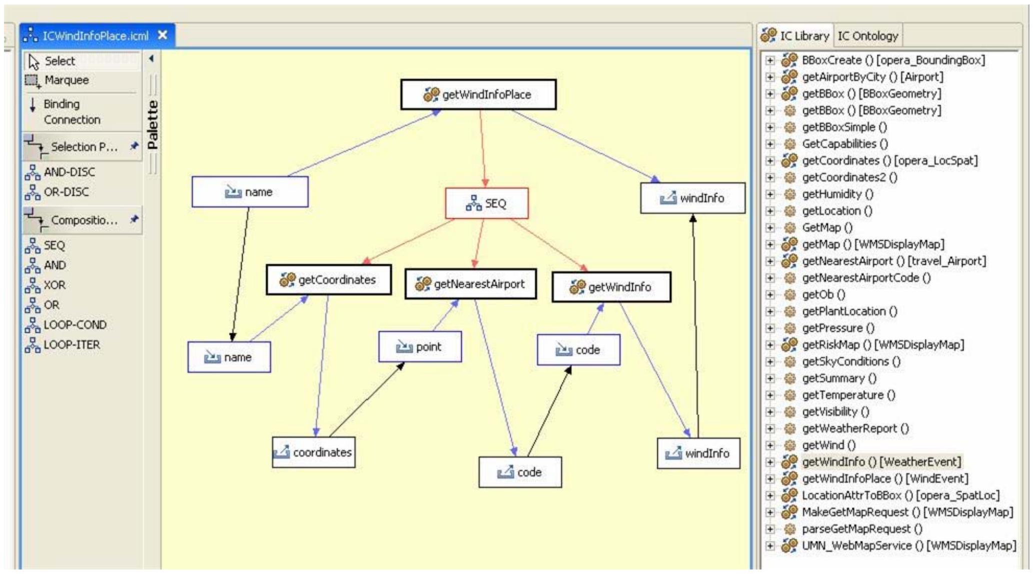
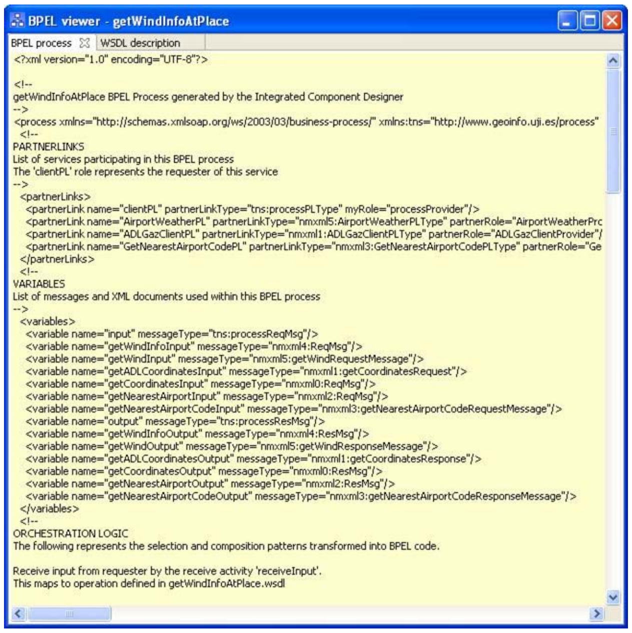


Figure 1. User interface for composing reusable services using abstract patterns.

Model creation and representation components are concerned both with creating reusable services (model) and with generating the corresponding descriptions that represent a reusable service. All of the reusable service's aspects (descriptive, functional, structural and bindings) are encoded in an XML-based description (private interface), and only functional aspects are publicly available in a WSDL-S description [4].

Reusable service management and composition components are responsible for constructing complex reusable services by incrementally aggregating and reusing existing ones taken from the repository. Figure 1 illustrates our prototype to compose graphically reusable services. The left side of the figure shows the abstract patterns palette that allows users to select the proper abstract pattern (composition pattern SEQ in Figure 1). The graphical editor is in the center with boxes representing reusable service operations (black border), input and output parameters (blue border), and abstract patterns (red border). The *Library* view, in the right side, permits users to inspect the service operations available of registered reusable services. Reusing an exiting reusable service is as easy as dragging it from the *Library* view and dropping it into the graphical editor.

Model-driven transformation components maps reusable services and abstract patterns into a WSBPEL process. At this stage we could model this task manually using directly WSBPEL constructs in any existing editor. Yet, the model-driven transformation components permits generating WSBPEL processes (code ready to be executed not only skeletons) from reusable services (model), as for example the WSBPEL description generated in Figure 2.



```
<?xml version="1.0" encoding="UTF-8"?>

<!--
getWindInfoAtPlace BPEL Process generated by the Integrated Component Designer
-->
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/" xmlns:tns="http://www.geoinfo.uji.es/process"
  <!--
PARTNERLINKS
List of services participating in this BPEL process
The 'clientPL' role represents the requester of this service
-->
  <partnerLinks>
    <partnerLink name="clientPL" partnerLinkType="tns:processPLType" myRole="processProvider"/>
    <partnerLink name="AirportWeatherPL" partnerLinkType="nmxml5:AirportWeatherPLType" partnerRole="AirportWeatherPrc
    <partnerLink name="ADLGazClientPL" partnerLinkType="nmxml1:ADLGazClientPLType" partnerRole="ADLGazClientProvider"/
    <partnerLink name="GetNearestAirportCodePL" partnerLinkType="nmxml3:GetNearestAirportCodePLType" partnerRole="Ge
  </partnerLinks>
  <!--
VARIABLES
List of messages and XML documents used within this BPEL process
-->
  <variables>
    <variable name="input" messageType="tns:processReqMsg"/>
    <variable name="getWindInfoInput" messageType="nmxml4:ReqMsg"/>
    <variable name="getWindInput" messageType="nmxml5:getWindRequestMessage"/>
    <variable name="getADLCoordinatesInput" messageType="nmxml1:getCoordinatesRequest"/>
    <variable name="getCoordinatesInput" messageType="nmxml0:ReqMsg"/>
    <variable name="getNearestAirportInput" messageType="nmxml2:ReqMsg"/>
    <variable name="getNearestAirportCodeInput" messageType="nmxml3:getNearestAirportCodeRequestMessage"/>
    <variable name="output" messageType="tns:processResMsg"/>
    <variable name="getWindInfoOutput" messageType="nmxml4:ResMsg"/>
    <variable name="getWindOutput" messageType="nmxml5:getWindResponseMessage"/>
    <variable name="getADLCoordinatesOutput" messageType="nmxml1:getCoordinatesResponse"/>
    <variable name="getCoordinatesOutput" messageType="nmxml0:ResMsg"/>
    <variable name="getNearestAirportOutput" messageType="nmxml2:ResMsg"/>
    <variable name="getNearestAirportCodeOutput" messageType="nmxml3:getNearestAirportCodeResponseMessage"/>
  </variables>
  <!--
ORCHESTRATION LOGIC
The following represents the selection and composition patterns transformed into BPEL code.

Receive input from requester by the receive activity 'receiveInput'.
This maps to operation defined in getWindInfoAtPlace.wsdl
```

Figure 2. Excerpt of generated WSBPEL process.

Our prototype has been validated in real scenarios. In prior research we have investigated the notion of reusable service applied to a practical emergency management use case [9]. Also, this prototype has been part of a tool set to jointly carry out service discovery and composition to generate a risk map of potentially hazardous substances [11]. In this paper, we have described the abstract patterns and explored the use of reusable services with selection patterns involved in larger compositions by creating a gazetteer reusable service from two operation-level mismatching web services such as the ArcWeb Place Finder service (http://www.esri.com/software/aws-publicservices/index.html) and the Alexandria Digital Library Gazetteer (http://middleware.alexandria.uscb.edu /client/gaz/adl/index.jsp), which provide the same functionality but have different operation and parameters names. Using the Java-based prototype, such a gazetteer service with selection pattern can be automatically transformed in a process document by means of combinations of WSBPEL constructs, validating then the usefulness of our model-driven approach.

5. Related work

Benatallah et al. [6] propose a model-driven framework for web services life-cycle management, by analyzing and managing web service business protocols. They introduce the notions of compatibility and replaceability to validate whether two service protocols can syntactically interact and how. This work is focused on business protocols or service choreography while our approach takes place on service orchestration. In addition, they do not directly emphasize the service reusability goal as do the proposed abstract patterns and the reusable service concept.

From the component-based software perspective, Yang and Papazoglou [21] treat services as components in order to support basic software development principles such as reuse, specialization, and extension. The main idea is to encapsulate composite-logic information inside a class definition, which represents a web component. This concept has similarities with our reusable service model. However, we explore a wider variety of selection and composition patterns, for example loop or patterns based on split with join combinations, and their relationships to the reusability and flexibility features.

Regarding workflow pattern-based approaches, Jaeger et al. [10] propose an abstract model using workflow patterns that emphasize the aggregation of service properties regarding quality of service dimensions. The workflow pattern analysis is similar although it is applied to the goal of quality of service instead of service reuse. Moreover, we use both selection and composition patterns as part of the model (along with the reusable service) for defining the structural aspects, considering reusable services as building blocks for developing web applications. Medeiros et al. [12] propose an interesting approach to annotate and reuse scientific workflows. The authors present WOODSS, an infrastructure to help scientists to specify and annotate their experiment as well as documenting shared scientific activities. Their main goal is also workflow reuse and they define the concept of digital content components (DCC) as a reuse unit for encapsulating annotated workflows. DCC is similar to our notion of reusable service as it is composed, reused, and serialized into a WSBPEL process description. However DCC units are composed by keeping the original WSBPEL constructs. As discussed throughout this paper, the proposed abstract pattern set minimizes workflow constructs avoiding overlapping and redundant workflow patterns, leading to a straightforward and structured composition process.

Examples addressing the specification of services using model-driven development can be found in [5, 7]. The work of Anzböck & Dustdar [5] is focused on modeling medical web services. The authors look into coordination, transaction, and security aspects for web services instead of service reuse. They propose the definition of WSBPEL process documents from Unified Modeling Language (UML) use case models. Therefore it is similar to our approach, that allows us to generate WSBPEL process descriptions from reusable services expressed as high-level designs. Gannod et al. [7] present a tool set based on model-driven techniques to assist in the creation of semantic web services described in OWL-S –an upper ontology that provides a mechanism for describing service semantics in a standard manner. Users first create descriptions of web services in a UML model; next an interactive tool transforms these model descriptions into OWL-S descriptions. Although outputs are different, since an OWL-S description is generated in [7], both approaches use a model-driven approach demonstrating how the use of model-driven tools may facilitate the creation of complex specifications such as OWL-S and WSBPEL.

6. Conclusions and future work

Integration of distributed software and components has become a recent trend in the web services field. From a user's perspective, adopting a language such as WSBPEL to directly model web service composition may be difficult because of the learning curve and complexity. On the positive side the WSBPEL specification is in widespread use due to the presence of several editors and workflow engines that support it. Because of that, our model-driven approach presented here allows users and service developers to focus on creating a model in terms of proper reusable services that suit their needs rather than on solving syntactical issues of the WSBPEL specification.

The core of this work is based on creating a simple but expressive model for simplifying the design of complex and customized web applications and enabling high-level reuse and flexibility. The model for the proposed model-driven approach relies on the reusable service model, to represent and manipulate service descriptions and to place emphasis on service reusability, and abstract patterns for reusable services orchestration to facilitate the creation of flexible services. Reusable services stored in repositories are subsequently transformed for execution according to the model-driven transformation components. This allows users also keep their own indexed lists of useful reusable services, which are likely to be used for certain clients or providers. Although the approach has been tested for combining geo-services, we believe that it is independent-domain enough to be applicable to other contexts. Indeed, the capability of reusing existing compositions permits users to exploit past experiences to solve similar problems in other domains.

We are planning to continue integrating model-driven mechanisms and web services to add additional features for discovery and composition of web services, such as security and semantics. We are also investigating the use of novel model-driven mechanisms and architectures to minimize the network data transfer for data-intensive web service compositions as in the case of geographic information services.

7. References

[1] W.M.P. van der Aalst, “Don`t go with the flow: Web services composition standards exposed”, *IEEE Intelligent Systems*, 18(1): 72-76, 2003.

[2] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, “Workflow Patterns”, *Distributed and Parallel Databases*, 14(1): 5-51, 2003.

[3] S. Aissi, P. Malu, and K. Srinavasan, “E-Business Process Modeling: The Next Big Step”, *IEEE Computer Magazine*, 35(5), 55-62, 2002.

[4] R. Akkiraju, J. Farrel, J. Miller, M. Nagarajan, M-T Schmidt, A. Sheth, and K. Verma, “Web Services Semantics – WSDL-S”, v. 1.0, W3C Member Submission, November 2005, http://www.w3.org/ Submission/WSDL-S/ (accessed December 2006).

[5] R. Anzböck and S. Dustdar, “Modeling and implementing medical Web services, Data & Knowledge Engineering, 55(2): 203-236, 2005.

[6] B. Benatallah, H. Reza, M. Nezhad, F. Casati, F. Toumani, and J. Ponge, “Service Mosaic: A Model-Driven Framework for Web Services Life-Cycle Management”, *IEEE Internet Computing*, 10(4): 55-63, 2006.

[7] G. C. Gannod, J. T. E. Timm, and R. J. Brodie, “Facilitating the Specification of Semantic Web Services Using Model-Driven Development”, *Intl. Journal of Web Services Research*, 3(3): 61-81, 2006.

[8] A. Gómez-Perez, R. Gónzalez.Cabero, and M. Lama, “ODE SWS: A Framework for Designing and Composing Semantic Web Services”, *IEEE Intelligent Systems*, 19(4): 24-31, 2004.

[9] C. Granell, M. Gould, R. Gronmo, D. Skogan, “Improving Reuse of Web Service Compositions”, *Proc. E-Commerce and Web Technologies – EC-Web 2005*. Springer LNCS 3590, 358-368, 2005.

[10] M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl, “QoS Aggregation for Web Service Composition using Workflow Patterns”, *Proc. IEEE Enterprise Distributed Object Computing Conference – EDOC 2004*. IEEE CS Press, 149-159, 2004.

[11] R. Lemmens, A. Wytzisk, R. de By, C. Granell, M. Gould, and P. van Oosterom, “Integrating Semantics and Syntactic Descriptions to Chain Geographic Services”, *IEEE Internet Computing*, 10(5): 42-52, 2006.

[12] C. B. Medeiros, J. Perez-Alzacar, L. Digiampietri, G. Z. Pastorello, A. Santanche, R. S. Torres, E. Madeira, and E. Bacarin, “WOODS and the Web: annotating and reusing scientific workflows”, *SIGMOD Records*, 34(3): 18-23, 2005.

[13] M. Milanovic and M. Malek, “Current Solutions for Web Service Compositions”, *IEEE Internet Computing*, 8(6): 51-59, 2004.

[14] R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, 1999.

[15] C. Pautasso, *A Flexible System for Visual Service Composition*, PhD dissertation number 15608, Department of Computer Science, ETH Zurich, Zurich (Switzerland), 2004.

[16] K. Sycara, P. Paolucci, A. Ankolekar, and N. Srinivasan, “Automated Discovery, Interaction and Composition of Semantic Web Services”, *Journal of Web Semantics*, 1(1): 27-46, 2003.

[17] C. Szyperski, Component Software. Beyond Object-Oriented Programming, Addisson-Wesley, 1998.

[18] M. Vasko and S. Dustdar, “An analysis of Web Services Workflow Pattern in Collaxa”, *Proc. Web Services: European Conference – ECOWS 2004*. Springer LNCS 3250, 1-14, 2004.

[19] A. Voisard and H Schweppe, “Abstraction and decomposition in interoperable GIS”, *Intl. Journal of Geographical Information Science*, 12(4): 315-333, 1998.

[20] P. Wohed, W. M. P .van der Aalst, Dumas, A. H. M. ter Hofstede, “Analysis of web services composition languages: The case of BPEL4WS”, *Proc. Conceptual Modeling – ER 2003*. Springer LNCS 2813, 200-215, 2003.

[21] J Yang, M. P. Papazoglou, “Web Components: A Substrate for Web Service Reuse and Composition”, *Proc. of CAiSE 2002*. Springer LNCS 2348, 21-36, 2002.