BUSINESS MODEL FOR COMPOSING WEB SERVICES

Ahmed Khoumsi
University of Sherbrooke, Department of Electrical and Computer Engineering, Sherbrooke, Canada
Ahmed.Khoumsi@USherbrooke.ca
(Ahmed Khoumsi was on sabbatical leave at King Saud University, Dept of Computer Science, when the study was done.)

Zohair Chentouf, Ahmed Ghoneim, Samir Elmougy
King Saud University, Department of Computer Science, Riyadh, Kingdom of Saudi Arabia
(zchentouf, Ghoneim, mougy)@ksu.edu.sa

ABSTRACT

Web service composition denotes the design of new Web services by composing existing ones. An important challenge is to design a business model which makes Web service composition practical, by automating as much as possible the various composition steps. However, such a composition may generate undesirable behaviours, which are referred to as feature interactions. In this paper, we develop a new business model for composing Web services and detecting their feature interactions. The Web service composition and feature interaction detection process is described literally and as an algorithm. The use of our business model is illustrated in an example.

KEY WORDS

Business model, Web service composition.

1. Introduction

The increasing number of Web Services (WSs) and the advantages of software reusability have promoted compositions of existing WSs to synthesize new WSs. These compositions may generate unexpected behaviours which are commonly called Feature Interactions (FIs). The latter have been intensively studied in telecommunications, e.g. the series of feature interaction workshops (FIW) [1]. Solutions to FIs are often operated in two phases: FIs are detected and then resolved if they are undesirable. FI detection and resolution are qualified as off-line (resp. online) when they are performed before (resp. when) running the services. Our contribution is the development of a business model that offers an efficient and practical environment of WS composition and FI detection. The proposed system is designed with the idea to evolve in future with new functionalities, in addition to WS composition and FI detection. Examples of functionalities: assessing and guaranteeing a Quality of Service (QoS) and performances. We will use the following terminology:

- A behavior of a WS: the set of sequences of actions that the WS can execute.
- A description of a WS: a formal description of the behaviour of the WS.
- Designing a WS: developing a description of a WS (from scratch or by combining descriptions of 2 WSs).

- Constructing a WS: developing an implementation of the WS (synthesized from a WS description, or by combining the implementations of two WSs).
- Creating a WS: designing and constructing a WS.
- Composing two WS descriptions: designing a WS by combining the two descriptions
- Composing two WS implementations: constructing a WS by combining the two implementations.
- Composing two WSs S1 and S2: composing the descriptions of S1 and S2 and obtaining a description S; then detecting FIs, i.e., checking if S is FI prone; then composing the implementations of S1 and S2, if no FI is detected.

The rest of the paper is structured as follows: Section 2 presents related work on composing WSs and detecting their FIs. Section 3 introduces our approach of FI detection. Section 4 details the notions of authorization, ownership and deployment which are used by our business model. The latter is presented in Section 5 by its components and the main ideas on which it is built, and then Section 6 contains a procedural description of our business model. Section 7 illustrates by an example how our business model can be used to create some WSs by composition. Section 8 concludes.

2. Related Work on Composing WSs and Detecting Their FIs

[2, 3] raises interest of researchers to WS composition and FI detection. Moreover, [3] presents a series of examples of WS composition and FI detection which can be used as a benchmark to assess FI detection methods. Another contribution in raising interest is in [4] which shows that FIs in telecommunications are different from FIs in WSs.

Contributions on on-line composition can be found for example in [5-7]. [5] proposes an on-line FI detection method based on Situation Calculus. FI resolution is discussed but without proposing an FI resolution method. In [6], an on-the-fly WS composition approach is outlined. In [7], it is argued that WS composition can be done at run-time, and challenges and opportunities in FI detection and resolution are identified.

Contributions on off-line composition can be found for example in [8-12]. [12] presents an off-line FI
Some work on software-tooling and user-interfacing for WS composition can be found for example in [14, 15]. [14] proposes an environment using Mashup for composition of WSs, and [15] presents an integrated development environment for end-to-end WS composition. FI detection is not considered in [14, 15].

[16, 17] are particularly interesting since they consider both theoretical, software-tooling and user-interfacing aspects. [16, 17] make a rigorous study on WS composition and FI detection. It is proposed to use CRESS formalism which can be automatically translated into BPEL and LOTOS.

A business model for WSs has been proposed in [18]. WS-providers publish their WSs in a WS-registry. A WS-requester seeking for a WS finds the location of the WS in the WS-registry, and then it has access to the WS by communicating with the corresponding WS-provider. [19, 20] propose an extension of such a business model to support WS composition. Our main contributions compared to [19, 20] is that our business model is evolutionary and contains a module of FI detection.

3. Our Approach of FI Detection

Consider that a WS description S has been designed from scratch or by composing two WS descriptions S₁ and S₂. Then, a WS must be constructed (synthesized from S, if S has been designed from scratch; or by composing implementations of S₁ and S₂, if S has been designed by composition). Before such a construction step, it is necessary to check if S contains FIs. We model FIs as non-satisfied properties; we consider 2 types of properties:

Required-property: it reflects a requirement, e.g. the WS must not contain deadlocks. For example, with finite state automata (FSA), this requirement can be specified by using a variable NbOutTrans (number of outgoing transitions), and requiring that the property NbOutTrans > 0 holds in all states.

Semantic-property: it reflects how some variables of WSs are interpreted from the user or designer point of view. For example, assume that two WS descriptions S₁ and S₂ to be composed use respectively two variables: x₁ and x₂. If the user understands that x₁ and x₂ have the same semantics (e.g. the price of an ordered product), then when composing S₁ and S₂, the user or designer can define the property x₁ = x₂.

Some required-properties may be generic and automatically associated to each WS description (e.g., absence of deadlock), while other properties (some of the required-properties, and all semantic-properties) may be defined when designing a WS description. Every property is formalized by an invariant, i.e. an expression that must always be respected during the execution of the WS. For example, with FSA, invariants must be respected in all states of an FSA modeling a WS. Invariants can also be used to specify properties which must be satisfied in some (but not all) states. For example, if a property P must be satisfied uniquely in a given state q, we use the following invariant in all states: if currentState = q, then P. Required-invariant (resp. semantic-invariant) denotes an invariant modeling a required-property (resp. semantic-property).

Hence, after the design of a WS description S, and the automatic selection of generic required-invariants, the designer must prepare for FI detection by specifying required-invariants (reflecting his specific requirements) and some semantic-invariants (reflecting his understanding). Here are two extreme cases for specifying the semantic-invariants:

Maximal invariant specification: the designer specifies, as much as possible, the semantic-invariants reflecting all the understanding he has of the WS. More precisely, he considers all variables used in WS description and expresses formally all relations between variables. The drawback of this approach is that it may generate a great number of invariants, and its advantage is that it increases chances to detect unsuspected FIs.

Directed invariant specification: the designer specifies only semantic-invariants that are not respected by some suspected FIs. For example, consider the composition of two WSs S₁ and S₂ using respectively variables x₁ and x₂ representing the price of the same product. If the designer suspects that S₁ and S₂ may unduly associate different prices to the same ordered item (e.g., x₁ includes taxes, and x₂ excludes taxes), then he can define the semantic-invariant x₁ = x₂. The drawback is the reduction of chances to detect unsuspected FIs, and the advantage is the relatively small number of generated invariants.

We opt for FI detection based on model-checking, where a reachability tree of the description of the composed WSs is synthesized, and the specified invariants are evaluated in each node of the reachability tree. A non-respect of an invariant corresponds to the occurrence of an FI.

4. Authorization, Ownership & Deployment

We call requester the entity that wants to compose two WSs S₁ and S₂, and owner of a WS the entity that has all rights on the WS. The owner is not necessarily the designer or the creator. In this section, we show how we treat the following aspects: 1) authorizations needed by the requester to have access to the descriptions and implementations of S₁ and S₂; 2) who will be the owner of the composed WS denoted Comp(S₁,S₂); and 3) where to store the composed Comp(S₁,S₂).
4.1 Authorization for accessing a WS description

Since for composing $S_1$ and $S_2$, the requester needs an access to their descriptions, we have to deal with the issue of access authorization to WS descriptions. Our solution is to require that the owner of each WS publishes an extended WSDL specification (denoted WSDL-ext) which contains (in addition to the usual content of WSDL) a parameter $Perm_{desc}$ specifying the necessary authorizations for accessing the WS description. Let us consider an example of the possible values of $Perm_{desc}$:

- **FREE**: a WS description is accessible without any restriction;
- **NO**: no description of the WS is accessible;
- **NON-PROFIT**: a WS description is available only to develop non-profit-making applications;
- **PAY(x)**: a WS description is available after payment of the amount $x$.

We may have conjunctions of conditions, e.g., “NON-PROFIT and PAY(200)”. When $Perm_{desc} \neq NO$, it must be accompanied by an URL $URL_{owner}$, which is used by the requester to access to the owner through an adequate communication interface. A description of the WS can be downloaded through this interface after payment (if necessary) and commitment to respect the conditions (if any). When the descriptions of $S_1$ and $S_2$ are obtained, they can be edited and composed.

4.2 Authorization for accessing a WS implementation

A WS composition may necessitate modifying the description of at least one of $S_1$ and $S_2$, and hence also the source code of its implementation. It is therefore necessary to consider the issue of access authorization to (source code of) WS implementations. Let us call requester the entity that wants to compose WSs. To understand implementation access authorization, let us compare it with description access authorization:

- Descriptions of $S_1$ and $S_2$ must be obtained by a requester who wants to compose them (possibly with some modifications, in case of choreography) and then to analyze them for FI detection.
- Source code of an implementation (of $S_1$ or $S_2$) is necessary only when the implementation must be modified. Besides, the modification may be done by the owner or by the requester. In the first case, the requester must specify the modifications to the owner, for example by sending him a description of the WS including the desired modifications. In the second case (modification by the requester), the requester must receive the source code from the owner, to proceed with the modifications. For simplicity, we consider uniquely the second case, i.e., the owner sends the source code to the requester who will produce a new implementation.

Hence, the descriptions of $S_1$ and $S_2$ must be obtained by the requester in all situations, whereas the source code of an implementation must be obtained by the requester only if it must be modified. The implementation access authorization is treated in the same way as the description access authorization, with the publication in WSDL-ext of a parameter $Perm_{imp}$. For example: $Perm_{imp} =$

- **FREE**: the source code is accessible without restriction;
- **NO**: no source code of the WS is accessible;
- **NON-PROFIT**: the source code is available only to develop non-profit-making applications;
- **PAY(x)**: the source code is available after payment of the amount $x$.

We may have conjunction of conditions, e.g., NON-PROFIT and PAY(200). When $Perm_{imp} \neq NO$, $URL_{owner}$ is used by the requester to access to the owner through an adequate communication interface. The source code of the implementation of the WS can be downloaded through this interface after payment (if necessary) and commitment to respect the conditions (if any). Note that the description and the source code of $S_1$ or $S_2$ are downloaded through the same $URL_{owner}$.

4.3 Requirements on access authorization

Composing two WSs requires accesses to their descriptions, hence $Perm_{desc}$ of each WS must be $\neq NO$. Access to a source code of a WS $S$ is required when the composition necessitates modifying the implementation of $S$. In the latter case, we must have $Perm_{imp} \neq NO$.

4.4 Ownership

After a WS has been composed, it is necessary to determine who is its owner. Let us give a simple example of how to handle the pairs of parameters $(Perm_{desc}^{S_1}, Perm_{desc}^{S_2})$ and $(Perm_{imp}^{S_1}, Perm_{imp}^{S_2})$ to analyze the ownership of the composed WS.

\[
\begin{align*}
&\text{if } (Perm_{desc}^{S_1}, Perm_{desc}^{S_2}) = (\text{FREE,FREE}) \text{ and } (Perm_{imp}^{S_1}, Perm_{imp}^{S_2}) = (\text{FREE,FREE}) : \\
&\quad \text{The requester is the owner of Comp(S_1,S_2) and can} \\
&\quad \text{publish it and specify his own access authorization.} \\
&\text{else if } (Perm_{desc}^{S_1}, Perm_{desc}^{S_2}) = (\text{FREE, FREE}) \text{ and } (Perm_{imp}^{S_1}, Perm_{imp}^{S_2}) \neq (\text{FREE, FREE}) : \\
&\quad \text{if } (\forall S = S_1, S_2: Perm_{imp}^S \neq \text{FREE}) : \\
&\quad \quad \text{source code of S needs not be modified} \\
&\quad \quad \text{The requester is the owner of Comp(S_1,S_2) and can} \\
&\quad \quad \text{publish it and specify his own access authorization} \\
&\quad \text{else: The requester has not full rights on the composed} \\
&\quad \text{WS. We may conceive that the requester can use} \\
&\quad \text{the composed WS but he cannot publish it.}
\end{align*}
\]

4.5 Necessity to specify who deploys the composed WS

An important issue is to specify who deploys the newly composed WS and in which server it is deployed. We will see situations where a WS is not deployed by its owner (Point 9 in Sect. 5, and the composition process in Sects. 6.1-6.2).
5. New Business Model for WS Composition and FI Detection

The two expressions “behavior of the business model” and “service provided by the business model” will be used as synonyms. We have developed a business model whose essential components and their interactions are illustrated in Figure 1. In addition to the components of the standard model (WS-registry, WS-requester, WS-providers) [18], five new components are proposed: WS-composition-registry (more briefly: WS-comp-reg), WS-registry-controller (more briefly: WS-reg-cont), WS-composer, WS-FI-detector, and WS-owner. Figure 1 represents also some unspecified modules (in dotted lines) to stress the fact that the proposed business model can evolve in future with new functionalities in addition to FI detection. We will mention, in Point 10 below, some examples of future functionalities. In the standard model, the term WS-owner is not mentioned explicitly because it is considered equivalent to WS-provider, i.e. the owner of a WS is the entity that provides that WS. With our model, there may exist WSs without owner (see Point 9 below). In such a case, WS-provider is not WS-owner, because the latter does not exist while the former exists. But when WS-owner exists, it is assumed to be WS-provider. In Figure 1, the numbers labelling the interactions identify the communications during the execution of our business model. Those numbers will be clarified in Section 6.2. The interactions in the standard model are indicated by the usual words find, bind and publish. Our proposed model comprises the standard model, in the sense that it behaves exactly like the standard model in the following two situations: 1) WS-requester asks WS-registry to find some WS; 2) WS-owner creates a WS and publishes it in WS-registry.

Before presenting in detail the behaviour of the business model in Section 6, let us present the essential ideas which have directed its design.

1. Three components are related to the publication of WSs: WS-registry, WS-comp-reg and WS-reg-cont. Consider that a WS-owner has created a WS S:
   - According to the standard model, WS-registry is a recipient where are published WSs that are publicly available for execution. In our framework, S is published by WS-owner in WS-registry if WS-owner authorizes its public execution.
   - Our new component WS-comp-reg is a recipient where are published WSs that are publicly available to be used in the design of new WSs. In our framework, S is published by WS-owner in WS-comp-reg if WS-owner authorizes that other providers create new WSs by using S, for example by modifying S and/or composing S with other existing WSs. The publication of S in WS-comp-reg may specify some restrictions on access.
   - The same WS is published in both WS-registry and WS-comp-reg if its WS-owner authorizes both its public execution and its composition with WSs. Instead of communicating with WS-registry and WS-comp-reg, WS-owner has also the choice to communicate uniquely with our new component WS-reg-cont, and the latter is responsible of publishing S in WS-registry and WS-comp-reg.
   - When a WS S=Comp(S₁,S₂) is published in WS-registry and/or WS-comp-reg, the names of S₁ and S₂ are indicated in the publication(s).

2. When WS-requester needs a WS, there are 3 situations:
   (a) If WS-requester needs to execute a given WS S, he behaves as in the standard model by asking WS-registry to search S.
   (b) If WS-requester needs to create a new WS Comp(S₁,S₂) from given S₁ and S₂, he can ask WS-comp-reg to search S₁ and S₂. The composition process can start when both S₁ and S₂ are found.
   (c) If WS-requester needs to execute or create a WS Comp(S₁,S₂), he communicates with WS-reg-cont. The idea is to execute Comp(S₁,S₂) if it exists, and to create Comp(S₁,S₂) if it does not exist. When WS-reg-cont receives the request from WS-requester, WS-reg-cont asks WS-registry to search Comp(S₁,S₂). If the latter is found, the process continues as in the standard model. If Comp(S₁,S₂) is not found, WS-reg-cont asks WS-comp-reg to search S₁ and S₂ before to start the composition process.

3. The main actors in the composition process are WS-requester and WS-composer. WS-requester is realistically assumed to be a human, because he is the intelligent part (designer and decision-maker). WS-composer is the automatic part in the sense that it has the task to execute, as much as possible, all the edition, composition and FI detection operations that can be automated, with the objective to make the composition as easy and intuitive as possible for WS-requester.

4. To create Comp(S₁,S₂), WS-requester needs an access to the descriptions of S₁ and S₂, which are necessary for designing the description of Comp(S₁,S₂) and...
executing FI detection (i.e., checking if Comp(S₁S₂) is FI prone).
5. To create Comp(S₁S₂), WS-requester needs an access to the source code of any S₁ or S₂ that is modified in the composition. This is because such a source code is necessary for constructing the implementation of Comp(S₁S₂).
6. If WS-owner of S₁ or S₂ has specified restrictions on access, then during the creation of Comp(S₁S₂), it is WS-composer that pays WS-owners and commits to respect their restrictions. This is because WS-composer is assumed to be an entity of a trusted organization, so that all WS-owners trust its commitment. Hence, WS-composer has the responsibility to respect the restrictions of WS-owners. Consequently, WS-requester has not to commit but he has to pay WS-composer
7. During the creation of Comp(S₁S₂), the descriptions and implementations of S₁ and S₂ are hosted by WS-composer instead of WS-requester. A first justification is that WS-composer should execute, as much as possible, all the edition, composition and FI detection operations that can be automated (see Point 3). A second justification (which holds when there are restrictions on access) is that WS-composer is responsible of respecting the restrictions (see Point 6). Consequently, the accesses of WS-requester to the descriptions and implementations are remote.
8. If both WS-owners of S₁ and S₂ have specified no restriction on access, then after the creation of Comp(S₁S₂), WS-requester is the WS-owner of Comp(S₁S₂) and hosts the final result of composition, i.e., the implementation of Comp(S₁S₂) and that of any modified source code of S₁ or S₂. Being WS-owner of Comp(S₁S₂), WS-requester has full rights to deploy (locally) and publish Comp(S₁S₂) (in WS-registries and/or WS-comp-reg). When Comp(S₁S₂) is published in WS-comp-reg, then WS-requester (who is WS-owner of Comp(S₁S₂)) may specify restrictions on access (see Point 1).
9. If WS-owner of S₁ or S₂ specifies restrictions (other than payment) on using his WS Comp(S₁S₂), Comp(S₁S₂) has no WS-owner and cannot be published, because only WS-owner of a WS can publish that WS. The final result of composition is deployed locally by WS-composer, and WS-requester can execute it remotely.
10. The proposed business model is designed with an evolutionary spirit. That is, it may evolve in future with new modules that provide services to WS-composer, in addition to FI-detector. Examples of services: estimating the Quality of Service (QoS) and performances of Comp(S₁S₂).

Remarks: Points 4 and 7 refer to descriptions specifying WS behaviors. As shown in [16, 17], CRESS (mentioned in Section 2) is a good example of formalism for describing WSs. In (a) and (b) of Point 2, services S₁ and S₂ are selected by WS-requester which is assumed human, i.e. a human decides to construct a new service from services he/she has in mind. We do not consider here the problem of automatic selection of services that may compose a new service.

6. Detailed Description of the Business Model

6.1 Literal presentation of the service provided by the business model

The situations where the composition is impossible (and hence abandoned) are emphasized.

When a WS-owner creates a WS S from scratch or by composition, WS-owner (who is also WS-provider of S) deploys S and can publish it in WS-registry and/or WS-comp-reg. When WS-owner wants to publish S in both WS-registry and WS-comp-reg, he can decide to ask WS-reg-cont to proceed with the two publications.

When a WS-requester needs a WS Comp(S₁S₂), he can inform WS-reg-cont of that. Then, WS-reg-cont will first ask WS-registy to search Comp(S₁S₂). If WS-registy finds Comp(S₁S₂), we proceed as in the standard business model. If Comp(S₁S₂) is not found by WS-registy, then WS-reg-cont asks WS-comp-reg to search S₁ and S₂. If at least one of S₁ and S₂ is not found by WS-comp-reg, the composition is impossible. If Comp(S₁S₂) is not found (by WS-registy) but both S₁ and S₂ are found (by WS-comp-reg), the composition process is initiated (see next paragraph). WS-comp-reg can also “short-circuit” WS-reg-cont by communicating directly with WS-registy or WS-comp-reg. In the 1st option, WS-requester asks WS-registy to search Comp(S₁S₂), and the system proceeds as in the standard model. In the 2nd option, WS-requester asks WS-comp-reg to search S₁ and S₂, and the system proceeds as in the case where WS-reg-cont communicates with WS-comp-reg.

Initiation of the composition process: WS-comp-reg launches some protocol so that WS-requester and WS-composer are connected through a Web interface. Then, WS-requester indicates to WS-composer the two WSs S₁ and S₂. WS-composer downloads from WS-comp-reg the WSDL-ext descriptions of S₁ and S₂, which contain their (Perm-desc, Perm-imp, URL-owner) (see Sects. 4.1 and 4.2). If some Perm-desc specifies that the description of S₁ or S₂ is unavailable, the composition is impossible. Otherwise, if some Perm-owner specifies that the source code of S₁ or S₂ is unavailable while the composition requires modifying such unavailable code, the composition is impossible. Otherwise (all descriptions and needed source codes are available), WS-composer informs WS-requester of the total price and the authorization required by the WS-owners of S₁ and S₂, if any. If WS-requester does not accept the price or some restriction, the composition is impossible (under acceptable conditions). Otherwise, the composition process can really start as follows.

The composition process: WS-requester pays WS-composer, and then the latter pays and commits with WS-owners of S₁ and S₂ to respect their restrictions. Then, the WS-owners permit WS-composer to download the
descriptions and any source code which needs modifications in S₁ and S₂. Then, WS-requester has access to WS-composer through a Web interface to edit the description of Comp(S₁,S₂) and that of any of S₁ or S₂ which needs modifications. Through this Web interface, WS-requester can also enable/disable generic invariants and edit some required or semantic invariants. Then, WS-requester launches the FI detection procedure. If some FI is detected, the composition result is FI prone and hence the composition is impossible (under acceptable conditions). Otherwise, WS-requester has access to WS-composer through the Web interface to modify the source code and generate the implementation of any of S₁ or S₂ which needs modifications. WS-requester generates also the implementation of Comp(S₁,S₂). Before deployment, WS-composer has to determine whether WS-requester is the owner of Comp(S₁,S₂). We consider here that WS-requester is the owner if no restriction has been specified by the WS-owners of S₁ and S₂. In the latter case, WS-requester can download from WS-composer the implementation of Comp(S₁,S₂) and the implementation of any of S₁ or S₂ which has been modified. Then, WS-requester deploys locally the downloaded implementations and publishes Comp(S₁,S₂) in WS-registry and/or WS-comp-reg. If some restriction (other than payment) has been specified by the WS-owners of S₁ and S₂, WS-requester is not WS-owner of Comp(S₁,S₂). In such a case, Comp(S₁,S₂) remains in WS-composer, because WS-composer has committed to respect the restrictions, and hence is responsible of any non-respect of restrictions. In this case, WS-composer deploys locally Comp(S₁,S₂) and any modified implementation of S₁ or S₂, and indicates to WS-requester an URL for executing Comp(S₁,S₂).

6.2 Algorithmic presentation of the behaviour of the business model

Let us now present the behavior of the business model algorithmically in several blocks. Interactions (or groups of interactions) between two components are indicated by numbers from 1 to 35. The interactions of the standard model are indicated by the standard operations: find, bind and publish. These operations and the interaction flows are also shown in Figure 1. Note that the numbers that represent the interaction flows do not indicate the order of interactions. For example, 12 does not follow 11, we have either 11 or 12.

Our business model consists of several components (see Figure 1). Our system is designed such that at any time before the completion of the composition process, two components are running while the other components are idle (i.e., in their initial states). When for some reason the composition is detected as impossible, each of the two currently running components terminates its execution and returns to its initial state. Such termination is emphasized for the sake of clarity. Note that every communication between WS-composer and WS-requester is made through a Web interface.

> When some WS-owner has created a WS S (from scratch or by composition):

<table>
<thead>
<tr>
<th>WS-owner deploys S and then has four options for publishing S:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- WS-owner can ask WS-registry to publish S (publish)</td>
</tr>
<tr>
<td>- WS-owner can ask WS-comp-reg to publish S (1)</td>
</tr>
<tr>
<td>- WS-owner can ask WS-registry (publish) and WS-comp-reg to publish S (1)</td>
</tr>
<tr>
<td>- WS-owner can ask WS-reg-cont to proceed with one of the 3 above options (2). In such a case, WS-reg-cont asks WS-registry (3) and/or WS-comp-reg (4) to publish S</td>
</tr>
</tbody>
</table>

> When WS-requester wants to use a WS Comp(S₁,S₂) (without creating it):

| - WS-requester asks WS-registry to search Comp(S₁,S₂) (find). |
| - Then, proceed as in the standard business model of [18] (bind the service if it is found). |

> When WS-requester wants to create a WS Comp(S₁,S₂) by composition:

<table>
<thead>
<tr>
<th>WS-requester asks WS-comp-reg to search S₁ and S₂ (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if WS-comp-reg does not find S₁ or S₂:</td>
</tr>
<tr>
<td>- WS-comp-reg informs WS-requester that S₁ or S₂ are not found (6)</td>
</tr>
<tr>
<td>- WS-requester and WS-comp-reg terminate.</td>
</tr>
<tr>
<td>// because S₁ or S₂ is not found.</td>
</tr>
<tr>
<td>else: - WS-comp-reg informs WS-requester that S₁ and S₂ are found (7)</td>
</tr>
<tr>
<td>- Initiate the composition process.</td>
</tr>
</tbody>
</table>

> When WS-requester wants to use a WS Comp(S₁,S₂) if it exists, or create it by composition if it does not exist:

<table>
<thead>
<tr>
<th>WS-requester informs WS-reg-cont that he needs a WS Comp(S₁,S₂) (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS-reg-cont asks WS-registry to search Comp(S₁,S₂) (9)</td>
</tr>
<tr>
<td>if WS-registry finds Comp(S₁,S₂)</td>
</tr>
<tr>
<td>- WS-registry sends to WS-reg-cont the WSDL description of Comp(S₁,S₂) (10)</td>
</tr>
<tr>
<td>- WS-reg-cont forwards the WSDL description of Comp(S₁,S₂) to WS-requester (11)</td>
</tr>
<tr>
<td>- Then, proceed as in the standard business model of [18] (bind)</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>- WS-registry informs WS-reg-cont that Comp(S₁,S₂) is not found (12)</td>
</tr>
<tr>
<td>- WS-reg-cont asks WS-comp-reg to search S₁ and S₂ (13)</td>
</tr>
<tr>
<td>if WS-comp-reg does not find S₁ or S₂:</td>
</tr>
<tr>
<td>- WS-comp-reg informs WS-reg-cont that S₁ or S₂ is not found (14)</td>
</tr>
<tr>
<td>- WS-reg-cont informs WS-requester that [Comp(S₁,S₂) and S₁ or S₂] not found (15)</td>
</tr>
<tr>
<td>- WS-requester and WS-reg-cont terminate</td>
</tr>
<tr>
<td>//because Comp(S₁,S₂)and S₁ or S₂ are not found</td>
</tr>
</tbody>
</table>
WS-comp-reg informs WS-requester that S_1 and S_2 are found (16)
- WS-reg-cont informs WS-requester that S_1 and S_2 are found but not Comp(S_1,S_2) (17)
- Initiate the composition process.

Initiation of the composition process

WS-comp-reg initiates some communication protocol so that WS-requester will be connected to WS-composer through a Web interface. Here is an example of protocol:

WS-comp-reg sends to WS-requester an URL consisting of: (18)
- an address where to join WS-composer through the Web interface, and
- a message (understood by WS-composer) meaning that 2 WSs must be composed.

WS-requester uses that URL to connect to WS-composer.

WS-requester indicates to WS-composer the two WSs S_1 and S_2 (19).

WS-composer downloads from WS-comp-reg the WSDL-ext of S_1 and S_2, which contains (Perm_desc, Perm_imp, URL-owner). (20)

If one of downloaded Perm_desc = NO
  // i.e., one of the descriptions is not accessible
- WS-composer informs WS-requester that the description of S_1 or S_2 is unavailable (21)
- WS-requester and WS-composer terminate.
  // because S_1 and S_2 description unavailable

// We reach this level when the descriptions of S_1 and S_2 are accessible.

If one of downloaded Perm_imp = NO
  // i.e., one of the source codes is not accessible
- WS-composer informs WS-requester that the source code of S_1 or S_2 is unavailable (22)
  if WS-requester realizes that he needs to modify WS whose source code is unavailable
    - WS-requester asks WS-composer not to continue
    // because needed source code unavailable
  // We reach this level when all descriptions and needed source codes are available.

WS-composer initiates some protocol of payment and commitment to respect conditions (if any). Example of protocol:

WS-composer informs WS-requester about the total payment and the authorization (24)

If WS-requester does not accept the price or some restriction:
- WS-requester informs WS-composer about his refusal
  // because price or restriction unacceptable
- WS-requester and WS-composer terminate

Composition process

// We reach this level when the price and all restrictions are accepted by WS-requester
WS-requester pays WS-composer (26)
for S = S_1, S_2: (27)

WS-composer uses URL_owner to connect to WS-owner of S
- WS-composer interacts with WS-owner of S as follows:
  // remainder of this for-loop
- WS-composer pays WS-owner of S and commits to execute the restrictions, if any.
- WS-composer downloads from WS-owner the description of S
if S needs modification
  - WS-composer downloads from WS-owner the source code of S

// We reach this level when WS-composer has obtained the descriptions and the needed source codes of S_1 and S_2.

WS-requester interacts with WS-composer by proceeding as follows: (28)
for each S = S_1, S_2:
  if S needs modification:
    WS-requester modifies the description of S.
    WS-requester edits a description of the desired
    Comp(S_1,S_2)
    WS-requester can enable/disable some generic required-invariants.
    WS-requester can edit required-invariants and semantic-invariants.

WS-composer launches the FI detection and transfers the invariants to WS-FI-detector. (29)

WS-FI-detector informs WS-composer of the FI detection result. (30)

if some FI has been detected:
  - WS-composer informs WS-requester that an FI is detected. (31)
  - WS-requester and WS-composer terminate.
    // because FI detected

// We reach this level when no FI is detected by the // WS-composer.

WS-composer informs WS-requester that no FI is detected. (32)

WS-requester has access to a development environment
of WS-composer and proceeds as follows: (33)
for each S = S_1, S_2:
  if S needs modification:
    - WS-requester modifies the source code of S and generates its implementation
    - WS-requester generates an implementation of Comp(S_1,S_2)

// We reach this level when Comp(S_1,S_2) is implemented.

WS-composer has access to some ownership and deployment management procedure to determine the WS-owner of Comp(S_1,S_2) and to deploy Comp(S_1,S_2).

Here is an example of procedure: // until the end of the block...
7. Example

In this section, we illustrate how our business model can be used to compose some WSs. We consider the example of WSs given in [17]. Let us qualify as basic any WS designed from scratch, and as complex any WS designed by composing other WSs. For the sake of simplicity, we have omitted some details (catching faults, invoking methods, …).

7.1 Basic and complex WSs

The example consists of four basic WSs (Dealer1, Dealer2, Approver, Assessor) and three complex WSs (Lender, Supplier, Broker).

- Dealer1 and Dealer2 provide services of two car dealers. Each Dealer can be invoked by a customer who specifies his need (e.g., type of car, maximum price, …), and then Dealer returns a quote (price, delivery delay, …) to the customer. When Dealer cannot satisfy the need, he answers with an infinite quote price.
- Approver and Assessor evaluate loans for customers. Approver is invoked to evaluate a loan proposal, and then decides to refuse or approve it. Assessor is invoked to evaluate the risk of a loan proposal, and returns such a risk.
- Lender = Comp(Approver, Assessor), i.e., Lender is designed by composing Approver and Assessor. It is invoked to approve or assess a loan of a given amount.
  - If amount ≥ 10000, Lender invokes Approver. If Approver approves the loan, an acceptance response is returned with a proposition of loan rate. Otherwise, a refusal response is returned.
  - If amount < 10000, Lender invokes Assessor that evaluates and returns the risk of the loan. If risk is low, an acceptance response is returned with a proposition of a loan rate. Otherwise (i.e., risk is high), Lender invokes Approver.
- Supplier = Comp(Dealer1, Dealer2), i.e., Supplier is designed by composing Dealer1 and Dealer2. It is invoked by a customer who specifies his need (e.g., type of car, maximum price, …). The order is forwarded to Dealer1 and Dealer2. Each dealer returns a quote (price, delivery delay, …). Supplier selects the dealer with the best quote (lowest price, earliest delivery date if prices are equal) and sends him a definite offer which is also returned to the customer. Supplier can also be invoked to cancel an order before it is completed.
- Broker = Comp(Lender, Supplier), i.e., Broker is built from Lender and Supplier. It is invoked by a customer who specifies his need (e.g., type of car, maximum price, …). The need is forwarded by invoking Supplier. We have the following two situations:
  - If Supplier answers with an offer (because at least one of the dealers has given a finite quote price), Broker invokes Lender who returns a refusal response or an acceptance with a loan rate.
  - If Supplier answers with an infinite quote price (because both dealers have given an infinite quote price), Broker replies with a refusal.

7.2 Publishing the WSs

After the creation of each of the basic WSs (Dealer1, Dealer2, Approver, Assessor), its WS-owner has several options for publishing it: in WS-comp-reg (1) and possibly also in WS-registry (publish); or ask WS-reg-cont to publish it in WS-comp-reg (2, 4) and possibly also in WS-registry (3). The most important thing is that each of those 4 WSs must be published in WS-comp-reg. This is because Dealer1 and Dealer2 are necessary to create Supplier by composition, and Approver and Assessor are necessary to create Lender by composition.

In the same way, after the creation of each of Lender and Supplier, his WS-owner has to publish it (or ask WS-reg-cont to do it) in WS-comp-reg and possibly also in WS-registry (publish, 1, 2, 3, 4). This is because Lender and Supplier are necessary to create Broker by composition. And after the creation of Broker, it is
sufficient to publish it in WS-registry (publish) or ask WS-reg-cont to publish it in WS-registry (2, 3).

7.3 Creating the complex WSs

With our model, we have emphasized the creation by composition. This is partly because service creation from scratch is now well automated, and partly because many WSs already exist [16]. Hence, we do not discuss the creation of the basic WSs (Dealer1, Dealer2, Approver, Assessor).

Each of the complex WSs (Lender, Supplier, Broker) can be created with the same scenario. Let us present the case of Lender = Comp(Approver, Assessor) with the scenario where WS-reg-cont is not used (i.e., short-circuited). A designer (WS-requester) asks WS-comp-reg to search Approver and Assessor (5). WS-comp-reg finds the two WSs and initiates some protocol so that WS-requester will be connected to WS-composer though a Web interface (16). WS-requester indicates to WS-composer the two WSs Approver and Assessor (17), and WS-composer downloads from WS-comp-reg WSDL-ext, which contains (Perm_desc, Perm_exp, URL_source) of Approver and Assessor (18). Assume that for the two WSs, the unique requirement on access is to pay, hence no commitment is required. WS-composer informs WS-requester of the total required payment (22). Assuming that WS-requester accepts that amount, WS-requester pays WS-composer (24) and then WS-composer pays WS-owners of Approver and Assessor and receives from them the WS descriptions and also the source code of each WS that needs modification (25). Then, WS-requester edits the description and the invariants of the composed Lender (26). Here, the Web interface (through which WS-requester communicates with WS-composer) targets to make the edition operations as intuitive as possible, and the fact to use the graphical formalism CRESS is helpful for reaching such a target. CRESS is also adequate in FI detection, because it is translatable into formal languages (such as LOTOS) for which there exist strong model checking tools. Then, WS-composer launches FI detection (27), receives the result of FI detection (28) and transfers it to WS-requester (29). Assuming that no FI is detected and that no source code needs modification, WS-requester generates an implementation of Lender (30). Assuming that there is no other requirement than payment, WS-requester downloads from WS-composer the description and implementation of Lender (31) before publishing it (see Section 6.2).

Then, we can proceed in the same way to create Supplier = Comp(Dealer1, Dealer2) and Broker = Comp(Lender, Supplier).

8. Conclusion

We have developed in detail a new business model for composing WSs and detecting their FIs. We first present the essential ideas on which our system is built. Then, we describe in detail the global behavior of the system. Finally, we illustrate the use of our business model to create some WSs by composition.

Our business model is designed with an evolutionary spirit, in the sense that it is intended to evolve in future with new modules that provide services to WS-composer. In the present version, FI-detector permits to verify the existence of FIs before composing WSs. An example of service that we plan to add is the assessment of the Quality of Service (QoS) and of the performances of Comp(S1, S2).

Our business model has been checked by applying it in some examples, such as the one presented in Section 7. We intend to develop a prototype of the business model to validate it more intensively. We also plan to validate formally our business model, for example by proving that it satisfies some desirable properties specified in temporal logic, e.g., if Comp(S1, S2) is requested but not found while S1 and S2 are found, then Comp(S1, S2) is created.

References


