Dynamic Component Selection for SCA Applications

Djamel Belaïd, Hamid Mukhtar, Alain Ozanne, and Samir Tata

Institut TELECOM, TELECOM & Management SudParis
9 rue Charles Fourier, 91011 Evry Cedex
France
{djamel.belaid,hamid.mukhtar,alain.ozanne,samir.tata}@it-sudparis.eu

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 have previously proposed an approach for dynamic selection of components in SCA, based on functional semantic matching and non-functional strategic matching using policy descriptions in SCA. In this paper, we extend our existing approach by providing further flexibility in component selection and present the architecture and implementation of our system. An evaluation of the system is also reported.

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, which is not always feasible. 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,

C. Godart et al. (Eds.): I3E 2009, IFIP AICT 305, pp. 272–286, 2009.

platform-agnostic computational elements that support rapid, low-cost composition of distributed applications [1]. Services are offered by service providers —organizations that procure the service implementations, supply their service descriptions, and provide related technical and business support.

However, even after arrival of SOC based approaches, the aforementioned problems have not been solved completely. Although the applications have started to become modularized in terms of services, they are still not decoupled from their underlying platforms —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, services can be provided by different providers differently and, hence, will have different implementations. Should a provider changes, the new implementation is to be reused with minimum of effort.

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 fact, given the large number of destinations and depending on the time of the year, different destinations are served by different service providers at different time of the year. 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 without letting the travel agent, who is using the application, knows how many and which service providers he is dealing with when making a transaction. The selection of a service provider for a particular service for a particular time period 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. Once all the new service providers have been identified, they are registered into the system and the selection of the proper agencies for each type of service is managed automatically, according to the conditions of the agreements and requirements of the travel agency. However, for certain destinations no service provider offers excursion activities. Thanks to the development approach used by IT personnel of the agency, the application will still be able to offer the rest of its services to the travel agent, even though it is missing some of the services for those destinations. This is possible because 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 is 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 functionalities can not be found. Both of these points formed the basis of our previous approach for service composition in SCA [2]. In this paper, we extend our prior approach to add further flexibility in the composition process. Also, an evaluation of our implemented system is provided in this paper.

The rest of this paper has been organized as following. First, in Sect. 2 we describe the some related work done by others. In Sect. 3 we describe the Service Component Architecture (SCA) upon which we build the rest of the paper. Sect. 4 discusses the notion of abstract and concrete composition and how it can be applied to SCA. Sect. 5 describes the architecture of our system, its implementation and usability while Sect. 6 provides its evaluation. Sect. 7 concludes this paper along with description of the intended future work.

2 Related Work

The idea of describing application as an abstract composition of services, which are resolved into service components dynamically, has been treated previously. However, existing works mostly treat the process from the point of view of a user in a pervasive environment. For example, in the COCOA approach [3], 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 being bound to a particular semantic description language such as OWL-S, we propose to use semantic annotations, which are independent of description languages. Also, our approach is more relaxed by providing the possibility to define the different levels of abstraction at different phases of application life-cycle as will be described in the paper.

The Aura project [4] defines an architecture that realizes user tasks in a transparent way. The user tasks defined in Aura are composed of abstract services to be found in the environment. Gaia [5] is a distributed middleware infrastructure that enables the dynamic deployment and execution of software applications. In this middleware, an application is mapped to available resources of a specific active space. This mapping can user-assisted or automatic. Gaia supports the dynamic reconfiguration of applications. For instance, it allows changing the composition of an application dynamically upon a users request (e.g., the user may specify a new device providing a component that should replace a component currently used). Furthermore, Gaia supports the mobility of applications between active spaces by saving the state of the application. Both of the previous platforms introduce advanced middleware to ease the development of

pervasive applications composed out of networked resources. However, they are too restrictive when it comes to interoperability between different applications, specifically when they are provided by different parties. Both approaches assume framework-dependent XML-based descriptions for services and tasks. In other words, both approaches assume that services and tasks of the environment are aware of the semantics underlying the employed XML descriptions. However, in it is not reasonable to assume that service developers will describe services with identical terms worldwide. It is for this reason that we base our approach on SCA (Service Component Architecture) which is an open standard, independent of any particular implementation technology or communication protocol.

The subject of semantic service description has also been treated by various research works. Semantic Annotations for WSDL (SAWSDL) [6] defines how to add semantic annotations to various parts of a WSDL [7] document such as input and output message structures, interfaces and operations. For this purpose, SAWSDL defines a new 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 [8]. 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. As we will describe in the rest of this paper, SCA applications can easily be described at various levels of abstraction and provide a flexible way of extension for supporting semantic descriptions.

There has been some recent work related to the use of policies in SCA. One such approach uses the SCA policy framework [9] for abstract and concrete resource specification [10] 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 our current approach, can be used as a component replacement strategy as described in Sect. 5.2. Similarly, in [11] 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.

3 Service Component Architecture

Service Component Architecture (SCA) [12] 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 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.

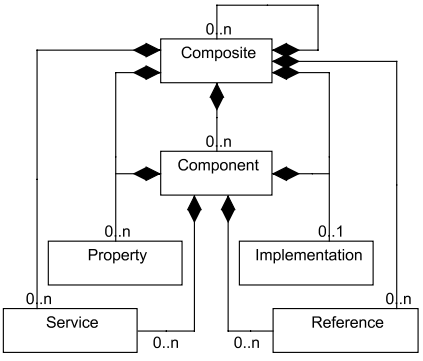

Fig. 1. A basic view of SCA meta model

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. For example, a component might rely on a property to tell it what part of the world it is running in, letting it customize its behavior appropriately. Figure 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 [13]. 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 defines a particular protocol that can be used to communicate with a service as well as how to access them. Because bindings separate how a component communicates from what it does, they let the components business logic be largely divorced from the details of communication. A single service or reference can have multiple bindings, allowing different remote software to communicate with it in different ways.

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.

3.1 SCA Example Application

First, we show how we can represent our example application in SCA. This has been done schematically in fig. 2(a). The listing below shows how the same SCA

application is defined in SCDL (Service Component Description Language), an XML-based description of SCA applications.

```
<composite name="TravelPlanner">

  <service name="TravelBookingService"
    promote="TravelBookingComponent/TravelBookingService"/>

  <component name="TravelBookingComponent">
    <service name="TravelBookingService">
      <interface/>
    </service>
    <implementation.bpel process="BookingProcess"/>
    <reference name="PlaneBookingService"/>
    <reference name="CarBookingService"/>
    <reference name="HotelBookingService"/>
    <reference name="ExcursionBookingService"/>
    <implementation.bpel process="TravelBoooking.bpel"/>
  </component>

  <component name="ExcursionBookingComponent">
    <service name="ExcursionBookingService">
      <interface/>
    </service>
    <implementation.composite name="ExcursionBooking"/>
    <!-- references to coach and restaurant booking components -->
  </component>

  <!--PlaneBooking, CarBooking and HotelBooking components definitions-->

  <wire source="TravelBookingComponent/ExcursionBookingService"
    target="ExcursionBookingComponent/ExcursionBookingService"/>
  <!-- wires between other components of the TravelPlanner composite -->

</composite>
```

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

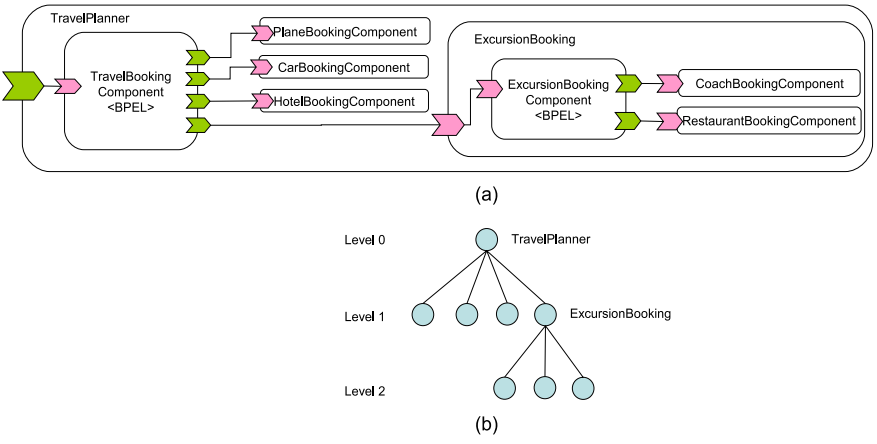

Fig. 2. The TravelPlanner application (a) SCA representation (b) representation as composite tree

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 ExcursionBooking-Component.

4 Abstract and Concrete Composition

As mentioned in the motivating example, an application composition can be described abstractly so that its concretization can be carried out dynamically depending upon the context in which it is used. In general, 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.

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

4.1 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 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*. However, it is shallow abstract if only the implementation of one of its subcomponents is not defined. 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 all of the composites enclosed by the root composite are shallow concrete, the root composite is *deep concrete*.

Figure 3 shows these various levels of abstraction diagrammatically.

Our aim is to build a concrete composition tree, which is semantically equivalent to a given (shallow or deep) abstract composition 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.

4.2 Transformation of Tree

Our aim is to transform the input abstract application into an equivalent concrete one. This transformation process consists of three intermediate stages:

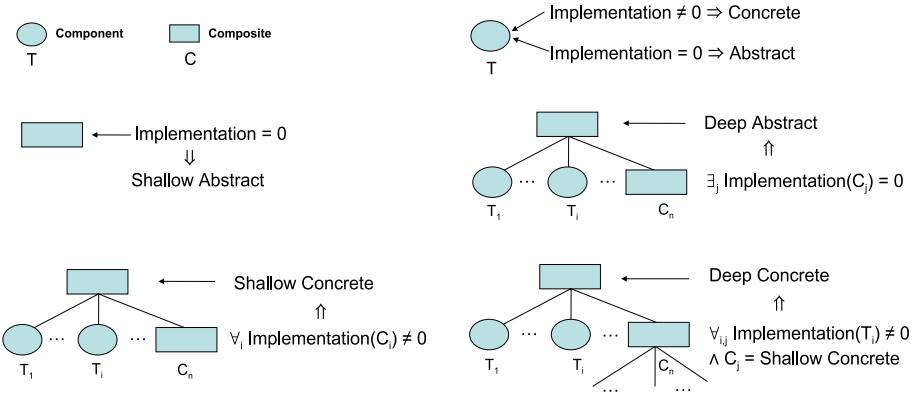

Fig. 3. The different levels of abstraction for SCA applications

1) First, the application is transformed into a composition tree structure as described previously. From the composition tree, a sub-tree is selected that keeps only those branches whose 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 abstract one by the concrete one.
3) During the second stage, we may find more than one component or no matching components at all for an abstract service. We need to determine a strategy for deciding on what to do in such a case.

   In the second stage of transformation process, semantic matching is used for matching. However, the SCA specifications [12] do not specify any mechanism for matching of services and their implementations (components). Thus, we propose a mechanism for semantic description of services and components for matching purposes.

4.3 Semantic Description

To be able to reason about the functional properties of SCA artifacts, we add semantic descriptions to them, 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 [6] and is

in accordance with the SCA extensibility mechanism [12]. 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 abstract CoachBookingComponent component:

```
<component name="CoachBookingComponent"
  sasca:modelReference="http://tp.org/booking.owl#CoachBooking">
  <service name="CoachBookingService">
    <interface.java interface="com.example.coachBookingServiceItf"/>
  </service>
</component>
```

Note that the component description now has a reference to an OWL ontology, which contains the definition of the CoachBooking concept. When this abstract component is matched with concrete components, it will be ensured that both of them refer to the same CoachBooking concept. Only if they match, the concrete component description can be used. For example, the coach booking service provided by an agency in Madrid is implemented in Java and described in the following listing:

```
<component name="MadridCoachBookingComponent"
  sasca:modelReference="http://tp.org/booking.owl#CoachBooking">
  <service name="MadridCoachBookingService">
    <interface.java interface="com.example.coachBookingServiceItf"/>
  </service>
  <implementation.java name="spaincoach.madrid.booking.CoachBookingServiceImpl"/>
</component>
```

Since the *modelreference* attribute in both the abstract and concrete descriptions refer to the same CoachBooking concept, they will match.

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. An application composer can process a shallow transformation on a composite located on its hosting computer, and delegate the transformation of the distant subcomposites to the composers located on those hosts.

5 System Architecture and Implementation

So far, we have discussed our approach for abstract and concrete composition. In this section, first we describe the architecture describing the entities involved in

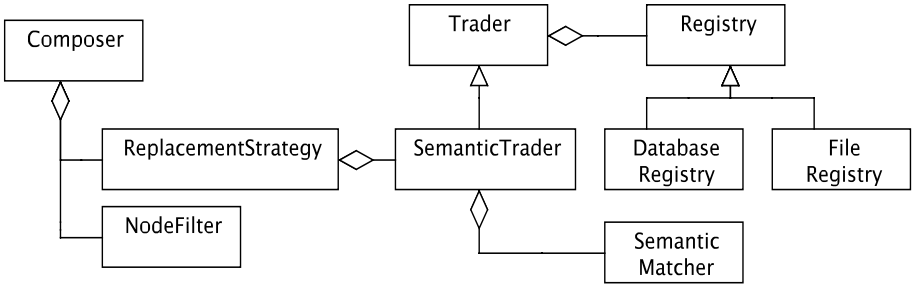

Fig. 4. The Semantic Trader architecture

realizing the abstract-to-concrete composition in Sect. 5.1 and then we provide the details of our implementation, developed as a proof of concept, in Sect. 5.2.

5.1 Architectural Components

Figure 4 describes the architecture of our proposed system. The Composer is the entity in charge of the transformation of the abstract composition description. In order to do so, it uses NodeFilter for selection of nodes in the abstract tree and a particular ReplacementStrategy for replacement of abstract components by concrete ones. Hence, the Composer walks through the abstract tree and when the NodeFilter accepts the current node, the description in the abstract composition is replaced with the one returned by the ReplacementStrategy.

In order to determine what to put in place of an abstract component description, the ReplacementStrategy uses the SemanticTrader which returns the description of a component semantically equivalent but concrete to a given abstract component description.

The SemanticTrader can do this because it specializes the Trader which provides an extensive access to the Registry that contains the concrete components descriptions. A SemanticMatcher is used to compare the abstract component with the concrete ones returned by the Registry.

5.2 Implementation

Currently, our implementation provides two different possibilities for use as a ReplacementStrategy. These strategies actually provide the possibility of replacement of either complete or partial description of abstract component by the concrete one:

– the ImplementationOnlyReplacementStrategy keep the complete description of the abstract component but add to it the implementation field in the description of the concrete component. This strategy is meant to be used when SCA wires refer explicitly to the component and interface names of the abstract component, which are then needed to be kept intact.

– The FullReplacementStrategy replaces the complete description of the abstract component with the complete concrete component. This strategy can be used when SCA wires to the replaced component are automatically generated. It is then possible to import the complete description of the concrete replacement component into the outer composite.

Our implementation also provides three NodeFilters: one that accepts only an abstract component, another that accepts only abstract composite, and the third one that accepts both. So it is possible to apply a specific replacement strategy to each type of abstract components. For example, the FullReplacementStrategy can be used for replacing a full abstract composite, as the concrete replacement composite may promote its subcomponents interfaces.

The Registry can also have various implementations depending on the way the available concrete components are serialized. Currently we provide a database implementation. The DatabaseRegistry uses a MySQL base in which components are stored in a simple table that contains for each component: its name, its XML description, its unique key determined on its registration request, and its provider id.

As we are looking forward to giving a public access to the Composer, the Registry also maintains the list of its authorized users. Indeed each operation on the trader, i.e., component request or publication, requires a user key. There are three kinds of users, each with different rights:

– the customer can request components from the trader,
– the provider can register components but also request them,
– the administrator can register new users, customer or provider.

Usability of Our Approach. By using a service component-oriented model and by dynamically selecting components using semantic description, our approach can be used for service and component bindings both at design time and at runtime. At design time, when the application designer defines the abstract composition, he can select and reuse the concrete components from the component repository —if they are available— and do the bindings at design time, i.e., an early binding is provided. However, if not all components can be found at design time, the designer can leave the choice to the container, which can carry out late binding depending on the available concrete components. This leads to greater flexibility.

6 Evaluation

In order to benchmark our approach we generated various sets of primitive SCA components counting from 100 to 600 elements. We semantically annotated their services and references with concepts taken from the Rosetta ontology (which has 63 classes and 30 subclass relations)[2]. We took care to have a uniform

[2] Rosetta is available at http://www.w3.org/2002/ws/sawsdl/spec/ontology/rosetta.owl

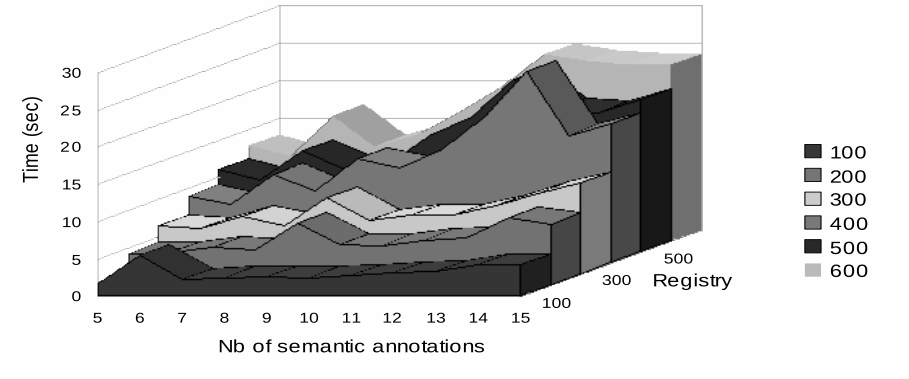

Fig. 5. The time for turning abstract primitive components into concrete ones

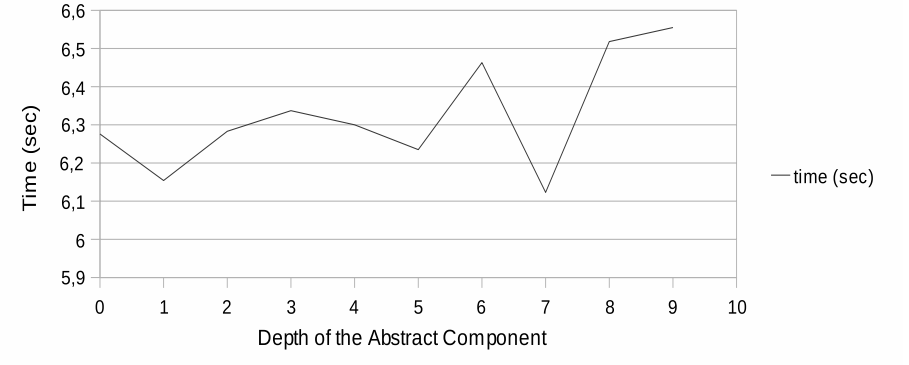

Fig. 6. The cost of the abstract tree building and exploration

distribution of components with respect to the number of semantic annotations they contain. The tests have been done on a 1.86 GHz Pentium M with 1 Gb RAM.

First, we look at how the time to turn an abstract primitive component into a concrete one grows. In order to do so, we took a set of 10 abstract components, holding from 5 to 15 semantic annotations. We measured how long it took to find their concrete equivalents in sets of concrete components varying from 100 to 600 components. What we noticed is that for small number of annotations, the time required to turn an abstract component into a concrete one grows linearly and slowly with the size of the registry. But the slope becomes abrupt for components holding large number of annotations, as shown in figure 5.

Then we look at the impact of the abstract tree building and exploration for the transformation of a deep abstract composite into a deep concrete one. To do so, we took a composite component of ten levels depth. We placed an abstract

primitive component to its deepest level and measured the time required for the transformation of the composite. Then we repeat the operation with the abstract component on the other levels of the composite, the result is shown in the figure 6. The additional cost of the building and exploration of the abstract tree is at most 0.25 second, to be compared to the 6.5 seconds took by the matching process. So we shall optimize this latter part of our tool in the future.

7 Conclusions and Future Work

We have presented an approach for dynamic composition of applications whose composition is described in terms of the services provided/required by the application; however, these services are bound to implementations either at design time or dynamically at the time of execution of the application depending on the availability of concrete components in the current context. The service implementations might be distributed and provided by different service providers whose selection is influenced by a particular replacement strategy. The selection of a particular implementation is made on the basis of a matching algorithm. We have discussed an implementation of our system, whose evaluation is also provided.

The applications we consider are described in SCA. To resolve an abstract component, our system looks for the corresponding concrete component. However, it is possible that the implementation of an abstract component may not be provided by any available concrete components; rather we may find more than one component providing the same functionality required by the abstract component. Similarly, if a composition tree requires to resolve several abstract components and instead of providing various concrete components, our system will provide a single concrete composite, which provides the same functionality; this might be either due to unavailability of some of the required concrete components or due to performance reasons.

Currently, we consider only applications whose composition in terms of services is defined statically. In the future, we are looking forward to having such applications created automatically in the pervasive environments in terms of the services available in the environment.

References

1. Papazoglou, M.P.: Service-oriented computing: concepts, characteristics and directions. In: Proceedings of the Fourth International Conference on Web Information Systems Engineering, WISE 2003, December 2003, pp. 3–12 (2003)
2. Belaïd, D., Mukhtar, H., Ozanne, A.: Service Composition Based on Functional and Non-functional Descriptions in SCA. In: Proceedings of The 1st International Workshop on Advanced Techniques for Web Services, AT4WS 2009, Milan, Italy (2009)
3. Ben Mokhtar, S., Georgantas, N., Issarny, V.: COCOA: Conversation-based service composition in pervasive computing environments with qos support. J. Syst. Softw. 80(12), 1941–1955 (2007)

4. Sousa, J., Garlan, D.: Aura: an Architectural Framework for User Mobility in Ubiquitous Vomputing Environments. In: WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture, Deventer, The Netherlands, pp. 29–43. Kluwer, B.V., Dordrecht (2002)
5. Román, M., Campbell, R.H.: A middleware-based application framework for active space applications. In: Endler, M., Schmidt, D.C. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 433–454. Springer, Heidelberg (2003)
6. Akkiraju, R., Sapkota, B.: Semantic annotations for WSDL. Technical report, W3C (September 2006), http://www.w3.org/TR/sawsdl-guide/
7. WSDL 2.0 Home Page: Web Services Description Language (2006), http://www.w3.org/TR/wsdl20/
8. Ould Ahmed M'Bareck, N., Tata, S.: How to consider requester's preferences to enhance web service discovery? In: Second International Conference on Internet and Web Applications and Services, ICIW 2007, May 2007, pp. 59–59 (2007)
9. Open SOA Collaboration: SCA Policy Framework v1.00 specifications (2007), http://www.osoa.org/
10. Mukhtar, H., Belaïd, D., Bernard, G.: A policy-based approach for resource specification in small devices. In: UBICOMM 2008: The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies. IEEE, Los Alamitos (2008)
11. Satoh, F., Mukhi, N.K., Nakamura, Y., Hirose, S.: Pattern-based Policy Configuration for SOA Applications. In: IEEE International Conference on Services Computing, SCC 2008, July 2008, vol. 1, pp. 13–20 (2008)
12. Open SOA Collaboration: Service Component Architecture (SCA): SCA Assembly Model v1.00 specifications (2007), http://www.osoa.org/
13. Chappel, D.: Introducing Service Component Architecture. White paper (July 2007), http://www.osoa.org