High-level Programming Support for Robust Pervasive Computing Applications

Wilfried Jouve
INRIA / LaBRI, France
wilfried.jouve@labri.fr

Julien Lancia
Thales / LaBRI, France
julien.lancia@labri.fr

Nicolas Palix
INRIA / LaBRI, France
nicolas.palix@labri.fr

Charles Consel
INRIA / LaBRI, France
charles.consel@labri.fr

Julia Lawall
DIKU, Denmark
julia@diku.dk

Abstract

In this paper, we present a domain-specific Interface Definition Language (IDL) and its compiler, dedicated to the development of pervasive computing applications. Our IDL 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 programming 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.

1 Introduction

Pervasive computing environments introduce new challenges for application development, due to the heterogeneity and dynamicity of the devices involved. Currently, middleware is a key enabling technology used to address these challenges in developing pervasive computing applications. Middleware abstracts over a number of implementation issues, enabling distributed, heterogeneous objects to interoperate. Most middleware approaches are furthermore general-purpose and highly dynamic, thus providing the expressiveness and adaptability needed for pervasive computing applications.

The complexity of the pervasive computing domain, however, implies that it is not sufficient to implement applications in an ad hoc manner; it is also necessary to be able to reason about their behavior and verify domain-specific and area-specific properties. Middleware uses generic mechanisms, such as strings, to describe the interaction between components. Such mechanisms are error prone (*e.g.*, strings can be misspelled) and not checked until run time. Many middleware approaches do provide some level of customization via an Interface Definition Language (IDL), such as OMG IDL, however, they do not permit domain-specific properties to be checked or domain-specific programming support to be generated. Thus, they do not adequately ensure the robustness of applications. Furthermore, pervasive computing development consists of common program patterns that are only partially supported by the IDLs of general-purpose middlewares, requiring the programmer to provide code to further tailor the middleware to this domain. In this case, the programmer lacks declarative means to express these commonalities, limiting the sharing of knowledge to code.

Our approach We introduce an IDL, named *PerIDL*, with pervasive computing concepts built-in. In our approach, an area expert uses PerIDL to describe a pervasive computing area. This description forms a repository of knowledge for programmers who need to develop applications in the area. It is furthermore passed to our compiler, *PerGen*, to automatically generate a software framework that provides customized verifications and programming support. Unique among existing IDLs, PerIDL provides native support not only for the *command* (*e.g.*, RPCs) and *event* iteraction modes, but also for stream-oriented entities, via a complementary interaction mode called a *session*. Sessions raise the level of abstraction of the implementation of stream-oriented interactions, as are widely required in many pervasive computing areas, and enhance the quality of verifications and programming support that can be provided for stream-based programming.

2 Defining a Pervasive Computing Area

Using PerIDL, an area expert describes a pervasive computing area as a taxonomy of the relevant entities. Creating

such a taxonomy requires first analyzing the pervasive computing area and then specifying the relevant entities and the relationships between them.

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 (*e.g.*, sensors, webcams) and software entities (*e.g.*, databases). Variations occur in both the functionalities provided by these basic building blocks and their non-functional properties. For example, at the level of functionalities, there are simple webcams, zooming webcams and webcams with a built-in motion detector. These functional variations typically entail significant differences in the implementation logic. Non-functional variations include properties such as location and orientation, which may also impact how an entity is used. Finally, these basic building blocks are not in general sufficient to describe all of the entities needed in a pervasive computing area; programmers also need to develop composite building blocks, *i.e.*, applications or managers, that coordinate the behaviors of the more basic building blocks.

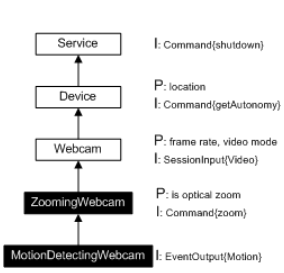
2.2 Specifying an area in PerIDL

Based on the results of the analysis, the area expert creates a PerIDL specification modelling the classes of entities relevant to the target area. We refer to a class of entities providing a particular set of functionalities as a *service class*. The area expert describes a service class in terms of 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 range of entities it corresponds to, in terms of the non-functional variations identified during the analysis phase, *e.g.*, the location of a motion detector or the codec being used by a webcam. Fig. 1 depicts a number of service classes, with their associated semantic properties indicated by "P". 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 mode corresponds to an RPC, typically to control a device (*e.g.*, `zoom` for a webcam in



(a) Graphical representation

```
interface Webcam : Device {
  attribute Framerate frame_rate;
  attribute VideoMode video_mode;
  SessionInput(Video video);
};

interface ZoomingWebcam :
                        Webcam {
  attribute boolean isOptical;
  void zoom(in Region region);
};

interface MotionDetectingWebcam :
                ZoomingWebcam {
  EventOutput(Motion motion);
};
```

(b) Textual representation

Fig. 1. The zooming and motion detecting webcam

Fig. 1).

Event. The event 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. A service class for webcams featuring motion detection could be modeled as a publisher of a motion event, as shown in Fig. 1 by the declaration `EventOutput{Motion}`. An event declaration indicates both the event type (*e.g.*, `Motion`) and the event direction, whether incoming (`EventInput`) or outgoing (`EventOutput`).

Session. The session mode natively supports 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: (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.

The declaration `SessionInput{Video}` in the definition of the service class `Webcam` (Fig. 1) 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).

Hierarchical relationships. The PerIDL description of a pervasive computing environment is structured as a hierarchy, as illustrated by the specification of the building surveillance area in Fig. 2. Starting at the root node, this description breaks down the set of possible entities of this area 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 the semantic properties and interaction modes of its ancestors.

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. Code that uses a service class should choose

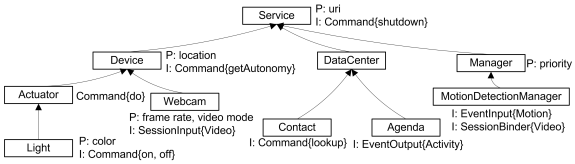


Fig. 2. Excerpt of the building surveillance area

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, and making it forward compatible with future or refined versions of the requested service class.

3 Developing Services for an 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 a domain-specific design framework for implementing all 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 by the compiler PerGen.

3.1 Definition of services

To create a service, the programmer extends the abstract class generated from the corresponding PerIDL service class. The skeleton of such an extension can be automatically generated by an IDE such as Eclipse (*e.g.*, Fig. 3). We now describe the subsequent programming process.

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 (Fig. 3) and a `MotionDetectionManager` service that uses such a webcam (Fig. 4).

```
 1 public class MyWebcam extends MotionDetectingWebcam {
 3   public MyWebcam(String uri) {
 4     super(uri);
 5     // TODO Auto-generated constructor stub
 6   }
 7   public VideoSession connect(IVideoSessionOutput service) {
 8     // TODO Auto-generated method stub
 9     return null;
10   }
11   public void disconnect(VideoSession activeSession) {
12     // TODO Auto-generated method stub
13   }
14   public void zoom(Region region) {
15     // TODO Auto-generated method stub
16   }
17   public Autonomy getAutonomy() {
18     // TODO Auto-generated method stub
19     return null;
20   }
21   public void shutdown() {
22     // TODO Auto-generated method stub
23   }
24 }
```

Fig. 3. The MyWebcam class skeleton

Command. Commands are represented by abstract methods in the abstract class. As shown on lines 14 to 23 of Fig. 3, for a motion detecting webcam, these methods are `zoom`, `getAutonomy`, and `shutdown`, declared respectively by the PerIDL nodes for `ZoomingWebcam`, `Device`, and `Service` (Fig. 1(a)).

```
 1 public class HallMotionDetectionManager
    extends MotionDetectionManager {
 2   private MotionDetectingWebcam myWebcam;
 3   [...]
 4   public HallMotionDetectionManager(String uri) {
 5     super(uri);
 6     MotionDetectingWebcamPart part =
         MotionDetectingWebcam.getPartition();
 7     part.location.setValue(hall);
 8     myWebcam = MotionDetectingWebcam.getService(part);
 9     myWebcam.subscribe(this);
10   }
11   public void receive(MotionEvent event) {
12     Region region = event.getValue().getRegion();
13     Display myDisplay;
14     [...]
15     myWebcam.zoom(region);
16     VideoSession webcamSession = myWebcam.connect(this);
17     VideoSession displaySession = myDisplay.connect(this);
18     bridge = bind(webcamSession, displaySession);
19     [...]
20   }
21   [...]
22 }
```

Fig. 4. The hall motion detection manager

Event. For a PerIDL specification that provides an event interaction mode, the corresponding abstract class defines a concrete `publish` method for each declared type of output event. Services implementing an `EventOutput` interaction mode 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. A service may subscribe to events from various sources (*e.g.*, line 9 in Fig. 4. For each type of event, it must define an appropriate `receive` method (*e.g.*, line 11-20 in Fig. 4.

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` abstract methods in the corresponding abstract classes. These methods are invoked by service classes declared as session initiator to receive a stream of data.

Services in service classes declaring session binder use the generated concrete `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 `MotionDetectionManager` service is a session binder. Thus, the hall motion detection manager can establish a session between the `MotionDetectingWebcam` service and the `Display` service (lines 16–18 of Fig. 4).

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.

3.2 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.

4 The PerGen compiler

The goals of the PerGen compiler are to verify the PerIDL specification and to generate code that supports relevant pervasive computing operations.

Area-specific framework. From a PerIDL specification, PerGen performs verifications that ensure consistency properties and generates code supporting basic pervasive computing operations. 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. This area can still evolve and new PerIDL declarations can be added to an environment specification, leading to the generation of a new framework. PerGen then checks that the event and session declarations in this taxonomy are consistent.

Service registration. To provide dynamicity, existing middleware service discovery manipulates component types and property names as strings. This approach, however, is error-prone, and errors such as misspellings are not detected until runtime. PerGen generates typed methods and classes for service registration and discovery, enabling compile-time verification of the use of these operations. To ensure that all deployed services are registered, service registration is automatically invoked in the constructor of the abstract class.

Service discovery. The generic nature of service discovery in existing middlewares 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 of the semantic properties. In the latter case, the generated methods check any constraints on the required values of the semantic properties. Two methods are generated to complete the discovery process: one to select a single service from a refined partition (see lines 8 in Fig. 4), and another to get all the services of a refined partition. Service discovery produces a typed reference to the discovered services.

Interaction modes. The safety of a pervasive computing application critically relies on how services are composed. PerGen carries out three key verifications that exploit the PerIDL declarations: 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).

5 Related Work

The genericity of standardized middlewares can be a burden when it comes to address requirements from a particular domain. To address this issue, several approaches propose to specialize or configure a middleware implementation to fulfill requirements from a particular domain [1, 4]. IDLs have been successfully used to facilitate the development of distributed systems [2]. However, none of these approaches, as far as we know, use specifications to generate a customized programming framework, to facilitate program development in the domain of pervasive computing. Finally, numerous programming frameworks and toolkits have been proposed [3, 5]. These approaches focus on the rapid prototyping of pervasive applications without ensuring the validity of the design prior to runtime.

6 Conclusion

In this paper, we have presented PerIDL and PerGen, language-based tools to improve the robustness of pervasive computing applications. PerIDL allows an area expert to provide a high level description of the entities in a pervasive computing area, and PerGen generates programming support specific to the area that amenable to static verification. PerGen is implemented and has been successfully used to specify three areas of pervasive computing (*i.e.*, home automation, telephony, building surveillance). Furthermore, applications within these areas are being developed.

References

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

[2] C. Becker and K. Geihs. Generic QoS-support for CORBA. In *ISCC '00*, 2000.

[3] A. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. In *Human-Computer Interaction*, 2001.

[4] W. Jouve, N. Ibrahim, L. Réveillère, F. Le Mouël, and C. Consel. Building home monitoring applications: From design to implementation into the Amigo middleware. In *ICPCA'07*, 2007.

[5] 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*, 2005.