Service Composition Based on Functional and Non-Functional Descriptions in SCA

Djamel Belaïd, Hamid Mukhtar and Alain Ozanne

Institute TELECOM, TELECOM & Management SudParis
9, rue Charles Fourier, 91011 Evry Cedex, France

Abstract. Service Oriented Computing (SOC) has gained maturity and there have been various specifications and frameworks for realization of SOC. One such specification is the Service Component Architecture (SCA), which defines applications as assembly of heterogeneous components. However, such assembly is defined once and remains static for fixed components throughout the application life-cycle.
To address this problem, we propose an approach for dynamic selection of components in SCA, based on functional semantic matching and non-functional strategic matching using policy descriptions in SCA. The architecture of our initial system is also discussed.

1 Introduction

In order to provide their services to a large variety of clients, enterprises often manage various contracts with other service providers. One problem faced by such enterprises is the emergence of new competing service providers, with better, cost-effective solutions. Thus, it would be natural that enterprises change partnerships in pursuit of better ones. However, in reality, it is much more different than that.

When inter-enterprise applications are developed on top of the existing Information System, they are created for particular service providers. This results in two major problems. First, if a change of any of the service provider is required, a whole new application needs to developed. Second, if only a part of the functionality of the application is required to be reused, again a new application needs to be deployed. Such problems arise due to the fact that most of the time, the description of service provider is hard-coded in the application logic instead of the service description itself. Thus, we can rightly call such applications as service-provider-dependent rather than service-dependent.

To overcome such difficulties, Service-Oriented Computing (SOC) has emerged recently. SOC is the computing paradigm that utilizes services as fundamental elements for developing applications/solutions. Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications [8]. Service providers procure the service implementations and descriptions, and provide related technical and business support.

However, even after arrival of SOC based approaches, the aforementioned problems have not been solved completely. The applications have started to become modularized

Belaïd D., Mukhtar H. and Ozanne A. (2009).
Service Composition Based on Functional and Non-Functional Descriptions in SCA.
In *Proceedings of the Joint Workshop on Advanced Technologies and Techniques for Enterprise Information Systems*, pages 52-61
DOI: 10.5220/0002201900520061
Copyright © SciTePress

in terms of services, but the definition of services is still dependent on their implementation. One particular approach for realizing SOC based applications, the Service Component Architecture (SCA) avoids such obstacle by separating the service definition from its implementation. However, as we will explore in this paper, SCA is also limited by the fact that applications defined using SCA are static. Once defined, services and their implementation remain intact afterwards. But in an ideal situation, should a service provider changes, the new implementation is to be reused with minimum of effort.

1.1 A Motivating Example

Consider a fictitious travel agency based in Paris. The agency provides services such as flight, hotel and car booking as well as arranging for excursions in a specific destination city. To offer its services, the agency relies on a number of other specialized service providers in France. In order to keep up with so many service providers, the IT personnel at the agency have set up an application that combines the various services from different service providers but that hides this complexity to the travel agent using the software. The selection of a service provider for a particular service, according to criteria such as availability etc, is managed automatically by the application.

Now assume that our agency wants to open a new branch in Madrid. In order to provide their services for various destinations in Spain, the travel agency settles up new agreements with local service providers, which are registered into the system. However, for certain destinations no service provider offers excursion activities. Thanks to the development approach used by IT personnel of the agency, if the application finds that a service provider is unreachable, it tries to find an alternative service provider. If it does not find any service provider for some service, it continues offering the rest of the services.

As the reader can observe, the above example requires several points: first, the application, whose composition defined in terms of services, should be deployable at different locations with different service providers. Second, an application designer should be able to make its application work in a kind of degraded mode if some of the service providers required for its full fonctionalities can not be found. Both of these points form the basis of our approach for service composition used in this paper.

The rest of this paper has been organized as follwing. In Sect. 2 we describe the Service Component Architecture (SCA) upon which we build the rest of the paper. Sect. 3 discusses the notion of abstract and concrete composition and how it can be applied to SCA. Sect. 4 describes the architecture of our system. Sect. 5 provides and overview of related work and Sect. 6 concludes this paper.

2 Service Component Architecture

Service Component Architecture (SCA) [7] provides a programming model for building applications and systems based on a Service Oriented Architecture (SOA). The main idea behind SCA is to be able to build distributed applications, which are independent

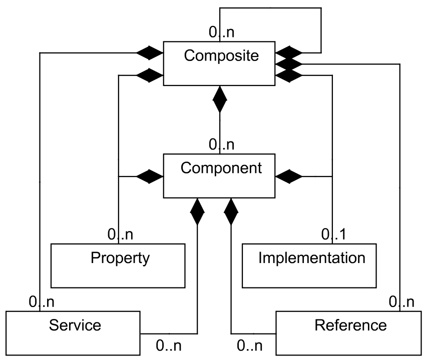

Fig. 1. A basic view of SCA meta model.

of implementation technology and protocol. SCA extends and complements prior approaches to implementing services, and builds on open standards such as Web services. The basic unit of deployment of an SCA application is composite. A composite is an assembly of heterogeneous components, which implement particular business functionality. These components offer their functionalities through service-oriented interfaces and may require functions offered by other components, also through service-oriented interfaces.

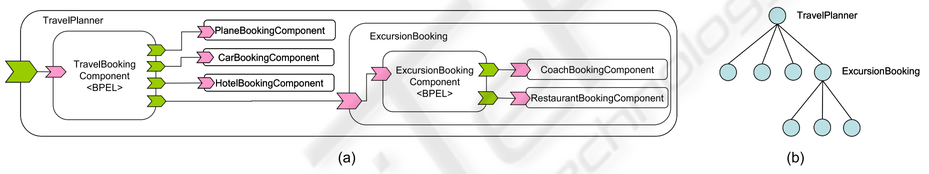

Fig. 2. The TravelPlanner application (a) SCA representation (b) representation as composite tree.

SCA components can be implemented in Java, C++, COBOL, Web Services or as BPEL processes. Independent of whatever technology is used, every component relies on a common set of abstractions including services, references, properties, and bindings. A service describes what a component provides, i.e. its external interface. A reference specifies what a component needs from the other components or applications of the outside world. Services and references are matched and connected using wires or bindings. A component also defines one or more properties, potentially holding its configuration. Fig. 1 shows the various SCA elements and their relationships in the SCA meta-model. As shown, the SCA definition of a composite is recursive, i.e., a composite can contain another composite and so on.

SCA allows dependency injection by relieving the developer from writing the code to find the required references and do the appropriate binding [3]. The bindings are taken care of by the SCA runtime and can be specified at the time of deployment. The bindings specify how services and references communicate with each other. Each binding, separating how a component communicates from what it does, lets the components business logic be largely divorced from the details of communication.

Since SCA already has the notion of services and components and since it allows dynamic binding of services to components, it is an ideal candidate for realization of our proposed approach and, hence, in the rest of the paper we will explain our approach using the SCA artifacts.

2.1 SCA Example Application

First, we show how we can represent our example application in SCA. This has been done in fig. 2(a). The application is described in the composite named TravelPlanner, which offers a single service to the user that is provided by the TravelBooking component. However, the TravelBooking component itself uses services provided by other components namely PlaneBookingCompnent, CarBookingComponent and HotelBookingComponent as well service provided by the ExcusionBooking composite. Finally, the ExcursionBooking composite is also composed of one component namely ExcursionBookingComponent. Note how the services provided by one component are used as references by another component. For example, the ExcursionBookingComponent references are connected to the services provided by the CoachBookingComponent and the RestaurantBookingComponent components.

The TravelPlanner application describes all the services required by the travel agent for a successful trip planning of a client. As mentioned previously, the selection of components implementing these services is made dynamically based on the availability of service provider. However, since the procedure for booking a travel or an excursion is known, such a procedure is already provided in the description of the TravelPlanner composite. Let us assume that this process has been described in BPEL. Our goal is, thus, to find the components that match the references required by the TravelBookingComponent and ExcursionBookingComponent.

3 Abstract and Concrete Composition

We say that a composition is abstract when its description lacks some of the information that defines the composition implementation. Such a composition describes the services participating in the composition, but does not tell about how the services are implemented.[1]

When this concept is applied to SCA, we say that an application described in SCA is abstract if its description does not contain complete implementation definition. However, since an SCA composite is defined recursively, we need to distinguish between various levels of abstraction depending on whether all or part of a composite is abstract. This notion can be better explained by using the composition trees.

3.1 SCA Applications as Composition Trees

The implementation of an SCA composite may be provided by one or more components. However, these components may themselves be defined in terms of other components and so on. This property can be explained easily by a tree structure, where the

[1] We assume the availability of the technical resources required for instantiating and running such a composition, and hence do not treat such aspects.

root is the application itself (i.e., the outermost composite) and its children represent the composites and components enclosed by it. With this tree structure, we observe that the inner nodes of the tree represent the composites and the leaves represent the components. The components, i.e., the leaves of the tree may be found at any level below the root depending on the application composition structure.

Figure 2(b) shows the tree representation of the example SCA application of fig. 2(a). Note that while a composite knows about its contents enclosed by it, it does not have any information about the contents of the composites enclosed by it. For example, in fig. 2(b), the root node (at level 0) knows if its children (at level 1) have known implementations or not, but it does not have this information about the nodes at level 2. To know them, we need to query the composite at level 1.

Bearing such a tree structure in mind, we distinguish between various levels of abstraction for an SCA application:

1) If any of the subcomponents of a composite have no defined implementation, then the composite is *shallow abstract*, e.g., the composite ExcursionBooking at level 1 of the tree in fig. 2(b) is shallow abstract.
2) By recursive definition, if any of the composite enclosed by the root composite is shallow abstract, the composite is called *deep abstract*. For example, the TravelPlanner composite is deep abstract because it encloses the ExcursionBooking composite, which is shallow abstract.
3) If all the subcomponents of a composite have known implementations, then the composite is *shallow concrete*.
4) By recursive definition, if any of the composites enclosed by the root composite is shallow concrete, the root composite is *deep concrete*.

Figure 3 shows these various levels of abstraction diagrammatically.

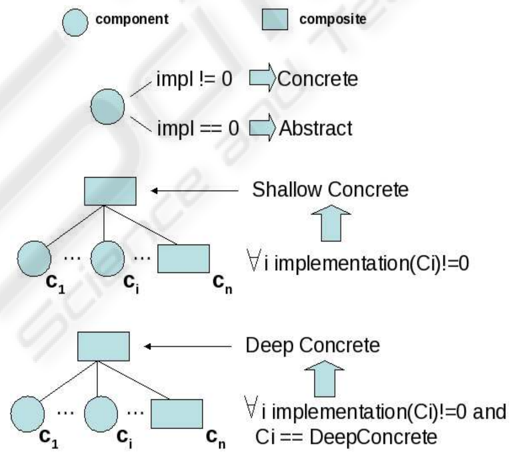

Fig. 3. The different levels of abstraction for SCA applications.

Our aim is to build a concrete composition tree, which is semantically equivalent to a given (shallow or concrete) abstract tree. Its fundamental principle is to replace the abstract components of a composition tree by semantically equivalent concrete ones. We assume that a number of concrete components are available in some repository, which is accessible to us and we need to make a selection out of them.

However, the SCA specifications [7] do not specify any mechanism for matching of services and their implementations (components). Thus, we propose an architecture for matching of services to components.

4 The System Architecture

Figure 4 describes the architecture of our proposed system. The Composer is the main entity, which initiates the composition process by accepting the abstract application as input. It uses services of the Semantic Trader for matching of abstract and concrete components. The Semantic Trader depends on the services provided by the Registry for requesting components and uses the services of Semantic Trader for matching services and components. The Composer uses the Selection Strategy entity for deciding which strategy to use based on the current policy employed by the system administrator. The role of these entities is discussed in the following sub-sections.

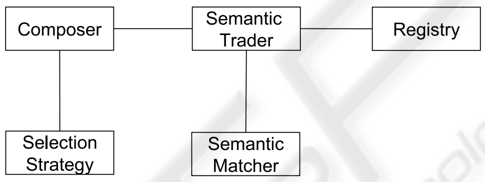

Fig. 4. The System Architecture.

4.1 Transformation of Tree

The Composer transforms the input abstract application into an equivalent concrete one. The transformation process consists of three intermediate stages:

1) First, the Composer transforms the application into a composition tree structure as described previously. From the composition tree, the Composer selects a sub-tree that only keeps branches of which leaves are abstract components. In other words, if some components have well-defined implementations, they are not considered for processing.
2) While walking down the abstract tree, for each component node, we look in the repository for a concrete component, which is semantically equivalent to the abstract one and replace the description of the latter by the former.
3) During the second stage, we may find more than one component or no matching components at all for an abstract service. The Composer uses a strategy for deciding on what to do in such a case.

4.2 Semantic Matching

To be able to reason about the functional properties of SCA artifacts, we use semantic matching, as described in the second stage of the transformation process.

SA-SCA:Semantic Annotations for SCA We propose Semantic Annotations for SCA (SA-SCA), which suggests how to add semantic annotations to various SCA artifacts like composite, services, components, interfaces, and properties. This extension is similar to the concept of annotations in SAWSDL [1] and is in accordance with the SCA extensibility mechanism [7]. Our proposed SA-SCA defines a new namespace called *sasca* and adds an extension attribute called *modelReference* so that relationships between SCA artifacts and concepts in another semantic model are handled. This choice is motivated by the fact that applications developers can use any ontology language to annotate services rather than be bound to one particular approach. The listing below shows the description of our TravelPlanner composite:

```
<composite name="TravelPlanner">
  <service name="TravelPlannerService"
   promote="TravelBookingComponent/
            TravelBookingService"/>
  <component name="TravelBookingComponent">
    <service name="TravelBookingService"/>
    <reference name="PlaneBookingService"/>
    <!-- ... (other references) -->
    <implementation.bpel
      process="TravelBoooking.bpel"/>
  </component>
  <component name="PlaneBookingComponent"
   modelReference=
   "booking.owl#PlaneBooking">
    <service name="PlaneBookingService"
      modelReference=
      "booking.owl#PlaneBooking"/>
  </component>
  <!-- ... (other component definitions) -->
</composite>
```

4.3 Selection Strategy

The selection of a concrete component can be filtered out by using a particular strategy employed by the administrator based on the organizational policies. For example, one policy might be the selection of the service provider based on seasonal variations as described in the example scenario, but other strategies could also be used. For example, when there are many concrete components, the choice of one concrete component over the other may be influenced by the QoS factors such as price, response time, reputation, etc. Such strategies can be defined along with SCA application description by using the SCA Policy Framework [6].

Partial Composition. It is possible that while walking the abstract tree, we do not find all the components. Instead of resulting in a failed composition, the component selection policy may specify to ignore such concrete components and build a composition tree with only the available components. For example, a transaction policy may require inter-component dependency and, hence, if any of such components is missing, the transaction will not be executable. In such a case, it is important to find all the relevant components and if they are not found, the policy may require to require abandoning the dependent components and continuing with components for the rest of services in the composition. The description and enforcement of such policies is dependent on particular use case and we do not consider them in this paper, but provide an overview in Sect. 5.

It is then important to notice that we provide the possibility for both a shallow and a deep transformation of the composite. In the first case, the composite description is brought to a shallow concrete state, while in the second case a deep concrete tree is created. Considering the TravelPlanner composite, its shallow transformation will replace the CarBookingComponent, HotelBookingComponent, and PlaneBookingComponent components with concrete ones, and its deep transformation will, in addition to these, replace the CoachBookingComponent and RestaurantBookingComponent components. This possibility is interesting in the case of a distributed composition. A Composer can process a shallow transformation on a composite located on its hosting computer, and delegate the transformation of the distant subcomposites to their colocated Composers.

We are currently implementing our proposed architecture as part of the French National project ANR-SCOrWare.[2]

5 Related Work

The idea of describing application as an abstract composition of services which are resolved into service components dynamically, has been treated previously. In COCOA [2], the objective is to find concrete components for abstract services defined in a user task. Their solution builds on semantic Web services (OWL-S) and offers flexibility by enabling semantic matching of interfaces and ad hoc reconstruction of the user tasks conversation from services conversations. Furthermore, COCOA allows meeting QoS requirements of user tasks. For this purpose, they have created COCOA-L, an extension of OWL-S, that allows the specification of both local and global QoS requirements of user tasks. Compared to their approach, our approach also proposes use of semantic matching but instead of using a fixed number of QoS attributes, we propose to use the SCA policy Framework for specifying organization-level policies. Our approach is more flexible because the service developers and the deployers have full control over defining the appropriate policy for component selection.

The subject of semantic service description has also been treated by various research works. Semantic Annotations for WSDL (SAWSDL) [1] defines how to add semantic annotations to various parts of a WSDL [10] document such as input and output message structures, interfaces and operations. For this purpose, SAWSDL defines a new

[2] http://www.scorware.org/

specific namespace *sawsdl* and adds an extension attribute, named *modelReference*, to specify the association between WSDL components and concepts in some semantic model. The matching between a concept and WSDL element is done by using a matching algorithm. One such matching algorithm is proposed in [4]. Following the example of WSDL extension, we have extended SCA to be able to carry out semantic matching for different SCA elements including services, components, interfaces, and properties.

Finally, there has been significant recent work related to the use of policies in SCA. One such approach uses the SCA policy framework for abstract and concrete resource specification [5] which is then used for matching abstract services with their concrete component implementations. However, the approach is based on syntactic matching of SCA artifacts. This approach, together with ours, can be used as a component selection strategy as described in Sect. 4.3. Similarly, in [9] the authors define patterns and roles for applying abstract policies in SCA to their concrete implementations. With an example application they show how their approach can be applied for transactional policies.

6 Conclusions & Future Work

We have presented an approach for dynamic composition of applications whose composition is described in terms of the services provided by the application; however, these services are resolved into component implementations at the time of execution of the application. The service implementations might be distributed and provided by different service providers whose selection is influenced by the policies used by the system administrator. The selection of a particular implementation is made on the basis of a matching algorithm.

The applications are described in SCA. Currently, we consider only applications whose composition in terms of services is defined statically. For the future work, we are looking forward to having such applications created automatically in the pervasive environments in terms of the services available in the environment. We also intend to consider the user's preferences apart from the organizational policies when looking for candidate service providers.

References

1. Akkiraju, R. and Sapkota, B., 2006. Semantic annotations for WSDL. Technical report, W3C. (cf. http://www.w3.org/TR/sawsdl-guide/).
2. Ben Mokhtar, S., Georgantas, N., and Issarny, V., 2007. COCOA: Conversation-based service composition in pervasive computing environments with qos support. *J. Syst. Softw.*, 80(12):1941–1955.
3. Chappel, D., 2007. Introducing sca. White paper. Available online:http://www.osoa.org.
4. M'Bareck, N. O. A. and Tata, S., 2007. How to consider requester's preferences to enhance web service discovery? In *Internet and Web Applications and Services, 2007. ICIW '07. Second International Conference on*, pages 59–59.
5. Mukhtar, H., Belaïd, D., and Bernard, G., 2008. A policy-based approach for resource specification in small devices. In *UBICOMM 08: The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE.

6. Open SOA Collaboration, 2007a. SCA Policy Framework v1.00 specifications. http://www.osoa.org/.
7. Open SOA Collaboration, 2007b. Service Component Architecture (SCA): SCA Assembly Model v1.00 specifications. http://www.osoa.org/.
8. Papazoglou, M. P., 2003. Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12.
9. Satoh, F., Mukhi, N. K., Nakamura, Y., and Hirose, S., 2008. Pattern-based Policy Configuration for SOA Applications. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 1, pages 13–20.
10. Web Services Description Language. WSDL 2.0 Home Page, 2006. http://www.w3.org/TR/wsdl20/.