Software Engineering of Pervasive Services

International Conference on Pervasive Services'06
Lyon, France, June 2006

Pascal Cherrier & Daniel Stern
France Telecom Orange
pascal.cherrier,daniel.stern@orange-ft.com

Holger Mügge
University of Bonn
muegge@iai.uni-bonn.de

Eric Tanter
University of Chile
etanter@dcc.uchile.fr

Wolfgang de Meuter
University of Brussel
wdmeuter@vub.ac.be

Abstract—The first workshop about Software Engineering for Pervasive Services [1] was organised during the International Conference on Pervasive Services [2] in Lyon, France, on June 2006. The program of the workshop included an invited talk, 5 presentations of the accepted articles and a discussion. This report gives the complete text of the short articles and summarizes the invited talk.

A. Introduction

The International Conference on Pervasive Services was organized in Lyon, France, on June 2006 [2], with the financial support of the IEEE, the Arizona University, the INSA Lyon, the INRIA Rhône Alpes, the région Rhône Alpes and the Université Lyon 2. Five workshops were organized on the following specific areas: security, health, multimodality, integration and software engineering. The SEPS workshop [1] proposed to work about Software Engineering of/for Pervasive Services. The slides and the articles are available on [1].

B. Program

One invited talk and 5 articles were exposed.

- Invited Talk: Ambient-Oriented Programming by Tom Van Cutsem, Vrije Universiteit Brussel
- Pervasive Applications Through Scripted Assemblies Of Services by David Svensson, Lund Institute of Technology
- QoS Specification in Ambient Intelligence Systems by Daniel Schneider, Fraunhofer Institute for Experimental Software Engineering
- Implementation of an AmI Communication Service Using a Federated Event System Based on Aspects by Daniel Jimenez Priego, LCC Malaga University.
- Engineering Pervasive Services for Legacy Software by Andrés Fortier, Universitat Politècnica de València
- Applying Models, Frameworks and Transformations by Javier Muñoz, Universitat Politècnica de València

BIBLIOGRAPHY


CONTENTS

-A Introduction . . . . . . . . . . . . . . . 1
-B Program . . . . . . . . . . . . . . . . . 1

Bibliography

References

1

II Invited Talk. Tom van Cutsem

II-A Introduction . . . . . . . . . . . . . . . 3
II-B Basic approach . . . . . . . . . . . . . . 4
II-C Assembly representations . . . . . . . 4
II-D Simple assemblies . . . . . . . . . . . . 5
II-D.1 A remote slide show assembly 5
II-D.2 Local device and service names 5
II-D.3 Global names of devices and services . . . . . . . . . . . . 5
II-D.4 Connections . . . . . . . . 6
II-D.5 Static-semantic constraints on the assembly . . . . . . . . . . . . 6
II-D.6 Dynamic constraints on the assembly . . . . . . . . . . . . 6
II-E Scripted assemblies . . . . . . . . . . . 6
II-E.1 GeoTagger as a scripted assembly . . . . . . . . . . . . . . . . . . . . . . . 7
I. INVITED TALK. TOM VAN CUTSEM

Ambient-Oriented Programming

Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, Theo D’Hondt
Vrije Universiteit Brussel
Email: {jessie.dedecker,tom.van.cutsem, smostinc,wdmeuter,tjdhondt}@vub.ac.be

Tom van Cutsem exposed a common work with Jessie Dedecker, Stijn Mostinckx, Wolfgang de Meuter and Theo d’Hondt. After having introduced Ambient Oriented Programming this work exposed Ambient Talk, a Java-J2ME based actor’s language which gives a distributed application support for Ambient Intelligence and Pervasive Services. More information about AmbientTalk may be found on [1].

II. DAVID SVENSSON

Pervasive applications through scripted assemblies of services

David Svensson
Görel Hedin
Boris Magnusson
Department of Computer Science, Lund University
Ole Römers v 3, 223 63 Lund, Sweden
Email: {david,gorel,boris}@cs.lth.se

a) Abstract: This paper proposes a technique for letting end users build pervasive applications by combining services on networked devices. The approach avoids relying on standardized service interfaces which are deemed too limiting, and instead makes use of migratable user interfaces and scripted combinations of services.

A. Introduction

In a world of pervasive computing, people will encounter a wealth of devices that offer software services in (typically wireless) networks. These services will often be tied to the particular devices, enabling control of and interaction with the devices in powerful ways. We argue that in this setting, interoperability is bound to become a major challenge.

A typical need that can be foreseen is the possibility to combine services, utilizing the combined functionality of several devices. Support for this can facilitate repeated use of a set of connected devices, and also provide inherently new functionality, not given by the individual devices themselves. However, the device vendors cannot be expected to foresee all possible combinations of services that can be demanded by future users; combinations possibly including future services and devices. This makes the usual approach, where one service interacts directly with another service through a standardized service-specific interface, too limiting. Instead, we propose that the combination of services should be separated from the services themselves, and that this combination is scripted, rather than programmed, to make it easy to adjust by end users. This will allow individual services to be developed independently of other services, but still be integrated into combined services.

In this paper, we describe a mechanism called scripted assemblies, that supports such combination of independent services. We have built an experimental system based on this approach, and tried it out on example scenarios. The ideas build on the MUI system [1], that supports remote control through migratable user interfaces. The work has been carried out within the EC-funded integrated project PalCom [2].

The rest of this paper is organized as follows. In Section II-B, we present our basic approach. Sections II-C to II-E go into more detail about non-scripted and scripted assemblies, dealing with issues in the assembly description language. Section II-F relates to other work in this area, and Section II-G
lists some things that remain to be investigated and developed. Finally, Section II-H concludes the paper.

B. Basic approach

Figure II.1 illustrates our approach to dealing with interoperability. There are two services, $A$ and $B$, located on two different devices, $D_A$ and $D_B$. The user wants to use and combine these two services. To accomplish this, the user has a third browser device that supports device and service discovery. Typically, the browser device is a handheld like a PDA or a mobile phone, but it could also be a general-purpose computer, e.g., a laptop. Each of the services $A$ and $B$ has a service description that can be migrated to the browser device, and rendered as a user interface there, in order to remotely control the service. In Figure II.1(a), the user interacts with $A$ and $B$ through these migrated user interfaces that are rendered on the browser device. This remote control mechanism is provided by a service $uiDisplay$ that runs on the browser device. The $uiDisplay$ service receives service descriptions from remote services, creates corresponding user interfaces that are shown on the screen of the browser device, and connects to the services over the network $\mathcal{I}$. In the figure, the service descriptions are the gray boxes with a small “hook”. After this setup process, there is a two-way, peer-to-peer communication between $uiDisplay$ and each of the remote services $A$ and $B$. When the user performs an action in one user interface, a command is sent over the connection to the remote service, which can react appropriately. When something happens at the remote service, typically as a result of physical interaction with the device on which the remote service lives, a command is sent in the other direction, typically updating status information in the user interface on the browser device.

The interaction through user interfaces solves parts of the problems with standardization of service interfaces. In this case, the human is in the loop, and can make intelligent interpretations of changes in the service descriptions (which show up in the user interfaces). When a new feature is added to a device, perhaps through an update of its firmware, a change in the service can be directly spotted and utilized. There is nothing in $uiDisplay$ itself that is tied to the specific service.

The other part of the figure, Figure II.1(b), shows how interaction with the services can be automated, realized through an assembly. The assembly is a service which performs much the same function as $uiDisplay$, but instead of rendering service descriptions as user interfaces, it coordinates the services according to an assembly descriptor (shown as a scroll). The assembly descriptor contains a specification of a combination of services, residing on different devices. Simple assembly descriptors just specify a number of connections between the services, while more advanced assembly descriptors also contain a script, which coordinates the interaction in a more fine-grained way. The assembly shown in Figure II.1(b) is of the latter type. It specifies how different events received from $A$ lead to one or more commands sent to $B$.

The key point in our approach is that interoperation is separated from services. This is what makes it possible to combine groups of services that were not created together, without restraining all of them to use standard service interfaces that were already established when the oldest service was created. In the case of user interface rendering, the user controls the interoperation directly, something which is good for trying things out in order to see how they work. This can be impractical, though, for more complex or long-lasting tasks. For these, the assembly works better. The goal is that assemblies should be possible to create and modify by end users. For this reason, we propose that assemblies should be created using a scripting language, rather than using a general-purpose programming language that would require programming skills. In some cases, the interoperation of two services might, however, require programming. For example, in order to convert between different kinds of real-time data streaming formats. To handle this, we propose that such problems are delegated to separate software services. Such software services need to be programmed, but can be used and combined in an assembly by an end user.

C. Assembly representations

Our model of an assembly consists of the following parts:

1) A set of devices
2) A set of services (on those devices)
3) A set of connections between those services
4) A set of offered synthesized services, generated by the assembly
5) Logic and scripts defining and constraining how the assembly should be deployed and executed.

Furthermore, an assembly can be fully bound, forming a composition of particular identified services on particular identified devices, or it can be in various ways partially unbound, e.g., to act as a template. In this paper, we will focus on fully bound assemblies.

It is useful to discuss the assembly from several different perspectives: the end user, the expert user, the tools manipulating the assembly, etc. In our system, we use the following important representations of the assembly:

1) An XML representation that is used for storing the assembly in a file system, and for moving or copying it between different devices.
2) A concrete syntax that is used in documents like this to show the same details as in the XML representation, but in a syntax more readable by humans. In principle, this concrete syntax could also be used by editing tools on laptops for creating or editing assemblies by expert users.
3) A representation as an attributed abstract syntax tree (AST) that is used internally by tools accessing and manipulating the assembly. We use the JastAdd compiler construction system for supporting AST programming [4], allowing the internal tools to add computations on the AST as modular aspects. The XML and concrete
syntax can trivially be unparsed from the AST representation, programmed as simple JustAdd aspects.

4) Tool-specific editing representations for displaying parts of the assembly information to end users, often in a visual way. E.g., a PalCom browser device can display the connections between services as lines between boxes, and provide graphical commands for composing or changing an assembly.

In this paper, we will use the concrete syntax when providing examples. In Section II-D we will discuss simple assemblies with devices, services, and connections, but without scripts. In Section II-E we will discuss how scripts can be added to capture the execution logic of an assembly.

D. Simple assemblies

1) A remote slide show assembly: Some simple assemblies consist only of a set of connections between services on particular devices, and have no logic of their own. As an example, consider an assembly RemoteSlideShow which composes a video projector, a laptop, and a PDA. Slides are sent from the laptop to the video projector, and the user controls the actions, next slide, previous slide, etc., from the PDA. The assembly itself resides on the PDA.

To illustrate the use of this assembly, consider the following scenario:

The user is a university professor who has weekly lectures in room E:1406 at the university. On the first lecture, she creates the assembly RemoteSlideShow by connecting her laptop, her PDA, and the video projector in room E:1406. This is done by a few visual commands on the PDA. On the PDA, she then uses commands to select the desired presentation, and to step through the slides. At the next lecture, she simply activates the existing assembly, which will then discover and connect the devices according to the assembly description. She can then immediately select the appropriate presentation and step through the slides.

2) Local device and service names: The RemoteSlideShow assembly is shown in Figure II.2 in concrete syntax. The assembly introduces a number of local device names: projector, laptop, pda; and a number of local service names: control, images, uiDisplay, and imageViewer. These local names are used within the assembly, e.g., to define the connections, and inside assembly scripts (discussed later).

Typically, the local names are taken from the logical names used in the device and service descriptions of actual devices. But refactoring to other names inside the assembly (for greater readability), would not affect the behavior of the assembly. These names are not used for binding the assembly to real devices and services. For such binding, the global names are used (see below).

In the proposed language, the service names are simple rather than structured. However, it could easily be generalized to support structured names. This would be useful since the services on a device are typically structured in a hierarchy, and it would be useful to keep that hierarchy in the local names of the assembly.

3) Global names of devices and services: Devices and services are identified by globally unique names. The globally unique names have an internal structure including a globally unique identifier, versioning information, and a logical name (which does not need to be unique). Typically, these names are quite long, and not intended to be very readable to a human. In the example in Figure II.2 we simply display them as “global-device-name” and “global-service-name”. These unique names are used for making it possible to reconnect an assembly to the same devices and services as used when the assembly was constructed. The assembly also has such a unique name itself (the value of “this”), with the same structure as a service name. The versioning information in the globally unique names is used by tools to make safe upgrades of an assembly when a service or a device has been upgraded. Note, however, that when an assembly is upgraded (rebound device and/or service), this has to be somehow visible to the user, and testing might be needed (unless it can be deduced that testing has already been carried out).
Two different devices, e.g., two projectors, can have (different instances of) the same service on them. To uniquely identify a service instance, i.e., a service on a particular device, the global names of the hosting device and the service are combined.

4) Connections: The connections part in the assembly specifies how the services in the assembly are connected to each other, using clauses on the following form:

\[ \text{providing-service on device-1} \rightarrow \text{customer-service on device-2} \]

Connections can be either data connections (unidirectional), sending messages from provider to customer, or control connections (bidirectional), where messages can go in both directions. For example, in Figure II.2, the connection \text{images on laptop} \rightarrow \text{imageViewer on projector} is a data connection where JPEG images are sent from the laptop to the projector. The connection \text{control on laptop} \rightarrow \text{uiDisplay on pda} is a control connection. As an example of messages over this connection, the PDA can send a message “next” to the laptop, to go to the next slide. The laptop will then send a status message back to the PDA, showing the name of the currently shown slide.

MIME types are used for specifying the types of connections. For example, the type of the \text{image-imageViewer} connection is \text{image/jpeg}, whereas the type of the \text{control-uiDisplay} connection is \text{application/x-palcom-control+xml}. These types are not explicitly visible in the assembly, but belong to service descriptions that are available for each device through the discovery protocol. The service description also classifies a service as being either provider or customer.

5) Static-semantic constraints on the assembly: There are certain semantic constraints on how assemblies may be constructed. The local names should be declared and used correctly. E.g., two devices named the same way is forbidden, using an undeclared name is forbidden, using a device name where a service name is expected is forbidden, etc. This boils down to normal name and typechecking rules similar to those in simple programming languages.

6) Dynamic constraints on the assembly: There are additional semantic constraints that can be checked only dynamically, i.e., when trying to activate the assembly:

- **Device bound.** When activating an assembly, the device declarations will be bound to descriptions of actually discovered devices. Naturally, it may be the case that it is not possible to discover a given device. It might be broken, turned off, not within range, etc. The operation of the assembly may then be limited for the moment.

- **Service bound.** Even if a device is bound, it is not guaranteed that all its services are available. Some services may be down, depending on the state of the device. It might also be the case that when an assembly is changed so that a device declaration is rebound to another device, the new device does not have all the declared services, and this is then flagged as errors or warnings. The tools can then guide the user in trying to rebind to another service on the same device, or possibly to a service on another device.

- **Connection well formed.** If the services of a connection are bound, it is checked that the connection is well formed. I.e., the service declared as the providing service should indeed be specified as a providing service in its service description, and similarly for the customer service. Furthermore, the MIME types of the provider and the customer should match.

E. Scripted assemblies

In the RemoteSlideShow example, the assembly simply connects existing services directly to each other. A more advanced assembly can itself receive and send messages and perform actions internally. These actions are written in a simple script language that can be used by an end-user. In the present experimental system, the script is edited as text, but in future versions, we plan to provide visual tools for editing the scripts. If the internal logic is more complex than the script language can handle, parts of the logic can be delegated to new software services, programmed in a general-purpose language.

Below, we extend the assembly representation to include a scripting possibility. The basic idea is that the assembly can be

```
assembly RemoteSlideShow {
  this = global-service-name;
  devices {
    projector = global-device-name;
    laptop = global-device-name;
    pda = global-device-name;
  }
  services {
    control on laptop = global-service-name;
    images on laptop = global-service-name;
    uiDisplay on pda = global-service-name;
    imageViewer on projector = global-service-name;
  }
  connections {
    control on laptop -> uiDisplay on pda;
    images on laptop -> imageViewer on projector;
  }
}
```

Fig. II.2. A simple assembly.
connected to other services to receive and send messages. The body of the script is an event handler that receives messages from other services and acts upon them. The possible actions (supported so far) are to send messages to other services and to store values in variables local to the script.

1) GeoTagger as a scripted assembly: GeoTagger is one of the end-user scenarios studied in PalCom, see Figure II.3. It is an application intended for use by landscape architects. The idea is that photos taken with a camera should be automatically tagged with the current GPS coordinates and stored in a backend database on a laptop. This application is realized as a scripted assembly running on a handheld PDA. The assembly combines and coordinates services running on the camera, the GPS device, and the laptop.

Figure II.4 shows the scripted assembly. In the event handler, clauses are written as

\[
\text{when message from service on device} \{ \\
\text{actions} \\
\}
\]

where the actions can access data in the message, send new messages to other services, and perform simple computations (assignments of local variables).

The service coordStuffer on the PDA device is a software service that can receive an image in JPEG format, and a GPS coordinate, and which sends out an image tagged with the GPS coordinate. This is a typical example of a computation that is too complex to express directly in the scripting language, and that is instead implemented as a software service.

As shown in the example, the assembly interacts with other services by receiving and sending messages. Thus, the assembly implicitly plays the role of a service that connects to the other services. The “this” expression used in the assembly script refers to the assembly itself viewed as a service. Received messages that are not listed in the event handler are simply ignored.

2) Additional constraints on the assembly: The introduction of the script in the assembly makes it possible and necessary to check additional constraints, statically and dynamically. In the static part, the name and type analysis is extended to the local variables. In the dynamic part, it is checked that the bound services actually have the incoming and outgoing messages used in the script, with the appropriate message structure.

3) Loopback mechanism: It might be the case that the assembly is located on the same device as some of the other services. A loopback mechanism is used which allows the assembly to communicate in the same way with these services as with services on other devices, without causing any messages to go out unnecessarily on the network.

The loopback mechanism is used also if the assembly connects two services on the same device: the network is transparent, and messages between services will only go out on the network if the services are on different devices.

Note that it is often the case that services on the same device are tightly bound and communicate with each other directly (not via an assembly). For example, when taking photos with a digital camera, the photos will be stored locally on the camera. This process is a bottleneck and needs to be carried out as efficiently as possible, to allow pictures to be taken at high speed.

Assemblies for connecting services on the same device are useful when the services are more unrelated, i.e., when they could in principle be located on different devices, but just happen to be located on the same device.

4) Moving the assembly?: For a scripted assembly, its location can dramatically affect the efficiency. For the GeoTagger, there will be large messages sent that include JPEG images. Suppose the assembly is located on the PDA (a natural choice since an assembly interpreter will need some kind of general-purpose platform to run on). In this case, JPEG images will be sent from the camera to the PDA, coordinates added to the image on the PDA, and the “stuffed” image is sent to both the camera and the laptop backend.

Clearly, if the coordinate stuffer and the assembly were moved to the camera or to the laptop, network traffic would be substantially reduced. It might be possible to move them to the camera if it is sufficiently advanced to serve as a general-purpose software service platform. And moving them to the laptop should be possible, but then the assembly would rely on the laptop which might be heavy for the user to always carry with him.

Note that if the assembly is moved to another device, its script does not need to change. There is nothing in the assembly script that makes it depend on its own platform.

If the assembly and coordinate stuffer are moved to the camera or laptop, it might still be the case that the user would like to control the assembly from the PDA, e.g., to activate it. Future versions of our system will support this by using an uiDisplay service for remote control of assemblies.

F. Related work

The scripted assemblies we propose are related to W3C’s Web Services Choreography Description Language [5]. WS-CDL choreographies are expressed in an XML language, and govern peer-to-peer interoperability between a number of services. Like our assemblies, the choreographies are external to all the participating services. One thing that differs is the context. WS-CDL is intended for E-business, taking place between Web services on the Internet. There is no notion of physical devices, which are important in our approach, and in pervasive computing in general. The purpose of WS-CDL is also not on keeping interoperability between services when facing service interface changes. Instead, the choreography is more like a contract that is decided on before a business relationship is started, making it possible for all parties to keep the internals of their services private.

Another closely related project is Obje at PARC [6]. Obje targets the same basic problem, and seeks to enable interoperability without relying on domain-specific standards. A difference is that Obje builds on mobile code. Using mobile code, in the form of a proxy object that is distributed to clients and executed there, services are able to “teach” clients how to communicate. This way, it is possible to let users
combine their clients with new services, some of whose features were unknown at the time the clients were written. There is also a possibility to let the proxy object generate a user interface, giving a situation similar to that of Figure ??(a), where the proxy object corresponds to our service description. Another difference, though, is the way of programmatically interfacing the proxy objects. Obje proxy objects are (Java) code, which requires the capability of running Java on clients, and their interfaces are so called meta interfaces, offering only very generic operations, such as reading a chunk of data. In contrast, our service descriptions are distributed as XML, which can be handled on almost any device, and they contain domain-specific operations: the operations are invoked by the user through a user interface, or by the assembly script. There is no concept in Obje corresponding to the assembly. Instead, the user directly connects components written in Java.

Cooltown at HP Labs [7] is an early pervasive computing project, whose target is to bring the Web to things in the physical world. By embedding wirelessly accessible web servers into things, it is possible for a user to interact with them in a Web browser on his handheld device. It is also possible to connect one device to another, by sending a URL to one of the devices, identifying a resource on the other. There is nothing domain-specific in the Web protocols involved, so this can be seen as a way of achieving basic parts of the interoperability we look for. But, apart from the client-server model being inherent in the interaction between Web clients and servers, one big difference is that our assemblies can define other aspects of a service interoperation than just pure connections.

Jini and UPnP are important technologies for network services. Jini [8] is tied to the Java programming language, and clients interact with services through proxy objects, distributed to the clients at discovery time. Our objection, also stated by Obje, is that this approach requires the interfaces of the proxy objects to be standardized at the domain level. To partly overcome this, there is a framework for user interface services built on top of Jini [9]. But, still, the tight connection to the Java language makes it inconvenient to build assemblies on top of Jini. UPnP [10] is not tied to Java, or to another programming language: devices and services are described using XML. But the focus in UPnP is on standardization of device types at the domain level. There are standards for devices such as printers, scanners, lighting controls, and digital security cameras, among others [11]. Therefore, UPnP is not directly usable as a platform for assemblies, either.

G. Future work

In our continued work on scripted assemblies, we will look into a number of issues including synthesized services, binding of services, message types, and service versions. Synthesized services are services that are offered by the assembly itself, allowing control of the combination of services, rather than of each service individually. A simple case of a synthesized service could be to collect the most important parts of the participating services’ interfaces into one interface, for convenience. Another case could be to have an interface for changing the activity state of the whole assembly.

So far, we have considered only fully bound assemblies where each service declared in the assembly is bound to a specific service on a specific device. We will look into more elaborate support for service bindings. One example is to investigate cases where an assembly can be functional without all services being present. This may give degraded, but acceptable, functionality of the assembly. In other cases, it may be enough with one out of a set of services for full functionality. There are also possibilities for experimenting with partially bound assemblies, where the identity of a device or service is not filled in, but can be specified later, perhaps when the assembly has been moved into a new context. In
assembly GeoTagger {
    this = global-service-name;
    devices {
        gps = global-device-name;
        camera = global-device-name;
        backend = global-device-name;
        pda = global-device-name;
    }
    services {
        gps on gps = global-service-name;
        photo on camera = global-service-name;
        storage on camera = global-service-name;
        display on camera = global-service-name;
        coordStuffer on pda = global-service-name;
        photo_db on backend = global-service-name;
    }
    connections {
        gps on gps -> this;
        photo on camera -> this;
        storage on camera -> this;
        display on camera -> this;
        coordStuffer on pda -> this;
        photo_db on backend -> this;
    }
    script {
        variables {
            text/plain latestReadableCoordinate;
            text/nmea-0183 latestStandardCoordinate;
        }
        eventhandler {
            when position from gps on gps {
                latestReadableCoordinate = thisevent.WGS84;
                latestStandardCoordinate = thisevent.NMEA-0183;
            }
            when photo_taken from photo on camera {
                send show(latestReadableCoordinate) to display on camera;
                send sendme_photo() to storage on camera;
            }
            when photo from storage on camera {
                send sendme_stuffed_image(latestStandardCoordinate, thisevent.Photo)
                to coordStuffer on pda;
            }
            when stuffed_image from coordStuffer on pda {
                send store_photo(thisevent.Image) to photo on backend;
                send store_photo() to storage on camera;
            }
        }
    }
}

Fig. II.4. A scripted assembly.

relation to this, versioning of assembly descriptors becomes important.

Currently, we demand exact matching of MIME types for connecting services. However, we will investigate the use of subtyping to allow services to be connected where the types match only partially.

H. Conclusions

This paper has presented scripted assemblies as a technique for letting end users combine services, and for letting them control the cooperation between services in a script. The assembly concept allows the interoperation between services to be separated from the services themselves. As a consequence, it is possible to adjust aspects of the interoperation at a later time, without re-programming the services, and to incorporate services with different, or changed, interfaces, by manipulating the assembly only. We see this as a way of easing interoperability in pervasive computing systems.

In the paper, the current language of assembly descriptors has been presented, exemplified by scenarios from the PalCom project, and possibilities for future development and experimentation have been discussed.

BIBLIOGRAPHY

QoS Specification in Ambient Intelligence Systems

Daniel Schneider, Michalis Anastasopoulos, Joachim Bayer and Martin Becker
Fraunhofer Institute for Experimental Software Engineering
Fraunhofer-Platz 1,
Christian Webel
Computer Science Department
University of Kaiserslautern
Erwin-Schrödinger-Str.
D-67663 Kaiserslautern, Germany
Email: {Daniel.Schneider, Michail.Anastasopoulos, Joachim.Bayer, Martin.Becker}@iese.fraunhofer.de, webel@informatik.uni-kl.de

a) Abstract: Quality of Service (QoS) is a central characteristic of human-centric Ambient Intelligence (AmI) systems, since it has a strong relation to the degree of satisfaction with the services provided to users. Hence, AmI applications must be able to describe their QoS needs as well as capabilities so that the corresponding infrastructures can match the latter to the preferences of users. Current QoS specification techniques are concentrated on special domains like communication systems and do not address special aspects of the AmI field like high dynamics. In this paper we, elaborate the special AmI requirements posed to QoS specifications and describe an initial approach towards a solution.

A. Introduction

Pervasive computer systems as described by Weiser [1] are characterized mainly by their non-invasive presence throughout the physical environment of human users. Ambient intelligence systems (AmI) [2] follow-up on this human-centric vision and enhance it with the idea that computer systems are able to adapt themselves proactively to the demands of the users and their environment. This idea is subsumed under the term adaptivity, which means a combination of changeability and intelligence. In other words, AmI systems must be open to changes at run-time, and these changes are performed under control of an effective reasoning about user, environment, and system context.

The openness for changes is twofold: AmI systems, including hosting infrastructures as well as applications, must recognize contextual changes on the one hand and must be able to react by changing their own internal behavior (i.e. application logic) accordingly on the other hand. The recognition and assessment of changes requires sound specification techniques that enable the description of contextual information and therefore the identification of changes therein. Providers of such specifications are all kinds of services that sense and publish contextual information. Providers are also AmI applications that publish their (changing) capabilities and
requirements. Consumers are infrastructures that host AmI applications and aim at matching service requestors with adequate service providers. Consumers are also other AmI applications that want to access contextual information for the sake of run-time adaptivity.

Specification techniques must also support reasoning, which is the second component of adaptivity. To this end, specification techniques must bear attributes of description logics [3]. In other words, they must enable the structuring and formalization of information in the various subdomains of AmI applications. This also entails the formal description of relations between information on different abstraction layers, ranging from the user interfaces down to the hardware devices.

The continuous satisfaction of user needs is the main reason for adaptivity in AmI applications. Naturally, there is strong correlation between user satisfaction and the perceived quality of services provided by AmI systems. This is especially true when it comes to the reproduction of time-based media (i.e., media that require timely delivery and processing).

The relation between Quality of Service (QoS) and adaptivity is clear: QoS depends on numerous factors in a service landscape and especially on the presence and sharing of resources. Moreover, the user’s QoS perception can be broken down to various cascading quality attributes. In other words, QoS is subject to frequent change as well as to sophisticated reasoning.

Due to the inherent scarcity of resources (like bandwidth, energy, processing power, and memory) in an AmI environment, the systems need to manage their resources in a QoS-aware way. That means that services must specify their QoS requirements and their related demand for resources in order to provide a basis for corresponding QoS mechanisms. Furthermore, it is essential to deal with QoS in a holistic manner. More specifically, QoS has to be considered from end-to-end and from user layer down to the resource layer.

Hence, the need of sound QoS specification techniques in the field of Ambient Intelligence becomes apparent. In fact, QoS specification can be seen as an indispensable complement of functional service specifications (i.e., features). Current QoS specification techniques are concentrated on special domains like communication systems and do not address special aspects of the AmI field like high dynamics. In this paper, we elaborate the special AmI requirements posed to QoS specifications and describe an initial approach towards a solution.

The paper is structured as follows. Section II-B introduces the reader to the QoS terminology and related work. Section III elicits the requirements posed to a specification technique and Section IV provides a metamodel for fulfilling these requirements. Section V describes how a QoS description can be integrated with a service description. The paper concludes in Section VI.

B. Quality of Service

The term Quality of Service (QoS) was coined in the distributed systems domain. The ITU-T defines Quality of Service (QoS) as the "collective effect of service performance that determines the degree of satisfaction of the user of that service" [4]. In more abstract terms, QoS determines how well a service performs. This is in contrast to functional properties that determine what a service does. However, in order to support QoS, a system has to meet certain prerequisites by means of QoS specifications and corresponding QoS mechanisms.

1) State of the Art in QoS Specification: Generally, there are two different approaches to QoS-aware system design that fundamentally affect the nature of a corresponding QoS specification. The first one is a static approach, in which QoS contracts are already negotiated at development time. The second one is dynamic and hence requires that QoS contracts can be negotiated at run-time. As the first approach is not applicable in the context of AmI systems due to the inherent dynamics, we focus on the latter in this paper.

Prerequisite to the dynamic negotiation of QoS contracts is an explicit and computable specification of a service’s required and provided quality attributes. To this end, the QoS specification of a service complements the specification of its functionality. The selection, composition and deployment of services at run-time must consequently be based on both, the functional and the QoS specification. According to the different requirements posed by different system designs, user preferences, environmental conditions, hardware, networking, operating systems and applications, numerous QoS specification techniques emerged over the last few years. Recognizing the importance of a taxonomy for QoS specification techniques [5] presents a classification based on the layer of the specification technique and subsequent refinements. The layer of the specification technique describes where it belongs in the end-to-end architecture:

- **User layer**: At user layer, the specification of QoS is based on the user’s perception. Thus, a user layer QoS specification abstracts from the actual technical details.
- **Service layer**: The perceivable user layer quality has to be translated to more concrete and technical attributes like number of frames per second and the frame resolution. However, those application layer attributes are still hardware and platform independent.
- **Resource layer**: Eventually, the service layer properties have to be mapped on attributes of the resource layer like bandwidth, jitter, etc. The values of the resource layer attributes are inherently hardware and platform dependent.

The QoS specification techniques on the different layers are then further classified based on their general properties. For the application layer, for instance, the classification is based on seven different paradigms: script-based, parameter-based, process-oriented, logic, markup-based, aspect-oriented, and object-oriented. For the resource layer, they distinguish coarse grained and fine grained QoS specifications [5].

C. QoS Specification Requirements

1) Requirements Elicitation: In the AmI domain, it is essential to follow a holistic QoS approach. In other words, QoS has to be considered:
(a) from one application service to another (end-to-end),
(b) across abstraction layers (e.g. from the user down to the resource layer).

Hence, a QoS specification technique has to provide sufficient support by means of appropriate concepts, syntax and semantics. Given the fact that many of the existing approaches arose from other domains or are focused on a specific layer of abstraction, finding appropriate candidates for the needs of AmI systems is not trivial.

Jin and Nährstedt [5] used expressiveness, declarativity, independence, extensibility and reusability as criteria for the evaluation of the surveyed specification language [5]. Those criteria constitute general requirements that apply to QoS specification techniques in any context. We have elaborated them into the following criteria:

1) **Expressiveness**: QoS specification techniques must provide adequate language constructs to express all relevant quality attributes, e.g. required resources, policies, etc.
2) **Declarativity**: QoS support is a cross-cutting concern that cannot be dealt with locally. Thus, services should declaratively specify required and provided QoS instead of how this is to be achieved.
3) **Independence**: For the sake of modularity, the QoS specification technique should be independent from any functional specification technique. Furthermore, the QoS specification technique must be independent from any technological platform.
4) **Traceability**: Even though functional and QoS specifications are independently modelled, the specification technique must enable explicit linking between the two.
5) **Reusability**: A QoS specification technique should foster reuse.
6) **Extensibility**: The specification technique should be easy to extend.
7) **Transformability**: A QoS specification should be machine readable and provide for the automated generation of specific QoS mechanisms (i.e. design decisions that achieve QoS, cf. [6]) and the run-time representation of the service specifications.

Further requirements can be identified by investigating specific characteristics of AmI systems. The latter can be illustrated with the following example:

Imagine the user of a video streaming application requesting a video stream at a specific quality (e.g. "good"). The application knows how to map this abstract quality characterization to ranges of acceptable qualities of underlying services. It also knows what policy to follow if underlying services do not provide adequate QoS. This information is submitted to the infrastructure as a look-up request. The infrastructure has then to match the request to corresponding services thereby following the declared policies.

Services in turn describe the mapping of their provided QoS to low-level resources. This information is also used by the infrastructure for eventually selecting a service or a service composite (i.e. service aggregate consisting of many other services) that suits the user input as well as the resource situation. This example is illustrated in Figure III.1.

![Fig. III.1. Services and QoS Specifications](image)

Given the above example and experiences we have made in the context of this work we can derive four additional requirements:

8) **Type-Safety**: Typed QoS attributes are essential for type checking whenever service specifications of different services (from different providers) have to be composed.
9) **Composition**: When services are composed at run-time (i.e. sequential and/or parallel composition) this affects also their QoS specifications.
10) **Adaptation**: The specification technique has to support the definition of adaptation behaviour. To this end, it must provide constructs to define ranges of qualities, guards and policies defining the adaptation related behaviour.
11) **Mappings**: The specification technique must support associations between quality characterizations across different abstraction layers.
• A video telephony service enables communication between elderly persons living in their home and relatives. In this case attributes that relate to media quality and communication system are most important. This includes: Frame Rate, Screen Resolution, Colour Depth, Compression Degree (Audio and Video), Audio Resolution, Sampling Rate, Audio/Video Synchronization, Frame Loss, Bandwidth, Delay, Jitter and Packet Loss.

• The living assistance system collects a series of data about the every-day behaviour of elderly persons. These data are highly confidential and the access to the database is allowed only to authorized service personnel. This scenario yields attributes relating to transactional behaviour and security. The transactional quality attributes are based on the ACID principle while the security attributes are based on the main drivers of security: Confidentiality, Integrity, Accountability, Authenticity, Consistency, Isolation and Durability. (Availability is deliberately omitted since it will be considered later on in this report).

• Contextual data are acquired with the help of small sensor nodes. They are usually battery-powered and have scarce resources in terms of CPU, memory and communication bandwidth. The relevant quality attributes in the third scenario obviously strongly relate to the required resources: Energy, Memory, CPU Utilization, Actuality, and Precision.

Apart from the attributes that have been derived from the above scenarios, some additional properties are to be specified in order to provide adaptive QoS in a dependable manner. The level of service (i.e. deterministic, predictive, and best-effort) specifies the degree of end-to-end resource commitment [6].

In other words, the level of service facilitates the distinction between hard and soft QoS guarantees. The specification of adaptive behaviour requires a further attribute. An adaptation policy defines the behaviour of the infrastructure, with regard to the service, on a changing resource situation ("Renegotiation when resources deteriorate/improve" e.g.) [8].

To this end, two further properties are of interest. Priority allows expressing a service’s rank compared to other services. Given two services with best-effort guarantee the one with the higher priority may be treated in favor. With relative importance it is possible to weight different qualities [8]. As an example, a user requesting data from a sensor node might prefer precision over actuality. However, those two will not be considered as separate quality attributes. It is sufficient to specify priority as a property of the level of service quality attribute and relative importance as a property of quality attributes in general.

The reliability of a service is also related to the quality that is perceived by the user. Thus, we add Availability and Mean Time Between Failures (MTBF) as additional quality attributes. While MTBF is self-explanatory, availability can be defined as "the probability that the system is operating at a specified time t." [9]. Availability is commonly related to both security and reliability. However, according to the definition of availability it is obvious that it is different in nature compared to the other security related attributes. Thus we decided to introduce it as reliability attribute.

It must be noticed that we do not address user layer QoS attributes in this paper. The main challenge regarding user layer attributes is to define appropriate mappings that reflect the actual user’s perception and is therefore out of the scope of this paper.

3) Initial classification of Quality Attributes: The following table summarizes the quality attributes collected above and categorizes them initially according to their relations to application services and underlying resources. The further classification corresponds to related work performed in [5] and [10] and is based on the quantitative or qualitative nature of attributes.

Quantitative QoS attributes are those that have quantifiable values. They are directly related to a specific parameter or property of the system. Thus, the parameters and properties are to be monitored and managed in order to facilitate (re)negotiation of QoS contracts, adaptation within the specified ranges and QoS feedback.

In this regard, qualitative QoS attributes are more abstract since they do not depend on a specific parameter of the system. They comprise policies that allege the behavior of the system with regard to a given service. As an example consider a specific level of service, an adaptation policy or a security policy. In each of those cases the system has to yield a specific behavior to cope with the service demands.

This leads to the final classification shown in Table 2.

D. QoS Specification Metamodel

1) Common Metamodel: In the following figure a metamodel is provided realizing the overall requirements analyzed in the previous section. This model is referred to as common because it is not bound to any case studies and can therefore be applied generally for the definition of QoS specification languages. The metamodel introduces the class QoS Profile, which practically captures the provided and required QoS. The latter is described in terms of quality attributes like the ones of the previous section.

2) Specialized Metamodel: With the given set of QoS attributes it is possible to derive further classes of attributes according to their role and their properties. To this end, also
Table II
Classification in Quantitative and Qualitative Attributes

<table>
<thead>
<tr>
<th>QoS Layers</th>
<th>Quantitative</th>
<th>Qualitative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Frame Rate, Screen Resolution, Colour Depth, Compression Degree (Audio and Video), Audio Resolution, Sampling Rate, Audio/Video Synchronization, Frame Loss, Actuality, Precision, MTBF, Availability</td>
<td>Confidentiality, Integrity, Accountability, Authenticity, Atomicity, Consistency, Isolation, Durability, Level of Service, Adaptation Policy</td>
</tr>
<tr>
<td>Resource</td>
<td>Bandwidth, Delay, Jitter, Energy, Memory, CPU Utilization, Packet Loss, MTBF, Availability</td>
<td>Level of Service, Adaptation Policy</td>
</tr>
</tbody>
</table>

We further classify the quantitative quality attributes into Reliability, Performance and Resource. The Reliability class subsumes all reliability related quality attributes. Those may occur on both, the service layer and the resource layer. To account for the different abstraction levels, found in the remainder of the service layer and the resource layer quality attributes respectively, we classify those quality attributes into Performance and Resource attributes. Accordingly, the Performance attributes belong to the service QoS layer and directly determine the performance of the service. Those attributes are mapped on corresponding attributes of the resource layer, which are subsumed by the Resource class. Thus, the set of quantitative quality attributes of a concrete QoS specification profile may include certain reliability attributes and either performance attributes or resource attributes.

As for the qualitative QoS attributes, we further classify into Level of Service, Adaption Policy, and Optional Policy. Each instance of a QoS profile needs to specify a Level of Service and an Adaption Policy since both are essential to the negotiation process and the behaviour of the infrastructure with regard to the corresponding service. As already described above, priority is not considered as a quality attributed but assigned to level of service as an optional property. In case of non-deterministic level of service it may be used to add a relative weight to the service’s demands. The Optional Policy class subsumes optional policies that may be added to a QoS specification: Confidentiality, Integrity, Accountability, Authenticity, Atomicity, Consistency, Isolation, and Durability. The specialized model is depicted in Figure III.3.

![Common QoS Specification Metamodel](image)

Fig. III.2. Common QoS Specification Metamodel

![Specialized QoS Specification Metamodel](image)

Fig. III.3. Specialized QoS Specification Metamodel

the work of Liu et. al. [8] and Sabata et. al. [10] is a valuable input.

We further classify the quantitative quality attributes into Reliability, Performance and Resource. The Reliability class subsumes all reliability related quality attributes. Those may occur on both, the service layer and the resource layer. To account for the different abstraction levels, found in the remainder of the service layer and the resource layer quality attributes respectively, we classify those quality attributes into Performance and Resource attributes. Accordingly, the Performance attributes belong to the service QoS layer and directly determine the performance of the service. Those attributes are mapped on corresponding attributes of the resource layer, which are subsumed by the Resource class. Thus, the set of quantitative quality attributes of a concrete QoS specification profile may include certain reliability attributes and either performance attributes or resource attributes.

As for the qualitative QoS attributes, we further classify into Level of Service, Adaption Policy, and Optional Policy. Each instance of a QoS profile needs to specify a Level of Service and an Adaption Policy since both are essential to the negotiation process and the behaviour of the infrastructure.
• **Relative Importance**: Indicates the relative importance of the quality attribute compared to other ones. This is of interest in the context of (re)negotiation and adaptation.

As an example for concrete quality attributes and their actual needed properties consider the following 8-tuple:

- **FrameRate** = (frame rate, "video stream xy", #frames/sec, absolute/average, N+, ratio, increasing, 5)
- **CPU** = (CPU utilization, "cpu probe", percentage %, absolute/average/high/low, [0..1], ratio, decreasing, 8)
- **Conf** = (confidentiality, -, -, -, {true, false}, nominal, -, 3)

**E. Service Specification**

In this section, we embed the QoS specification elaborated above in our approach to specify services.

Conceptually, a service consists of a number of parts, as shown in Figure III.4. On one hand, there is the service implementation that contains all information necessary to invoke a service, such as the implementation reference. On the other hand, there is the service specification that is used to make a decision on whether and how a service should be used in a specific situation, that is, the service specification is used for identification, negotiation, or composition of services. The service specification itself is decomposed into the syntactic service specification and the optional semantic service specification. Semantic service specifications are required if the identification, negotiation, and composition is to be done automatically and will not be considered in the following. The syntactic service specification contains functional aspects of services, that is, their signature. The syntactic, as well as the non-functional service specification are considering quality aspects: the functional service specification contains information on the quality of data provided to or returned by the service (qod) and the non-functional service specification captures information on the quality of the service itself (QoS).

Figure III.5 depicts our conceptual model for the syntactic service specification integrating the quality of service metadata.

As described above, the functional service specification contains the service’s signature, that is, its operation name, input, as well as output type. The input and the output data can be related to quality expressing quality of data for the respective input or output. Quality can also be assigned to a complete service, denoting its quality of service. Quality is associated in the form of quality profiles that encompass a number of quality attributes along with possible values and actual values. Quality attributes can be organized hierarchically. Additionally, mappings are used to model relationships among quality attributes such as the mappings discussed above.

**F. Conclusion and Future Work**

QoS is a central characteristic of AmI systems and requires holistic approaches. Unfortunately, due to some limitations, such approaches are not available yet. To pave the way towards a holistic QoS treatment in AmI systems, in this paper we elaborated characteristic AmI requirements posed to QoS specifications, developed a model for fulfilling these requirements, and illustrated how such a QoS specification can be integrated with a service description. Our next steps comprise a detailed analysis of different QoS specification techniques and mechanisms and the synthesis of appropriate concepts into a sound QoS approach for AmI systems.

**Acknowledgment**

The work presented in this paper was partially carried out in the context of following projects:

- The BelAmI (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) project, funded by German Federal Ministry of Education and Research (BMBF), Fraunhofer-Gesellschaft and the Ministry for Science, Education, Research and Culture (MWWFK) of Rhineland-Palatinate, in Germany.
- The integrated project Adaptive Service Grid (ASG) (FPS-IST 004617) funded by the European Commission.
BIBLIOGRAPHY


IV. DANIEL JIMENEZ

Implementation of an AmI Communication Service Using a Federated Event System Based on Aspects

Lidia Fuentes
Daniel Jimenez
LCC Malaga University
Rene Meier
DSG Trinity College

Email: {lff,priego}@lcc.uma.es, rmeier@cs.tcd.ie

a) Abstract: Event-based communication can be considered naturally suited to support Ambient Intelligence and Ubiquitous Computing applications due to its asynchronous nature and due to loose coupling between application components. Event systems support different properties depending on the specific problem domain for which they have been developed. Using these event systems in a federated way, where events are disseminated across the boundaries of a single event system, has been possible in some areas. However, such federation has typically been realized as bilateral inter-working federation between designated pairs of event systems that rely on hardcoded architectures, which are inherently difficult to maintain when systems evolve over time. This paper presents an Ambient Intelligence platform, called AOPAMI, that uses aspects to enable truly multilateral inter-working heterogeneous event systems. AOPAMI also solves the technology evolution problem using the Aspect Oriented Software Development paradigm.

A. Introduction

Ambient Intelligence applications are becoming the new revolution in computer science that is being supported by several organizations such as [1], [2], and [3] as well as by the work of individuals such as [4], and [5]. The next 10-15 years will likely be the era of the Ambient Intelligence (or AmI) and we will be surrounded by thousands of intelligent devices and applications that will help to fulfill everyday tasks. Today this kind of application is gaining acceptance due to the reduced manufacturing cost of the supporting hardware devices, the increase of users of portable devices such as mobile phones, PDAs and laptops and the development of new communication technologies such as WiFi or Bluetooth.

AmI applications are distributed and loosely coupled in nature and as a consequence, event systems are a natural choice for realizing communication. Unfortunately, there exist different event systems, each one providing solutions for specific domains and problems such as Siena [6], Steam [7] and CNS [8]. As a result, if we select one of these systems to implement an AmI application we might find that this application will be unable to interact with others applications that use different event systems.

Meier and Cahill [9] provide a classification of event systems that categorizes event systems according to a set of properties. Ryan et al. [10] has used this taxonomy and has shown that it is possible to implement a Federated Event System (FES). The FES provides a direct event translation between systems providing an adaptation mechanism for event system properties. This mechanism adapts the events produced in one system to a common and neutral representation, called the FES event model. This model enables the transformation from its representation to other event systems representations. As a consequence, this mechanism enables transparent access to the services provided by a diverse set of event systems, thereby creating a multilateral inter-working federation of event systems while maintaining the independence and coherence of the individual event systems.

However, this adaptation mechanism has some limitations. The first limitation is that it is not always possible to perform an adaptation of the events from one system to another. This limitation is due to the heterogeneity in the capabilities of the event systems and due to differences in event formats. Thus, only a subset of event system properties and features is available to be used in the FES. The second limitation is that it neither provides mechanisms to handle the evolution of event systems over time, for example, due to technological changes, nor considers possible changes to the event system configuration used at runtime, for example, due to modifications to the environment. In the first case, if we need to add support for a new event system or an existing event system changes, it will be necessary to modify and to recompile the entire application and then to redeploy the system on all devices in order to update the application. The workload associated with this task is expected to be considerable given the scale of such applications. In the second case, the application always uses a fixed set of event systems and it can not be adapted to change these at runtime. This feature allows, for example, to disable or to unload those event systems that are not anymore used, thus, saving significant computational resources. Finally, it also enables us to decide, even at runtime, which one event system the platform will use to forward the events. This decision can be taken at every moment depending on the environment variable values and state.

Based on our previous experiences refactoring applications using the Aspect Oriented Software Development (or AOSD [11]) paradigm [12] [13], we have developed a middleware platform, called AOPAmI. This paper shows the benefits of using the AOSD paradigm for addressing the heterogeneity, evolution and adaptation issues in FES applications. We show how the AOPAmI platform can be effectively applied to develop FES applications.

The rest of this paper is structured as follows. In section 2 we discuss the benefits of using a federated event system. Section 3 describes an example application. Section 4 presents the AOPAmI platform describing its main features. In section 5, we explain the application implementation process, indicating the problems faced when adapting existing applications to the AOPAmI platform. Finally, in the last section we present
our conclusions and future work.

B. Federating Event Systems

Event based middleware is nowadays applied in a growing number of different application domains including finance, telecommunications, smart environments, health care and entertainment. The use of event systems allows the integration of heterogeneous applications in an anonymous and scalable way, due to the properties inherent to event systems [10]. These event systems provide a wide range of services to the applications that use them. However, the problem of integrating different event systems in order to use their services in a multilateral inter-working and federated way has not been widely considered by researchers. The use of a FES provides a powerful tool to build truly distributed applications that can use a wide range of services provided by the different event systems integrated in the federation.

For example, suppose we design a control application that detects invalid value ranges in multiple applications controlling a production chain. The applications throw control events every time that a task finishes, and we have different event systems for each application used in the production chain. If we want to avoid the problems derived from the integration of these event systems, our choice will be to use a FES. This FES integrates all the used event systems and enables the easy introduction of new event systems in the production chain. Otherwise, we will be forced to modify the implementation of the control application each time a new event system is added. The use of a FES is justified here, because it provides an elegant solution to the heterogeneity problem associated with the use of different event systems.

Another problem that can be solved used FES is system scalability. Suppose that the event systems do not provide content-based filter capabilities. But, they must send only those events whose contents match a specific criterion, for example, values that are out of range. Consider the importance of this problem if the communication channel used by the system only support a limited amount of events. As a consequence, we must minimize the number of events thrown by the system. An adequate way to achieve this is to develop a FES that provides filtering capabilities for all the event systems integrated into it.

As a conclusion, we must say that implementing a FES is complex, but it provides a transparent communication mechanism that does not interfere with the original event system behaviour. Additionally, the resulting system maintains all the benefits associated with the use of event systems, such as scalability and loose coupling between entities that use the FES.

C. Using Federated Event Systems

The distributed application example, which we have selected to evaluate our approach, is split in several parts that are executed on different hardware devices, as is illustrated by Fig. [IV.1] Together these applications form a typical FES application that exhibits the problems associated with distributed event based applications such as device heterogeneity, scalability and limitation on communication. To illustrate all these problems and how the FES solve them, we have taken the following example from the work of [10]. In this paper, we show how to adapt this example from the FES to AOPAmI as well as the advantages of using aspects.

The first application, identified as Vehicle in the Fig. [IV.1] is located inside a moving vehicle and sends events containing information about the vehicle identifier, position (GPS) and speed at regular time intervals using an ad-hoc wireless connection. These events are gathered by one or more fixed traffic monitor applications, noted as TrafficMonitor in the figure. These events are forwarded to a traffic control center, indicated by SienaServer, using fixed communication connections. The delivered events and the information that they contain will be used by the final application, named TrafficMonitoringApp, in order to determine if the vehicle is circulating at an adequate speed or in the right direction.

The main problem with this scenario is that each application is using a different event system for disseminating events and that each application is implemented in a different programming language. Concretely, the Vehicle application is running a CORBA ORB implemented in Java. This application uses the CORBA Notification Service CNS to send location events to the nearby traffic monitor applications acting as event producers. Each instance of the second application type, the TrafficMonitor, runs a C++ application and the STEAM event system. This application receives STEAM events from the Vehicle application and sends them to the control center using the Siena event system. Therefore, this application acts both as an event consumer and as an event producer. Finally, the third application, the traffic control center, is implemented in Java and uses a Siena client connected to a Siena server. The Siena server catches the events produced by TrafficMonitor applications acting as an event consumer.

Now we will explain how events are disseminated from the Vehicle application to the traffic control center application. In the first place, the Vehicle application sends a CNS event. This event is sent to other CNS enabled applications in the environment. Additionally, this event is sent through a gateway element identified as G1 in Fig. [IV.1]. This element converts the event format from CNS to FES and from FES to STEAM and delivers the adapted event to the second application. Then, the TrafficMonitor application receives the STEAM event and process it determining that it must be sent to the control center through a Siena server. In order to achieve this, the STEAM event is converted to the FES representation by the G2 gateway shown in Fig. [IV.1] and encoded again as a Siena event to be delivered to the Siena server. Finally, the Siena server receives the event and delivers it to all the Siena clients interested in it, in our case the TrafficMonitoringApp application.
D. AOPAmI and Federate Event System

The first application, identified as Vehicle in the Fig. IV.1 is located inside a moving vehicle and sends events containing information about the vehicle identifier, position (GPS) and speed at regular time intervals using an ad-hoc wireless connection. These events are gathered by one or more fixed traffic monitor applications, noted as TrafficMonitor in the figure. These events are forwarded to a traffic control center, indicated by SienaServer, using fixed communication connections. The delivered events and the information that they contain will be used by the final application, named TrafficMonitoringApp, in order to determine if the vehicle is circulating at an adequate speed or in the right direction.

The main problem with this scenario is that each application is using a different event system for disseminating events and that each application is implemented in a different programming language. Concretely, the Vehicle application is running a CORBA ORB implemented in Java. This application uses the CORBA Notification Service CNS to send location events to the nearby traffic monitor applications acting as event producers. Each instance of the second application type, the TrafficMonitor, runs a C++ application and the STEAM event system. This application receives STEAM events from the Vehicle application and sends them to the control center using the Siena event system. Therefore, this application acts both as an event consumer and as an event producer. Finally, the third application, the traffic control center, is implemented in Java and uses a Siena client connected a Siena server. The Siena server catches the events produced by TrafficMonitor applications acting as an event consumer.

Now we will explain how events are disseminated from the Vehicle application to the traffic control center application. In the first place, the Vehicle application sends a CNS event. This event is sent to other CNS enabled applications in the environment. Additionally, this event is sent through a gateway element identified as G1 in Fig. IV.1.

E. The AOPAmI Framework

As we have indicated at the beginning of this paper, we have extended the AOPAmI framework in order to implement a FES. The AOPAmI platform (Aspect Oriented Platform for AmI) is an aspect-oriented middleware (AOM) platform. An AOM alleviates much of its complexity by allowing that concerns such as communication, coordination, location, persistence and security that crosscut the application to be modularized. This modularization hence facilitates the system evolution and makes it more robust to accommodate new application requirements. In AOPAmI we put special emphasis in describing the evolution and adaptation issues.

Aspect technologies separate and encapsulate crosscutting concerns in modules called aspects. Aspects can only be invoked at some well defined execution points inside components called join points (e.g. component creation, disseminating a message or an event). The aspects code is weaved into the components that are crosscut by the aspect obliviously from the point of view of the components. The information about which aspects have to be weaved into a component and when, is specified in an aspect language using a set of composition rules. In our approach the aspect language is a Domain Specific Language or DSL for AmI applications (AOPAmI-DSL) that defines the platform architecture as is shown in [14]. The file holding the configuration of the AOPAmI platform, as is shown in Fig. IV.2 is parsed by the platform when it is started. After this phase the platform instantiates the required base elements, identified by a series of InstantiateBaseElement tags in Fig. IV.2 which we explain later. This configuration information is internally stored by the platform so it can be changed at runtime. We call base elements to both components and aspects. The difference between them is the way in which they are composed in the application. On the one hand, aspects modify the component’s normal behaviour by intercepting and modifying the disseminated events. On the other hand, components are not conscious about the existence of aspects that change their behaviour. In Addition, base elements can take on both the component and the aspect role in the application.

In AOPAmI, the relations between base elements are defined outside their source code, in the AOPAmI-DSL as a set of rules. Two of these rules are shown in Fig. IV.2 enclosed by the compositionRules tag. As a consequence, the base elements do not maintain direct references among themselves. Therefore, we can provide different implementations for the same role name in the application in order to address the device specific requirements such as memory and execution constraints. For example, we can provide several different implementations of a base element that model different communication technologies such as WiFi or Bluetooth and refer to it using the same role name. In Fig. IV.2 it is shown the definition of a gateway base element (baseElement tag) that provides only one implementation impl tag of this element. In order to add new implementations of this base element, we simply add new impl tags and identify them using the id attribute. Finally, we need to change the selected attribute in the impls tag in order
Using the AOPAmI platform we are able to modify the event model configuration depending on the application requirements and on the hardware restrictions. We can adapt this configuration even at runtime without stopping the application or modifying a single line of code. For example, we can change the set of event systems within the FES, adding or removing them from the application, or decide to which ones the system will use to disseminate events by simply modifying the composition rules shown in Fig. IV.2. We can also accommodate all the different event types (untyped, typed and structured) to the AOPAmI event representation without changing the system implementation. Using AOPAMI, we are able to change the event propagation model if the adapted event system supports more than one. For example, we can model the push and pull event propagation models in the CNS event system using base elements. Switching between the two models will be as simple as changing the selected implementation of the required base elements and some rules.

F. AOPAmI and Federate Event Systems

In this section, we show how we have modelled the original FES application using the AOPAmI platform, the challenges that we have faced and the advantages derived from this implementation. In order to implement the distributed application presented in section 3 using AOPAmI, we have to implement two different applications.

1) Adapted Applications: The first application, the Vehicle application shown in Fig. IV.3 is modelled in Java using JacORB [15], which provides a CNS implementation. As JacORB and AOPAmI are implemented using Java, this adaptation is as easy as instantiate a new java object.

The second application, shown in Fig. IV.3, is in charge to receive STEAM events and to send Siena events to the traffic control center. This application is implemented in C++ and, thus, we need an integration technology such as JNI Java [16] in order to create a bridge between both languages. But, two limitations arose from this solution.

The first limitation is that the target device must support a Java Virtual Machine (JVM). The second one is that only primitive Java types are supported as parameters and return values for the functions written in C++. Thus, in order to do not overload the AOPAmI platform with superfluous methods, we created a java class that acts as a launcher for the AOPAmI platform. This class implements all the methods needed by the TrafficMonitor application to disseminate events and to perform all the necessary parameter transformations.

2) Application Base Elements: Now we are going to describe the set of base elements used by the applications and the functionality that they model. In both applications we have followed the same model, but the base element implementations differs depending on the application’s purpose. Fig. IV.2 describes the base elements used by the Vehicle application using a series of baseElement tags.

The most important base element in both applications is the gateway. This element models the object that the application will use to disseminate events using the FES. We can change the set of event systems within the FES, adding or removing them from the application, or decide to which ones the system will use to disseminate events by simply modifying the composition rules shown in Fig. IV.2. We can also accommodate all the different event types (untyped, typed and structured) to the AOPAmI event representation without changing the system implementation. Using AOPAMI, we are able to change the event propagation model if the adapted event system supports more that one. For example, we can model the push and pull event propagation models in the CNS event system using base elements. Switching between the two models will be as simple as changing the selected implementation of the required base elements and some rules.

We can add new features to event models such as content and subject filters, real time restrictions or security by adding the adequate base elements and rules. Another advantage is that we are able to add new features incrementally to the system. Moreover, these features can be enabled or disabled in the application whenever it is needed if they are orthogonal.
that we describe later. This composition is reflected in the composition rules that describe the platform architecture.

After the gateway elements, we found the adaptors. We need a pair of them to perform the event adaptation between a specific event system and FES and vice versa. Concretely, in the first application we found two of these named CNStoFES and FEStoSTEAM as it is shown in Fig. [IV.3]. The first one transforms the CNS events received by the STEAMGateway base element to the FES event format. The second one transforms the FES events into the STEAM format using conversion table [III].

Other data associated to the event, as for example the event type, are also adapted automatically from one event system to another. Notice that these base elements are usually composed with other base elements acting as aspects, because they modify the normal application behaviour. Notice also that how base elements are composed is completely described in the composition rules shown in Fig. [IV.2] and that as a consequence; base elements have not knowledge of how they are being composed by the platform.

### TABLE III

<table>
<thead>
<tr>
<th>AOPAmI FES Type</th>
<th>STEAM parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>S_EventParameterDeclaration.S_STR</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>S_EventParameterDeclaration.S_INT</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>S_EventParameterDeclaration.S_INT</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>S_EventParameterDeclaration.S_DBL</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>S_EventParameterDeclaration.S_DBL</td>
</tr>
<tr>
<td>javaSLSLocation</td>
<td>S_EventParameterDeclaration.S_POS</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>S_EventParameterDeclaration.S_TIM</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>S_EventParameterDeclaration.S_TIM</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>Not Supported by STEAM</td>
</tr>
</tbody>
</table>

If we examine carefully conversion table [III] we notice the adaptation problem that arises here. The problem is that STEAM does not support all possible parameter types provided by CNS, for example Boolean or Object values. The opposite it is also true, because STEAM defined a local S_POS type that it is not supported directly by CNS. As a result, in some cases we must perform some parameter adaptations. For example, in AOPAmI we have defined a class called JavaSLSLocation that models the S_POS type. Other adaptation is from the CNS numeric parameters to the FES numeric types (Integer and Double). In this case some loss of precision may occur when adapting the numeric values. Finally, the conversion of Boolean values is an extreme case. We have no other choice that to convert them to integers. This is needed because STEAM does not support Booleans. Additionally, we need to remember the name of these parameters in order to convert them back to CNS.

The second application uses another pair of adaptors named STEAMtoFES and FEStoSIENA that are composed similarly to the two previously presented. The conversion table from STEAM to FES is the inverse of table [III]. Finally, the conversion between FES and Siena is straightforward because there is a direct correspondence between Siena and FES formats.

The following important base elements used by the applications is the locator, named STEAMLocation and SIENALocation and identified as Location role name in Fig. [IV.2]. These elements maintain a list of available STEAM and SIENA devices in the environment. When an event is disseminated through the STEAMGateway, this list is added to the event as a property that is used by the platform to send the event. Notice that we have composed this base element with the gateway element, as an aspect, using a rule instead of providing a hardcoded implementation of this property inside the coordination or gateway elements. Doing this, we provide a more flexible implementation and we make use of a platform mechanism to add information to the event as needed. This information can be used afterward by the coordination element to process the event throw by the gateway base element.

One of the most important base elements in any AmI applications is the coordination base element. Each AOPAmI application defines and uses one of these events to process the events received by the platform. STEAMCoord and SIENACoord are our example coordination elements identified in Fig. [IV.2] as the Coord base element. These elements can be implemented using a transition diagram or a hard coded decision tree. This decision is up to the programmer and depends on the device capabilities. In the examples we used the hardcode version because the coordination model is very simple. We only need to redirect the events to all the devices in the environment indicated in the property added by the Location base element. Notice that when an event is thrown by a base element and a composition rule is applicable, then the platform always evaluates a coordination element. An example of this is shown in the first evaluation rule on Fig. [IV.2] where the coordination element, indicated by the coordElement attribute, is evaluated as an aspect. Notice that this element can be used as a normal component when it sends messages to other base elements. This behaviour is depicted in the second composition rule (see from tag) shown in Fig. [IV.2].

Finally the last base element used by the applications is the communicator, identified using the Communication role name in Fig. [IV.2]. Each application defines and uses one of such elements, identified as STEAMCommunication and SIENACommunication. The functionality of these base elements is sent the events to the target devices and usually they are composed as components in the application.

3) The Application Execution: After explaining which base elements are defined by the applications, we will show how they are used. Firstly, when the Vehicle application is started, it instantiates and configures the AOPAmI platform using the platform architecture file shown in Fig. [IV.2]. This file describes the base elements used by the application, as is shown in Fig. [IV.3]. After AOPAmI has instantiated all the required elements, indicated by a series of instantiateBaseElement tags, the Vehicle application obtains a reference to the STEAMGateway base element. This object is equivalent to the gateway G1 object in the original example and it is used to disseminate events using the AOPAmI platform.

When the application sends a CNS event, this event is
passed to the gateway base element. The gateway generates an AOPAmI event that it is intercepted by the platform. The platform checks for rules that can be applied on this situation. In this case a rule, the first rule shown by Fig. IV.2 is found. We determine that the rule is evaluated only when an event is thrown because of the EVENT value assigned to the when attribute in the compositionRule tag. This rule also indicates that when a sendEvent event (events tag) is thrown by the gateway base element (from tag), then the platform will compose this base element with the Location and the CNStoFES base elements (evaluation tag). The base elements will be evaluated as aspects in the order indicated by the rule, and they will probably modify the event contents and properties. Finally, the rule states that a coordination base element, indicated by the coordElement property of the rule, will be evaluated. This last base element corresponds in our example to the Coord element.

At the application level, by applying these two aspects we achieve two goals. Firstly, the platform adds a list of available STEAM devices to the event using the Location base element. And secondly, the platform converts the event format from CNS to STEAM using the CNStoSTEAM base element. The base elements will be evaluated as aspects in the order indicated by the rule, and they will probably modify the event contents and properties. Finally, the rule states that a coordination base element, indicated by the coordElement property of the rule, will be evaluated. This last base element corresponds in our example to the Coord element.

The new message is sent to the platform and the platform checks for the applicable rules. In the example describe by Fig. IV.2, a rule is found that states that when a message sendEvent (events tag) sent by the Coord base element (from tag) to the Communication element (to tag) is found. Then, before the message is delivered to the Communication element (the attribute when takes the BEFORE__SEND value in the compositionRule), the message will be composed with the FESstoSTEAM base element.

At the application level the platform adapts the event format from FES to STEAM using the FESstoSTEAM element as an aspects that modifies the normal element composition. Finally, the event is delivered to the STEAM device or devices selected by the coordination element using the STEAMCommunication element.

The second application works similarly to the first one. After receiving a STEAM event, the TrafficMonitor application decides to forward this event using the SIENAGateway base element. But in this case we made the transformation from STEAM to SIENA event format. Finally, the event originally generated using a CNS event system is delivered and processed by a Siena event system.

Note that in both previous examples it is not necessary that the applications receive and handle events coming from other event systems. This functionality can be easily implemented in AOPAmI by adding the appropriate base elements and composition rules. For example, to receive STEAM events in the Vehicle application, we only need a STEAMtoFES and a FEStoCNS elements and a rule to compose them. This composition will take place when the STEAMCommunication element receives a remote event from other STEAM application.

G. Conclusion

This work has investigated the possibility and benefits of applying the AOSD paradigm using the AOPAmI platform to develop a FES. Starting from the example and the set of properties identified in the work of [9] and [10], we have been able to implement a functional AOPAmI FES prototype. This prototype has combined three different event systems namely Siena, STEAM and CNS establishing a communication channel among them. The use of AOPAmI has clearly added some benefits to the FES application, such as adaptability, modularisation and reusing. A consequence of this is that using AOPAmI, we will be able to extend this system by identifying new event system properties not considered previously. These properties then can be modelled and integrated in the FES as new base elements and rules adding even more functionality to existing and new applications. Additionally, both the base elements and the platform architecture definitions developed for this example can be reused in other applications by simply adapting the platform configuration to these systems. Indeed, we reuse the Platform Architecture file shown in Fig. IV.2 in both the Vehicle and the TrafficMonitor application examples. Moreover, by modifying the platform architecture file, we can develop more complex event system federation configurations adding new functionality to existing applications. An additional issue that was raised when adapting the original application was that the AOPAmI platform was originally designed
to develop J2ME applications. But, when using Siena and JNI we force to create an extended version of the platform able to use the J2SE features due to the requirements of these event systems. Finally, we have shown that AOPAmI is able to deal with the integration issues of different programming languages (Java and C++) in an elegant way. In contrast, other AmI platforms such as PCOM [17] or EMI2Lets [18] only consider a fixed set of technologies and development languages and applications are hardcoded, which limits the application development. As future work, we intend to integrate other event systems in our FES modelling additional properties as new base elements and refine the already developed ones to improve their reusability. Additionally, we are working on a set of tools to automate.

Acknowledgment

This work is partially financed by IST-2-004349-NOE AOSD-Europe and the Spanish Ministry of Technology and Science, CICYT, under grant TIN2005-09405-C02-01.

BIBLIOGRAPHY


V. ANDRÉS FORTIER

Engineering Pervasive Services for Legacy Software

Silvia Gordillo, Gustavo Rossi
LIFIA, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina.

Andrés Fortier
LIFIA, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina.


Email: {gordillo,gustavo,andres}@lifia.info.unlp.edu.ar

a) Abstract: In this paper we present a novel architectural approach to engineer applications that provide location-aware services; in particular, we explain how to extend existing software systems with location-aware services. We show how a clear separation of design concerns (e.g. applicative, context-specific, etc) helps to improve modularity. We stress that, by using dependency mechanisms among outstanding components, we can get rid of explicit rule-based expressions thus simplifying evolution and maintenance. We first motivate our research with a simple example. Next, we present the big picture of our architectural approach. Then we detail how to specify location-aware services; we present details of the services’ activation mechanisms. We finally we discuss some related work in the field. We conclude with some further issues in which we are now working.

A. Introduction and Motivation

Building applications that provide context-aware services has proved to be difficult; the most important reasons have been extensively reported elsewhere [1], [2]. The only way to guarantee seamless software evolution is to rely on solid software engineering practices; in particular, in order to assure software modularity, a clear separation of design concerns is a must [3]. To make matters worse, applications that provide context-aware pervasive services are not built from scratch; they emerge as a consequence of the evolution of existing software systems, which are modified or extended to provide brand new functionality, according to new communication and hardware possibilities. We have devised a design approach and an implementation framework to engineering location-aware services [4]. Instead of using rule-based approaches, we based our approach on an extensive use of well-known dependency mechanisms in the object-oriented field [5]. In this paper we elaborate our approach and show how to use it to improve existing systems with location-aware services.

This paper has been partially supported by the Argentine Secretary of Science and Technology (SeCyT) under the project PICT 13623
As a motivating example, suppose an academic system which provides information (e.g. using a Web interface) on courses offered by the university, time-tables, teaching material, etc. For the sake of simplicity suppose that the system has been designed using good object-oriented practices, e.g. it follows the model-view-controller metaphor, in which model, interface and interaction issues are clearly separated and we can identify a set of model’s behaviors that provide the intended information [5].

How do we extend this system in order to provide pervasive services? For example, when a student is in a room in which a course is to be given, he has access to the material of the course; the professor meanwhile can access the list of students in the course and can upload material (e.g. for homework).

In the rest of the paper, we show in a step by step way how to seamlessly enrich existing applications with location specific information (e.g. locations of rooms), and how to engineer pervasive services. We treat services as light-weighted objects that are attached to physical locations (service areas) and made available to the user when he enters into the corresponding location; additionally, these locations may eventually refer to specific application objects (e.g. a room).

The rest of the paper is organized as follows: In Section 2 we present an outline of our architectural approach. In Section 3, we discuss the specification of location-aware services; details on service activation are presented. In Section 4 we analyze some related work and in Section 5 and present our concluding remarks.

B. High-Level Architectural Decisions

For the sake of comprehension, we will describe each design problem using a pattern format [3], [6], stating the problem we faced, the context and the solution. This style, which follows the ideas in [7], allows us to show that the proposed solutions are more general than a particular framework implementation (like ours), and thus can be used in similar situations. We use a coarse grain for describing these architectural design decisions. Many of them deserve a longer explanation at a lower (say, micro-architectural) level, but we omit this discussion for conciseness. We will first concentrate on the most outstanding components and design decisions; details on architectural and lower-level issues related with hardware abstractions, sensing concerns and location models can be read in [4], as we only describe them in a high level way.

1) Dealing with Location and Location Models: Problem: In many applications (such as in the example), we need to determine the position of the user in relationship with physical regions that correspond to application objects. For example, to decide whether a user is in a room or in its vicinity, we need to know the room’s position in some location model [8], [9]. Coupling application objects with their positions or making them location-aware (e.g. adding a variable in class Room for representing its positions) have many disadvantages. The most relevant problem is that it pollutes the application model with location information (which tends to be volatile), and with objects that are not important for the original application, e.g. a corridor in which we want to provide certain services. When we are extending legacy software, this problem is even more evident. In summary, how do we enrich applications with location information transparently?

Solution: Build a separated Location layer which contains the abstractions which are necessary to maintain and process location information. Objects in this layer may or may not have a counterpart in the application layer. For example, in this layer we model the located counterpart of application objects, e.g. location.room; we also model other objects which have not been defined in the application, but have a spatial meaning, such as corridor or building and which may be related by spatial relationships (corridor connects rooms). The Location layer also comprises lower-level abstractions that implement different location models (symbolic, semantic, geometric, etc) and a component, Location.User which contains the actual user position (See 2.2). The relationships between the Location and the Application Layer is shown in Figure 1. Notice that the relationship between objects in the Location and the Application layers resembles the Decorator [6] pattern as located objects “extend” application objects functionality with spatial information and behaviors. This solution makes application objects oblivious of their spatial extensions, therefore allowing their evolution, and the evolution of the located components independent and thus less error prone.

There are three important abstract classes in the framework: LocatedModel, LocatedObject and Location. Every class that stands for an application counterpart (as in the case of Location.Room) is defined as a subclass of LocatedModel. Purely spatial classes (like Corridor) are derived from LocatedObject. The position of all objects in the Location layer are described by an object of a class implementing the Location type. This type abstracts different location models (geometric, symbolic, etc) which we do not describe in Figure 1. Decoupling located objects from the location model allows us to reason in a higher level way, as the existence of a corridor or a room and their spatial relationships are independent of the way they are represented as locations (e.g. with coordinates, symbols, code bars, etc).

![Fig. V.1. The Location Layer as a decorator on the application model](image-url)
contextual data, and to query this object to know the actual user’s position. There are two problems with this approach: First, considering this object as just a data repository tend to delegate some of its responsibilities to other objects (thus, compromising modularity). Besides, in many applications such as in the university campus, there may be already an object which represents a possible user, e.g. a student object in the location model. Which is the relationship between these objects?

Solution: We model the user’s location in the Location layer (Location.User). Similarly to other objects in this layer, this location is described using an object of type Location. Location.User may be also related with the corresponding application object (if any). We consider Location.User as a critical object in the process of triggering the activation of services instead of a passive data repository. Each time this object changes (i.e. the user changes his location), it communicates the change to its counterpart in the Service Layer (See Section 3), thus changing the possible available services. Figure 2 shows Location.User and its two subclasses: Application.User and Located.User which play the same role as Located.Model and Located.Object in Figure 1. When the user has a counterpart in the application model (e.g. he is a registered student or teacher), we use Application.User which extends the corresponding class (in this example Person). A casual user is represented by Located.User objects. As we will concentrate ourselves on the process of activating services, we will not address authorization issues in this paper. The description in Section 3.4 holds for both Application and Located users.

![Fig. V.2. Locating the user](image)

3) Dealing with Hardware and Sensing: Problem: Hardware for sensing the user’s location (and other context variables) evolves constantly. Sensing policies (e.g. push vs. pull) vary according to hardware capabilities. It is clear that low-level details have to be hidden from the application. However, sensed data is in the best case string data and it has to be interpreted to fit into the needs of the application. Moreover, location models (See 2.1) should evolve independently of sensing hardware. How do we provide this independence?

Solution: Decoupling sensors and their logic, from application concerns has been the driver of many research projects. While context widgets in [2] and adaptors in [10] isolate hardware from the application software, we still need a higher level of interpretation to relate sensed data, first with location objects and then with application objects. We thus decided to further decouple the hardware abstractions (similar to Dey’s widgets [2]), and the logic for sensing the user’s position. These two layers, namely Hardware Abstractions, and Sensing Concerns and their relationships with the previously described components are shown in Figure 3. In Figure 4 we present a more detailed diagram, exemplifying with a simple location sensing widget, an IR port. The Sensing concern acts as a dependency transformer between the hardware and the location layer. The dependency relationship implies that every time something changes in the sensor, the sensing concern object is notified. This object abstracts the sensing policy (e.g. push or pull), as a Strategy object [6], represented by a subclass of SensingPolicy. Once it has the new position, it maps the sensed value into an object of the Location layer (e.g. a room, a corridor, etc). There might be different algorithms for performing this mapping, which obviously depend on both the sensed data and the actual location model (e.g. symbolic or geometric).

![Fig. V.3. Sensing Concerns and Hardware Abstractions](image)

This new location is sent to the Location.User object, indicating that the user has changed his position. Notice that the sensing concern layer plays the role of an interpreter, enhancing the behavior of Dey’s interpreters [2] to get a slightly higher level location object, which can be related with an application object.

![Fig. V.4. Refinement of Sensing Concern and Hardware Abstractions](image)

C. Engineering Location-Aware Services

We consider services as full-fledged software artifacts which may be (as in the examples in this paper), extensions of an application’s behavior. They may already exist as methods in some application object, e.g. providing the material of a course or they may not use application’s methods, but involve application-related objects, e.g. informing which rooms are in this floor.

Some services will be autonomous; others might require the use of “external” (eventually Web) services, e.g. returning the
actual temperature in the area. As others, we envision that service engineering will be a critical software design activity and, therefore, design issues related with location-aware services are fundamental to assure the quality (e.g. modularity, reuse, etc) of the engineered services. Following the style of Section 2, we next analyze the most important problems we faced and the solutions we propose and implemented.

1) Describing Services as Modular Application’s Extension: Problem: Services should be engineered independently of other application components. It should be possible to make them dynamically available to users, according to different conditions (location, role of user, etc). It should be possible to specialize them or compose them to obtain more complex services. However they may have a close relationship with applications’ behaviors, i.e. activating a service might imply the invocation of an application object’s method. How do we balance modularity with the need to relate services with application objects?

Solution: First, we created a Service Layer separated from the others framework layers. Then we defined Services as objects, following the Command [6] design pattern. An abstract class Service is defined to contain the interface common to all services. For each possible service, we defined a class, representing the concrete service, e.g. GetMaterial. When a service is activated (e.g. because a user enters in a place where the service is available), we instantiate the corresponding class and allocate this object to serve the user. Allocating a service to the user means to initialize an instance variable of the service (defined in the abstract class) to refer to the user (See 3.2) and add the service object to the set of active services for that user (See 3.2). Other relevant information is defined when the service is started (See 3.3 and 3.5).

Treating services as first-class citizens allows manipulating them uniformly, building service compositions, making their activation dependent of the behavior of other objects, etc. We also decouple the instantiation of a service from its execution, which allows us to manage services’ queues, logs, etc. The relationship between Services and application objects is a knowledge relationship as shown in Figure 5. We will discuss the relationships among the Services and the Location layer in Section 3.2.

3.2) Linking Services to Locations and Providing Location-Awareness: Problem: How do we state that a certain service should be active (available to the user) in a certain location? How do we proceed to its activation? The usual solution is to write a set of rules in which the condition checks whether the user is inside an area and the action consists in activating the service. While rules can be decoupled from the application (e.g. similar to business rules) and designed as objects (See for example the material in [4]), they tend to become monolithic when the application evolves; for example when new services emerge, or a service needs to be attached/detached to a different area we have to edit (add or modify) rules. Besides, the information on which services are provided in a location, and the code for invoking those services, are tangled in the corresponding rules; this makes the maintenance activity more difficult as we need to read all rules to grasp the big picture.

Solution: Register services to locations using a dependency mechanism such that when a user enters into a place, all services registered to that place are notified, and thus they are made available to the user. This solution involves the specification of several components to guarantee flexibility (of service definition), and modularity (of the underlying software). The most relevant are the following:

Service.User: This object has two fundamental roles; one which is purely informational: it knows the actual available services for the user; the other role relates with the process
of updating these services: it is dependent (a kind of Ob-
server [6]) on Location.User; therefore every time the user
changes his position (and therefore Location.User changes),
Service.User is notified and it triggers a set of behaviors to
determine which services are not longer available and which
ones should be added. Service.User implements the most
important dependency mechanism for implementing location-
awareness. In Figure 7 we show these relationships.

**Service Area:** Services are not always provided in logical
areas, such as a room, a bar, or a corridor, nor should they
depend on sensing hardware (e.g. if a room has many sensors
inside it); instead they may be defined opportunistically in
aggregations of areas or part of areas; we call these aggregation
Service Areas. For example, we might want that the services
corresponding to a room are provided also in the surrounding
of the room, e.g. in a part of a corridor (assuming that the user
can be sensed to be there). A Service Area has a knowledge
relationship with the corresponding Location object which
defines it (in the Location layer), and with the Services which
are provided in the Area. The Service.User object also has a
knowledge relationship with the area in which he is located
(See Figure 7).

**Service Environment:** The Service Environment acts as
areas (instances of Service Area). When the user changes
his position, Service.User collaborates with the environment
to determine if the user has left or entered a new area.
The Service Environment then sends the message leaveArea
(or enterArea) to Service.User which will update the current
services accordingly.

Figure 7 shows a static diagram with the relationships
among these classes and also showing how these classes
interact with classes in the Location Layer. Figure 8 shows
a simplified sequence diagram with the process of activating
a set of new services for a given user, i.e. making these
services available to the user. When the Location.User
object receives the message indicating a change of position, it
notifies Service.User by means of the dependency mechanism.
Service.User gets the new position and interacts with Ser-
vice.Environment to analyze if the new position implies that
the user entered or left a service area. Service.Environment
interacts with Service.Area (not shown in the sequence dia-
gram) and sends either the message enterArea or leaveArea to
Service.User. The effect of executing these methods is that a
new set of services is allocated to the user: those corresponding
to the actual service area in which the user is located.

3) **Relating Services with Application Objects:** *Problem:*
As previously discussed, there are services which may need
a strong interaction with application objects; for example
suppose that we specify a service which may be provided
in rooms, and which returns the material of the course that
is currently scheduled in the room. We may also associate a
service with research laboratories to provide information on
the corresponding research projects. In both cases, the service
object needs to interact with a specific application object (a
room, a research lab, etc.) which will eventually mediate with
other objects (course, project, etc.). However, services which
only deal with application data should not be cluttered with
details on location issues; they should only concentrate on
their task. How do we instantiate services to realize the correct
relationship?

*Solution:* We provide a pre-built initialization method in
the abstract class Service; when a service is started, ex-
cuting a method in Service, it must first initialize itself
(this process is performed using a template method [6]). The
standard initialization method (init) returns the application
object corresponding to the actual user’s location (e.g. a room
object), which is sent as a parameter for service execution.
This object is obtained by collaborating with the actual user’s
location object, which may have an explicit relationship with
an application object. For pure spatial objects, i.e. those which
do not correspond to any application object a nil object is
returned. Designers can either re-define init, by including it
in the concrete class (therefore being invoked by the template
method) or perform other additional initializations, e.g. when
more application or spatial information is needed for the
service execution. For example the service can the scheduled
course and set the relationship shown in Figure 6.

4) **Managing Service Granularity:** *Problem:* In most ap-
lications, services are associated with coarse-grained physi-
cal objects, e.g. meaningful physical areas in the university. Moreover, once the user is sensed to be in a Service area, we can assume that the services allocated to the area are bound to the corresponding application object as dis-cussed in Section 3.3. For example when the user enters in a room, the service GetCourseInformation will refer to the course object being scheduled in that room. Suppose, however that we want to provide services with a finer grained physical scope. For example we might have an art exhibition in the room and we want to provide additional information on art-works, as in augmented reality applications, e.g. when the user stands in front of an artwork he can get information on the painter or technical data (such as material, painting technique, etc). Assuming that we can sense the user’s position precisely, should we define finer grained service areas and allocate the same service (type) to each of these areas?

Solution: In our conceptual schema, services are allocated to areas. In the example above, it is clear that the service area is the room and that defining new areas for each of the physical objects (artworks), poses a problem of maintenance; adding a new artwork requires the definition of a new service area. We instead need to adapt the behavior of the location-aware services to the specific object the user is facing. The solution emerges by analyzing the flow of control that results when the user is sensed to be in a new location (in front of an artwork). As explained in Section 3.2, the Service environment detects that the user did not exit a service area and he did not enter another one (this conclusion follows by analyzing the physical object in the corresponding Location model). Therefore, the previously allocated services still apply. However, the physical position of the user has changed, and this change has been registered in Location.User. Then, when the service GetArtworkInformation is started, the process described in Section 3.3 is performed and the service is bound to the correct physical object (the artwork). In other words, we can make (location-aware) services adaptable to finer grained physical objects, without changing the overall architecture, nor the under-lying location model: we just need to slightly rewrite the start behavior of those services.

D. Related Work

Context Toolkit [2] has been our first source of inspiration for providing a clear separation of concerns in our architecture. Our hardware abstractions and sensing concerns are similar to Dey’s [2] context widgets. Hydrogen [10], meanwhile, introduces some improvements to the capture, interpretation and delivery of context information with respect to the seminal work of the Context Toolkit. Both the Context Toolkit and Hydrogen are aimed at providing a reusable context infrastructure that can be used by several applications. In this sense, the application concern is not dealt with and therefore there are no cues about how to structure application objects, particularly when they involve some information which is important for deciding about context changes.

Our view is slightly different; we focus on how to seamlessly extend existing applications with location-aware services. Even though our architecture also provides reusable components, our main goal resides in how to bridge context information with application objects. Our approach proposes a clear separation of concerns between those object features that are “context-free” (attributes and behaviors), those that involve information that is context-sensitive (like location and time) and the context-aware services. In this sense our approach proposes a set of micro-architectural styles to add location and services to application objects, which inverts the usual relationship between these aspects. While naïve software approaches make objects aware of their positions and services, we use decorators and commands [6] to achieve the same result but making the application oblivious of these additions. This approach also improves traditional rule-based approaches like [11], which tend to hardcode service activation conditions in rule conditions; these conditions either refer explicitly to application objects (which as a consequence must know their location), or contain location information, thus making them a critical point during maintenance.

By clearly decoupling these aspects in separated layers, we obtain modular applications in which modifications in one layer barely impact in others. Our idea of connecting services to places has been used in [12], though our use of dependency mechanisms improves evolution and modularity following the Observer’s style [5], [6]. From an architectural point of view, our work has been inspired in [7]: the sum of our micro-architectural decisions, such as using dependencies or decorations also generate a strong, evolvable architecture.

E. Concluding Remarks and Further Work

In this paper, we have described a set of abstract architectural components and their associated communication mechanisms, which provide a substrate for seamlessly extending existing object-oriented software to support location-aware services. The most important goal of our approach is that it provides an original way of mapping application objects to their located counterparts (i.e. the objects which describe their positions in a particular reference system). By using dependency mechanisms instead of rules we improve maintenance; the cost we pay is that the underlying design is more complex than typical rule-based systems which usually comprise a rule model, a context model and application objects. We are now researching on the following areas:

- Regarding service specification we are now studying how to use services as proxies of Web Services; while the use of objects to manipulate services is straightforward, many of our design structures relies on pure object-oriented constructs which have to be slightly modified to deal with XML-based services.
- We are studying abstraction and composition mechanisms at the service level, both to express service’s behaviors and activation conditions. For example we may have an abstract service which is activated in every room (e.g. Get material) but which may be refined into more specific ones according to the room, or other conditions hold on application objects.
• We are extending the approach to other kind of context data. Traditionally, context data has been treated as plain data which can be queried (e.g. activities are described as a string such as “working”); by objectifying such data, services dispatch can be dealt with by delegating the corresponding decisions to the involved object (e.g. an instance of a subclass of Activity or Role). For other kinds of contextual information, e.g. measurable context data, we are extending the notion of service area to use the same kind of strategies which we use for spatial information. We are also studying how to deal with n-dimensional areas, where each dimension deals with a different kind of context data.

• We are improving the architecture by incorporating an event model to simplify the management of dependencies. Event models help objects which receive a notification to delegate to specific event managers: the impact of this approach is that we can dynamically add new kind of events (e.g. to manage a new kind of context information), without having to edit the working code.

BIBLIOGRAPHY


VI. JAVIER MUÑOZ


Javier Muñoz
Vicente Pelechano
Carlos Cetina

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camí de Vera s/n, E-46022, Spain
Email: {jmunoz, pele, ccetina}@dsic.upv.es

a) Abstract: Current pervasive systems are developed ad-hoc or using implementation frameworks. These approaches could be not enough when dealing with large and complex pervasive systems. In order to improve the productivity and reduce the number of errors, we propose to apply the newest trends in software engineering (the MDA and Software Factories approaches) to the development of pervasive systems. These strategies propose to use models for automatically generating the final system, and not only for generating documentation or for guiding the implementation process. The application of model driven approaches to the development of pervasive systems can provide many relevant benefits. We have developed such a kind of method providing several assets for supporting the development process.

A. Introduction

Most of current pervasive systems prototypes are developed ad-hoc, since the current challenge is to achieve functional pervasive systems for proving the feasibility of the Weiser’s vision. Therefore, researchers (and practitioners) don’t use to take into account any software engineering method. This approach can be useful for building proof-of-concepts prototypes or the very first commercial systems, but this way of developing pervasive and ubiquitous systems can not scale. Following a craftsmanship strategy is error-prone and the resulting product use to be buggy and hard to evolve.

We state that software engineering practices must be applied in the area of pervasive systems in order to develop high-quality systems in a efficient way. The software engineering strategy that is proposed in this paper is inspired by the OO-Method [1], a model driven method for Information Systems with full code generation capabilities. The OO-Method makes a clear distinction between the problem space (centered on what the system is) and the solution space (centered on how it is implemented as a software product). It is based on the idea that every implementation decisions is based on an abstract concept. Therefore, we must provide conceptual modelling techniques which supply to the analyst with primitives for specifying these abstract decisions. The OO-Method has been implemented in the OlivaNova Model Execution System [2] and it is currently used to develop software for international companies. Our aim has been to apply these successful ideas to
pervasive systems development taking into account the newest trends in software engineering (MDA and Software Factories).

In this paper, we present the motivation of this approach and how it has been applied in a method for developing pervasive services for ubiquitous environments. The structure of the paper is the following: In the following Section we argue why solid engineering methods for pervasive systems development are needed and we propose a model driven approach. In Section VI-C we introduce the Software Factories and MDA proposal, which are the basis of our approach. Section VI-E presents a method for the development of pervasive service which has been built to follow a model driven strategy. Finally, Section VI-F includes the conclusions of this paper.

B. The need of solid engineering methods for pervasive systems development

As stated in [3], the next few years will determine the success or failure of Ubiquitous Computing research. Pervasive systems are moving from the academy to the industry, and this evolution implies that the systems under development are more complex and with more quality requirements than in research prototypes. Thus, solid engineering methods are needed in order to produce robust systems in an efficient way. If we fail in providing specific methods which solve the challenges that pervasive systems have introduced, it will be very hard to completely achieve the vision that was disseminated by Weiser.

In order to deal with the increasing complexity, these kind of methods must also increase the abstraction level of the concepts that are used for building pervasive systems. Following this strategy, the developers describe the system using primitives from the problem domain (specifying what the system must do) instead of using technological concepts (specifying how the functionality must be implemented). In this context, a model driven approach seems a good option, since this strategy is supposed to provide this characteristic. Unhappily, traditional methods and CASE tools that use models for developing software have many drawbacks. These traditional methods did not provide mechanisms for ensuring that the generated product is equivalent to the conceptual model. Usually, these methods did not specify precisely the primitives for building the conceptual model. Therefore, it was very hard to define transformations from the abstract specifications. As a consequence of this situation, the effort that the developers invest in the conceptual modeling phase is only useful to generate documentation. The models had not a great influence on the final software product and, thus, they were not seen as a valuable asset.

If we want to effectively use model driven approaches, we need methods and tools with the following characteristics [4]:

- they must face the development process rigorously and in a systematic way.
- they must focus on the precisely specification of the system in the Problem Space.
- they must help the analyst to make decisions in the conceptual modelling step.
- they must automatically generate fully functional systems (structure and behaviour) taking as input conceptual models.
- they must provide technological independence hiding the implementation details.
- they must produce high quality software functionally equivalent to the abstract specification.

Last trends in the Software Engineering area aims to achieve these goals. Both the Model Driven Architecture (MDA) [5], that is promoted by the OMG, and the Software Factories approach [6], that is promoted by Microsoft, propose the use of models as the main artifacts in the development of software. These proposals are supported by big players in the industry market, which ensures a strong tool support for developing practical techniques and methods.

1) Benefits of the Proposed Approach: The application of a model driven approach to the development of pervasive systems can provide many relevant benefits to pervasive systems development.

- The adaptation of a pervasive system to new implementation technologies is easier using abstract models than working with low level of abstraction code. This feature is really important in the development of pervasive systems, where new technologies continuously arise.
- The development of pervasive systems is more intuitive using model driven methods than using classical approaches because the developers deals with domain concepts avoiding technical details.
- The application of implementation best practices is embedded inside the implementation framework and inside the transformations. Currently, developers must worry about applying best practices in the development of every project.
- Models can be used for simulating the pervasive system. Pervasive systems are highly hardware dependent. In order to test a system, the developer needs the devices that the system will manage in the final deployment environment. Following a model driven approach, models can be animated with prototyping tools. This characteristic allows an early validation of system requirements.

C. Software Factories and MDA

A Software Factory, as defined in [6], is a software product line that configures extensible tools, processes and content [...] to automate the development and maintenance of variants of an archetypical product by adapting, assembling and configuring framework-based components. Therefore, Software Factories focus on the development of similar systems encouraging the reuse of architectures, components and know-how.

In order to achieve these goals, Software Factories integrate several existing techniques. The main activities promoted by Software Factories are:

- Building families of similar software. This activity involves the analysis and design of a common architecture for a set of systems and the development of a framework to support this architecture.
• **Assembling components.** The construction of a new system implies the use, assembly and/or configuration of the components provided by the framework.

• **Developing domain specific languages and tools.** Developers use this language in order to describe the specific requirements of a member of the systems family. Then, the specific source code is automatically generated.

• **Using constraint based scheduling and active guidance.** All the steps of the development project must be taken according to a well-defined process properly adapted to the domain.

On the other hand, MDA is, as described in the IEEE Software special issue on Model Driven Development [7], "a set of OMG standards that enables the specification of models and their transformation into other models and complete systems.

Following this approach, system developers build high level abstraction models (called Platform Independent Models, PIM) and transform them obtaining models that directly represent the final software product (called Platform Specific Model, PSM).

Therefore, there is a natural integration of this two approaches. MDA techniques can be used to support the development of domain specific languages for building high level abstraction models. Then, these models can be transformed in order to obtain the specific source code of a system in the context of a family of systems.

In short, we are interested in the strengths of both approaches:

• **from Software Factories** we get their focus on reuse by means of domain specific development.

• **from MDA** we get their focus on high level abstraction models and automatic code generation.

We think that this merging can provide important benefits to the state of the art in the development of pervasive systems. The different kind of systems that are developed in this area (context providers, services suppliers, personal clients, etc) can be precisely bounded. Therefore, the construction of domain specific languages and frameworks supporting these kind of systems seems feasible.

**D. Applying the Approach: A MDD Method for Developing Pervasive Services**

In order to illustrate the proposed approach, this section briefly describes a method that has been developed by the authors for developing pervasive services for ubiquitous environments. The method, which was presented in [8], provides the following assets:

• a **domain specific language (PervML)** [8], [9] for specifying pervasive systems using conceptual primitives suitable for this domain. The language provides technology independent primitives like service, device or interaction. The graphical representation of PervML is based on UML diagrams in order to facilitate the comprehension by current software engineers. Figure VI.1 shows an excerpt of a pervasive meeting room specified using PervML.

• an **implementation architecture** which provides a common structure and execution strategy for all the systems which are developed using the method. This architecture has been designed with the aim of making easy the integration of technologies (EIB networks, web services, etc.) and for supporting multiple user interfaces. More information about the architecture can be obtained in [10] and [11].

• **transformation mappings** that translates the PervML specifications into Java code, since this is the programming language of the destination environment (as it is going to be described in the following section).

1) **Supporting Assets:** In order to achieve most of the the benefits that are promised by a model driven approach, the method must provide concrete techniques and tools. Our method is currently supported by three assets: (1) a PervML editor for the creation of PervML specification, (2) an implementation framework which supports the architecture of the pervasive systems that are developed by the method and (3) an implementation of the PervML-to-Java transformation.

![PervML editor automatically generated by the EMF plug-in](image)

**Fig. VI.2.** PervML editor automatically generated by the EMF plug-in

• **PervML Editor:** The language has been implemented using the EMF [12] plug-in for the Eclipse environment (see Fig. VI.2). This plug-in provides facilities for managing models (edition, storage, manipulation, etc.) from the specification of metamodels. Currently we are working on a more user-friendly editor using the functionality supplied by the Eclipse Graphical Modeling Framework (GMF) [13]. Tools like Eclipse GMF and the DSL Tools of Microsoft [14] are dramatically increasing the facilities for developing graphical editors for modeling languages.

• **Implementation framework:** This framework provides similar abstract classes to the PervML conceptual primitives (Service, Trigger, Interaction, etc.) in order to facilitate the translation process. It has been built on top of the OSGi middleware [15] which is a standard Java-based and dynamic execution environment for services. This
environment has bridges to many of the technologies that can be used in pervasive systems and provides high level of abstraction implementation constructs. Figure VI.3 provides an overview of the structure of the systems that generated by our method. The implementation framework uses the concept of drivers in order to integrate external devices or software systems. A driver in our framework is a manually developed service that implements a known interface and that embeds the code for accessing the external element. Moreover, we apply the Model-View-Controller pattern in order to support multiple user interfaces.

- **Transformation engine:** In order to implement the model to text transformation, we have used the MOFScript tool, which is included in the Generative Model Transformer (GMT) [16] Eclipse project. The MOFScript project "aims at developing tools and frameworks for supporting model to text transformation". The MOFScript tool is an implementation of the MOFScript model to text transformation language. This language was submitted to the OMG as response to the "MOF Model to Text Transformation Language RFP".

In the following sections we describe when to use this assets during the development of a pervasive systems.

2) **Applying the Method:** Figure VI.4 presents an overview of the approach. The figure shows the developers roles that are in charge of performing the method steps, the assets that are produced during the development process and the tools and infrastructure elements that are provided by the software factory in order to facilitate the development process. In summary, a development team must carry out the following steps in order to produce a pervasive system using our method:

1) Systems analysts capture system requirements and describe the pervasive system at a high level of abstraction using the service metaphor as the main conceptual primitive. Analysts build three graphical models that constitute what we call the Analyst View. In these models the analyst describes (1) the kinds of services (by means of their interfaces, their relationships, their triggers and a State Transition Diagram for specifying the behaviour of each service), (2) the components that are going to provide the defined services and (3) how these components interact to each other.

2) The system architect selects the kind and number of devices or software systems that are more suitable in order to provide the services specified by the analyst. The selection could have into account economical reasons or constraints in the system physical environment. We call binding providers to the elements that are responsible of binding the software system with its physical and logical environment. For instance, a lighting sensor is in charge of measuring a physical feature of the environment, whereas an e-mail server allows sending information to agents that are out of the scope of the system. Architects build other three models that constitute what we call the Architect View. We need to build a detailed specification of the lower level artifacts that realize system services in order to have a complete and operative pervasive system description. In order to achieve this goal, the architect describes (1) every kind of binding provider (their interfaces and their relationships), (2) the binding providers which are used by each system component, and (3) which actions should be executed when a component operation is invoked.

3) An OSGi developer implements the drivers for managing the devices or software systems which were selected by the system architect. These drivers provides access from the OSGi-based framework to the devices or
external software systems. They must be developed by hand, since they deal with technology-dependent issues. If any device or external software system was used in a previous system, the same driver can be reused.

4) The **transformation engine is applied to the PervML specification**. Many Java files and other resources (Manifest files, etc.) are automatically generated as a result of this action.

5) The **Java files are configured in order to use the selected drivers**. This configuration only implies to set up the drivers identifiers.

6) Finally, the **generated files are compiled, packaged**
Note that the proposed method is focused on the development of the software system part of the pervasive system. The physical installation of devices, networks, etc. is out of the scope of this work.

E. Conclusions

From our point of view, if pervasive systems want to move from the laboratories to the real world not only technology but also solid engineering methods are required. We have stated in this paper that a model driven approach can help to build quality pervasive systems in a cost-effective way. The strength of this approach relies on the use of high level of abstraction primitives for the specification of the systems, and on precise transformations from these abstract models to concrete technologies. We have briefly described a method with the proposed characteristics for the development of pervasive services in ubiquitous environments.

Note that a method following the proposed approach is focused on very specific kind of systems. For instance, the method that has been presented in this paper should be extended (or another method may be developed) for producing GIS pervasive systems for mobile devices. These situations encourage the construction of specific methods for every specific kind of systems that can be found on the pervasive systems area. For instance, the work of Henricksen [17], which also applies a model driven approach, is very focused on the specifications and implementation of context information.

But this approach has also some drawbacks. For making true the benefits promised by model driven approach, tool support must be provided. These tools must assist the developer in the construction on the models and must automate as much as possible the transformation of the models into the final products. The development of such a kind of tools used to be a hard task but, hopefully, big player in the software industry like Microsoft and IBM are betting strong for the model driven development. As a result of these efforts, new tools are rising (for instance, the GMF project and the DSL Tools) that facilitate the development of applications which need to manage models. This important trend can change radically the way we develop software, and the pervasive systems field could take benefit of these advances.

BIBLIOGRAPHY

VII. APPENDIX: TOM VAN CUTSEM’S SLIDES
Fig. VII.2. Invited talk: AOP, page 2