A Domain-Specific IDL and its Compiler for Pervasive Computing Applications

Wilfried Jouve, Julien Lancia, Nicolas Palix, Charles Consel, Julia Lawall

A Domain-Specific IDL and its Compiler for Pervasive Computing Applications

Wilfried Jouve — Julien Lancia — Nicolas Palix — Charles Consel — Julia Lawall

N° 6213

Juin 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6213--FR+ENG

A Domain-Specific IDL and its Compiler for Pervasive Computing Applications

Wilfried Jouve [*], Julien Lancia[†] [*] , Nicolas Palix [*] , Charles Consel [*] , Julia Lawall[‡]

Thème COM — Systèmes communicants
Projet Phoenix

Rapport de recherche n° 6213 — Juin 2007 — 24 pages

Abstract: Pervasive computing environments introduce new challenges for application development, due to the heterogeneity of the devices involved. In practice, pervasive computing applications rely on general-purpose middleware to manage this heterogeneity, but this approach does not provide programming support and verifications specific to the pervasive computing environment.

In this paper, we present a domain-specific IDL and its compiler, dedicated to the development of pervasive computing applications. Our IDL is based on that of CORBA and provides declarative support for concisely characterizing a pervasive computing environment. This description is (1) to be used by programmers as a high-level reference to develop applications that coordinate entities of the target environment and (2) to be passed to a compiler that generates a framework dedicated to the target environment. This process enables verifications to be performed prior to runtime on both the declared environment and a given application. Furthermore, customized operations are automatically generated to support the development of pervasive computing activities, such as service discovery and session negotiation for stream-oriented devices.

We have implemented a framework generator and have used it to generate frameworks targeting pervasive computing areas such as building surveillance, advanced telecommunications and home automation.

Key-words: middleware, pervasive environments, programming framework, CORBA

[*] Equipe INRIA Phoenix, Bordeaux, France
[†] Thales, Bordeaux, France
[‡] DIKU, University of Copenhagen, Copenhagen, Denmark

Spécialisation d'IDL et de son compilateur pour le développement d'applications ubiquitaires

Résumé : Les environnements pervasifs ou ubiquitaires sont constitués de nombreuses entités hétérogènes qui complexifient le développement d'applications. Dans la pratique, cette hétérogenéité est gérée par des intergiciels génériques qui ne fournissent pas le support et les vérifications requis par les environnements ubiquitaires.

Dans ce papier, nous présentons un IDL (PerIDL) et son compilateur (PerGen) dédiés au développement d'applications ubiquitaires. Notre IDL est basé sur celui de CORBA et fournit du support déclaratif pour caractériser de façon consise un environnement ubiquitaire. Cette description est (1) utilisée par les programmeurs comme une référence haut-niveau pour développer des applications qui coordonnent les entités de l'environnement cible et (2) est utilisée par un compilateur qui génére un cadre de programmation dédié à l'environnement cible. Ainsi, des vérifications peuvent être effectuées à la compilation sur l'environnement cible et une application donnée. De plus, des opérations dédiées sont automatiquement générées pour supporter l'utilisation de mécanismes spécifiques aux environnements ubiquitaires ciblés, tels que la découverte de services et la négotiation de session de flux de données.

Nous avons implémenté un générateur de cadre d'applications et l'avons utilisé pour générer des cadres d'applications ciblant des environnements ubiquitaires tels que la surveillance d'un immeuble, la télécommunication et la domotique.

Mots-clés : intergiciel, environnements ubiquitaires, cadre de programmation, CORBA

1 Introduction

Pervasive computing applications such as medical monitoring, assisted living, and home automation operate in a distributed networked environment, consisting of heterogeneous computational entities. These applications must collect, coordinate, process, transmit, and react to information emanating from entities distributed over a network. For example, a building surveillance application must coordinate motion sensors to detect intrusion, control webcams to record video streams, and send alarm messages to activate guards.

Middleware is a key enabling technology used in developing pervasive computing applications [7, 9, 10]. It abstracts over a number of implementation issues, enabling distributed, heterogeneous objects to interoperate. Most middleware approaches are general purpose and highly dynamic, but provides some level of customization via a compiler of an Interface Definition Language (IDL) (*e.g.,* [8]). The IDL provides the developers with a declarative approach to defining the API of a component, including its attributes, its parent classes, the relevant typed events and methods, and distributed properties. Still, a general-purpose IDL does not permit domain-specific properties to be checked, or domain-specific programming support to be generated.

Let us examine the domain-specific needs of pervasive computing applications.

Domain-specific interaction modes. From an application viewpoint, pervasive computing entities, whether hardware or software, are characterized by the interaction mechanisms they provide. Commonly available interaction mechanisms are the Remote Procedure Call (RPC) and events. RPC is typically used for controlling entities (*e.g.,* an On/Off command). We refer to this interaction mode as *command mode.* Event notifications enable applications to react to changing conditions of the environment (*e.g.,* fire alarm events). We refer to this interaction as *event mode.* Existing IDLs allow command and event modes to be typed to improve safety.

Yet, pervasive computing entities increasingly require interactions based on data streams. Common examples of such entities include audio and video devices, but this interaction mode encompasses any kind of sensor producing a stream of measurements. To address stream-oriented entities, a session approach is required to negotiate the parameters of a stream (*e.g.,* data format and sampling rates) before creating it. We call this kind of interaction mode a *session mode.* Of course, this mode can be implemented in terms of RPCs, but this approach entails developing a lot of administrative code that is essentially boilerplate.

Domain-specific service discovery. To cope with the dynamicity of pervasive computing environments, applications rely on service discovery. Service discovery allows applications to query the environment and dynamically bind functionalities to entities. Commonly, queries rely on general-purpose mechanisms that do not assume any explicit organization of the entities. Consequently, discovering entities is only as disciplined and predictable as the programmers registering these entities. In practice, however, there are often hierarchical relationships between conceptual entities, such as the relationship between a generic webcam and webcams providing specific features. Exposing these relationships to the service discovery process would allow specific support for different views on a concrete entity to be provided. Such support is particularly needed when a class of entities (*e.g.,* webcams) offers multiple interaction modes, requiring an abstraction layer to shield the application code from these variations.

Domain-specific verifications. The general-purpose nature of middleware implies that few static verifications are possible. For example, when using a generic service discovery mechanism that is not aware of the possible classes of services (*i.e.,* entities) available in a given environment, requests for services must be expressed using correspondingly generic mechanisms, such as strings, that are error prone and not checked until run time. Yet, the safety of pervasive computing applications is critical to most target environments, and the correct deployment of such applications would be greatly facilitated if more safety properties could be ensured.

Domain-specific programming support. Experience shows that pervasive computing development consists of common program patterns that are only partially supported by an IDL compiler, requiring the programmer to provide code to further tailor the middleware to this domain. Examples include session-oriented entities and service management. Common program patterns also occur when considering a specific pervasive computing *area* (*e.g.,* building surveillance and assisted living). In this case, the programmer lacks declarative means to express these commonalities, limiting the sharing of knowledge to code.

Our approach

We propose to extend a CORBA-based IDL with pervasive computing concepts. The resulting pervasive-computing-specific IDL, named *PerIDL*, enables an expert to define an environment for a given pervasive computing area in terms of descriptions of pervasive computing services, organized hierarchically. A service description represents a class of concrete components that can be deployed at run time. It comprises a collection of attributes that express various domain-specific concepts such as stream-negotiation parameters and service-discovery parameters. Programmers use the IDL-based description of a pervasive computing environment to develop a variety of applications in a given area.

Besides being a repository of knowledge, a PerIDL description is passed to our compiler, named *PerGen*, to automatically generate a software framework that provides programming support customized with respect to the target pervasive computing area. This customized framework guides programmers when developing application code. It supplies environment-specific APIs to perform operations. For example, service registration and discovery operations are generated with respect to the hierarchy of entities defined by the PerIDL declarations. Coupled with an Integrated Development Environment (IDE), such as Eclipse [6], it enables generation of class skeletons (*i.e.,* class stubs), customized with respect to the target environment, raising the abstraction level of the development effort.

Finally, the pervasive computing concepts encapsulated in the PerIDL declarations make verifications possible statically (*i.e.,* at framework-generation time) and dynamically. At framework-generation time, a PerIDL description of a pervasive computing environment is analyzed to check the consistency of the the service dependencies induced by its interactions. For example, if a service class is declared as consuming a fire alarm event, the PerIDL description of the environment must include a service class supplying such an event. At run-time, each deployed component must correspond to a PerIDL description. The consistency of the pervasive computing environment is furthermore continuously checked. For example, an exception is raised in a component consuming a fire alarm event if the expected event supplier is not present at deployment time, or later becomes unavailable. Consistency checking is essential to making pervasive computing applications reliable. PerIDL and PerGen contribute to the reliability of pervasive computing applications by making this checking systematic.

Currently, we are working with a telecommunications company that is studying the use of an ADSL modem as a gateway to enable a homeowner to observe his home remotely. The first step of this project was to characterize the set of entities (*e.g.,* devices and servers) that would be required to carry out a number of existing and future scenarios. This phase led to an environment description written in PerIDL, enabling this knowledge to be shared among area experts and application programmers. PerIDL was then passed to PerGen to generate a framework customized with respect to home automation. Various scenarios are now being developed to validate the approach.

Contributions

The contributions of this paper are as follows.

- We introduce a pervasive-computing-specific IDL based on that of CORBA, named PerIDL. It allows creating a hierarchical description of a pervasive computing environment, raising the level of knowledge that can be shared among area experts and application programmers.
- We show that the PerIDL description of an environment can be used to carry out domain-specific verifications, permitting consistency checking to be performed at compile time as well as at run time.
- We present PerGen, a compiler for PerIDL declarations. PerGen takes a description of a pervasive environment and generates a customized programming framework, assisting the programmer by leveraging IDE capabilities.
- PerIDL has been used to describe various pervasive computing environments, including building surveillance and home automation. These environment descriptions have been used by PerGen to generate customized frameworks, enabling applications to be developed.

- We show that there is a need to generate programming frameworks customized with respect to pervasive computing areas that go beyond the code generated by existing IDL compilers.

Terminology. A pervasive computing service class (*service class* for short) is a functional unit that captures a class of concrete entities (hardware or software); it is defined by a PerIDL declaration. Services (or concrete entities) correspond to implementations (*i.e.,* instances) of a PerIDL-declared service class.

2 Defining a Pervasive Computing Area

A PerIDL specification is a taxonomy of entities relevant to a given area of pervasive computing (*e.g.,* building surveillance), enriched by attributes that further characterize possible variations. To create such a specification, an analysis is needed of the area. This analysis is performed once, by an *area expert*, who creates a PerIDL specification based on the result. This specification will then be used by application developers to guide and structure service implementations. In this section, we describe the process of constructing a PerIDL specification for a pervasive computing area.

2.1 Analyzing a pervasive computing area

The goal of the analysis of a pervasive computing area is to identify the basic building blocks of the area and their range of possible variations. The basic building blocks are the relevant hardware and software entities. For example, in the surveillance area, basic building blocks include hardware entities such as various kinds of sensors (*e.g.,* motion detectors, intrusion beams, and webcams) and actuators (*e.g.,* lights, intrusion sirens, and an SMS server for contacting building personnel), as well as software entities such as a database that stores information about building opening hours and employee access privileges. Variations occur in both the functionalities provided by these building blocks and their non-functional properties. For example, there are simple webcams, zooming webcams and webcams with a built-in motion detector. These may provide some functionalities in common, but typically entail significant differences in their implementation logic to manage their specific features. Other kinds of variation include *e.g.*, the location of a motion detector or the codec currently being used to encode the stream of images emitted by a webcam. These non-functional variations describe an entity's current state.

The basic building blocks are not in general sufficient to describe all of the functionalities needed in an area; programmers also need to develop composite building blocks that combine and coordinate the behaviors of more basic building blocks. For example, in the context of surveillance, a service class may be introduced to coordinate a set of sensors, abstracting over implementations of coordination strategies.

2.2 Creating the PerIDL specification of an area

The area expert next uses the results of the analysis to create a PerIDL specification consisting of declarations that model classes of entities relevant to the target area. Each service class is characterized by high-level information, shielding application code from implementation details. This information comprises the *semantic properties* characterizing the service class, the *interaction modes* provided by the service class, and the *hierarchical relationships* between service classes.

Semantic properties

The semantic properties of a service class describe the scope of variation of entities it corresponds to. These are derived from the nonfunctional variations identified during the analysis phase, *e.g.*, the location of a motion detector or the codec currently being used by a webcam. Semantic properties are represented by name-value pairs as in CORBA. PerIDL goes one step further in introducing a set of properties dedicated to the domain of pervasive computing. The resulting domain-specific vocabulary provides predictable matching between service providers and service consumers.

In our approach, when an entity is deployed, it must be associated with a service class. To make this association, the deployment code must initialize the values of each of the semantic properties. Programmers that want to interact with services can then write service discovery logic in terms of the PerIDL description of the target environment, by selecting a service class and refining it with the desired values of the semantic properties.

Interaction modes

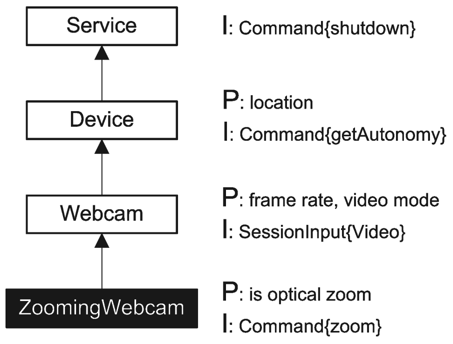
The interaction modes associated with a service class describe how the service produces or consumes data. Service programmers must implement each specified interaction mode. Our approach supports three interaction modes: command, event and session.

Command. The command interaction mode corresponds to a RPC, enabling networked services to interact with each other. It is typically used to control a device. For example, a webcam that can zoom in on a particular region may have a `zoom` command, which takes a region (*e.g.,* the top left quadrant of the captured images) as an argument. Figure 1 shows the PerIDL specification of such a webcam, which includes a `Command{zoom}` declaration.[1]

Event. The event interaction mode is analogous to the push-oriented event mechanism offered by most middleware approaches. It allows services to be aware of and react to conditions in their environment.

Consider a service class for webcams featuring motion detection. This class of devices could be modeled as a publisher of a motion event, as shown in Figure 2 by the declaration

[1]Note to the reviewer: for lack of space, we have kept the PerIDL declarations short. Complete specifications of pervasive computing environments are available from the authors upon request.

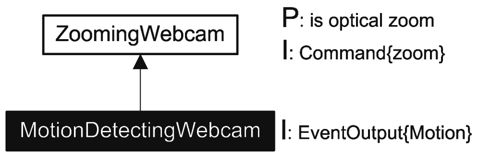


(a) Graphical representation
(P:Semantic properties, I:Interaction modes)

```
interface Webcam : Device {

  attribute Framerate frame_rate;
  attribute VideoMode video_mode;
  SessionInput(Video video);
};

interface ZoomingWebcam : Webcam {

  attribute boolean is_optical_zoom;
  void zoom(in Region region);
};
```

(b) Textual representation

Figure 1: The zooming webcam interface definition



(a) Graphical representation

```
interface MotionDetectingWebcam :
                    ZoomingWebcam {

  EventOutput(Motion motion);
};
```

(b) Textual representation

Figure 2: The motion detecting webcam interface definition

`EventOutput{Motion}`. An event declaration indicates both the event type (*e.g.,* using the `Motion` IDL attribute) and the event direction, whether incoming or outgoing with respect to the entity, as represented by the `EventInput` and `EventOutput` keywords.

Session. The session interaction mode goes beyond existing IDLs by natively supporting entities that exchange a stream of data. The main difficulty in managing a stream is the variety of possible data formats. Our approach consists of two phases: (1) a setup phase, in which a consumer and a producer agree on data stream parameters; then, (2) a session creation phase, in which a session of data exchange is created and configured with respect to the negotiated parameters. This process is inspired by the Session Initiation Protocol (SIP [11]) which is a signaling protocol widely used in telephony. We have extended it to go beyond this domain.

Consider the class of simple webcams (*i.e.,* the `Webcam` service class), defined as an ancestor of the `ZoomingWebcam` service class (Figure 1). The declaration `SessionInput{Video}` indicates that this class of devices can accept requests to create a stream. This declaration defines the type of the stream items (here `Video`) and the session initiation capabilities – input for *invitee* and output for *initiator*. Furthermore, it specifies the negotiation parameters (not shown). For video, these parameters may include the video mode (codec and resolution) and frame rate. Deploying a concrete webcam service requires instantiating the negotiation parameters according to the webcam's capabilities.

Hierarchical relationships

The PerIDL description of a pervasive computing environment is structured as a hierarchy, as illustrated by the specification of the building surveillance domain, shown in Figure 3. For simplicity, we omit some service classes. Starting at the root node, this description breaks down the set of possible domain entities into increasingly specific classes. Each successive entry adds new semantic properties and interaction modes that are specific to the service class that it represents. A service class furthermore inherits all of the semantic properties and interaction modes of its ancestors.

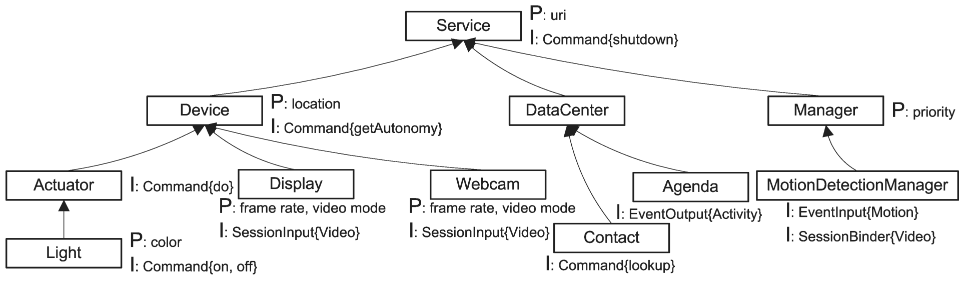


Figure 3: Selected nodes in the PerIDL specification of the building surveillance domain

In our approach, inheritance not only enables reuse and makes explicit the relationships between entities, but it also plays a decisive role in service discovery. Conceptually, a user who wants access to a service class designates the corresponding node in the service hierarchy, and receives all of the services corresponding to the service classes contained in the subtree. This strategy implies that the hierarchical relationships between services have a dual impact on the code that implements a service class and the code that uses a service class. Code that implements a service class should be associated with a node as low as possible in the hierarchy, to most precisely expose to users the variety of functionalities provided by the service. On the other hand, code that uses a service class should choose the least detailed class of services that meets its needs. In this way, (1) a service discovery request is more likely to be successful and to return a larger number of entities; (2) the resulting application only exposes the functionalities it requires, thus improving its portability, including making it forward compatible with future or refined versions of the requested service class.

2.3 An example

As a concrete example, we consider in more detail the PerIDL specification of Figure 3. At the root of the PerIDL specification is a general service class, named `Service`, containing the semantic properties and interaction modes that will be the least common denominator of any entities in the given domain. In our example, the semantic properties include the

`URI`, which each entity must provide so that it can be referenced by other entities in the environment. The interaction modes include the `shutdown` command, which enables any entity to be shut down. `Service` has three successor service classes.

The left node `Device` assembles the classes of devices that the area expert has identified as supporting the building surveillance activities. A service class may correspond to a class of actual devices, or some extended version of devices, provided by some logic implemented by the programmer. For example, the area expert could define a class of extended webcams with motion detection capability. For a given webcam providing this capability, motion detection could be built-in or implemented as a software service combining the webcam with a motion detector.

The middle node `DataCenter` assembles the service classes that serve various kinds of data, such as the calendar information for building opening hours and the contact information of guards on duties. Interaction modes may be events for the building opening hours and simple lookup operations for the contact information of guards.

The right node `Manager` assembles the service classes dedicated to the application area that need to be developed. For building surveillance, the area expert may identify the need for various managers to coordinate the activities of more basic services. For example, the motion detection manager is responsible for collecting and cross-checking detection information, and taking appropriate actions based on specific rules.

3 Developing Services for a Pervasive Computing Area

To develop a new service in our approach, the programmer first determines the service class it should belong to. The declarations of the selected service class then provide the programmer with a domain-specific design framework for implementing all the facets of the service, ranging from its operations to its deployment. This design framework is supported by a software framework that is automatically generated from the PerIDL specification, as described in Section 4. We assume that code is implemented in Java.

3.1 The interface provided to the programmer

The software framework generated by PerGen from a PerIDL specification provides the programmer with an abstract class corresponding to each PerIDL service class declaration. This abstract class contains methods for those functionalities that are completely determined by the information provided in the PerIDL specification, abstract classes for those functionalities that have a service-specific behavior, and instance variables for the semantic properties. For example, a method for publishing an event depends only on the type of the event, and thus is defined in the abstract class, whereas a commands are represented by abstract methods that the programmer must implement in the service. This organization implies that the programmer can focus his development effort on the application logic. This use of abstract classes can also trigger generation of programing support in Integrated Development Environments (IDEs), such as Eclipse.

3.2 Definition of services

To create a service, the programmer begins by extending the abstract class corresponding to the selected PerIDL service class. In an IDE such as Eclipse, this action triggers the creation of a class skeleton (or class stub in Eclipse parlance), which the programmer must then fill in with the service code. We now describe the subsequent programming process, and the support that Eclipse provides.

3.2.1 Interaction modes

The interaction modes are represented by the methods and abstract methods of the abstract class. As examples, we use the implementation of a `MotionDetectingWebcam` service (Figure 4) and a `MotionDetectionManager` service that uses such a webcam (Figure 5).

```
 1 public class MyWebcam extends MotionDetectingWebcam {

 3   public MyWebcam(String uri) {
 4     super(uri);
 5     // TODO Auto-generated constructor stub
 6   }

 8   public VideoSession connect(IVideoSessionOutput service) {
 9     // TODO Auto-generated method stub
10     return null;
11   }

13   public void disconnect(VideoSession activeSession) {
14     // TODO Auto-generated method stub
15   }

17   public void zoom(Region region) {
18     // TODO Auto-generated method stub
19   }

21   public Autonomy getAutonomy() {
22     // TODO Auto-generated method stub
23     return null;
24   }

26   public void shutdown() {
27     // TODO Auto-generated method stub
28   }
29 }
```

Figure 4: The `MyWebcam` class skeleton, as generated by Eclipse

Command. Commands are represented by abstract methods in the abstract class. From these declarations, Eclipse generates method stubs[2] for all of the commands that need to be implemented. As shown on lines 17 to 28 of Figure 4, for a motion detecting webcam, these

[2]We refer to *method stubs* as methods generated by an IDE like Eclipse, whereas stubs refer to client stubs created using an IDL compiler.

methods are `zoom`, `getAutonomy`, and `shutdown`, declared respectively by the PerIDL nodes for `ZoomingWebcam`, `Device`, and `Service` (Figure 2(a)).

When a service needs to invoke a command from another service, it calls the associated method of this service, which embeds an RPC call. For example, the `HallMotionDetection-Manager` service shown in Figure 5 may need to be able to make a webcam zoom in on a particular region. Line 19 of this service invokes the `zoom` method of a previously obtained webcam.

```
 1 public class HallMotionDetectionManager extends MotionDetectionManager {
 2
 3   private MotionDetectingWebcam myWebcam;
 4   [...]
 5   public HallMotionDetectionManager(String uri) {
 6     super(uri);

 8     MotionDetectingWebcamPart part = MotionDetectingWebcam.getPartition();
 9     part.location.setValue(hall);
10     myWebcam = MotionDetectingWebcam.getService(part);
11     myWebcam.subscribe(this);
12   }
13
14   public void receive(MotionEvent event) {
15     Region region = event.getValue().getRegion();
16     Display myDisplay;
17     [...]

19     myWebcam.zoom(region);

21     VideoSession webcamSession = myWebcam.connect(this);
22     VideoSession displaySession = myDisplay.connect(this);
23     bridge = bind(webcamSession, displaySession);
24     [...]
25   }
26   [...]
27 }
```

Figure 5: The hall motion detection manager

Event. For a PerIDL specification that provides an event interaction mode, the corresponding abstract class defines a `publish` method for each declared type of output event. Services implementing the interface definition invoke these `publish` methods to publish the corresponding event. In our webcam example, the only output event is a motion event. Consequently, the service publishes an event whenever motion is detected. This event will be received via an event channel by all services that have subscribed to it.

For a PerIDL specification that provides an event interaction mode, the corresponding abstract class also defines a `receive` abstract method for each declared type of input event. Such a method has an argument of the given event type. From this declaration, Eclipse generates a method stub, which is to be filled in by the programmer. Lines 14-25 of Figure 5 show the definition of this method provided by the programmer for the `HallMotion-DetectionManager`, which implements the `MotionDetectionManager` sevice class that can receive motion events. The hall motion detection manager may then subscribe to motion events from various sources. In Figure 5, line 11, it subscribes to the motion event provided by the previously obtained motion detecting webcam.

Session. The code relevant to a session is similar to that of an event: services declared as session invitee lead to the creation of the `connect` and `disconnect` method stubs, whereas services in service classes declared as session initiator invoke these methods to receive a stream of data. Services in service classes declaring session binder use the `bind` method to establish a session between two services that have session capabilities. In our example, the `MotionDetectingWebcam` and `Display` services are invitees and the `MotionDetection-Manager` service is a session binder. Thus, the hall motion detection manager can establish a session between the `MotionDetectingWebcam` service and the `Display` service, as shown in lines 21 to 23 of Figure 5.

3.2.2 Semantic properties

The programmer must initialize the values of the semantic properties in the constructor of the service. In doing so, the service is characterized, enabling other services to discover it. An excerpt of the constructor of the motion detecting webcam is shown in Figure 6. This constructor first invokes the constructor of the abstract class, via a call to `super`, to register the service in the runtime of the framework. It is an invariant of our approach that this constructor requires a URI as an argument, as this information is necessary to identify the service.

```
public MyWebcam(String uri, Location myLocation) {
  super(uri);

  // from the Device service
  location.setValue(myLocation);

  // from the Webcam service
  videomode.addValue(new VideoMode(VideoCodec.MJPEG, Resolution.4CIF));
  videomode.addValue(new VideoMode(VideoCodec.MPEG4, Resolution.2CIF));
  framerate.setValue(25);
  [...]
}
```

Figure 6: An MotionDetectingWebcam constructor (extracted from the `MyWebcam` service)

3.3 Service discovery

The software framework that is generated from a PerIDL specification provides the programmer with methods to select any node of this specification. The result of this selection is a set of all services corresponding to the selected node and its subnodes, which we refer to as a *partition*. From a partition, the programmer can further narrow down the service discovery process by specifying the desired values of the semantic properties. Eventually, the method `getServices` or `getService` is invoked to obtain a list of matching services or one service chosen at random from this list, respectively. A key advantage of our approach is that this discovery process is not based on strings but makes use of selection operations generated from the environment specification, making the discovery logic safe with respect to the service class hierarchy.

As an example, consider a `MotionDetectionManager` service that should get a video stream from all webcams that detect motion. This service must initially subscribe to the motion detection event of all of the motion detecting webcams that are located inside the building. A fragment of this service is displayed in Figure 7. It obtains a partition by selecting the `MotionDetectingWebcam` node (line 5) and then sets the `location` semantic property to limit the scope of webcam to those that are indoors (line 6). The operation `getServices` (line 7) then returns all webcams corresponding to this request, including its subnodes, if any. Finally, the last two lines iterate over all of the obtained webcams to subscribe to each one's motion detection event.

```
 1 public MyMotionDetectionManager(String uri) {
 2   super(uri);
 3   priority.setValue(Priority.HIGH);
 4
 5   MotionDetectingWebcamPart part = MotionDetectingWebcam.getPartition();
 6   part.location.setValue(indoor);
 7   LinkedList<MotionDetectingWebcam> myWebcamList =
 8                         MotionDetectingWebcam.getServices(part);
 9   for (MotionDetectingWebcam myWebcam:myWebcamList)
10      myWebcam.subscribe(this);
11 }
```

Figure 7: Discovering motion detecting webcams

4 The PerGen compiler

The goals of the PerGen compiler are both to verify the PerIDL specification and to generate code that supports relevant pervasive computing operations. To present this compiler, we first give an overview of its processing of a PerIDL specification of a pervasive computing area. Then, we examine key parts of the generated framework. Each aspect of the PerGen compiler is discussed from two viewpoints: the verifications enabled by our approach and the administrative support code generated from PerIDL declarations. We also highlight the relationship between our approach and CORBA.

4.1 Area-specific framework

A PerIDL specification describes a pervasive computing area. Based on this, PerGen performs verifications that ensure consistency properties and generates code supporting basic pervasive computing operations.

Verification. Existing middleware approaches are designed with the goal of allowing distributed systems to be constructed dynamically. Thus, these systems allow new IDL specifications and their implementations to be added at any time. While this approach is highly flexible, it interferes with static verification of dependencies between service classes. For example, a service class could be specified as a consumer for an event for which there is no supplier. This would lead to errors at runtime when services in the service class are deployed.

In contrast, our approach is built on a static model, in which an expert in the area provides a complete taxonomy of the service classes relevant to the target area. PerGen then checks that the event and session declarations in this taxonomy are consistent. For both event and session declarations, every event or session provider must have at least one matching consumer, and vice versa. Matching events or sessions have the same type.

These verifications of course do not completely prevent run-time errors. Even if a hierarchy contains a service class with given properties, there may be no corresponding service available at run time. Further verifications are thus performed at service registration time, as noted below. The checks performed on the hierarchy, however, are sufficient to identify guaranteed inconsistencies, and do so before application deployment.

Code support. The framework generated by PerGen from a PerIDL specification contains code specific to each service class as well as code managing the pervasive computing environment.

A PerIDL service class declaration compiles into a Java abstract class that plays the role of a CORBA server skeleton, as generated by a traditional IDL compiler. It implements the interfaces representing the various interaction modes used by the service class, declares instance variables representing the semantic properties, and defines abstract methods that must be implemented by the service, *e.g.*, to provide the behavior for a command. PerGen also generates environment-wide operations concerning, for example, service registration and service discovery.

Even though a pervasive computing area needs to be fixed at some point in time, it can still evolve. New PerIDL declarations can be added to an environment specification, leading to the generation of a new framework, requiring both verification and generation of code support. If the new PerIDL service class declarations are only introduced as leaves of the hierarchy, then existing application code does not need to be updated. However, when nodes are inserted in the middle of the hierarchy of service classes, some parts of an application may need to be changed. Future work involves providing developers with support for making these changes.

4.2 Service registration

When a service is deployed, it needs to be registered in the framework runtime. In this, semantic properties guide the service programmer in writing the registration logic: setting their values amounts to exposing the characteristics of the service, making it possible for client services to find the best match for their requirements.

Verification. CORBA manages service registration and discovery via a *trading service.* To provide dynamicity, this trading service manipulates component types and property names as strings. This approach, however, is error-prone, and errors in these strings such as misspelling the name of a property are not detected until runtime. Our approach builds on the top of the CORBA trading service, but PerGen generates typed methods and classes for service registration and discovery, enabling compile-time verification of the use of these operations.

Semantic properties defined as PerIDL attributes are transformed into Java classes. In addition to the primitive types, a PerIDL specification consists of semantic properties that are specific to pervasive computing like ranges (*e.g.,* a rotation angle is an integer between 0 and 180), enumerations (*e.g.,* a list of codecs), sequences (*e.g.,* a resolution consists of a width and a height), and hierarchies (*e.g.,* the location). Domain-specific properties that are constructor-based (*e.g.,* enumeration) are checked at compile time, when registering or querying a service. When semantic properties are domain specific but not constructor-based (*e.g.,* ranges), verification is postponed until runtime via generated tests.

Code support. To ensure that all deployed services are registered, service registration is automatically invoked in the constructor of the abstract class. The registration code attaches the service to the node corresponding to its service class in the service class hierarchy.

4.3 Service discovery

A service discovery query is processed with respect to the hierarchy of service classes and their semantic properties.

Verification. In the CORBA trading service, service discovery is performed by the `query` operation. The generic nature of this operation prevents any verification at compile time. In contrast, we introduce a two-step service discovery process: (i) selecting a node in the hierarchy of service classes and (ii) refining this selection by setting values to semantic properties.

Specifically, for each Java abstract class generated from a PerIDL service class, PerGen produces a method to select the corresponding node from the hierarchy of service classes. When invoked, this method yields a partition containing the services corresponding to the selected node (see Line 8 of Figure 5). PerGen produces further methods that allow refining this partition by setting values for the various semantic properties. These methods check any constraints on the required values of the semantic properties, as in the service registration case.

Code Support. Two methods are generated to complete the discovery process: one to select a single service from a refined partition, and another to get all the services of a refined partition. These methods return respectively a service object or a `LinkedList` parameterized by an abstract service class (see Figure 7).

A service is not manipulated directly: service discovery produces a reference to the selected services. These references point to proxies (*i.e.,* stubs), implementing the corresponding Java abstract class. For example, in Figure 5, the service discovery code in line 8-10 obtains a `MotionDetectingWebcam` proxy, whose `zoom` method is invoked in line 19.

4.4 Interaction modes

Our command mode is identical to CORBA methods. Thus, we focus on the event and session interaction modes.

4.4.1 Verification

The safety of a pervasive computing application critically relies on how services are being composed via events and sessions. We have identified three key verifications that exploit the PerIDL declarations and are carried out by the PerGen compiler: direction of interaction mode (*i.e.,* a supplier must only interact with a consumer), connectivity of services (*i.e.,* there should not be dangling suppliers or consumers), and typed service interaction (*i.e.,* a supplier and a consumer should have a strongly typed interaction).

Direction of interaction mode. The PerGen compiler generates interfaces that ensure at compile time that an event or a session consumer always interacts with a supplier.

Connectivity of services. CORBA allows suppliers and consumers to be loosely coupled in that an event consumer can subscribe without there being a registered event supplier, whereas the reverse is not allowed. As a result, an environment could consist of a siren service, expecting a non-existent (or faulty) fire detection service.

PerGen ensures that suppliers and consumers of information (whether events or sessions) are connected, preventing information from being lost or expected endlessly. For example, in the case of events, in our generated frameworks, an event supplier is connected to the appropriate event channel during the service registration process. When a consumer finds an event supplier, using service discovery, this event supplier is thus known to be connected to the event channel; the consumer can then subscribe to the desired events. This process ensures that all consumers have valid suppliers. If a supplier is no longer available, a lease mechanism detects it and an exception is raised on the consumer side, allowing the consumer to select a substitute supplier, if desired.

Typed service interaction. The CORBA event service was not primarily designed to perform typed interactions between consumers and suppliers. Instead, string matching is used to link consumers and suppliers; an error prone cast is required when publishing an event.

PerGen produces proxies for events and sessions that ensure, at compile time, that interactions between consumers and suppliers are well-typed. We consider the case of events; sessions are treated similarly. As in the CORBA event service, the generated framework includes two generic interfaces: the `EventOutput` interface for event suppliers and the `EventInput` interface for the event consumers. These interfaces enable a service to register as a supplier or a consumer of events. Event typed interfaces (*e.g.,* `EventOutputMotion`) compile into a Java class (*e.g.,* `MotionEvent`) describing the event and an associated event interface (*e.g.,* `IMotionEventOutput`). These interfaces extend the generic event interfaces, defining publish/subscribe methods with respect to the event type. In doing so, type mismatch can be detected when a service subscribes to an event. A consumer furthermore implements a `receive` method and a supplier inherits a `publish` method implementation from the server skeleton. These methods are typed according to a given event. A supplier also implements a `subscribe` method. A reference to the consumer is passed to this `subscribe` method, ensuring that the consumer is able to receive the events published by the given supplier (see Line 11 of Figure 5).

4.4.2 Code support

We concentrate on the support generated by PerGen for sessions, as support for sessions is a unique aspect of our approach. Creating a session involves first negotiating the values of session parameters (*e.g.,* codec, sampling rate or transport protocol) and then exchanging the data stream. Sessions can be created in two ways: a `SessionOutput` service can establish a session with a `SessionInput` service (*e.g.,* an air conditioner may establish a session with a temperature sensor) or a `SessionBinder` service can establish a session between two `SessionInput` services (*i.e.,* two invitees). PerGen accordingly creates generic `SessionOutput`, `SessionInput`, and `SessionBinder` interfaces, where the `SessionBinder` extends the `SessionOutput` interface. The `SessionInput` interface declares the `connect` method, allowing services in `SessionOutput` or `SessionBinder` service classes to invite services in a `SessionInput` service class. The `SessionBinder` interface declares the `bind` method, allowing services in the `SessionInput` service class to establish a session.

The `connect` and `bind` method depend on the information provided in the corresponding `SessionInput` or `SessionBinder` service class, respectively. They both consist of negotiating session parameters. The `connect` method receives a session invitation along with the session initiator capabilities and returns a subset of these capabilities. The `bind` compares the session capabilities of the services to be bound and determines the session parameters to be used to configure the exchange phase. PerGen generates a `match` method for each session type (*e.g.,* the video session type) to perform this negotiation process. PerGen also generates the implementation of a `bind` method in the corresponding abstract class. A `bind` method calls the corresponding `match` method. Unlike the `bind` method, the `connect` method must be implemented by the developer to support dynamic mechanisms (*e.g.,* resource control policies). The `match` method facilitates the implementation of the `connect` method. The negotiation mechanism is thus hidden from developers.

5 Evaluation

Our approach is implemented in the context of JacORB [1], a Java implementation of CORBA. PerGen was developed on the top of the JacORB IDL compiler, reusing as many CORBA components as possible.

This section examines what additional benefits our approach offers as compared to CORBA. To do so, we first present a qualitative comparison, identifying the additional features provided by our approach, and then present a quantitative study, measuring the amount of additional code support generated by PerGen.

5.1 Qualitative comparison with CORBA

Table 1 summarizes the additional features provided by our approach, in terms of code support and verification. The column for CORBA indicates whether CORBA provides ("yes") or not ("no") the given feature. The column for PerGen indicates how our approach compares to CORBA with respect to the given feature. "Yes" indicates that our approach provides a feature that CORBA does not, "+" indicates that our approach provides an enhanced version of the feature, and "=" indicates that our approach provides essentially the same functionality as CORBA.

		CORBA	**PerGen**
Interaction modes	Command	yes	=
	Event	yes	+
	Session	no	yes
Service Discovery		yes	+
Verifications	Framework-generation time	no	yes
	Compile time	partial	+
	Runtime	yes	+
Programming Support		stubs & skeletons	every stage

Table 1: Generated features

Regarding interaction modes, our contribution focuses on events and introduces sessions, which are critical to pervasive computing. We extend the service discovery facility of CORBA by generating high-level support to register and discover services. PerIDL declarations are extensively exploited to carry out verifications at every development stage, from the specification of a pervasive computing area to the execution of an application. Finally, our approach provides the programmer with support that is customized with respect to a target pervasive computing area, enabling the generation of area-specific class skeletons by an IDE such as Eclipse.

5.2 Quantitative comparison with CORBA

Let us now measure how much additional code support is provided by PerGen compared to CORBA. To do so, we evaluate our approach in three application areas: building surveillance, telephony, and home automation. The surveillance area was introduced earlier in the paper. We briefly examine the entities covered by our PerIDL specifications for the two other areas. The telephony area consists of a variety of forms of multimedia communication and supporting components. It involves the following kinds of entities: communication terminals (*e.g.,* GSM smart phones and WIFI phones), various kinds of displays, devices for sound and image capturing (*e.g.,* microphones and webcams), devices for motion capturing (*e.g.,* RFID readers and motion detectors), rendering devices (*e.g.,* voice recognition and synthesizer devices), message-based servers (*e.g.,* e-mail and MMS servers), personal information manager (*e.g.,* presence, agenda and contacts), *etc.* The home automation area is composed of the following categories of entities: appliances (*e.g.,* dishwashers and ovens), safety devices (*e.g.,* smoke and fire detectors), security devices (*e.g.,* door/window locks and motion detectors), entertainment entities (*e.g.,* multimedia center, HIFI systems and home cinemas), *etc.* As for the building surveillance area, the above mentioned entities have been organized hierarchially into PerIDL declarations. Specific attributes and operations have been defined. These PerIDL declarations have been validated in the context of various applications.

Table 2 compares the number of interfaces and operations that CORBA IDL and PerIDL require to describe the various pervasive application areas. The numbers for CORBA IDL are consistently higher than those for PerIDL. This difference is due to the domain-specific nature of PerIDL that makes implicit various pervasive computing aspects.

	Interfaces		Operations	
	IDL	PerIDL	IDL	PerIDL
Surveillance	49	28	39	27
Telephony	63	35	44	41
Home automation	108	48	74	51

Table 2: Comparing PerIDL with IDL

PerGen extends the JacORB IDL compiler to generate additional features such as interaction modes and service discovery on top of the CORBA middleware. Figure 8 compares the number of lines of code generated by PerGen with the number of lines of code generated by the JacORB IDL compiler in total, over the three application areas. The PerGen specific part of the bar charts corresponds to the additional features generated by PerGen, whereas the JacORB IDL compiler part mainly corresponds to stubs and skeletons. As shown by the rightmost bar, when considering all three areas, the part of code generated by PerGen represents 37% of the overall generated code. Finally, note that regardless of the application area, the percentage of code generated by PerGen is stable.

Figure 9 decomposes the contribution of PerGen with respect to key aspects of a pervasive computing, namely, service discovery and interaction modes, over our three application areas.

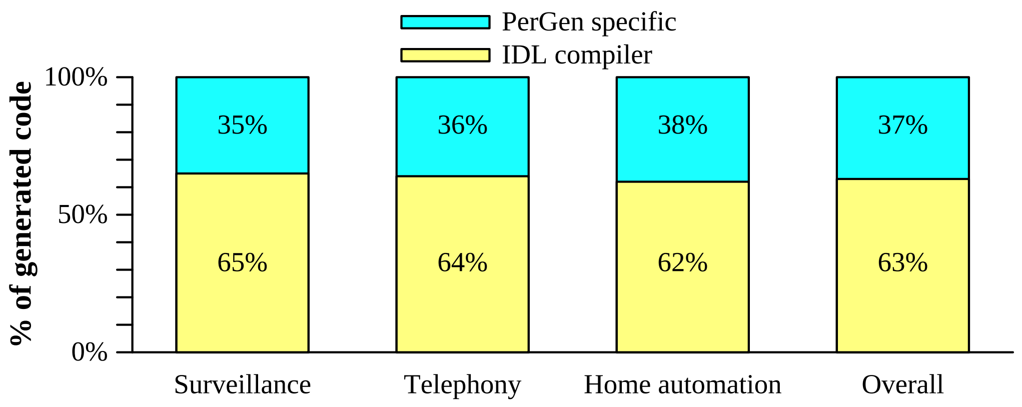


Figure 8: Application areas: Code generated by the IDL compiler in PerGen

As can expected, service discovery requires much more generated code than the interaction modes. The three bars on the right break down the interaction mode result in terms of the three kinds of interaction modes. PerGen relies entirely on the JacORB IDL compiler for the command interaction mode (100%). In constrast, the session interaction mode relies on more PerGen generated code than the event interaction mode (39% vs. 29%). This difference is due to the fact that the session interaction mode is not built into the CORBA middleware and thus requires specific code to be generated.

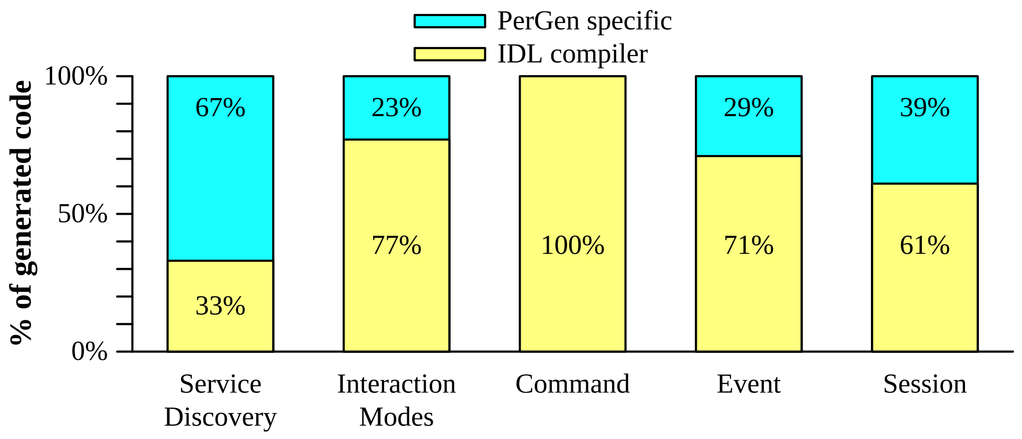


Figure 9: Features: Code generated by the IDL compiler in PerGen

6 Related Work

Standardized middlewares (*e.g.,* CORBA, DCOM) are highly flexible and support a large number of features, targeting a wide range of application areas. This genericity can, however, be a burden when it comes to address requirements from particular domains, as adaptation code must often be developed to match the application's needs. To address this issue, several approaches propose to specialize or configure a middleware implementation to fulfill requirements from a particular domain. For example, Gilani *et al.* propose an aspect-oriented programming approach to statically and dynamically configure an ORB middleware [2].

The IDL approach has been successfully used to facilitate the development of distributed systems. Although this initial target domain is the main focus of work, researchers are proposing new IDL approaches. QIDL extends an IDL with QoS aspects [3]. Demir *et al.* enriches an IDL to bypass middleware layers under certain conditions [5]. The DADO approach facilitates the integration of late-bound, crosscutting features in distributed systems using the concept of adaplets. DADO uses an enhanced IDL named DAIDL that defines crosscutting services [13]. Vergmaud *et al.* propose to generate a customized middleware from an Architecture Description Language (ADL) specification and a code repository [12]. Glue for source code from the repository is generated from an ADL. The Q-WSDL extension proposes to specify QoS requirements, to establish service level agreements (SLA) [4].

These above approaches leverage on various forms of specifications such as ADLs, WSDLs or IDLs to generate additional crosscutting features (*e.g.,* security, performance, and real time) to adapt a middleware implementation to an application domain (*e.g.,* embedded systems). However, none of them, as far as we know, has targeted the domain of pervasive computing. Also, these approaches do not use specifications to check domain-specific properties in applications. Finally, they do not exploit specifications to generate a customized programming framework, to facilitate program development.

Most related to our approach is Olympus [9]. This work proposes to ease the development of applications and services in pervasive computing environments. Olympus enhances Gaia, a distributed middleware infrastructure for pervasive environments, by providing the programmer with a high-level programming model. Similar to PerIDL, a pervasive environment is specified, taking the form of ontologies. Unlike our approach, Olympus provides the developer with an untyped and generic programming model. In particular, service discovery operations are passed strings to represent values of semantic properties. In constrast, our approach integrates PerIDL specifications into the programming framework. This integration enables a variety of compile-time verifications to be performed, improving the application reliability.

7 Conclusion

In this paper we have identified key requirements in the development of pervasive computing applications. To address these requirements we have proposed PerIDL, a CORBA-based IDL dedicated to the domain of pervasive computing. We have shown that PerIDL is expressive enough to characterize environments in a given pervasive computing area. In our approach, a PerIDL specification is used by programmers to guide the development of applications in a target area.

We have presented PerGen, a compiler for PerIDL specifications. PerGen performs a number of verifications at various stages of the development, making pervasive applications more reliable. Additionally, PerGen generates a programming framework customized with respect to a given pervasive computing area, facilitating the development work. PerGen is implemented and has been successfully used to specify three areas of pervasive computing. Furthermore, applications within these areas are being developed.

In the future, we plan to cover more pervasive computing areas, studying in particular assisted living and medical patient monitoring. We also plan to extend PerGen to offer non-functional features like environment monitoring and encrypted data.

References

[1] JacORB: The free Java implementation of the OMG's CORBA standard, http://www.jacorb.org.

[2] S. Apel and K. Bohm. Towards the development of ubiquitous middleware product lines. In *ASE'04 SEM Workshop*, 2004.

[3] C. Becker and K. Geihs. Generic QoS-support for CORBA. In *ISCC '00: Proceedings of the Fifth IEEE Symposium on Computers and Communications (ISCC 2000)*, pages 60–65, Washington, DC, USA, 2000.

[4] A. D'Ambrogio. A model-driven WSDL extension for describing the QoS of web services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 789–796, Washington, DC, USA, 2006.

[5] O. E. Demir, P. Devanbu, E. Wohlstadter, and S. Tai. An aspect-oriented approach to bypassing middleware layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 25–35, New York, NY, USA, 2007.

[6] The Eclipse Foundation, http://www.eclipse.org/. *Eclipse - an open development platform.*

[7] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PERCOM'04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 263, Washington, DC, USA, 2004.

[8] OMG. *CORBA: The Common Object Request Broker: Architecture and Specification.* Framingham, 1995.

[9] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PERCOM'05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Kauai, Hawaii, 2005.

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

[11] Rosenberg, J. et al. SIP : Session Initiation Protocol. RFC 3261, IETF, June 2002.

[12] T. Vergmaud, J. Hugues, L. Pautet, and F. Kordon. Rapid development methodology for customized middleware. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 111–117, Washington, DC, USA, 2005.

[13] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: Enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 174–186, Washington, DC, USA, 2003.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399