Ontology-Directed Generation of Frameworks For Pervasive Service Development

Charles Consel, Wilfried Jouve, Julien Lancia, Nicolas Palix

Ontology-Directed Generation of Frameworks For Pervasive Service Development

C. Consel[1], W. Jouve[1], J. Lancia[2], and N. Palix[1]
[1]*INRIA / LaBRI, Bordeaux, France,* [2]*Thales / LaBRI, Bordeaux, France*
{consel, jouve, lancia, palix}@labri.fr

Abstract

Pervasive computing applications are tedious to develop because they combine a number of problems ranging from device heterogeneity, to middleware constraints, to lack of programming support. In this paper, we present an approach to integrating the ontological description of a pervasive computing environment into a programming language, namely Java. From this ontological description of a pervasive computing environment, a framework is automatically generated. It provides the developer with dedicated programming support to manage, discover and invoke services. Besides, it performs a number of verifications both at compile and run time, ensuring the robustness of applications.

1. Introduction

Pervasive Computing relies on an environment filled with devices and communications with which users interact. A key enabler to make this interaction possible is services that exploit the functionalities of the pervasive computing environment. These services must cope with a wide variety of entities and support a range of interactions while abstracting over entity details to prevent hardware dependencies as much as possible. Furthermore, services should cope with an environment that is highly dynamic. In fact, developing applications for a pervasive computing environment is a major challenge combining a number of issues including (1) describing, organizing and using environment entities, (2) designing and developing applications (3) ensuring the robustness of the resulting pervasive computing system.

Middleware-based approaches (*e.g.*, [1] [2]) have been developed to address a number of key features like mobility, service discovery and distributed applications. Their key benefit has been to propose a unique platform that offers as many generic features and mechanisms as possible to cover the needs of application developers. The limitation to these middleware-based approaches is that they do not necessarily match the constant flow of new devices and new application requirements that are inherent to a rapidly emerging area. Also, because of the generic nature of a middleware, it tends to act as some kind of interpreter, processing the computations of an application at run time. Yet, some processing, especially verifications (*e.g.*, for service invocations), could be done at either compile or deployment time, drastically improving the application reliability. Finally, a middleware-approach does not provide the developer with a programming model. Consequently, the developer is still left producing glue code to bridge the gap between the middleware and its application domain. A first step toward bridging this gap is proposed by Olympus [2]. This approach enables ontological descriptions of entities to be integrated into the development of an application. A middleware based on Gaia resolves these descriptions into actual entities depending on various aspects such as resource availability and developer-supplied constraints. Also, Olympus provides developers with a set of high-level functions to perform common operations like start and stop a component.

This paper proposes to push the Olympus approach further by integrating the ontological modeling of a pervasive computing environment into a programming language, namely Java. To do so, we introduce two syntactic constructs to Java that permit ontological descriptions of entities to be defined. One construct defines an abstract service, which abstracts over variations of a category of entities. An abstract service is defined with respect to an ontological hierarchy based on service inheritance. An abstract service consists of semantic properties, characterizing variations of entities, and interaction modes, defining ways in which it can interact with other services. We introduce another syntactic construct that enables a concrete service to be implemented; it must be in conformance with an abstract service. A concrete

service can compose other services. Because the modeling of a pervasive computing environment is integrated into Java, we can provide the developer with dedicated programming support for managing, discovering and invoking services. This programming support takes the form of a framework that is automatically generated with respect to an ontological description of a pervasive computing environment. As well, various verifications are performed at compile time whenever possible; otherwise code is generated in the framework to perform verifications that depend on run time values.

The rest of this paper is organized as follows: Section 2 presents the ontological description of a pervasive computing environment and associated abstract services. Section 3 describes the pervasive service development within the automatically generated framework. Section 4 details benefits from the framework generation in terms of programming support. Section 5 reviews some related work and Section 6 concludes.

2. Abstracting Pervasive Environments

Application domains of pervasive systems can widely vary; it can range from building surveillance to elderly health care. Our approach covers this great diversity of domains by giving the tools to describe and use heterogeneous entities present in these domains. Specifically, entities may be external to our framework; they correspond to either devices or software components (*e.g.*, databases and Web services). Also, entities may be internal to our framework; they are user-defined software components. They allow developers to coordinate other entities. All these kinds of entities, whether internal or external, are covered by the notion of a service; this notion provides a uniform view of these heterogeneous building blocks.

2.1. Abstract services

To abstract over the variations of a type of services, we introduce the notion of abstract services. The scope of an abstract service is specified by three key aspects: its parent abstract service, the semantic properties characterizing its variations, and the modes of interaction it supports. An abstract service declaration is displayed in Figure 1. It defines a class of services dedicated to measuring luminosity. Let us now examine each aspect of an abstract service declaration.

```
1. AbstractService LightSensor extends
                              MeasurementSensor{
2.  constrained unit {Lux, Phot, Foot-candle};
3.  Command Luminosity getLuminosity(void);
4.  EventOutput {Luminosity};
5.}
```

Figure 1. The LightSensor abstract service

2.1.1. Abstract service Inheritance. To capture all heterogeneous and domain-specific services in a consistent way, our approach allows an abstract service to be defined as an enrichment of another service. As an example, in Figure 1, the abstract service for light sensors is defined as an enrichment of a measurement sensor. This relationship is expressed using the `extends` clause of the declaration. As such it inherits a semantic property defining the measurement unit. As well, it may inherit a requirement for providing interaction mode operations. Abstract service inheritance is further described below as interaction modes and semantic properties are introduced.

2.1.2. Interaction modes. Regardless of the application domain, developing pervasive systems critically relies on interacting with services. Most existing approaches [2] consider that this interaction is achieved using some form of procedure invocation or event mechanism. Procedure invocation typically addresses the need to control a service. We call this kind of service interaction the command mode. An example of a command mode is shown in Line 3 of Figure 1. This operation is named `getLuminosity`; it takes no argument and returns the current luminosity. Besides the command mode, an abstract service may also offer the event mode, either as a subscriber or as a publisher, using the publish/subscribe mechanism. In our light sensor example (Figure 1), the abstract service is defined as a publisher of light measurements, as shown in Line 4. Events are characterized by their direction (input or output) and the class defining the data they publish or receive. Another important mode of interaction consists of interacting with a service by exchanging a stream of data. This interaction mode is called the session mode because it requires the consumer and the producer of the data stream to set up a session to communicate. Besides multimedia, stream-oriented services may also produce a stream of arbitrary data. We illustrate the session mode by extending our light sensor abstract service, as shown in Figure 2. This extended form of light sensors produces a stream of measurements. A service can either be invited or

invite other services, as indicated by the Input/Output keyword.

```
1.AbstractService ExtendedLightSensor extends
                                    LightSensor{
2.   SessionInput {Luminosity};
3.}
```

Figure 2. The ExtendedLightSensor abstract service

2.1.3. Semantic properties. Not only is a range of concrete services defined by its supported interaction modes, but it is also defined by the semantic properties that may hold. These properties are included in the ontological description of an abstract service and further characterize the target range of concrete services. Examples of properties include measurement unit, priority or location, as displayed in Figure 3. Semantic properties can either be inherited from a parent abstract service or introduced by the abstract service being defined. A property can be assigned a value or constrained. In our example, Line 2 of Figure 1 defines a constraint on the unit property inherited from the `MeasurementSensor` abstract service. The key feature of our approach is to provide the developer with a typed interface to semantic properties. Specifically, querying the unit property of light sensors requires a well-typed value, belonging to the enumeration Lux, Phot and Foot-candle. This strategy enables a range of errors to be detected at compile time.

2.2. An ontological hierarchy of abstract services

To further explain our approach, we now examine fragments of an ontological hierarchy defining abstract services developed in the context of building manager applications; this hierarchy is displayed in Figure 3. The ontological hierarchy uses abstract service inheritance to propagate semantic properties and interaction modes of one abstract service to all its child nodes (i.e., classes). As a result, all abstract services define the shutdown command and the type property inherited from the root class (i.e., the Service abstract service). This inheritance allows an abstract service to abstract over the variations of its sub-classes. A group of heterogeneous abstract services can thus be considered as one homogeneous entity. For instance, every abstract service extending the `Light` abstract service (*e.g.*, `DimmerLight`) can be considered as a `Light` abstract service, hiding their extra interaction modes. The abstraction given by the service ontology allows developers to use services that expose the level of abstraction that is appropriate for their needs. This inheritance allows abstract services to incrementally reveal new interaction modes, as illustrated by our light sensor example extended with the session mode (see Figure 2).

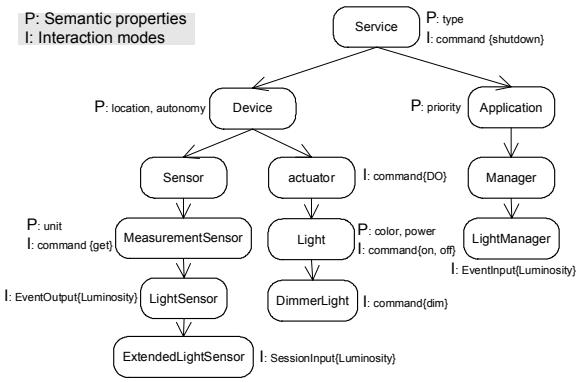

Figure 3: An example of ontological hierarchy

3. Application-Specific Framework

Once an application domain has been analyzed and organized in the form of ontological descriptions of abstract services, the application logic can be developed. Let us now describe the application-specific framework generated for the application developer, from ontological descriptions.

3.1. Concrete Services

Abstract services can be seen as a specification to which concrete services need to conform. In our programming model, a concrete service is defined via the `ConcreteService` construct. This construct defines a concrete service that must conform to the abstract service included in the `from` clause. Let us start by examining external concrete services. To do so, a concrete service needs to implement each functionality of the abstract service in terms of device driver operations, in case of a hardware entity, or in terms of object method invocations, in case of a software component. To illustrate this kind of concrete services, consider the example displayed in Figure 4. This `Light` wrapper service must implement all the `Light` operations (*e.g.*, the `on` and `off` operations). Besides, the inheritance requires us to implement the `DO` operation from the `actuator` service class and the `shutdown` operation from the `service` service class.

```
ConcreteService MyLight from Light {
   public MyLight(String uri) { (…) }
   void on() {(…)}
   void off(){(…)}
   void DO(){(…)}
   void shutdown(){(…)}
}
```

Figure 4: Definition of a concrete service

3.2. Service discovery

The heterogeneity and dynamicity of services in pervasive environments must be supported by an adapted service discovery mechanism. The goal of our approach is to allow the application logic to be decoupled from a concrete pervasive computing environment. At the basis of service management is the ontological hierarchy of services: it is used both to register concrete services and select specific concrete services.

3.2.1. Service registration. When a concrete service declaration is executed, it is automatically registered in the service hierarchy at the node corresponding to its abstract service. Consequently, any registered service behaves like a service of its class as well as a service of its parent classes; as such it is also registered as a concrete service of its parent classes. For example, a concrete `Light` service is also registered as an actuator because, as a child node of the actuator, it supports the `DO` operation.

3.2.2. Browsing in the pervasive computing environment. We propose to use the ontological descriptions to partition the environment. To choose a node in the hierarchy, the developer must identify an abstract service. The higher the abstract service node in the ontological hierarchy, the more re-targetable the application logic will be. This strategy makes it possible to maximize the number of services that belongs to the target partition of the pervasive computing environment. This situation demonstrates the key importance of the partitioning of services represented by the ontological hierarchy. This notion is illustrated by the fragment of code browsing shown in Figure 5 that selects the `Light` partition in Line 1. The semantic properties of an abstract service are used to further refine a partition of the pervasive computing environment. Each property covers a specific dimension. For example, once the `Light` partition is chosen, it can be further refined by selecting lights with respect to their location. Besides exact matching, our framework provides the developer with various other matching strategies, including value ranges, enumeration of values and existence of properties. Once the selection criteria have been set, the application can invoke a framework operation to collect the corresponding concrete services available in the pervasive computing environment. This collection of concrete services is illustrated in Line 3 of Figure 5.

```
1. LightPart part = Light.getPartition();
2. part.location.setValue(mylocation);
3. LinkedList<Light> lights = part.getServices()
```

Figure 5: Browsing a Light environment partition

3.3. Service composition

The development of an application for a pervasive computing environment critically relies on the composition of services. We now examine how this aspect is tightly integrated into our proposed ontology-based approach.

3.3.1. Composition with respect to the ontology of services. Prior to developing the logic of a new service, the programmer, or a project architect, needs to study how to place it into the ontological hierarchy of services. To do so, he needs to determine whether this new service falls into an existing category of services defined by an abstract service. If so, the developer leaves the hierarchy of services unchanged and simply implements a new concrete service. The key benefit of this strategy is service re-use.

In other cases the logic to be developed refers to a new category of services. Typically, this situation occurs for services that are inherent to the application domain. For example, managing a building requires the definition of various managers operating building resources. To address this category of situations, we define the `LightManager` abstract service, as shown in Figure 6. Its aim is to turn on/off the lights in the building hallways depending on the outside luminosity. To do so it extends the `Manager` abstract service and is declared as a receiver of `Luminosity` event.

```
AbstractService LightManager extends Manager {
   EventInput {Luminosity};
}
```

Figure 6: The LightManager abstract service

3.3.2. Composition with respect to the service logic. Service logic typically coordinates a number of other services. The example in Figure 7 illustrates service composition with the `MyLightManager`

concrete service. The constructor is first defined: it implements the discovery process of the `LightSensor` service and subscribes to a `Luminosity` event. Conforming to the abstract service, `MyLightManager` defines a receive operation to handle `Luminosity` events.

```
ConcreteService MyLightManager from LightManager
{
 LightSensor myLightSensor;
(…)
 MyLightManager(Uri uri) {
   super(uri);
   priority.setValue(Priority.LOW);

   myLightSensor = sensorPart.getService();
   myLightSensor.subscribe(this);
 }
 void receive(LuminosityEvent e){
   (…)
 }
 (…)
}
```

Figure 7: The MyLightManager concrete service

3.4. Service verification

An important contribution of our approach is to perform verifications at every stage of a service lifecycle. To do so, verifications are performed at both compile and run time.

When an abstract service is created, verifications are performed on the consistency of the ontological hierarchy of services, the type signature of the interaction mode operations, and the constraints on semantic properties. When a concrete service is created, verifications are similar to the ones on abstract services. In addition, the conformance of the concrete service with its abstract service is checked.

Every step of the service discovery performs verifications: the selection of the abstract service in the service hierarchy and the refined selection using the semantic properties. In particular, semantic properties are strongly typed unlike other approaches based on strings (*e.g.*, Olympus [2]). Importantly, the verifications of service discovery are performed at compile time.

The interaction modes of a concrete service are strongly typed with respect to the signatures defined by its abstract service. This applies to command operations, events and sessions. Like service discovery, service invocation is verified at prior to run time.

4. Framework Generation

Abstract services defined in the ontological hierarchy are used to generate programming support for managing, discovering and invoking a service. Our current prototype uses Java as the implementation language.

4.1. Programming support

Abstract service declarations produce abstract classes in Java. Each semantic property is mapped into a Java field in the abstract class. Each interaction mode of an abstract service generates a Java interface. These abstract classes implement methods to support browsing in the corresponding partition of the pervasive computing environment. Specifically, a method is generated to select a node in the ontological hierarchy of services. When invoked, this method produces a partition containing the concrete services corresponding to the selected node (see Line 1 of Figure 5). Furthermore, methods are generated to manipulate each semantic property of the abstract service, according to its nature (see Line 2 of Figure 5). These methods are used by the developer to refine the set of target concrete services. Two methods are also produced to complete the discovery process: one to select a unique concrete service and one to get all the concrete services. When a concrete service is selected, it is not referenced directly. Instead, a reference to proxy is returned. This proxy is an implementation of the abstract Java class associated with the selected abstract service. This strategy has two major benefits: actual concrete services may be spread over a distributed system; and, concrete services can be hot swapped.

4.2. Assessment

Ontology descriptions are specified in OWL [5] and created with Protégé [6]. The framework generator was developed in Java and is based on the JENA API. The prototype implementation uses the Java Remote Method Invocation to invoke service operations. We are conducting experimental studies to measure how much code is generated from ontologies in various application domains.

5. Related Work

As introduced earlier, two research projects are most related to our work, namely, Gaia and Olympus. Gaia is a distributed middleware infrastructure that coordinates software entities and heterogeneous

networked devices contained in a physical space [1]. Active Spaces enable user mobility and application portability. Olympus enhances Gaia by proposing a high-level programming model [2]. Bodhuin *et al.* use an entity description graph to abstract the physical world [3]. This graph contains a host of hierarchical tuples to define a device and its functionalities. Each class of devices inherits functionalities from its parent's interface. These functionalities are limited to two interaction modes: command and event. Unlike our approach, the session mode is not addressed. Also, entity descriptions are not integrated in a programming model and entity composition is not examined. Kalyanpur *et al.* propose an approach to generating Java APIs from OWL ontologies. It attempts to map OWL ontology semantics into Java [4]. This approach is quite appropriate for distributed architectures (*e.g.,* multi-agent environment) in that it allows agents to share a common view. However, by targeting no particular application domain, this approach is too generic to leverage the power of mapping ontologies to Java in the framework of pervasive computing.

6. Conclusion

We have introduced an approach to integrating the ontological description of a pervasive computing environment into a programming language. Entities in a pervasive computing environment are uniformly captured by the notion of a service, whose creation is supported by syntactic constructs. Abstract services form a hierarchical ontology that represents a design framework for developers. An abstract service defines semantic properties that characterize variations of concrete services. Also, it specifies the supported interaction modes. Our proposed interaction modes cover a wide range of situations and, in particular, enable stream-based services to be handled. Verifications on applications are performed both at compile and run time. Finally, a framework is automatically generated from an ontological description of a pervasive computing environment, providing the programmer with environment-specific operations.

7. References

[1] Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. "Gaia: a middleware platform for active spaces". SIGMOBILE Mob. Comput. Commun. Rev. 6, 4 (Oct. 2002), 65-67.

[2] Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R. H., and Mickunas, M. D. "Olympus: A High-Level Programming Model for Pervasive Computing Environments". In Proceedings of the Third IEEE international Conference on Pervasive Computing and Communications (March 08 - 12, 2005). PERCOM. IEEE Computer Society, Washington, DC, 7-16.

[3] Bodhuin, T., Canfora, G., Preziosi, R., and Tortorella, M. 2006. "Hiding complexity and heterogeneity of the physical world in smart living environments". In Proceedings of the 2006 ACM Symposium on Applied Computing (Dijon, France, April 23 - 27, 2006). SAC '06. ACM Press, New York, NY, 1921-1927.

[4] A.D. Kalyanpur, et al., "Automatic Mapping of OWL Ontologies into Java", Proc. 16th Int'l Conf. Software Eng. and Knowledge Eng. (SEKE 2004), 2004, 98–103.

[5] M. Dean and G. Schreiber. "OWL Web Ontology Language Reference". W3C Recommendation, February 2004.

[6] Protégé, http://protege.stanford.edu/