

An Aspect-Oriented Ambient Intelligence Middleware Platform

Lidia Fuentes, Daniel Jiménez

Departamento de Lenguajes y Ciencias para la Computación.

Málaga University

+034 952132810/+034 952132846

{lff,priego}@lcc.uma.es

ABSTRACT

Currently, the interest in Ambient Intelligence (or AmI) has increased exponentially due to the widespread use of portable devices. Users demand more and more functionality from these devices, especially in order to perform collaborative tasks and interchange information. As a result, this technology proposes new challenges that must be addressed by both the hardware manufacturers and Software Engineers. The first challenge is to provide a middleware platform providing specific AmI services like communication or device discovery and able to cope with several challenges posed by AmI applications. One of these challenges is to manage heterogeneity of devices that are present in AmI environments in a transparent way, and in particular to manage the evolution of such devices without breaking the code of already developed applications. Moreover, an AmI middleware platform has to support the evolution of the software architecture of AmI applications over time, making it possible to add new functionalities, to adapt applications to any technological changes, and to produce a product line of AmI platforms to be executed in different devices. A final challenge is that the middleware platform has to support the adaptation of AmI applications to unexpected and dynamic changes in AmI environments. Aspect Oriented Programming (AOP) provides good and advanced solutions to the evolution management problem at different levels, so our work focuses on applying AOP to develop an AmI platform. In this paper, we will discuss these problems, propose solutions and present how these problems are handled in our AmI aspect-oriented platform named AOPAmI. This paper describes the internal platform structure and how it provides support to AmI devices. We will focus on the dynamic nature of AmI applications and on how to solve the above-mentioned problems.

Categories and Subject Descriptors

D.2.11 [Software Architectures]: *Domain-specific architectures*

General Terms

Management, Design, Experimentation, Standardization.

Keywords

Ambient Intelligence, Middleware, Aspects, Components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MPAC '05, November 28- December 2, 2005 Grenoble, France Copyright 2005 ACM 1-59593-268-2/05/11... \$5.00

1. INTRODUCTION

Currently, the use of Ambient Intelligence Applications (or AmI) is spreading quickly throughout all human activities, ranging from new personal devices able to perform computing tasks and communicate with each other, like mobiles or PDAs, and complex entities like domotic houses that are capable of using, coordinating and combining information coming from heterogeneous devices in order to perform complex tasks. As we can imagine, the development of these applications is complex due to the special nature of the devices where applications are executed (limited memory, processing power and energy), also the distributed nature of the applications added to the heterogeneity and diversity of participants (human, hardware and software) contributes to add complexity to such applications.

The use of an adaptive middleware platform, addressing all these issues, would facilitate the development of AmI applications rather than always starting from scratch. But, regarding the development of an AmI platform we have identified three main problems. The first is how to manage the heterogeneity of AmI environments at different levels ranging from hardware and communication capabilities to data format heterogeneity. Such heterogeneity makes it hard to build truly reusable computation units. The second problem, is the management of software evolution, and arises from the difficulty of adapting the applications behaviour to new requirements and technological changes once the application is deployed and running. Currently, most AmI applications are built as monolithic and compact applications, mainly due to issues of efficiency and therefore, the reusability and evolution of developed applications is severely limited. In addition, the middleware platform has to be able to support the runtime adaptation of AmI application functionalities to events and changes in the environment, even to unexpected ones. We call this the application dynamic adaptability problem. Aspect Oriented Programming (AOP) [10] and Aspect Oriented Software Development (AOSD) [1] provides advanced solutions to the evolution management problem, by modelling these properties as aspects applications would evolve over time. Since aspects can be weaved with objects or components in an oblivious way [19], they are the perfect solution for managing software and hardware evolution of AmI applications without breaking the code of already developed components.

In this paper we will present AOPAmI, a component and aspect based middleware platform as a solution to the evolution problem which supports the development of applications in the AmI domain. After this introduction, in the second section we introduce the main motivations and challenges faced when

developing Aml applications, and some related work and solutions to these challenges. In section three, we describe the AOPAmI platform and how it solves the challenges previously presented and finally in section four we outline some conclusions and future work.

2. AMBIENT INTELLIGENCE CHALLENGES

In the previous section we have briefly presented the main challenges that we face when developing Aml applications. In this section we are going to show each one of these problems in detail.

The first problem, and one of the most important challenges in the developing of Aml applications, is dealing with the heterogeneity at different levels. We mean that Aml applications should be able to be run on different hardware devices (such as for example mobile phones or PDAs), communicate using various communication technologies (like IrDA, Bluetooth or WiFi) and transport protocols (like UDP, TCP or OBEX). Additionally, the applications must be able to interpret different data codifications like XML, data streams, etc. Therefore, Aml applications should be developed with software technologies providing mechanisms to manage all these kinds of heterogeneity in an oblivious way [7], without breaking the code of already developed components.

2.1 Hardware heterogeneity

Regarding hardware heterogeneity, J2ME is one of the software technologies available for developing applications that can be used on different resource-constrained devices like mobile phones or PDAs. However, each hardware device has different capabilities and resources, so the same Aml application has to be able to be adapted to different execution environments. Therefore, advanced software technologies should be used to produce different application configurations, it being desirable that the configuration of the application for each case is not hard coded as part of the application code.

In this sense, on one hand, specific domain languages (or SDLs) are an emergent technology that can give a good solution to this problem for specifying specific application behaviour, detaching this from the source code. The use of hardware profiles similar to J2ME device profiles will help us to deal with hardware diversity, categorizing different device types and their capabilities as is shown in [6]. J2ME is a good choice for solving the heterogeneity problem but it lacks of some features that would make it ideal for the development of Aml applications. These features have to be provided by an Aml framework built on top of this language. Two of the desirable features lacking in J2ME are firstly that it does not provide a mechanism to manage software evolution. Thus, any change in technology or functional requirements must be managed by the programmer at code level. This is due to the fact that the minimum functional unit of the J2ME is the class and the programmers and designers have to work at a very low level.

The second feature lacking in J2ME is the absence of support of a full reflection and class loading mechanism at runtime. This means it is very difficult to develop applications which are truly dynamic and evolvable because all the application resources must be available before the execution process has begun and once it has been initiated, the application can not be changed. A framework can partially palliate this situation using different

techniques as will be shown in the following sections, but it is a very important implementation issue to consider.

2.2 Communication heterogeneity

On the other hand, J2ME can be used to provide the application with a wide range of communication technologies like WiFi, Bluetooth or IrDA and the associated communication protocols like UDP, TCP or OBEX, partially solving the communication heterogeneity problem. But, the J2ME API does not provide advanced mechanisms to adapt the application in order to use a new communication technology dynamically without either code recompilation or redeploying the application, but simply plugs new hardware and a new functionality unit into the application. In this sense the use of aspect oriented platforms [1] could be a good option for adding new functionality units in an oblivious way coding them as aspects. This will allow us to add, remove or replace a functionality in the application in an elegant and non-intrusive way. The work of [15] uses AspectJ [10], an AOP language built as an extension of the Java language developed originally by Xerox PARC, to build an Aml application development platform. However this solution is static since if we change an aspect code all of the application must be recompiled, so this approach does not provide a way of performing the configuration on the fly.

2.3 Data heterogeneity

However, the use of J2ME alone will not be sufficient to solve all Aml heterogeneity problems because at communication level each device will manage its own set of information using its own vocabulary and proprietary formats. This is especially true if we use J2ME because it only provides the API to send and receive data but it does not directly support XML. It is the programmer's task to provide concrete data encoding that will be understood by all applications. Additionally if we examine existing Aml platforms, such as for example PCOM [3], the data sent between applications is encoded using different formats (binary stream, text, etc.) depending on the application. This fact makes communication among remote Aml applications difficult because a specific Aml application will probably not be able to understand all the possible data encodings.

In order to deal with this problem we need to be sure that communication among devices in the environment is possible. To achieve this we need to use a compatible and adaptable data interchange format. We think that XML or some variants of it, such as for example the WAP binary XML (or wbxml) format, provides the homogeneity of representation demanded by the Aml applications and additionally, it is flexible enough for new data structures and information which could potentially be demanded by applications in the future.

2.4 Evolution Management

Previously we have shown how to address the heterogeneity problem regarding different technologies and techniques used, now we face another of the hardest problems of Aml application development, the management of software evolution reusing the developed application code.

In most applications the common functionality is usually encapsulated in functional units like classes or components, that can be (re)used using the typical object- and component-orientation techniques, but in some cases these techniques are not

enough to achieve an effective reutilization without making any special adaptation or using adaptors. Very often, achieving an effective reuse of these functional units is difficult due to code tangling and code scattering of those properties that cut across several functional units. To explain these two problems with an example, we will show part of the source code of PCOM [3], a component based middleware platform for the development of Aml applications, which encapsulates the middleware communication functionality. As does our approach, PCOM provides a component-based application. Software components used in Component Based Software Development (CBSD) are independent deployment units that can be reused in order to build new applications [13]. Components are especially useful for building applications adaptable to different environments since by a definition of a common interface, different component implementations implementing the same interface can be (re)used to build the same applications for different target environments. For example a GUI component with the same interface can have different implementations, such as a PDA or a mobile phone.

We are going to illustrate the *code entangled problem* by inspecting the code of the PCOM platform depicted in Figure 1. In this platform, the communication property is encoded inside a specific set of classes that are deployed with the application at compile time as plugins. Therefore, the encoding of the communication protocols is fixed once the application has been deployed. Figure 1 shows the *SimpleTransceiver* class that implements the sending of data to other PCOM applications using the TCP protocol, this class extends the *TransceiverPlugin* interface that defines all methods needed by a PCOM transceiver plugin. To obtain a more detailed view of PCOM see [3].

As we mentioned before, PCOM is a component based platform and the use of components eases the application development, but usually we can find some properties that crosscut one or more components in our applications. Some examples of this are the application code corresponding to the error handling, tracing or communication properties. If we examine carefully the source code shown in Figure 1, we can see this point illustrated. As a result, the source code of this class mixes up different application properties like communication protocol configuration and initialisation, message definition, message sending, application tracing and error handling. As already mentioned, such mixing of functionality in the same computational unit is known as the *tangled code* problem. Now we are going to show where these properties are found in the source code and the problems related to them. Specifically in Figure 1, we can see in (a) and (c) the communication protocol configuration property in the definition *HOST_IP*, *HOST_PORT* and *CONNECT_PERIOD* variables that are hard coded in the class. The code that implements this property defines some variables used to configure the communication protocol. This definition makes it impossible to change these values once the application is compiled and deployed because they are declared as *final*. Moreover the use of an ip and a port value limits the kind of communication protocols that can be modelled by this plugin implementation to those protocols that are able to manage these concepts (like UDP or TCP). This problem is even more evident in (b) where a concrete TCP class, *ServerSocket*, is declared, also in (f) where the *HOST_IP* and *HOST_PORT* values are retrieved and in (g) where the concrete communication protocol is initialised using a *Socket* class and the previously retrieved variables host and port, making

the reuse of this class for other protocols impossible. The two last code snips (f) and (g) can be seen as the communication protocol initialisation property.

Another property previously mentioned as message sending is found in (h). This property manages the actions of sending data to a remote application using the communication protocol. Notice that this property is heavily dependant on the data encoding used to represent the messages sent and received by the class and with the protocol used to send these messages. This is the message definition property that we found in different places in this class as in (d), (f), (g) and (h) that makes this class implementation very dependent on another application class named *Message*, which encodes the message data. Additionally we can see at different places in this code other properties such as for example application tracing in order to generate a log from the execution in (e), (f) and (i) or error handling to control applications errors and throw exceptions when needed in (d), (f) and (i).

```

public class SimpleTransceiver extends TransceiverPlugin {
    // The property identifier for host ips.
    private static final String HOST_IP = "IP";
    // The property identifier for host ports.
    private static final String HOST_PORT = "PORT";
    // Flag that indicates whether the plugin is running.
    private boolean active = false;
    // Thread that listens for incoming messages.
    private Thread receptor;

    // The server socket used for incoming connections.
    private ServerSocket sSocket;

    // The reconnect period
    private static final int CONNECT_PERIOD = 5000;

    /**
     * Transfers a message to a remote system.
     */
    public void transmit(Message msg) throws InvocationException {
        if (msg == null) {
            Logging.error(this.getClass().getName(), "Not message to transmit.", null);
            return;
        }
        if (msg.getProperties() == null) {
            Logging.error(this.getClass().getName(), "No properties available.", null);
            return;
        }

        String ip = (String) msg.getProperties().get(HOST_IP);
        Integer port = (Integer) msg.getProperties().get(HOST_PORT);
        if (ip == null || port == null) {
            Logging.debug(this.getClass().getName(), "No ip and/or port information available.");
            throw new InvocationException("No ip and/or port information available.");
        }

        try {
            InetAddress address = InetAddress.getByAddress(ip);
            Socket s = null;
            synchronized (this.connections) {
                String key = address + " " + port;
                s = (Socket) this.connections.get(key);
                if (s == null) {
                    s = new Socket(address, port.intValue());
                    this.connections.put(key, s);
                }
            }

            try {
                synchronized (s) {
                    s.getOutputStream().write(msg.getPayload());
                    s.getOutputStream().flush();
                }
            }

            catch (IOException e) { Logging.error(this.getClass().getName(), "IOException", e); }
            catch (IOException e) { }
            printStackTrace();
            throw new InvocationException(e.getMessage());
        }
    }
}

```

Figure 1. PCOM source code example.

As we can observe in the example, the functionality of the properties is mixed in the application code, creating undesirable dependencies among them. This is especially true in (f) where we can see how the communication protocol initialisation property is mixed with the tracing and the error handling properties. Note that if it is necessary to modify the implementation of any of the

previously mentioned properties, the *SimpleTransceiver* class must be re-implemented, modifying each point where the property is used and maybe performing changes in other application classes. And it will be even worse if we have to modify the message class because we will then have to modify every class that uses it, such as for example all classes implementing the *TranceiverPlugin* interface. In conclusion, the use of components to model the application functionality is good but it is not enough for handling the evolution of Aml applications without breaking the code of already implemented components. If we were able to extract this code from the application components and model it as independent entities, called aspects, we would be able to produce much more reusable and modular code. Moreover, we could completely replace the functionality enclosed in these aspects without modifying the rest of the application. Remember that composition, or weaving in aspect terminology, of components and aspects will be done in an unconscious way from the point of view of components, so that to apply a new property (or aspect) to an existing component will not affect its internal implementation.

With respect to the code scattering problem, some of these properties, such as tracing and error handling can be found scattered and replicated throughout several classes in the application. Extracting this code from the different classes and grouping it in an aspect will provide a simpler and more reusable code, because we will centralize all possible changes in the application in one specific place. For example, after the corresponding testing of an application, the tracing information should be removed from all the components that formed the application, an operation which is usually performed manually. Using aspects, we only need to change the weaving information of components and aspects that is normally specified outside them, without modifying any source class.

2.5 Application dynamic Adaptation

Previously we have shown how it is possible to solve the heterogeneity and evolution management problems in Aml applications in order to build truly reusable software. In this section we will show how this adaptability can be achieved even at runtime providing a solution for the application dynamic adaptation problem.

As we have mentioned before Aml applications are distributed and dynamic applications, as a result, they are executed in a continuously changing environment and they must be ready to react to the different situations that occur in it. The problem is that the number of possible situations, sometimes changes over time, and most of these changes were not originally considered during implementation. Moreover, usually these behaviours are to do with several different environment properties like communication, device discovery, device location, data encoding, etc, making it impossible to code in advance any possible reactions to any new environment requirements. It is not possible to foresee which responses an Aml application may give for unpredictable events in the environment, at implementation time. But it will be interesting to define an application able to incorporate new strategies or actions in response to some events, or add new strategies, without needing to be recompiled.

Normally, this adaptation is currently limited to the strategies coded in the application and in most cases all the possible alternatives or reactions to changes in the environment have to be foreseen by the programmers. For example in PCOM all possible

events affecting the application, and strategies for adapting it must be coded internally, as part of the application code, before this application is deployed. They can not then be changed without stopping and recompiling the application. In other cases as in Aura [9] all possible events are deterministic and the environment behaviour is closed. In both cases the evolution capacity of the application is almost zero.

The approach that we propose is to use the components and the aspects in order to code this behaviour separately as rules and actions interpreted by the platform using some logic like [11] or [14] and not code them directly as part of the application. Doing this we extract the adaptive behaviour from the application code allowing its reuse, and we provide a way of letting the application change its behaviour dynamically, even at runtime, simply by modifying this strategy definition and not the application.

3. ARCHITECTURE OF AOPAmI

Originally this work has been derived from our experiences developing a component and aspect based middleware platform for distributed applications [12]. We have obtained good results solving the adaptability and the software evolution problems by using components [13] and aspects [1] as first class entities and combining them using a weaving mechanism in order to build the applications. Using this approach we are able to change the application behaviour without modifying a single line of code. After developing this middleware platform, we thought about applying these advantages to a specific application domain that demands high flexibility and adaptability, as is the case of Aml environments, and at the same time solving the specific problems of this specific domain, hiding them in the middleware platform. The identification of aspects specific to Aml applications and the benefits of using them, was mainly done by studying the PCOM platform, see [5], [6] and [8]. Currently, we are exploring the possibilities of the middleware platform for Aml.

In the previous sections we have shown some of the most important problems related to the Aml environments and propose general solutions to address them. In this section, we will first show an overview of our AOPAmI platform proposal and then a detailed view of it and how these solutions are implemented.

In AOPAmI we use J2ME to implement the component and aspect-based platform to deal with every heterogeneity problem. First, at the hardware level we realized that providing a fixed platform implementation would not be enough to develop truly reusable Aml applications because not all devices will support an entire platform implementation. To handle this issue we decide to explore the concept of product lines [2] and its application to the CBSD. Additionally we took some ideas from the application of AOSD to product lines from [4] and this led us to think about how combine all of these elements. This was the motivation for the definition of device profiles in our AOPAmI platform.

We have defined a set of device profiles similar to J2ME profiles, but always bearing in mind that we are developing a product line of Aml platforms. These profiles specify the variable concerns of Aml platforms as a set of device properties, like available memory, device processing power or the communication technologies supported. By configuring the AOPAmI platform externally we can instantiate the most adequate platform implementation for each device taking into account its capabilities. To perform this selection, we based our decision on

the requirements about these parameters demanded by both the components and the aspects used to build the application. An example of a profile for a PDA is shown in Figure 2. Each profile is identified using a name, *PDA_Profile*, in this example. Also shown are the set of parameters that encode the device capabilities using a predefined vocabulary. Using these parameters we can decide in first phase, which components and system aspects will build our platform implementation and in the second phase which components and user aspects can be used by the application. Currently we are working on a tool to automate this task.

Note that this vocabulary is implemented using XML and thus it is easily extensible. Moreover, it provides quantitative information for some properties, such as for example the screen size, which can be used by automatic tools to select the most adequate components and aspects for building up the application. Obviously both components and aspects should provide this quantitative data in their descriptions in order to take advantage of device profiles. We will see below how this information is part of the component and aspect description files.

```

<deviceProfile name="PDA_Profile">
  <description>A simple PDA profile for DAOPAmI</description>
  <parameters>
    <parameter name="processor" unit="Mhz">200</parameter>
    <parameter name="floating point support">false</parameter>
    <parameter name="memory" unit="Mb">48</parameter>
    <parameter name="screenWidth" unit="pixels">200</parameter>
    <parameter name="screenHeight" unit="pixels">300</parameter>
    <parameter name="filesystem">FlashCard</parameter>
    <parameter name="filesystemSpace" unit="Mb">128</parameter>
    <parameter name="protocol">UDP</parameter>
    <parameter name="protocol">TCP</parameter>
    <parameter name="connection">Bluetooth</parameter>
    <parameter name="connection">WiFi</parameter>
    <parameter name="connection">IrDA</parameter>
    ...
  </parameters>
</deviceProfile>

```

Figure 2. AOPAmI Device Profile.

To handle heterogeneity at the communication level, AOPAmI model communication as an aspect that is always present inside the middleware platform. In the interest of providing an open-ended list of common platform services instead of a closed list as in component platforms, platform services are provided as a list of aspects. We call the common platform services system aspects. AOPAmI provides different system aspect implementations for modelling different communication primitives like UDP or TCP and technologies such as WiFi, IrDA or Bluetooth. Since system aspects encapsulate all these evolvable technologies, and the platform defines a way of adding and modifying system aspects by changing the platform profile, we can add or modify a communication technology by simply specifying different weaving information inside the platform profile. We will provide a detailed view of these aspects in the following subsections.

Finally, regarding heterogeneous problem, at the data interchange level our platform uses XML and binary XML to encode the information interchanged among applications. We should provide a homogeneous and extensible way to interchange data and the possibility of communicating with AOPAmI applications that supports XML data encoding. In the following subsections we will explain briefly how each of these design decisions impact upon and are integrated into the middleware platform.

With respect to the management of software evolution, we should point out that after examining several platform proposals for the AmI application development such as Aura [9] or PCOM [3], we have found that the common functionality in these systems is modelled, in a more or less homogeneous way, as components.

But as we mentioned earlier, the use of components alone is not sufficient for achieving good modularity and reuse of the developed code. Therefore, working on the basis of our own experience we have decided to combine the CBSE and the AODS paradigms in our platform to overcome this problem. To show this weaving mechanism is outside the scope of this paper and to obtain a detailed view of it we recommend the reviewing of [6].

As a consequence, in order to achieve better modularisation, reusability, adaptation and flexibility, we have divided the platform into two functional levels as is shown in Figure 3. The lower level, the platform level, is composed of a set of managers that implement the basic platform functionality that we will describe later. We provide a common interface for each of them and also different implementations in order to allow the adaptation of the platform to the specific device that will finally execute it. Additionally, we reduce the framework complexity splitting its functionality and we get a better application performance because we are able to disable those platform parts that are not used by the AmI application without affecting the rest of the platform functionality. The upper level is called the application level. At this level, the application is executed. These applications, composed by components and aspects, use the services offered by the platform level to interact with remote peer AmI applications. Additionally the platform provides a mechanism that in some cases performs static composition of components and aspects and when necessary it performs the composition dynamically.

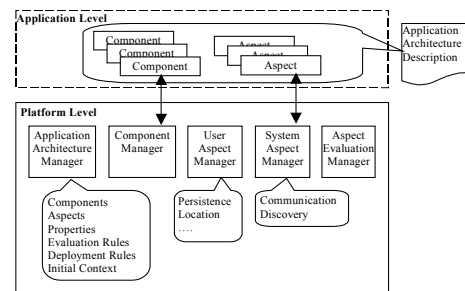


Figure 3. AOPAmI Architecture Overview.

Finally with respect to the dynamic adaptation problem, we again use aspects as a way of managing the changes, but our platform defines a special mechanism that allows us to extract the dynamic adaptation strategies and communication primitives from the platform source code. The way in which our platform solves this problem is outside the scope of this paper and it is shown in [6].

3.1 AOPAmI Layers

Having provided an overview of the platform, we will now describe the lower platform layer in detail. The description of the top layer is outside of the scope of this paper. As mentioned previously, in order to deal with the complexity of the application development process and to obtain better modularity, we have divided the platform complexity into five main modules, in order to achieve a better modularisation and support the evolution management in a simple and homogeneous way. But, in these AmI platforms we found that the name of the implementation class of components is hard coded as part of client components, creating undesirable dependencies between them. We address this issue by providing managers that choose the implementation

version that best matches the target device needs and restrictions. Now we are going to briefly describe each one of these managers.

Firstly, the Application Architecture Manager (or AAM) is in charge of loading the application architecture (or AA) definition stored as an XML file from the application resources or from a remote web server. Indeed, the whole platform can be downloaded and installed automatically in devices supporting Middlets. This file contains information about how instantiate components, aspects and the properties used by them. It also provides information about how the weaving process, previously mentioned, must be performed by the platform. Additionally, it contains information about where to deploy the components and aspects, using deployment rules, and finally, it describes which components must be initially instantiated when the application is started using the initial context rules. Additional information about the data contained in the AA can be found in [6]. Note, that AOPAmI will perform component and aspect composition by consulting the AA information file (previously loaded inside the AAM), so any change regarding application composition only entails the modification of an XML file.

Secondly, the Component Manager (or CM) is in charge of storing and managing the components lifecycle and state. It creates new components and destroys them using the information in the AAM.

Thirdly, the User Aspect Manager (or UAM), manages the application aspects lifecycle creating and destroying user aspects when needed. The information for doing so is provided by the AAM and these aspects are used in the upper layer. Some examples of these aspects are authentication or location.

Fourthly, the System Aspect Manager (or SAM), manages the platform system aspects, such as communication or discovery. This manager applies strategies when events or changes in the environment occur and are detected by these aspects [6]. The information used to define the strategies and how the system aspects are deployed and executed are defined in the AA file.

Finally, the Aspect Evaluation Manager (or AEM), is in charge of applying the composition rules when a *join point* is reached, in our case when a message is received or a component is created or destroyed by the application. This information is also managed by the AAM. To get a more detailed description of the AA see [6].

In order to understand how the platform uses these managers, we will explain the role that they play when an AOPAmI application is started. After starting the Middlet, the AAM is instantiated and the AA is loaded locally or from a web server. After loading the file, its contents are processed and the information is stored in AAM internal data structures. As a result, all information about how to create components and aspects, how to weave them and which system aspects must be started is accessible at runtime and can be easily changed externally to implementation classes.

While this information is processed the rest of the managers are instantiated and the system aspects are initialised by the SAM. Finally, when all the information is loaded, the CM is used to create the initial components that were specified in the AA file. If we need to apply any aspect to the component creation (one of the join points defined by AOPAmI) this will be detected by the AEM that will use the services of the UAM to perform this action. After loading the AA file and initialising the platform managers and System aspects, the application is started, the initial

components are created and the appropriate evaluation rules are applied. After the execution starts, user aspects are applied each time the application reaches a join point specified inside the AA file as aspect evaluation rules

3.2 System Aspects

In previous work [5], [6] and [8], after studying some AmI platforms we have identified several properties that are always present in AmI applications such as communication and device discovery and others that are present but only sometimes, such as device location, persistence and security. In AmI platforms the implementation of these properties is commonly intermingled with specific application code. Moreover, although an application will never need some of these properties, the middleware has them pre-installed and sometimes they waste precious resources such as the device location service. As we mentioned in previous sections, due to this coupling with the application code, it is very hard to modify the application if we have to replace or upgrade some properties when a technological change occurs (e.g. a new communication protocol appears). As a result, we have applied AOSD techniques to model these properties extracting them from the application code and some of them, specially the most used, were modelled as system aspects. These aspects are computational units executed continuously by the platform. They provide a common and well defined interface and the platform can manage several implementations of the same system aspects running concurrently. The System Aspects are located in the lower part of the platform, inside the SAM. The difference between implementing a platform offering the typical common services or implementing these common services as system aspects is that in the latter case the platform provides a mechanism for adding new system aspects when needed, so users do not have to wait for new releases of the platform implementing a new service.

Examples of system aspects are the discovery aspect that searches in the environment for other AmI devices executing the AOPAmI platform or the communication aspect that provides communication primitives for the platform. Each one of these aspects can be implemented according to different technologies. For example, one communication aspect implementation may model WiFi communications, whereas another may model a Bluetooth system, but both implementations can be used at the same time by the platform, or one can be replaced by the other, even at runtime, if they implement the same interface and are supported by the hardware. So, these aspects can be enabled, disabled or replaced at runtime by the platform. They commonly send events to the middleware platform when some change or problem in the environment occurs. For example a new device is found or a connection error happens.

Figure 4 shows the part of the AA file in XML which is in charge of defining the system aspects present in the application and its parameters. This definition can be seen as the list of specific system aspect implementations that will be instantiated for each platform version before the application starts its execution. This feature eases the modification of different application versions decoupling the application parameterisation from the application source code. Note that aspects are identified using descriptive role names for each *systemAspect* tag such as for example *Discovery* and they define the interface that must be implemented by the instances of this aspect. The aspect will be referenced in the

middleware platform by the role name instead of using the interface name, decoupling the platform from the specific implementation of the system aspect. The interface defines the set of common functionalities provided by the aspect. Each system aspect defines a series of implementations enclosed by *impl* tags. For each implementation we must indicate a *name*, an implementation class, *impl-class*, and the aspect implementation initial *status*. This status can be *ENABLED* or *DISABLED* and indicates whether the aspect implementation must be executed after loading the configuration. After specifying these mandatory tags, we can provide a set of parameters used to initialise the implementation aspect. In Figure 4 we define, for example, two parameters to initialise the *BluetoothDiscovery* implementation aspect. These parameters noted as *uuid* and *interval*, serve respectively to identify a Bluetooth device and to determine the search frequency of the aspect. Note that for the *UDP* implementation the set of parameters (*interval*, *readPort* and *writePort*) are different. Therefore, each implementation of the aspect is independent.

```
<systemaspects>
<systemAspect name="Discovery" interface="SystemAspect.DiscoveryInt">
<impls>
<impl name="Bluetooth" impl-class="System.Aspects.discovery.BluetoothDiscoveryImpl" status="ENABLED">
<parameter name="serviceName">BluetoothDiscovery</parameter>
<parameter name="uuid">1234567890</parameter>
<parameter name="interval">15000</parameter>
</impl>
<impl name="UDP" impl-class="System.Aspects.discovery.UDPDiscoveryImpl" status="DISABLED">
<parameter name="interval">9000</parameter>
<parameter name="readPort">5000</parameter>
<parameter name="writePort">5000</parameter>
</impl>
</impls>
<strategies>
<strategy name="strategy-1">
<onevent><event from="Bluetooth"> ConnectionError</event><event> ConnectionLost</event></onevent>
<execute script="XEXPR"> <href>strategy1.xml</href></execute>
</strategy>
</strategies>
</systemAspect>
</systemaspects>
<systemAspect name="Communication" interface="SystemAspect.CommunicationInt">...</systemAspect></systemaspects>
```

Figure 4. System Aspects Configuration File.

The system aspect definition does not end here. Additionally, we can define a set of strategies that must be applied when a system aspect sends events. This usually happens when something new occurs in the system. For example when a new device is found or the connection with another device is lost. These strategies are identified by a *name* and are activated when an event from a local aspect implementation is thrown. The list of events that triggers the strategy execution is indicated by a series of *event* tags enclosed by a *onevent* tag. Each *event* tag can optionally specify the name of the implementation aspect that sends the event. We do not need to know all the events thrown by aspects but only those specified in a specific strategy, which will be processed by the platform, the rest will be ignored. After detecting the event the platform will execute the script contained in the *execute* tag. This script can be expressed using any codification that can be supported by the device and that is indicated in the *script* attribute of the *execute* tag. Currently our platform provides an implementation of a subset of the XEXPR [14] notation, but we plan to provide support to other scripting languages and notations taking into account the limited computational resources provided by the AmI devices.

3.3 AOPAmI communications

In our previous work with CAM/DAOP we defined four communication primitives. These primitives were events, broadcasting messages, asynchronous messages and synchronous messages [12] that are the classical primitives supported by most distributed systems. In most cases all these communication

primitives are usually coded inside the application or the platform supporting the application. But when we think about AmI devices, we realize that not all these primitives will be available in all devices. This issue is especially true with devices that can not assure synchronous communications due to communication channel availability issues or simply because they do not have hardware and software capable of supporting all these primitives. Therefore, we have designed our AOPAmI platform to provide support only for those communication primitives actually supported by each specific device.

As a result, in our model we have decoupled the communication primitives from the middleware code and provide an alternative mechanism in order to handle this communication. This mechanism consists of, on the one hand, codifying the communication primitives in independent classes implementing the same interface, and on the other hand, we define for each output message of a component and outside the component code, the type of communication primitive that will be used. The latter definition is written in XML and provided as part of the application resources or retrieved from a remote web server as part of the AA file of the application. At the application start up this information is loaded and available so it can be changed even at runtime, thereby making our platform more dynamic and adaptable. An example of this file definition is shown in Figure 5 (b) along with a schematic view of the platform and an application running on it (a). In this figure, we indicate that when the *Component1* (*fromComponent* tag) sends an event (*onEvent* tag) identified as *event1* (*name* attribute), this event must be sent to the component *Component2* (*toComponent* tag) using a *message* primitive (*send* tag and *type* attribute).

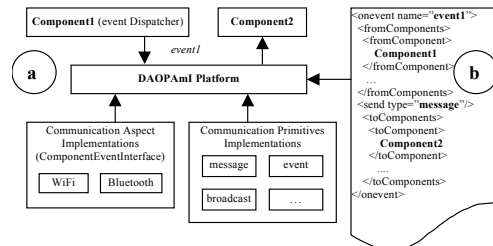


Figure 5. Communication primitives.

To implement this mechanism, each communication aspect implements an interface, named *ComponentEventInterface*, that allows it to process events implementing an *EventOccured* method. Whereas the components have to implement a *fireEvent* method and an *addListener* method to register the communication aspect in order to send these communication primitives. Figure 6 depicts the implementation of the *fireEvent* method from the component class. Additionally, the platform provides a *ComponentEvent* class that encapsulates the information needed to use this mechanism such as for example the source component, the event name and its parameters. As can be observed, this is the classic event listener pattern provided by Java. But unfortunately, in this case the J2ME implementation does not include a default implementation of this pattern. As a result, we have to implement it manually and add it to our middleware platform.

The advantages that this mechanism provides to the platform are obvious: firstly, the communication protocols (WiFi and Bluetooth in the example) are specified as independent classes. As a result, we can provide a concrete set of communication

protocols to the middleware without modifying the basic functionality provided by it and even add new protocols to the application simply adding new communication aspects and redeploying the application, without recompiling the source code. Secondly, components do not hold references to the class processing the communications or to the components that will receive them. Moreover, they do not know which communication strategy (message synchronous or asynchronous, event, broadcast, etc.) will be used to deliver the event. And thirdly, the information about how to send the messages is coded outside the application, and this information is available at runtime. As a result, we can change the application communication behaviour even while the application is running, increasing the evolvability of final applications once more.

```

void fireEvent(ComponentEvent evt) {
    Enumeration listeners = listener.listElements();
    while(listeners.hasMoreElements()){
        ComponentEventListener listener=(ComponentEventListener)listeners.nextElement();
        listener.EventOccurred(evt);
    }
}

```

Figure 6. Component fireEvent method.

Comparing this approach to other Aml platforms like PCOM or Aura, we see that all of them provide communication primitives that must be used as part of the application source code, it being impossible to change them without recompiling the source code. As a practical example of this we can observe Figure 1 where in (h) the communication primitive is message based and it is impossible to change it without recompiling the full application.

4. CONCLUSIONS AND FUTURE WORK

In this work we have shown some of the main challenges and problems that programmers must address in order to develop flexible and evolvable Aml applications. We have indicated the main problems and some possible solutions to them and we have presented AOPAmI as a platform for the development of Aml applications that implements these solutions. With the development of this platform we have proven that it is possible to apply CBSD and AOSD techniques to solve problems in specific application domains. The advantages of using a middleware platform to develop Aml applications are obvious, because the platform hides the specific domain complexity in the middleware saving the programmer from worrying about these details, providing a homogeneous environment where applications are developed and allowing interaction with other platforms that support XML encoding and interchange of data. Additionally the use of components and aspects allows us to develop much more dynamic, adaptable and evolvable applications.

Our future work is focused on providing new support tools for developing Aml applications using AOPAmI, and in particular, a tool for the automatic processing of device profiles. Additionally our intention is to integrate the application development process with MDA to be able to adapt existing application designs to our platform. Finally from the platform implementation point of view we are working on building more reusable components and aspects and at improving the existing ones, especially the system aspects and managers to achieve a better platform performance.

5. ACKNOWLEDGMENTS

This work is partially financed by IST-2-004349-NOE AOSD-Europe and the Spanish Ministry of Technology and

Science, CICYT, under grant TIC2002-04309-C02-02 and special acknowledgments to C. Becker and M. Handte for providing us with the PCOM source code. This code has helped us to identify specific aspects in the Aml domain.

6. REFERENCES

- [1] AOSD Web Site. <http://www.aosd.net>
- [2] Batory, D., et Al. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology*, Vol. 11, N° 2, (April 2002), Pages 191-214.
- [3] Becker, C., et Al. PCOM – A Component System for Pervasive Computing. *Second IEEE International Conference on Pervasive Computing and Communication. (PerCom'04)* (Orlando, USA, 2004).
- [4] Colyer, A., Rashid, A., and Blair, G. On the Separation of Concerns in Program Families. *Technical Report COMP-001-2004*, Lancaster University, Uk, 2004.
- [5] Fuentes, L., Jimenez, D., and Pinto, M. Towards the development of ambient Intelligence Environments using Aspect-Oriented techniques. *Aspects, Components, and Patterns for Software Infrastructure workshop (ACPSI, AOSD 2004)* (Lancaster, UK, 2004).
- [6] Fuentes, L., Jimenez, D., and Pinto, M. An Ambient Intelligent Language for Dynamic Adaptation. *Object Technology for Ambient Intelligence workshop (OT4AmI)* (Glasgow, Uk, 2005).
- [7] Fuentes, L., and Sanchez, P. AO Approaches for Component Adaptation. *Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'05)* (Glasgow, Uk, 2005).
- [8] Fuentes, L., Jimenez, D., and Pinto, M. Development of Ambient Intelligence Applications using Components and Aspects. *Ubiquitous Computing and Ambient Intelligence Conference, (UCAmI 2005)*(Granada, Spain, 2005).
- [9] Garlan, D., et Al. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, vol.1, no.2, pp.22-31, April-June 2002.
- [10] Kickzales, G., et Al, Aspect-Oriented Programming. *European Conference on Object Oriented Software Development (ECOOP'97)* (Jyväskylä, Finland, 1997).
- [11] OWL: <http://www.w3.org/2004/OWL/>
- [12] Pinto, M., Fuentes, L., and Troya, J.M. A dynamic component and aspect oriented platform. *Computer Journal*. 48(4): 401-420, 2005.
- [13] Szypersky, C. *Component Software. Beyond Object-Oriented Programming*, Addison-Wesley, 2002.
- [14] XEXPR: <http://www.w3.org/TR/xexpr/>
- [15] Young, T., and Murphy, G. Using AspectJ to Build a Product Line for Mobile Devices. *Demonstration in Aspect Oriented Software Development Conference (AOSD'05)*, Chicago, Illinois, USA, 2005.