Dynamic Contextual Service Ranking

André Bottaro[1,2] and Richard S. Hall[2]

[1]Francetelecom R&D, 28 Chemin du Vieux Chêne
38243 Meylan Cedex, France
andre.bottaro@orange-ftgroup.com
[2]LSR-IMAG, 220 rue de la Chimie - Domaine Universitaire, BP 53
38041 Grenoble, Cedex 9, France
richard.hall@imag.fr

Abstract. This paper explores service composition in pervasive environments with a focus on dynamic service selection. Service orientation offers the dynamism and loose coupling needed in pervasive applications. However, context-aware service composition is still a great challenge in pervasive computing. Service selection has to deal with dynamic contextual information to enable context-aware behaviors to emerge from the environment. This paper describes how to add dynamic contextual service ranking mechanisms to the service-oriented OSGi framework and further discusses service re-composition decisions.

1 Introduction

Pervasive environments emphasize the need for application dynamism and autonomic behaviors. These environments are characterized by the variability and mobility of acting entities on the network. Dynamically composing applications and guaranteeing service continuity to the users is a grand challenge in pervasive computing.

The home network is such an environment. Heterogeneous mobile and fixed devices join and leave the network. Numerous protocols coexist and provided interfaces need to be adapted. These devices offer various features to the users. Home context dynamism comes on the one hand from device mobility and evolving capacities and on the other hand from user mobility and activity.

Our previous work answered pragmatic problems related to dynamic service availability, distribution and interface heterogeneity [3]. In this work we also encountered other issues regarding dynamic adaptability. Context awareness is the key aspect in pervasive computing on which our present work focuses. Context awareness based on a context management system is a non-functional need that can be automatically dealt with in pervasive software composition.

Device self-organization and feature composition fulfilling the needs of the users are the ultimate goals for work in pervasive environments. Since service orientation is at the basis of much work in pervasive computing, many attempts to reach "self-organization" focus on service composition. Available plug-n-play middleware relies

on protocols for service discovery, service description, service control and address some other features from Quality of Service (QoS) to security [19].

Service selection and ranking, which consist of selecting the most appropriate service provider out of a list of discovered ones, is however not fully addressed by available service-oriented protocols today. The mechanisms are complex due to the complexity and dynamism of the home context. This task is often left to manual configuration. Automatic service selection is a difficult task that a lot of work tries to tackle. We propose a realistic approach in this domain and describe the implementation of our ideas on top of the OSGi framework [14].

This work is carried out within the ANSO[1] project, which brings together major European actors interested in the development of pervasive home services, including telecommunication operators (France Telecom), video companies (Thomson) and home control solution providers (Schneider Electric). This work is demonstrated on a home application jointly developed by France Telecom and its ANSO partners [2].

The next section describes our vision of context management in service-oriented applications. The service selection problem is defined in section 3. The framework implementation on top of the OSGi platform [14] is detailed in section 4. The realization of a context-aware application on top of this framework is depicted in section 5. Comparisons with related work are found in section 6. The last section concludes on the use of this framework and discusses the opportunity to leverage this work to further design generic service rebinding mechanisms.

2 Vision

Application intelligence emerges from the behavior of its constituent components. In order to let the intelligence emerge from software composition, we strongly believe in a component-oriented vision where components are self-descriptive and self-adaptive. Autonomic mechanisms are to be implemented at the fine-grained level of the architecture. Our approach follows the grand challenge vision of autonomic computing [12].

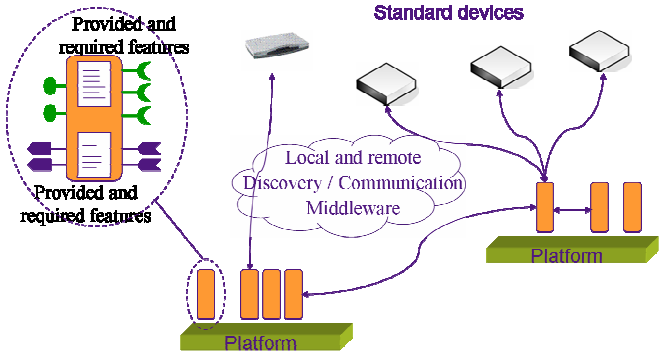


Figure 1. Autonomic elements represented on service platforms

1 ANSO is supported by the French Ministry of Industry under the European ITEA program.

Autonomic elements in our system are software components that provide and require services (see Figure 1). These services are distributed on the network or only locally visible on service platforms. Many works show how to reify distributed components on service platforms [3]. So the composition issue here is reduced to composition of local components on a unique service platform. Provided services and required services are declared at the component level in a specific component description. We leverage a service-oriented component architecture where service dependencies are dynamically resolved on every component thanks to a generic container dedicated to service publication, service discovery and service binding [5].

The target environment of our prototype is the home network [2]. In this environment, context-aware applications must adapt themselves according to dynamic variables. The home context consists of (i) the user context comprised of location, preferences and activity, (ii) the device context comprised of location and capacities (iii) and external physical context such as place properties (weather parameters, physical location, etc.) [16].

This context is assessed by a context management system reasoning on information coming from sensors distributed in the environment. This system is visible on our service platform in the form of services identified as *context sources*. Based on reasoning techniques, context sources are able to identify, locate and assess the activity of users and devices in the home. Some context sources directly represent underlying sensors and some others provide treated and aggregated information from one or several context sources.

Context sources are the entities responsible for the contextualization of service trading. On the one hand, high-level context sources dynamically qualify provided services with contextual properties. On the other hand, they dynamically refine service requirements with contextual information.

We consider that the service platform is the smart dynamic receptacle of pervasive entities. This adaptable receptacle turns pervasive software composition into the composition of uniform contextualized service components. Service components are running on an adaptable executive environment, which we call a service platform. The main role of the platform is to adapt to the environment in locally representing all the relevant external entities with their context on the platform.

The contextualization of service trading in our platform-centric vision is discussed in the remainder of this paper.

3 Service Selection

3.1 Concepts

Service selection consists of selecting the most adequate service provider according to service requester needs. A service is described by a type (a.k.a. an abstract interface) that determines a set of operations (a.k.a. methods or actions) and by service

metadata (e.g., properties) that qualifies the service. Service metadata is linked to contextual properties like location, features quality, current available capacities, etc.

Service metadata is often attribute-value pairs in various service architectures (OSGi[14], SLP[10], UPnP[17], Web Services[9] with ws-metadata-exchange and ws-policy). Services in the OSGi framework and in protocol middleware like SLP and UPnP can have an arbitrary set of attribute-value pairs attached to them. WS-Policy defines an extensible grammar to express capabilities for Web Service Provider and requirements for Web Service requester. Multiple attribute-value pairs with scoring functions are particularly common in e-commerce (see [13] for a list of technologies).

Many concepts are defined in existing middleware and in scientific literature to allow service selection:

- A **scope** defines the perimeter of a search request. Scopes are usually determined by network topology, user role or application context [19]. For instance, multicast communication methods naturally limit UPnP search requests to topological networks. Contextual scopes in SLP are freely assigned by service providers to their services at registration time and service requesters use scopes as a primary search filter. The use of hierarchical composition scopes is promoted in some component oriented architectures [5].

- A **filter** narrows the set of available services according to the service requester needs. It defines an acceptance threshold. In service-oriented architecture, the main filter concerns the service type. A service requester usually looks for precise service types. In models defining service properties as a set of attribute-value pairs [5], the equality operator and threshold operators are possible basic filter operators. We may distinguish mandatory and optional filters. A **mandatory filter** is always evaluated before service binding and eliminates the services that do not match from further evaluation. **Optional filters** are evaluated if the remaining number of services after the mandatory filter evaluation is greater than the number of services expected to be bound. They refine the first filter in order to narrow service selection (e.g., [8]).

- The objective of a **scoring function** (a.k.a. utility function, objective function, etc.) is to rank all the filtered services in a total order according to the adequacy of their ability to meet the service requirements. Many approaches on service ranking are based on the utility theory [11]. The latter defines properties as a set of attribute-value pairs qualifying resources, which may be services. A scoring function is a sum of weighted placeholders. Placeholders are attached to sub-scores evaluating property values of the potential services against the resource requester needs. The scoring function enables the service requester to rank available service providers. The service provider getting the best score is considered the one that optimizes service composition.

- **Policies** define ranking algorithms and may be implemented with scoring functions [13]. A policy has a very generic meaning. Some definitions embrace a more generic rule-based mechanism like Event-Condition-Action rules. For instance, reactive adaptation policies are defined in [7].

3.2 Problem definition in context-aware applications

Most of the service middleware only provides simple means to select services. Selection most often relies on scopes and filters due to the simplicity of these mechanisms. Scopes and filters narrow the set of possible bindings, but they make the service requester consider the remaining possibilities as identical. Service ambiguity occurs when several service providers fulfill the same service requirement. Therefore, these mechanisms do not avoid service ambiguity, which leads to composition unpredictability. This problem raises the need for the introduction of ranking algorithms in service composition mechanisms.

In context-aware applications, algorithms target the ranking of service providers against service contextual requirements at runtime. High-level mechanisms are required to separate the definition of contextual behavior from the core application. The problem can be decomposed into 4 subjects that are discussed in the next section:

- **Contextual service property management**: Service provider context may be acquired into the device itself or externally. This information has to be dynamically added in the description of the provided services.
- **Contextual service requirement management**: Static preferences and dynamic activity may be acquired through some internal inputs or external ones to the service requester. This information has to be dynamically added to the requirements of service requesters.
- **Dynamic service ranking**: Requirements can be designed into scopes, a mandatory filter, optional filters, and a sorting algorithm in order to dynamically rank the available services. A change in rankings generates rebinding actions.
- **(Re-)binding behaviors**: The designer has to map (un, re-)binding actions on ranking change events. Rebinding actions are connected to service continuity concerns, which lead to tasks that are difficult to automate and are often left to developers. Ranking change events are categorized in the next section.

4 Framework implementation

Our contribution targets improving the OSGi Declarative Services model, which is based on early work described in [5]. This model is a service-oriented component model on top of the OSGi core framework. In this model, a component declares service requirements and provided services in a static file. A service dependency manager parses this declaration and manages the component life cycle with regard to service dependency resolution. The service dependency manager is responsible for service filter evaluation, service tracking according to this evaluation, service selection, service binding and service publication. On service arrival, service departure and service property change, service events are generated by the service registry. These

events are received by the service dependency managers that are looking for matching services. Re-composition is triggered by these events at the component level.

Some features are missing in this model to fully adapt component life cycle to internal and external context dynamism:

- Component description dynamism: The description of the component's service requirements and provided services is dynamically populated with contextual information in our architecture whereas Declarative Services only addresses static component description. Our contribution introduces writable service properties and writable service requirements at runtime.
- Enhanced service selection mechanisms: Optional filters and a ranking policy are added to the declaration of service requirements. Declarative Services only supports a unique mandatory filter.

In our extended framework, the service dependency manager enables service properties dynamic evolution, re-evaluates filters, adapts service tracking (active and passive service discovery), and adapts service selection. The dependency manager listens to service and contextual changes to dynamically bind the best subset of services. Thus, a specific view of the service registry is maintained for every service requester. Service requester bindings are dynamically changed by the dependency manager whenever a filtered unbound service gets a higher ranking than a bound one. Recomposition is then triggered by internal and external contextual events and is enhanced with complete service selection mechanisms.

4.1 Contextual service properties management

Entities attaching scores and entities evaluating the scoring function must share the same context evaluation grid. It is possible to let every service provider declare their own contextual metadata and to let service requesters evaluate it. This is naturally done in most research work [13]. However, the fact that these entities share the same context semantics implies strong coupling between service providers and service requesters. This assumption goes against the loose coupling predicate of our service-oriented architecture.

In order to maintain loose coupling, we advocate that contextual property attachment be dealt with by the service context management layer itself. In our architecture, high-level context sources are capable of identifying software components and attaching the relevant contextual properties to the service specification at runtime.

Service providers are their own context sources for some contextual information (service state, service features, estimated QoS, etc.) and rely on external context sources for other contextual information (location, tested QoS, user rating, user immersion level, etc.).

In our framework, service provider metadata is populated with contextual properties by context source components. This feature makes use of the Configuration Admin model of the OSGi specification. In this model, the service provider has to additionally publish a *Managed Service* in the service registry. This enables external components, with the appropriate permission, to add, modify, and delete the properties of the provided service. Whenever the properties change, the service object (ob-

ject representing the service provider) is directly called to make it aware of the update. The service object may internally react to this change.

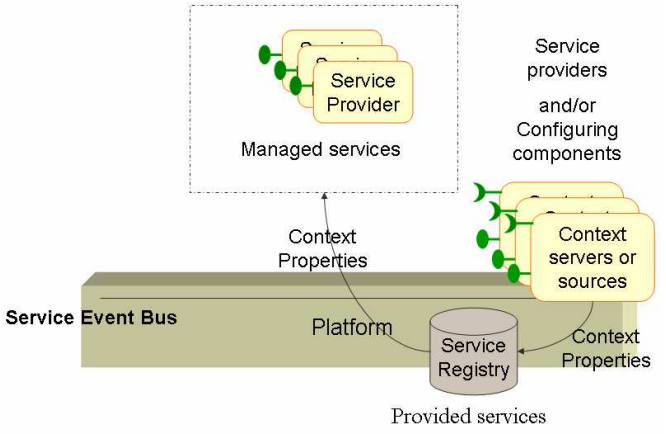


Figure 2 Context sources populate services with contextual properties

In a service-oriented model, it is natural to design context sources as service providers. Some entities are responsible for aggregating context from context sources and assigning contextual properties to the relevant managed services. For instance, a location server may infer location information from context sources and attach location properties to *managed services* representing UPnP devices on the network. Another example could be a user preferences server that adds user rating information to *managed services*.

Thanks to this model, application components need not adhere to context management service interfaces. The only interface between application components is the shared service registry, which enables service properties to be writable (see Figure 2). The *Managed Service* interface is a neutral interface enabling the components to be aware of contextual changes and to declare the ability to accept property modification.

4.2 Contextual service requirements management

Service requesters are also their own context sources for some contextual information (application state, user input through graphic interfaces, etc.) and rely on external context sources for other contextual information (user location, user activity, user preferences, desired QoS, etc.).

Service requesters follow the same pattern as service providers: High-level context sources attach contextual information to service requirement descriptions. Service requirements must then be accessible to components other than the service requester. We propose to register them in the service registry in our architecture with a simple interface which is defined below:

```
public interface ManagedRequester{
       ComponentInstance getInstance();
```

```
        void filterPartsUpdated(Filter f) throws ConfigurationException;
        addOptionalFilter(Filter f) throws ConfigurationException;
}
```

Every time a new component declared to have a context-aware behavior is instantiated, the service dependency manager associated with it registers a *Managed Requester* service for it. The *ComponentInstance* object mentioned in the interface provides the service requirements of the component. The method *filterPartsUpdated* enable context sources identifying the component to add some complementary contextual filters to the component filter. *addOptionalFilter* is a method enabling the allowed configuring components to add an optional filter to the managed service requester. The filters of the component are then writable in our architecture.

In order to show the specific interests of the requester, it is appropriate to allow placeholders to be added to the filter. These placeholders are interpreted by context sources, the configuring components in our model. For instance, an application looking for services in the room where a user called Maxandre is found would have the following filter part expression: location=$location-room{Maxandre} following the context-sensitive syntax of [4]. Location information sources would then be able to write the value of any property written this way.

Thanks to this model, the context management system is partly responsible for service binding decisions. In order to promote self-adaptability prior to manageability, components are responsible for property propagation: In our architecture, service providers and service requesters can ignore or modify the parameters which are submitted to them.

4.3 Dynamic service ranking

Scoring functions may be described in a complex language. The following examples are ranked from the easiest to the most complex. The first two functions are easily described with standard mathematical functions; the third one refers to a function attached to a specific semantic data model. It is noticeable that the last two also refer to a maximum value that is application-specific:

- Giving a score to a price, looking for a minimum, after having filtered the service with mandatory filter (&(price<100)(currency=dollar):

$$(100 - price) / 100$$

- Giving a score to the distance between a user and devices defined with geometrical parameters in a basic model, looking for a minimum:

$$(MaxDistance - (\sqrt{((x_{device}-x_{user})^2+ (y_{device}-y_{user})^2+ (z_{device}-z_{user})^2)}))/ MaxDistance$$

- Giving a score to the distance between a user and devices defined in a semantic data model, looking for a minimum:

$$distance(room_{device} , room_{user}) / MaxDistance$$

The examples show that scoring functions can be made of complex sub-functions. Moreover, it shows that evaluating the adequacy of declared service properties to

service requirements is an application-specific task. We then consider the scoring function as a method in object-oriented programming.

From a component's perspective, service ranking may occur either internally or externally depending on the circumstances. Service context management must be able to support both. However, mixing complex ranking policies coming from inside and outside of the component appears to be very complicated. Since components are self-adaptable we define the ranking policy in the component itself in the present architecture. In our framework, it is called by the service dependency manager of the component. We are currently thinking about a more generic ranking algorithm based on scores attached to service providers and correspondent utility weights populating service requirements, but big issues are raised with this approach. A generic utility function is also to be called on the service dependency manager of the component.

Two selection mechanisms are added to the Declarative Services unique mandatory filter:

- Optional filters are added to the first one. Optional filters enable further selection when many service providers are fulfilling mandatory ones. DSCL language [8] also defines mandatory and optional filters for better service selection. Optional filters may be numerous and organized from the most important to least important. This ordering enables progressive selection.

- A method name referring to a Java callback method in the component code allows developers to program a complex ranking method in the component. The ranking method is called with the remaining service references to be ranked as input arguments in order to let the component sort them in a total order. The component is then bound to the best available service (see the mentioned *sort-method* in the XML description of the *user control factory* of the application example in section 5).

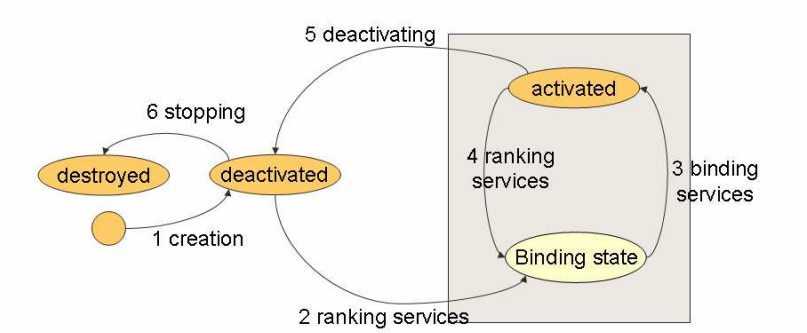
The limitations of the Declarative Services model for service selection are overcome in our model with writable filters, optional filters declaration, and the addition of a method declaration in the XML component requirements description (see sort-method attribute in XML descriptions below). This way, filters may be defined internally and externally at runtime and the sorting algorithm is expressed by the developer of the service requester with the expressiveness of the Java programming language. A hook method is also given in order for the component to warn the component manager to evaluate this ranking method again on internal events.

4.4 (Re-)binding behaviors

The component life cycle is affected by dynamic service ranking improvement. Optional filters and the ranking method are only processed on components with a service requirement that is declared to be fulfilled with a limited number of service providers (i.e., a binding cardinality of 0..x and 1..x where x belongs to $]0,\infty[$; in standard OSGi Declarative Services x may only take the value "1"). Otherwise, the component binds to all the available service providers fulfilling the mandatory filter – narrowing service selection seems useless in the case where $x=\infty$. Optional filters and the rank-

ing method are called after mandatory filter processing and before calling the binding method. It is called in 3 types of situations:

- whenever a service is registered (or modified) and fulfills the service mandatory filter,
- whenever a current bound service is modified or unregistered,
- and whenever the component requirements change (mandatory filter, optional filters, or service ranking method).



1. Creation
2. Ranking available filtered services with optional filters and ranking method
3. (Un,Re-) binding best services
4. Service ranking on following events: Bound service leaving or new service registered or requirements changed
5. Deactivation on following events: Mandatory service missing leading or stopping.
6. Stopping

5 Realization: A context-aware application

Building a context-aware instant messaging application consists of dynamically binding to the adequate input/output services according to the user location and the user activity and interacting with the other devices that have an impact on this activity. When the user sits in front of the TV, the system may direct the communications to a part of the TV screen using the Picture In Picture system. If the user is having a nap on the couch, the state of the communications is viewed through colors and brightness on a lamp emitting a low-level light. When the user enters the kitchen to drink a soda, he may listen to the communications through a text-to-speech system if no display is available. When he arrives at his desk, the communications are set in an usual instant messaging user interface on the computer. Moreover, as the instant messaging application is informed about the user context, device context and physical context through the use of a context management infrastructure, user state usually typed by the user are inferred by the system and redirected on the current output interface.

The application is implemented as components in our extended model:

- A location server and a context server are simulated on a service platform of the network. Both simulated components require and configure all the *Managed Services* and the *Managed Requesters*, especially those which are qualified with a property querying the location of known users and devices (e.g., see user control component factory description below). High-level key dynamic pieces of information are the willingness and the ability to communicate of every identified user [6].
- Some components represent physical devices (ambient lamp, computer, text-to-speech system) and are registered as delivering input/output user interfaces. These are dynamically categorized with location and contextual properties by the location server and the context server. Contextual properties are the user rating, the user willingness threshold and the necessary ability threshold.

```
<!– Ambient lamp –->
<component name="AmbientLamp">
<implementation class="device.AmbientLamp" />
   <service> <provide interface=” api.InstantMessagingUI,
                       org.osgi.service.cm.ManagedService” /> </service>
   <property name=”location” value=”” type=”string” />
</component>
```

- A proxy representing a server of the instant messaging application located on the Internet.
- A user control component factory is responsible for instantiating as many user control components as the number of identified users. Every user control component is the instant messaging client for a particular user. This component links the instant messaging application server on the Internet and the input/output devices used by the user. It keeps control of the application state, i.e., maintaining the user contextual state in the instant messaging application, maintaining the communication history and ensuring communication continuity at service re-composition time.

```
<!-- user control component factory -->
<component name="UserControlFactory">
<implementation class="pack.UserControlFactory" factory=”true”/>
   <provide interface=”org.osgi.service.component.ComponentFactory,
                       configuration.ManagedRequester” />
   <reference name="imServer"
     interface="api.InstantMessengerServer"
     cardinality="1..1" policy="dynamic"
     bind="bindServer" unbind="unbindServer" />
   <reference name="imUserInterface"
     interface="api. InstantMessagingUI"
     target="( & (location=$location-room{$user{unknown}} )
                 (willingness < willingness{$user{unknown}} )
                 (ability < ability{$user{unknown}} ))"
     optionalfilters="(sound.system=5.1)"
     cardinality="1..1" policy="dynamic"
```

```
    bind="bindUI" unbind="unbindUI"
    sort-method="sortUI"/>
</component>
```

The developer of the user control component factory defines three methods (*bindUI, unbindUI, sortUI*). *bindUI* and *unbindUI* are called respectively at binding and unbinding time. *sortUI* is called if service ranking is necessary. The service dependency manager is notified whenever new user interfaces are registered, modified or unregistered. Any time such an event occurs, it automatically updates the ranked list of filtered services. If the best available service has changed, the binding method is called and then the unbinding method is called to unbind the previous player.

As the user moves across rooms, the component instance attached to this user is notified by the location server through the use of writable filters when the user moves or begins a well-defined activity. Any time such an event occurs, service dependency tracking is reconfigured. The service ranking list is then updated and automatic rebinding behavior is triggered as mentioned before if the best service changes.

These components are packaged into bundles running on top of any OSGi platform implementation, e.g., Apache Felix. The implementation of the service dependency container (Declarative Services implementation) extends the one provided in the Apache Felix project.

6 Related work

Our model on top of the OSGi framework fits a part of the architectural approach described by Steve White et al [18]. Our autonomic elements are self-described components able to classify provided services thanks to high-level internal policies. It *enables self-assembly without requiring central planning*. However, we are thinking of a more generic approach where internal policies can be mixed with external policies, which can be added at runtime in order to externally refine statically defined incomplete behaviors.

Among the described infrastructure elements described in [18], we make heavy use of the registry provided by the OSGi framework. The service registry offers the means to publish, find and bind services. It is the interface enabling the contextualization of software composition in our architecture.

David et al. [7] describe the implementation of self-adaptive mechanisms on top of the Fractal component model. The authors define reactive adaptation policies with Event-Condition-Reaction rules that are externally attached to the components. Internal and external events trigger the rule-based reactions. Reaction code is weaved into the base component implementation at runtime thanks to powerful aspect-oriented programming features. However, the architecture description language is made at a high level in the architecture whereas the components are self-descriptive in the model we chose. This aspect and the generic service trading mechanisms at the basis of our framework are more appropriate for representing the pervasive elements of the targeted environments. It lets software composition spontaneously emerge at runtime.

Chen et al. [4] introduce a context-sensitive model for resource discovery that uses INS (Intentional Naming Service) [1]. The proposed architecture enables distributed

entities to publish and discover context attributes expressed in a simple syntax. This work could plug a complementary context management infrastructure into our design. Context sources could publish information through the use of *context-sensitive names*. Service requirements may be described with placeholders that the context management system could interpret as *context-sensitive name queries*. Context streams of Chen's architecture could then populate the service registry with contextual information.

A rich language dedicated to contextual service composition is described in [8]. It introduces a self-descriptive component model close to ours with service selection mechanisms based on attribute-value properties including mandatory and optional filters, contextual composition rules, and utility functions. Dynamic service availability and contextual service selection are clearly addressed by the work of Funk et al, nevertheless the definition of contextual behavior remains static.

The architectural design of a context-aware service discovery is described by participants in the European IST Amigo project [15]. The objective of the architecture is similar to ours: it links service discovery with a context management system and targets dynamic service composition in pervasive environments. The work carried out goes further in contextual information description in dealing with context ontology. It appears complementary to our work that targets clear component architecture in a software engineering study.

7 Conclusion

In this paper, we described a service-oriented component framework to structure the implementation of context-aware applications in pervasive environments. This framework enables dynamic service composition of pervasive entities in the network. The originality of the compositional approach relies on the contextualization made at the service registry level and on the automated decision-making mechanisms at the component level. Smart behaviors of the applications emerge from this autonomous composition of self-descriptive and self-adaptive components.

We strongly believe in a platform-centric vision where the execution environment reflects the pervasive aspects of the environment: Dynamic service availability, protocol heterogeneity, interface fragmentation and context dynamism. Every relevant entity on the network is reified on the platform and visible in the service registry. The registered services are also dynamically populated with contextual properties. In the described architecture, service requirements are declared at the component level and are also refined by dynamic contextual information with a similar model. The framework makes heavy use of the local dynamic service orientation of the component model.

Service selection mechanisms are needed in service-oriented architectures in order to overcome service ambiguity, which leads to composition unpredictability. Most service middleware defines scopes and filters to narrow unpredictability. Service ranking is another mechanism to achieve service selection. Moreover, service ranking is at the basis of self-optimization, which is one of the main design patterns in autonomic computing architectures [18].

In order to insert dynamic service ranking algorithms into service compositions, a compromise is made between the need to externally configure requirements and the need for powerful expressiveness in ranking algorithms. The algorithm is expressed in the programming language of the component and is called by an enveloping container.

In an attempt to further simplify the adoption of this architecture design, we are investigating using a POJO (Plain Old Java Object) programming paradigm. Existing implementations show that some non-functional needs are cleanly masked from the developer thanks to these techniques. This new trend is gaining momentum in the Java community, but it obviously raises other issues.

8 References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, Jeremy Lilley, "The design and implementation of an intentional naming system", Proc. 17th ACM SOSP, Kiawah Island, SC, Dec. 1999
2. André Bottaro, Johann Bourcier, Clément Escoffier, Didier Donsez, Philippe Lalanda, "A Multi-Protocol Service-Oriented Platform for Home Control Applications", Demonstration at IEEE Consumer Communications and Networking Conference (CCNC 2007), Las Vegas, 2007
3. André Bottaro, Anne Gérodolle, Philippe Lalanda, "Pervasive Spontaneous Composition", Proceedings of the First IEEE International Workshop on Service Integration in Pervasive Environments, 2006
4. Guanling Chen, David Kotz, "Context-Sensitive Resource Discovery", Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, p. 243-252, Fort Worth,Texas, March 2003
5. Humberto Cervantes, Richard S. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model", Proceedings of the International Conference on Software Engineering, May 2004
6. Jim Christensen, Jeremy Sussman, Stephen Levy, William E. Bennett, Tracee Vetting Wolf, Wendy A. Kellogg, "Too much Information", ACM Queue, Vol. 4, N°. 6, July-August 2006
7. Pierre-Charles David, Thomas Ledoux, "An Aspect-Oriented Approach for Developing Self-Adapting Fractal Components", Proceedings of the 5th International Symposium on Software Composition (SC2006), March 2006.
8. Caroline Funk, Jelena Mitic, Christoph Kuhmuench, "DSCL: A Language to support Dynamic Service Composition", 1st IEEE International Workshop on Services Integration in Pervasive Environments, June 2006, Lyon, France
9. Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Koenig, Claudia Zentner, "Building Web Services with Java", Sams Publishing, Second Edition, 2004
10. Erik Guttman, Charles Perkins, John Veizades, Michael Day, “Service Location Protocol, Version 2”, RFC 2608, June 1999

11. Ralph L. Keeney, Keshavan Nair, "Decision Analysis for the Siting of Nuclear Power Plants-The Relevance of Multiattribute Utility Theory", Proceedings of the IEEE, vol. 63, no. 3, march 1975

12. Jeffrey O.Kephart, "Research challenges of autonomic computing", Proceedings of the 27th international conference on Software engineering, ICSE 2005, St. Louis, MO, USA, 2005

13. Steffen Lamparter, Anupriya Ankolekar, Daniel Oberle, Rudi Studer, Christof Weinhardt: "A Policy Framework for Trading Configurable Goods and Services in Open Electronic Markets", Proceedings of the 8th Int. Conf. on Electronic Commerce (ICEC'06), Fredericton, New Brunswick, Canada, August 2006.

14. OSGi Alliance, "OSGi R4 Core Specification and compendium", October 2005

15. Pravin Pawar, Andrew Tomakoff, "Ontology-based Context-Aware Service Discovery for Pervasive Environments", Proceedings of the First IEEE International Workshop on Service Integration in Pervasive Environments, 2006

16. Fano Ramparany, Jérôme Euzenat, Tom Broens, Jérôme Pierson, André Bottaro, Remco Poortinga, "Context Management and Semantic Modeling for Ambient Intelligence", Proceedings of the First Workshop on Future Research Challenges for Software and Services (FRCSS), 2006

17. UPnP Forum, "UPnP AV Architecture v1.0", June 22, 2002

18. Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart, "An architectural approach to autonomic computing", Proceedings. International Conference on Autonomic Computing, May 2004.

19. Feng Zhu, Matt W. Mutka, and Lionel M. Ni, "Service Discovery in Pervasive Computing Environments," IEEE Pervasive Computing, vol. 4, pp. 81-90, 2005