

ULRR

Designing mobile aspect-oriented software architectures with ambients

Item Type	Book chapter
Authors	Ali, Nour;Ramos, Isidro
Citation	The Handbook of Research on Mobile Software Engineering: Design, Implementation, and Emergent Applications Alencar, Paulo Cowan, Donald (eds);chapter 29
Publisher	IGI Global
Download date	2026-06-09 09:10:30
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/2693

Handbook of Research on Mobile Software Engineering:

Design, Implementation, and Emergent Applications

Paulo Alencar
University of Waterloo, Canada

Donald Cowan
University of Waterloo, Canada

Volume II

Managing Director: Lindsay Johnston
Senior Editorial Director: Heather A. Probst
Book Production Manager: Sean Woznicki
Development Manager: Joel Gamon
Development Editor: Hannah Abelbeck
Acquisitions Editor: Erika Gallagher
Typesetter: Jennifer Romanchak
Cover Design: Nick Newcomer, Lisandro Gonzalez

Published in the United States of America by
Engineering Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: <http://www.igi-global.com>

Copyright © 2012 by IGI Global. All rights reserved. No part of this publication may be reproduced, stored or distributed in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher. Product or company names used in this set are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark.

Library of Congress Cataloging-in-Publication Data

Handbook of research on mobile software engineering: design, implementation, and emergent applications / Paulo Alencar and Donald Cowan, editors.

p. cm.

Includes bibliographical references and index.

Summary: "This book highlights state-of-the-art research concerning the key issues surrounding current and future challenges associated with the software engineering of mobile systems and related emergent applications"--Provided by publisher.

ISBN 978-1-61520-655-1 (hardcover) -- ISBN 978-1-61520-656-8 (ebook) -- ISBN 978-1-4666-1612-7 (print & perpetual access) 1. Mobile computing--Handbooks, manuals, etc. 2. Mobile computing--Research--Handbooks, manuals, etc. 3. Software engineering--Handbooks, manuals, etc. I. Alencar, Paulo, 1957- II. Cowan, D. D. (Donald D.), 1938-

QA76.59.H34 2012

004--dc23

2012002505

British Cataloguing in Publication Data

A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this book is new, previously-unpublished material. The views expressed in this book are those of the authors, but not necessarily of the publisher.

Chapter 29

Designing Mobile Aspect-Oriented Software Architectures with Ambients

Nour Ali

Lero - The Irish Software Engineering Research Centre, University of Limerick, Ireland

Isidro Ramos

Polytechnic University of Valencia, Spain

ABSTRACT

This chapter focuses on designing software architectures of mobile applications using an Aspect-Oriented Architecture Description Language (AOADL). The AOADL follows an approach called Ambient-PRISMA which enables designers to address, in an explicit and abstract way, the notion of location and mobility. Concretely, the AOADL extends the PRISMA AOADL by introducing a primitive called an ambient which is inspired by Ambient Calculus. An ambient defines a bounded place where other architectural elements (components and connectors) reside and are coordinated with elements that are outside an ambient's boundary. Architectural elements can enter and exit ambients. Ambients, as well as other architectural elements, are defined by importing aspects. Thus, behaviours that change the location of architectural elements are specified separately in distribution aspects. The objective of this chapter is to explain the steps that have to be followed when designing architecture configurations of distributed and mobile systems using the Ambient-PRISMA AOADL. This is explained by using a running example of a distributed auction system.

1. INTRODUCTION

In the last few decades, the information society has undergone important changes that have increased the complexity of software development. The software, the devices (PCs, laptops, PDAs, smart phones, etc) and people involved in current busi-

ness processes are often distributed, and mobile. As a result, the structure of software systems has become complex due to the new requirements that these systems need to satisfy such as mobility and adaptability.

Software architecture is a discipline that focuses on the design and specification of overall

system structure (Shaw & Garlan, 1996) (Bass, Clements, & Kazman, 2003). It is considered to be the bridge between the requirements and implementation phases of the software life cycle. The software architecture of a system describes its structure in terms of components (computational units), connectors (coordination units), and configurations (connections of components and connectors) (N. Medvidovic & Taylor, 2000).

Mobility causes changes in the structure of a distributed software system (Carzaniga, Picco, & Vigna, 1997). In this way, software architecture techniques can be useful to support the representation of these changes and the design of mobile software systems. Software architecture approaches have been proposed for modeling mobility (e.g. MobiS (Ciancarini & Mascolo, 1998) and Community (A. Lopes, Fiadeiro, & Wermelinger, 2002)). A comparison between different approaches has been made in (Ali, Solis, & Ramos, 2008). It can be inferred from this comparison that most of the proposed approaches do not provide an explicit notion for the location and mobility primitives.

An approach which provides an explicit notion of location and mobility is Ambient Calculus (AC) (Cardelli, 1999; Cardelli & Gordon, 1998), which is a process algebra which provides a primitive called ambient that represents a bounded place where computation happens. Ambients can form hierarchies and mobility is modelled by using ambient capabilities. The exit capability allows an ambient to exit its parent ambient. The enter capability provides an ambient to move in a sibling ambient in the ambient hierarchy.

A technique that aims to reduce complexity by increasing reusability, flexibility, and maintainability through the software development process is Aspect-Oriented Software Development (AOSD) (Filman, Elrad, Clarke, & Aksit, 2004). AOSD allows the separation of concerns by modularizing crosscutting concerns in separate entities called aspects (Kiczales, et al., 2001). Distribution and mobility have been identified as

crosscutting concerns (Lobato, et al., 2004; C. V. Lopes, 1997; Soares & Borba, 2002); separating them from other concerns of the software system will increase their reusability, and decrease cost and effort to maintain them.

Some researchers have proposed the integration of AOSD and software architecture (Chitchyan, 2005; Cuesta, del Pilar Romay, de la Fuente, & Barrio-Solórzano, 2005). PRISMA is an approach that integrates AOSD and Component Based Software Development (CBSDD) for describing software architectures (Pérez, et al., 2008). The crosscutting concerns of the software architecture are specified in aspects. Architectural elements (components and connectors) are then defined by using aspects.

Ambient-PRISMA (Ali, 2008) enriches PRISMA with the ambient concept inspired from Ambient Calculus (AC) in order to design, in an explicit and abstract way, the notion of location and mobility of architectural elements. AMBIENT-PRISMA introduces ambients as new kinds of architectural elements which define a bounded place where other architectural elements reside and can be coordinated with the exterior. Ambients can be hierarchically organized, conforming to a tree structure which is used to model distributed systems hierarchies. Architectural elements (including other ambients) can move by entering and exiting ambients. The functionality (behaviour) of an ambient is defined through mobility, coordination and distribution aspects.

This chapter focuses on explaining the steps that have to be followed by a user of the Ambient-PRISMA approach to specify mobility properties and design topologies of software architectures of mobile systems. The AOADL primitives will be illustrated by designing the aspect-oriented architecture of a distributed auction system with mobile agents.

The structure of the chapter is the following: Section 2 gives an overview of Ambient-PRISMA. Section 3 presents the distributed auction system

example that is used in the next sections to illustrate the Ambient-PRISMA AOADL. Section 4 explains how a user of Ambient-PRISMA designs a software architecture of a mobile system. The section gives guidelines to the users for detecting Ambient-PRISMA primitives, and then shows how the AOADL is used to design the software architecture. Section 5 presents related work. Finally, Section 6 presents conclusions and future work.

2. AMBIENT-PRISMA OVERVIEW

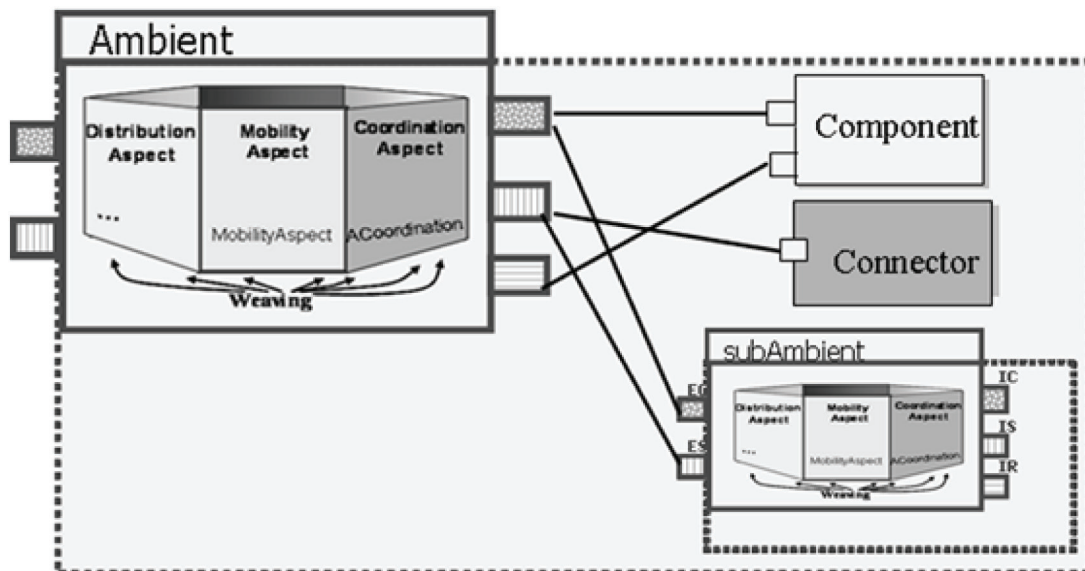
Ambient-PRISMA is an approach that enriches PRISMA with a concept called ambient, which represents a bounded place where architectural elements are located and provides them with mobility features. An ambient has been introduced by extending the concept of a connector for coordinating and separating what is inside and outside its boundary (see Figure 1). In addition, an ambient is an architectural element that needs to locate other architectural elements in its boundary. In Ambient-PRISMA, an ambient inherits the CBSD and AOSD characteristics of PRISMA

(as do other architectural elements of PRISMA such as components and connectors). The CBSD characteristics describe an ambient as a black box where it communicates with others by using ports that send and receive invocations of services. To enable the communication among architectural elements, channels called attachments are defined. Each attachment is defined by attaching two ports of different architectural elements.

In Figure 1, the *Component* and *Connector* are located in the same ambient. Also, an ambient can have other subambients. This allows the hierarchy of distributed and mobile systems to be modelled. An ambient offers two kinds of services to architectural elements located in it: mobility services and distributed architectural element services. Architectural elements that need these services are connected to the ambient through attachments (the lines in Figure 1).

The AOSD view describes an ambient through a set of aspects that can be weaved (as components and connectors). Weavings indicate that the execution of an aspect service can trigger the execution of services in other aspects. Weavings use operations called weaving operators that indi-

Figure 1. An ambient locating other architectural elements



cate that services of different aspects are executed. For example, if a weaving with the *after* operator is specified between service *s1* of aspect *A1* and service *s2* of aspect *A2*, this means that *s2* of *A2* is executed after *s1* of *A1* (Pérez, et al., 2008).

The ambient uses different aspects to specify the services it offers and requests. As Figure 1 shows each ambient must have a *MobilityAspect*, a *CoordinationAspect*, and a *DistributionAspect* (which can be different for each ambient).

As ambients are responsible for the mobility concern, all ambients must have the *MobilityAspect* to provide mobility services to their children architectural elements. The services of this aspect differentiate an ambient from other PRISMA connectors. This aspect is unique in Ambient-PRISMA and it is called *MobilityAspect*. The following are some of the *MobilityAspect* services:

- ***exit***: architectural elements that need to exit an ambient must invoke the *exit* service from their parent ambient (AC *exit* capability).
- ***enter***: architectural elements that need to enter an ambient must invoke the *enter* service from their parent ambient (AC *enter* capability).
- ***startMovement***: architectural elements that need to make a sequence of exits or enters must invoke the *startMovement* service. This service allows an ambient to prepare an architectural element to move.
- ***finishMovement***: architectural elements that finish making a sequence of exits or enters must invoke the *finishMovement* service.
- ***changeLocation***: an ambient invokes the *changeLocation* service of an architectural element in order to change the value of their location when an architectural element is moved.

MobilityAspect is a unique aspect that is reused by all ambients. As a result, ambients are defined by importing the generic mobility aspect and adapting it to the needs of the software system through weavings. For example, a LAN ambient may need some security policies that are different from a PC ambient inside of a LAN. Therefore, both the LAN and PC ambients import the same *MobilityAspect*, but the *MobilityAspect* is weaved with different security aspects.

The *ACoordination* aspect is also a unique aspect of an ambient and an ambient needs it for coordinating its architectural elements with external elements. This coordination aspect receives calls from external architectural elements of the ambient (distributed calls) and redirects them to corresponding architectural elements of the ambient. It also receives calls from its architectural elements and redirects them to the exterior.

A distribution aspect is used by architectural elements (ambients and others) in order to specify their distribution policies and allow them to be aware of their parent ambient. In this way, a distribution aspect is not a generic aspect that is reused by all ambients. A designer can specify a number of distribution aspects depending on the distribution requirements of a system. A distribution aspect stores the name of the parent ambient of an architectural element. This aspect also includes the specification of whether an element is mobile or not. A mobile element requests mobility services from its parent ambient. When an element is mobile, its parent ambient can be changed.

In Ambient-PRISMA, there are three kinds of ambients. This separation is necessary since each one of them has its own semantics and constraints.

- **Group Ambients**: They represent a collection of architectural elements. They can be used for grouping a set of architectural elements that share specific security poli-

cies or for moving architectural elements together. A Group ambient can lodge other Group ambients, Components or Connectors.

- **Site Ambients:** They represent physical locations; that is, Site ambients have a physical address. They can be devices or physical regions. A PC or a PDA are examples of ambients that can be modelled as Sites. Site ambients can only contain Components, Connectors or Group ambients.
- **Virtual Ambients:** They represent domains which can contain other Virtual or Site ambients. Virtual ambients model networks or virtual spaces where sites are located. A Virtual ambient that consists of a collection of Sites represents the boundary of a network (e.g. a LAN). The modelling of a Virtual ambient hierarchy can represent the hierarchy of LANs to form a Wide Area Network (WAN). In Ambient-PRISMA there is always a default Virtual Ambient that represents the root of an ambient hierarchy called Root.

3. AUCTION SYSTEM RUNNING EXAMPLE

The example used throughout the paper consists of an electronic Auction System which offers its products that can be bid and sold. This is a simplified version of a case study used in (Ali, 2008). The Auction System allows its customers to use mobile agents to search for interesting products and bid on their behalf. In this way, the customers can instruct the agents and disconnect from the network. This reduces network latency and traffic, and agents can respond to changes in the auction quickly.

The Customer moves a Bidder agent to the site of the Auction House (AuctionSite) which offers products for bidding. Once the Bidder is at the Auction House site, it keeps bidding for a specific product until it wins, the bid reaches a

maximum price limit, or when it loses. When the auction has finished (won or lost) or when the current biddings exceed the maximum price limit, the Bidder moves back to the site of the Customer (ClientSite). In addition, the Customer can decide to stop the Bidder and move it back whenever appropriate.

Figure 2 shows the software architecture configuration of the example. The Customer1 and the Bidder1 are component instances that are coordinated through an *AgentCustCnct1* connector instance. The Bidder1 component instance is coordinated with the AuctionHouse1 component instance through the AuctionHouseCnct1 connector instance.

4. DESIGNING MOBILE SOFTWARE ARCHITECTURES USING THE AMBIENT-PRISMA AOADL

This section shows how software architectures with distribution and mobility characteristics can be designed using the Ambient-PRISMA approach. First, guidelines for identifying these characteristics are presented. Then, the AOADL is used to specify distribution and mobility characteristics of the example presented in section 3.

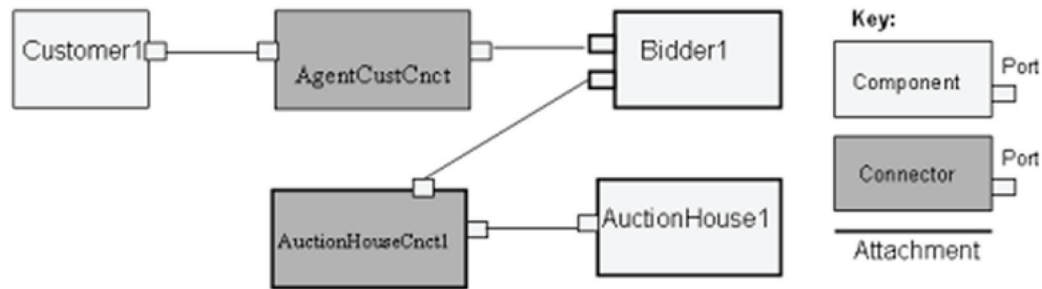
4.1 Guidelines for Identifying Distribution Aspects and Ambients

In Ambient-PRISMA, the software architecture is composed of architectural elements which can be ambients, components, and connectors. The main aspects that have to be detected for Ambient-PRISMA are the distribution aspects. The following steps can be taken in order to identify them.

4.1.1 Identification of Virtual and Site Ambient

A distributed system consists of physical locations that can be located on different domains. The kinds of site ambients and virtual ambients

Figure 2. Software architecture of the auction system example without ambients



depend on the physical locations and the domains, respectively. As a result, the analyst has to know the kind of network to be modelled. For example, in the Mobile Agents case study the network is a WAN that consists of hosts located on different LANs. However, a wireless sensor actuator network would consist of regions and a host.

In addition, the analyst has to know the properties of each virtual or site ambient. This is due to the fact that site ambients of the same network can have different characteristics. For example, two site ambients may need different security properties.

In the example, we have two Site ambients the ClientSite and the AuctionSite ambients. If we assume that these two ambients have the same characteristics they can be instances of a HostSite. In addition, we can have a Root ambient as a Virtual ambient type to represent the WAN network.

4.1.2 Identification of Mobility Properties for Components and Connectors

Once components and connectors are identified using the guidelines discussed in (Pérez, et al., 2008), the analyst has to identify which of them need to be mobile. In Fug et al. (Fuggetta, Picco, & Vigna, 1998) migration is classified to be proactive or reactive depending on who decides that an element needs to move. In subjective migration (sometimes called proactive or autonomous

migration), a mobile entity decides when it needs to move and to where it has to move. On the other hand, in objective migration (sometimes called reactive or passive migration) the migration decision is determined by another executing entity.

In this way, some of the following questions can be asked at this step:

- Is an architectural element mobile or not?
- Does a mobile architectural element need objective moves?
- Does an architectural element cause objective moves?

Once an analyst answers these and similar questions, the mobility services can be detected. As a result, interfaces needed for requesting and providing objective moves are detected. If there are no objective moves, then there may be no interfaces for distribution or mobility. In addition, ports and attachments can also be identified.

In our example, the Customer component is stationary (not mobile). However, it can order the Bidder component to move. As a result, the Customer component needs a port that allows the Customer to request the Bidder to move. In addition, the Bidder needs a port that allows it to receive the mobility request. This means that the Customer and the Bidder components need ports that publish the same interface for a service that causes mobility.

4.1.3 Identification of Group Ambient

Group ambients are identified after components and connectors since group ambients represent logical domains, i.e, a group of architectural elements that share a set of characteristics. As a result, an analyst can identify group ambients by assessing whether there is a set of architectural elements that need to move together or not, share some resources, or support similar security characteristics.

Once group ambients are identified, the analyst can answer the questions presented in section 4.1.2, such as whether a group ambient is mobile or not, or whether it is mobile through objective or subjective moves. Interfaces, ports, and attachments that a group ambient needs can also be identified by answering similar questions.

In our example, there are no group ambients. However, in the case that a Customer needs to move two agents to the AuctionSite, e.g., an agent for searching for the best product and another for bidding, a group ambient can be designed for locating the two agents. In this case, the Customer could order the group ambient to move and as a consequence both agents would move.

4.1.4 Identification of Distribution Aspects

All architectural elements in Ambient-PRISMA need a distribution aspect. Each kind of ambient or architectural element can have a different distribution aspect. In addition, an analyst can answer the following questions in order to identify the different distribution aspects of architectural elements:

- What causes an architectural element to move?
- Does an architectural element have any constraints on the locations it can move to?

- Does a mobile architectural element need to perform specific tasks after or before moving?

When these questions are answered, an analyst can identify the different kinds of architectural element behaviours. In this way, the behaviour needed in a distribution aspect, and the required weavings that relate this aspect to other aspects can be specified. For example, the Bidder component can move when certain conditions occur (e.g., when an auction has finished, or when the current biddings exceed the maximum price limit.

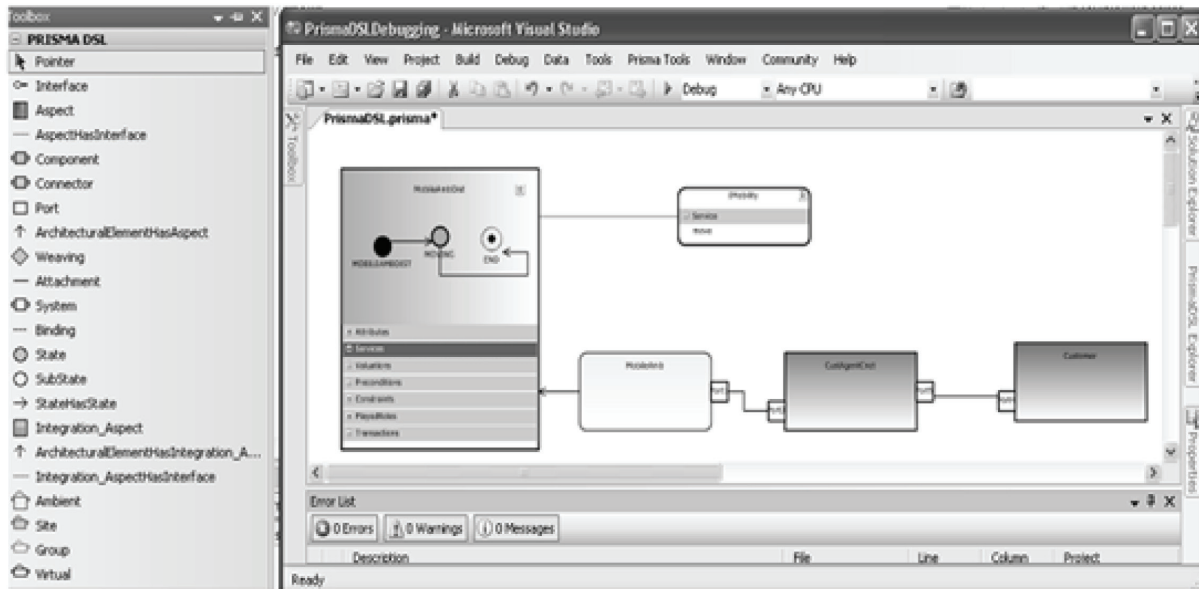
4.2 Designing the Auction System Mobile Architecture with the AOADL

Once the architectural elements and their distribution and mobility properties are identified, the analyst can either model graphically using the Ambient-PRISMA CASE Tool modelling graphically (see Figure 3) or textually specifying a software architecture using the AOADL. The modelling process is flexible enough to allow a user to choose between two options for including distribution and mobility characteristics in the modelling process:

- 1st Option:** To model the PRISMA aspect oriented software architecture as presented in (Pérez, et al., 2008) and then complete it with the distribution and mobility characteristics such as distribution aspects, and ambients.
- 2nd Option:** To model the distribution and mobility characteristics in parallel with the PRISMA aspect-oriented software architecture.

The methodology includes five modelling steps: Step 1 – interface; Step 2 - distribution aspects; Step 3 - components and connectors; Step 4 - ambient, and 5 - instantiation and configuration (see Figure 4). The enumeration of these steps is not restrictive. Similarly to PRISMA,

Figure 3. Toolbox and the modelling tool



this enumeration only indicates the dependencies between the concepts (Pérez, et al., 2008). For example, the modelling of ambients can be done before, during or after the components and connectors modelling step. Thus, the modelling process is incremental.

The following sections explain the steps for modelling the distribution and mobility characteristics using the textual AOADL are explained.

4.2.1 Definition of Interfaces

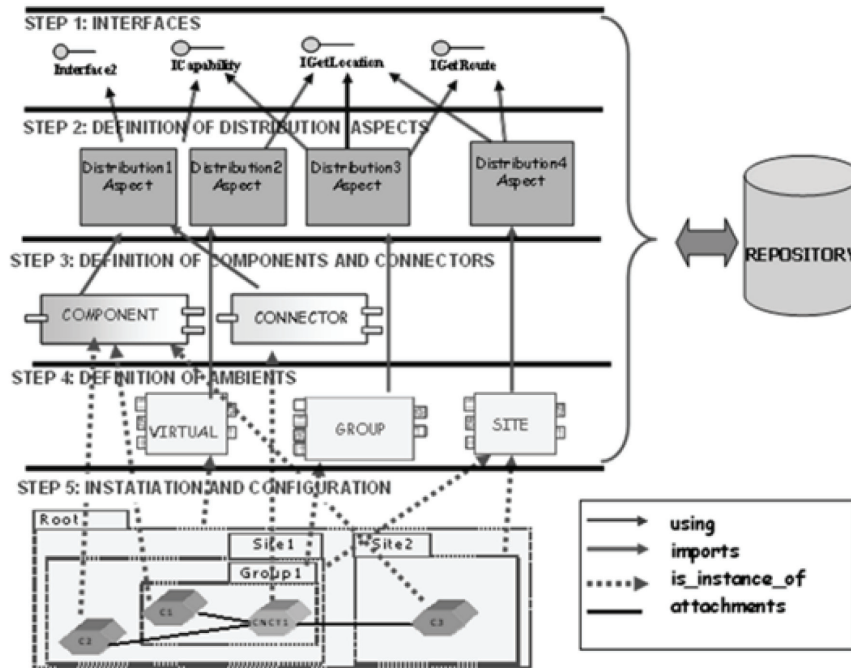
Interfaces publish a set of services. The interfaces identified through steps 4.1.2 and 4.1.3, are specified by modelling the services and their signatures. The interfaces specified are stored in a reuse repository (see step 1, Figure 4). Interfaces defined for distribution and mobility can be defined in parallel with interfaces for functional or other concerns of the software architecture. For example, an interface called *IMobility* is specified in Figure 5. The interface publishes a service called *move* which has an input parameter called *NewAmbient* in order to indicate its destination ambient.

Ambient-PRISMA also provides predefined interfaces: the *ICapability*, the *IGetLocation*, and the *IGetRoute*. The *ICapability* interface publishes a set of services which include: the *exit* service, the *enter* service, the *startMovement* and *finishMovement* and the *changeLocation* (described in section 2). The *IGetLocation* interface publishes a service called *getLocation*. This interface is used in order to inform the name of a parent ambient. *IRoute* interface publishes a service called *getRoute*. This service allows an ambient to provide its internal architectural elements for consulting a route of any architectural element of a software architecture. A route of an architectural element consists of the names of ambients where the architectural element is positioned in a tree hierarchy of ambients.

4.2.2 Definition of Distribution Aspects

The distribution aspects identified in section 4.1.4 are modelled after the specification of interfaces because many aspects have to use interfaces if they need to offer and provide public services.

Figure 4. The modelling stage of the Ambient-PRISMA methodology based on the PRISMA methodology (Pérez, et al., 2008)



Distribution aspects can be defined in parallel with aspects that describe other crosscutting concerns of PRISMA software architecture.

An example of a distribution aspect called *BidderDist* is specified in Figure 6, which will be imported by a Bidder component (see Figure 8). The *BidderDist* aspect uses the *IMobility* interface (specified in Figure 5) and the *ICapability* interface.

The *BidderDist* aspect has a variable attribute called *location*. This attribute is variable because its value can change during the life of the aspect. The *location* attribute is marked with bold due to the fact that location is an Ambient-PRISMA

keyword. The *location* attribute stores the name of the parent ambient of the architectural element that imports the aspect. The *location* attribute is also a **NOT NULL** attribute due to the fact that its value can never be NULL. This attribute exists in all distribution aspects of architectural elements except for the *Root* architectural element (the Virtual ambient that represents the root of an ambient hierarchy). In addition, the *BidderDist* attribute has a constant attribute called *originLocation*. The *originLocation* is a **NOT NULL** attribute that saves the value of the origin ambient of the architectural element which imports this aspect. This allows an architectural element to move back to its origin when needed.

The *begin* service has a *Valuation* for assigning the value of the attributes specified as **NOT NULL**. As a result, the *location* and the *originLocation* attributes are assigned with the value stored in the *input* parameter called *ParentAmbient*.

Figure 5. IMobility interface

```

Interface IMobility
    move(input NewAmbient: Ambient);
End_Interface IMobility
    
```

Figure 6. Specification of the BidderDist distribution aspect

```

Distribution Aspect BidderDist using IMobility, ICapability
Attributes
Variable
  location: Ambient NOT NULL;
Constant
  originLocation: Ambient NOT NULL;
Services
begin(input ParentAmbient: Ambient)
  Valuations
  [begin (ParentAmbient)]
    location := ParentAmbient, originLocation:= ParentAmbient;
in changeLocation(input Name: ArchitecturalElement,
  input NewLocation: Ambient)
  Valuations
  [changeLocation()] location:=NewLocation;
out startMovement(input Name:ArchitecturalElement);
out exit (Name: ArchitecturalElement);
out finishMovement(input Name:ArchitecturalElement);
out enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
end();
Preconditions
  in changeLocation(Name, NewLocation) if {self.Name==Name};
Played_Roles
  CapParent for ICapability ::= startMovement!(Name) →
    exit!(Name) →
    enter!(Name, NewAmbient) →
    finishMovement!(Name) →
    changeLocation?(Name, NewLocation);
  CUSTMOVESBIDDER for IMobility ::= move?(NewAmbient);
TRANSACTIONS in MOVE (NewAmbient:string)
  move = CapParent_startMovement!(self.Name) → MOVE1;
  MOVE1 = CapParent_exit!(self.Name) → MOVE2;
  MOVE2 = CapParent_enter!(self.Name, NewAmbient) → MOVE3;
  MOVE3 = CapParent_finishMovement!(self.Name);
  MOVE4 = CapParent_changeLocation?(Name, NewLocation);
Protocol
  BIDDERDIST:= begin → BIDDERDIST1;
  BIDDERDIST1:=(CUSTMOVESBIDDER_MOVE?(NewAmbient)+ MOVE?(originLocation)
    + end) → BIDDERDIST1;
End_Distribution Aspect BidderDist
    
```

The *changeLocation*, *startMovement*, *exit*, *enter* and *finishMovement* services of the *ICapability* interface are specified. The *changeLocation* service has an *in* behaviour, i.e., it is provided and executed by the aspect. When the *changeLocation* is executed, the value received through the *NewLocation* parameter is assigned to the *location* attribute. A *Precondition* is associated to the *changeLocation* service. The precondition specifies that the *changeLocation* is executed only if the value received in the *Name* parameter is equal to the name of the architectural element that imports the aspect (*self.Name*). The *startMovement*, *exit*,

enter, and *finishMovement* services are requested by the aspect (*out*).

The *BidderDist* aspect has two played_roles: *CapParent* and *CUSTMOVESBIDDER*. The *CapParent* played specifies how the services of the *ICapability* interface can be invoked. The *CUSTMOVESBIDDER* played_role specifies how the service of the *IMobility* interface can be invoked. It specifies that the *move* service can only have a server behaviour (*in*) when it is invoked through the *CUSTMOVESBIDDER* played_role.

The *BidderDist* aspect specifies that the architectural element which imports it, is a mobile architectural element. This is specified in a trans-

action called *MOVE* that uses the services of the *CapParent* played_role. The *MOVE* transaction consists of invoking the services of the *CapParent* played_role. The *MOVE* transaction consists of requesting the *startMovement* service, receiving the result of the *startMovement* service (the *CommuniList* output parameter), requesting the *exit*, the *enter*, and the *finishMovement* services, and receiving the *changeLocation* service.

Finally, the protocol section of the *BidderDist* aspect specifies how its services can be executed. The protocol specifies that the *begin* service is first executed. Then, the *BIDDERDIST1* process is executed. The *BIDDERDIST1* process consists of executing the *MOVE* transaction when it is invoked through the *CUSTOMOVESBIDDER* played role (an external architectural element invokes it), executing the *MOVE* transaction when it internally invoked (triggered by the architectural element that imports the aspect) or the aspect is ended. The *BIDDERDIST1* process can be invoked many times.

Distribution aspects of ambient must save the name of its parent ambient (as the ones of components and connectors), must provide a service which allows others to consult its current location, and provide a service which allows architectural elements located in it to consult routes of architectural elements. Therefore, a distribution aspect uses the predefined interfaces *IGetLocation* interface and the *IGetRoute* interface (see Figure 7).

Figure 7 also shows the specification of an aspect called *ADist* which is imported by a site ambient (see Figure 9). The aspect specifies the *location* attribute which stores the name of the parent ambient to be a constant. This indicates that the ambient importing this aspect cannot change its location, i.e. it is not mobile. A distribution aspect of a Site ambient also includes an attribute called *physicalLocation* of type *loc*. The *physicalLocation* attribute stores the physical location that the site represents e.g. the URL or URI. This attribute is a *NOT NULL* attribute because it always has to have a value. As a result,

the *begin* service must have a parameter that gives a value to the *physicalLocation* attribute as well as to the *location* attribute when the aspect starts executing. The aspect specifies that the *getLocation* service is of type *in/out* i.e., the service first receives the request (*in*), then the value of the *location* attribute is assigned to the *Location* parameter through the *Valuation*. Finally, the service sends the *Location* parameter (*out*). In addition, aspects specify that the *getRoute* service is also of type *in/out*.

4.2.3 Components and Connectors

Components and connectors are defined in step 3 because they are defined by importing aspects, defining ports, and defining weavings (see step 3, Figure 4). In Ambient-PRISMA, the specification of components and connectors has to import a distribution aspect. As a result, in this step each one of the architectural elements identified in step 4.1.2 has to indicate which distribution aspect it imports.

In addition, architectural elements that are mobile have to define a port called *DCapPort* for requesting capabilities from their parent ambients. Furthermore, an architectural element that needs to request routes of other architectural elements has to define the port called *DRoutePort*. Other ports can be defined depending on whether there are objective moves.

Figure 8 shows the specification of a *Bidder* component. It can be observed, that the *Bidder* component imports the *BidderDist* distribution aspect (specified in Figure 6) and the *BidderFunct* functional aspect. These two aspects are coordinated through a weaving which specifies that after the execution of the *finishedAuction* service of the *BidderFunct* aspect, the move service of the *BidderDist* aspect is executed. The *Bidder* also specifies four ports: *DCapPort*, *DMovingPort*, *CustBidderPort*, and *BidderAuctPort*. The *DCapPort* port has to be specified in order to indicate that

Figure 7. A distribution aspect of a site ambient

```

Distribution Aspect ADist using IGetLocation, IGetRoute
Attributes
  Constant
  location : Ambient NOT NULL;
  physicalLocation: loc NOT NULL;
Services
  begin(input ParentAmbient: Ambient, input PhysicalLocation: loc)
    Valuations
    [begin (ParentAmbient, PhysicalLocation)]
      location := ParentAmbient,
      physicalLocation:=PhysicalLocation;
  in/out getLocation(output Location: Ambient)
    Valuations
    [in getLocation(output Location)] Location := location;
  in/out getRoute(input Name: ArchitecturalElement, output Route[]: Ambient)
    Valuations
    [in getRoute(input Name, output Route)] Route:= deriveRoute(Name);
  end;
Played_Roles
  INTRROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);
Protocol
  DIST:= begin(ParentAmbient, PhysicalLocation) → DIST1;
  DIST1:=(getLocation?(Location)→ getLocation!(Location))+
    (INTRROUTE.getRoute?(Name, Route)→ INTRROUTE.getRoute!(Name, Route))+
    end;
End_Distribution Aspect ADist

```

the *Bidder* component needs to request mobility services from its parent ambient.

Each port of the *Bidder* component exports the services of different interfaces and has a different played_role defined for the interface. The played_roles of the *CustBidderPort* and the *BidderAuctPort* ports are defined by the *Bidder-Funct* aspect. The played_roles of the *DCapPort* and the *DMovingPort* ports are defined by the *BidderDist* aspect.

According to the specification in Figure 8, an architectural element needs to import a distribution aspect in order to indicate where an instance is located at instantiation time (see *new* in Figure 8). In this way, each time an architectural element is instantiated, the name of the ambient where the instance is located has to be indicated.

Once components, and connectors are defined, they are stored in the repository. The storage of these architectural elements implies the storage of the aspects they import. In this way, they can be reused in other software architecture descriptions.

4.2.4 Ambients

Ambients that are identified in steps 4.1.1 and 4.1.3 are defined in a similar way to the rest of architectural elements. The difference is that ambients have predefined constructs such as the *MobilityAspect* aspect, the *ACoordiantion* aspect, and the *ICapability* interface. As a result, some ambients need interfaces and aspects that should be defined by the user and others do not (see step 4, Figure 4). Ambients that define extra behaviours can specify extra ports, aspects and weavings. For example, an ambient that needs to be mobile must define the *DCapPort* (as in a mobile component (see section 4.2.3)).

It can also be noticed, that an ambient type is defined independently from the other types of architectural elements. Therefore, some ambients can be defined at any step of the modelling process due to the fact that they do not need extra interfaces or aspects. Once ambients are defined, they are

Figure 8. Specification of the Bidder component

```

Component_type Bidder
Import Distribution Aspect BidderDist;
Import Functional Aspect BidderFunct;
Weavings
  //This weaving is for moving to the customer location after the auction
  has finished.
  BidderDist.move(CustLocation) after BidderFunct.finishedAuction(BidWon);
End_Weavings
Ports
  DCapPort: ICapability Played_Role BidderDist. CapParent;
  DMovingPort: IMobility Played_Role BidderDist.CUSTOMMOVESBIDDER;
  CustBidderPort: ICustBidder Played_Role BidderFunct.CUSTBIDDER;
  BidderAuctPort: IBidderAuct Played_Role BidderFunct. BIDDERAUCT;
End_Ports
new(input ParentAmbient: Ambient)
  { BidderDist.begin(ParentAmbient);
    BidderFunct. begin();
  }
...
End Component_type Bidder;

```

stored in the repository in order to be reused in other software architecture descriptions.

Figure 9 shows the specification of a site ambient called *HostSite*. The ambient imports the predefined *MobilityAspect* aspect, the predefined *ACoordination* aspect, and the *ADist* aspect defined in Figure 7. The five predefined ports of

an ambient are specified with their played roles. These ports are the ones which appear marked in Figure 1. The predefined weaving which triggers the *getLocation* service of the *ADist* aspect instead of the *getParent* service of the *MobilityAspect* aspect is also specified. In addition, the sections which create and destroy the ambient are specified.

Figure 9. Specification of HostSite ambient

```

Ambient_Site type HostSite
Import Mobility Aspect MobilityAspect;
Import Coordination Aspect ACoordination;
Import Distribution Aspect ADist;
Weavings
  ADist.getLocation(Location) instead
  MobilityAspect.getParent(Parent);
End_Weavings
Ports
  InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
  ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
  EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
  InServicesPort: ICall Played_Role ACoordination.INTERIOR;
  InRoutePort: IGetRoute Played_Role ADist.INROUTE;
End_Ports
new(input ParentAmbient: Ambient, input PhysicalLocation: loc)
  { MobilityAspect.begin();
    ACoordination.begin();
    ADist.begin(ParentAmbient, PhysicalLocation);
  }
...
End Ambient_Site type HostSite;

```

The template of the ambient *HostSite* does not include the architectural elements that it locates or their attachments. It is assumed that ambients of an architectural model can locate all architectural elements and that all the possible attachments that connect architectural elements with ambients are created automatically. In this way, an ambient can be reused for different architectural models with different architectural elements.

In Ambient-PRISMA, if the analyst does not specify any ambient in its architectural model the Case Tool assumes that there is a default site ambient. In addition, the analyst does not have to define any virtual ambient in the architectural model. It is assumed that when no virtual ambient is defined, there will only be a Root ambient of type virtual.

4.2.5 Instantiation and Configuration

Finally, a specific configuration of a distributed software architecture is defined. Types of ambients, components, and connectors are instantiated by specifying where the instances are located. Then, attachments are created for connecting component and connector instances with each other.

The following steps are followed to define a configuration with ambients:

- Physical locations of the software architecture are instantiated by the *loc* data type.
- Ambients are instantiated in three ways: First, virtual ambients are instantiated by indicating their parent ambient with the exception of the Root ambient which does not have a parent ambient. Second, each site ambient is instantiated by indicating both its parent ambient and an instance of the *loc* data type. Finally, group ambients are instantiated by indicating their parent ambient.
- Constraints for specifying which ambients cannot be connected.

- Components and connectors are instantiated by indicating the name of their parent ambients.
- Attachments that connect component and connector instances are created.

Figure 10 shows the specification of the configuration called *MobileAgentsAuctionConf* of the *MobileAgentsAuction* architectural model. The configuration is defined by giving the possible physical locations that are instances of the *loc* data type. The *loc* data type is needed in order to model the different physical locations that form a specific distributed system. Then the ambients are instantiated. First, the *Root* ambient is instantiated. It can be noticed that the constructor of the *Root* ambient does not have any parameters because the *Root* ambient is not located. Then, site ambients are instantiated by indicating, in their constructor, the name of their parent ambient (their location) and the physical address they represent. For example, an ambient instance called *ClientSite* is created from the *HostSite* type ambient (see Figure 9). The constructor of the *ClientSite* instance indicates that it is located in the *Root* ambient and that its physical location is *IPI*. Then, components and connectors are instantiated by indicating their locations in their constructors. In addition, the attachments are also instantiated by indicating the component and connector instances, and their ports that need to be connected.

5. RELATED WORKS

One of the first approaches for the specification of software architectures for distributed systems is known as Darwin (Magee, Dulay, Eisenbach, & Kramer, 1995). However, Darwin does not support mobile systems. Some architectural approaches have modelled mobility in a technology independent way, such as Community (A. Lopes, et al., 2002), which is based on category theory; MobiS (Ciancarini & Mascolo, 1998) based on

Figure 10. A specification of the MobileAgentsAuctionConf configuration

```

Architectural_Model_Configuration MobileAgentsAuctionConf =
New MobileAgentsAuction
{
    IP1 = new loc(ip.of.host.1);
    IP2 = new loc(ip.of.host.2);

    ROOT = new Root() ;

    ClientSite = new HostSite(ROOT, IP1);
    AuctionSite = new HostSite(ROOT, IP2);

    AuctionHouse1 = new AuctionHouse ("AuctionSite",1,1,1,1,1,1,1)
    Customer1 = new Customer("ClientSite");
    Bidder1 = new Bidder("ClientSite");
    AgentCustCnct1 = new AgentCustCnct("ClientSite");
    AuctionHouseCnct1 = new AuctionHouseCnct("AuctionSite");
    AttchAuct1Cnct = new AttchAuctCnct (AuctionHouseCnct1,
                                        CnctAuctPortBidder,
                                        AuctionHouse1, BidderAuctPort);
.....

```

a multiple tuple-space model, CHAM (Inverardi & Wolf, 1995) based on molecular solutions; Con-Moto (Schäfer, 2006) based on π -calculus; and LAM model (Xu, Yin, Deng, & Ding, 2003) based on Petri nets. Each one of these approaches models locations and mobility in a unique way. For example, Community models locations using a data type where a location is represented by a value of a variable. Other approaches such as MobiS and LAM-Model use composite components for representing locations. Con-Moto provides special kinds of components called physical components to model locations. In most approaches, mobility is modeled as a reconfiguration of the software architecture. However in Community, mobility is modeled as a change in a value.

The C2Sadel ADL has adapted a style to support both distribution and mobility. The style (N. Medvidovic, Rakic, M., 2001) provides software connectors that are able to move components. It also has an implementation infrastructure to support this architectural style. However, there is no separation between coordination and distribution. Therefore, the components are the only architectural elements that are mobile, and the connectors are static. In our approach, we have introduced the ambient concept where mobility

can be applied to a wider range to include connectors and ambients. Also, Ambient-PRISMA combines CBSD and AOSD, achieving a better separation of concerns, such as separating the specification of the distribution concern from the mobility concern.

The work of Lopes (A. Lopes, et al., 2002) describes the semantics of externalizing the distribution dimension to support distribution and mobility for software architectures. This distribution dimension is very similar to a connector, but instead of containing the business logic, it controls the rules for mobility and location. In this way, a separation between computation, coordination and distribution is achieved. A difference between our work and Lopes's work is that our work defines the semantics of distribution and mobility by using Ambient Calculus. This allows our approach to have an explicit primitive to represent a location with boundaries.

6. CONCLUSION AND FUTURE WORK

This chapter has presented Ambient-PRISMA, an approach which models mobile systems us-

ing an aspect-oriented architectural approach with ambients. The approach provides the requirements analyst with a set of guidelines to facilitate its task for detecting distribution and mobility characteristics that can be modelled using Ambient-PRISMA. This chapter has also presented the Ambient-PRISMA AOADL. The Ambient-PRISMAAOADL enriches the PRISMA AOADL with new syntactical constructs for specifying aspect-oriented software architectures of distributed and mobile software systems in a transparent way. Predefined types such as interfaces, aspects and ports can be used in defining a software architecture. Designers can define the needed mobility behaviours in aspects and weave them with the predefined types. The Ambient-PRISMAAOADL also permits to specify locations using ambients and to deploy the architectural elements in ambients.

In the near future, Ambient-PRISMA is going to be enriched with security primitives. Although ambients, by definition are bounded places that control what can enter them and exit from them, our case studies have not considered modelling security scenarios yet. Different security protocols could be predefined in different security aspects which could be reused by the different ambients. In addition, group ambients can be used for supporting privacy and trustworthiness for protecting data.

Another direction that is worth exploring is Service Oriented Architecture (SOA), which is an architectural style that is based on loose coupling for enabling components to collaborate in distributed and highly dynamic environments (Singh & Huhns, 2004). Ambients can be environments for providing services, and where services can move by moving ambients. An approach that is already taking this direction is Ambient-SoaML (Ali & Babar, 2009) which extends the Service oriented architecture Modeling Language (SoaML) with ambients.

SOA has many applications. One of the most actual applications is Cloud Computing where scalable and virtualized resources are delivered

as services over the internet (Hayes, 2008). Services can deliver infrastructures, platforms and software. One possible architectural style that can be used in Cloud Computing is the one with ambients. An interesting research work would be to investigate how ambient architectures deliver cloud computing services.

Finally and not least, context awareness is an area which is worth exploring. In the context of the work presented in this paper, ambients can be extended in order to take into account information of the environments which are delimited by the boundaries they represent. In this way, Ambient-PRISMA AOADL can be used to model mobile context-aware applications.

ACKNOWLEDGMENT

This work was supported by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, METAproject TIN2006-15175-C05-01. This work has been performed while Nour Ali was affiliated to the Polytechnic University of Valencia. At the moment of writing the chapter, Dr. Nour Ali is a research fellow in Lero, funded by the Science Foundation Ireland grant 03/CE2/I303_1. The authors would also like to acknowledge Dr. Sarah Beecham, and Dr. Carlos Solis for their valuable comments after reviewing this chapter.

REFERENCES

- Ali, N. (2008). *Ambient-PRISMA: Ambients in aspect-oriented software architecture*. Valencia: Polytechnic University of Valencia.
- Ali, N., & Babar, M. A. (2009). *Modeling service oriented architectures of mobile applications by extending SoaML with ambients*. Paper presented at the 35th Euromicro SEAA Conference.

- Ali, N., Solis, C., & Ramos, I. (2008). *Comparing architecture description languages for mobile software systems*. Paper presented at the 1st International Workshop on Software Architectures and Mobility.
- Bass, L., Clements, P., & Kazzman, R. (2003). *Software architecture in practice* (2nd ed.). Addison-Wesley Professional.
- Cardelli, L. (1999). *Abstractions for mobile computation* (pp. 51–94). Secure Internet Programming.
- Cardelli, L., & Gordon, A. D. (1998). *Mobile ambients*. Paper presented at the Foundations of Software Science and Computational Structures: First International Conference FOSSACS '98.
- Carzaniga, A., Picco, G., & Vigna, G. (1997). *Designing distributed applications with mobile code paradigms*. Paper presented at the ICSE '97: The 19th International Conference on Software Engineering.
- Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., & Bakker, J. (2005). *Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design (No. AOSD-Europe Deliverable D11)*. Lancaster, UK: Lancaster University.
- Ciancarini, P., & Mascolo, C. (1998). *Software architecture and mobility*. Paper presented at the Third International Workshop on Software Architecture.
- Cuesta, C. E., del Pilar Romay, M., de la Fuente, P., & Barrio-Solórzano, M. (2005). Architectural aspects of architectural aspects. *Software Architecture, LNCS, 3527*, 247–262. doi:10.1007/11494713_18
- Filman, R. E., Elrad, T., Clarke, S., & Aksit, M. (2004). *Aspect-oriented software development*. Addison Wesley Professional.
- Fuggetta, A., Picco, G. P., & Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering, 24*(5), 342–361. doi:10.1109/32.685258
- Hayes, B. (2008). Cloud computing. *Communications of the ACM, 51*(7), 9–11. doi:10.1145/1364782.1364786
- Inverardi, P., & Wolf, A. L. (1995). Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering, 21*(4), 373–386. doi:10.1109/32.385973
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). *An overview of AspectJ*. Paper presented at the 15th European Conference on Object-Oriented Programming.
- Lobato, C., Garcia, A., Romanovksy, A., Sant'Anna, C., Kulesza, U., & Lucena, C. (2004). *Mobility as an aspect: The AspectM framework*. Paper presented at the 1st Brazilian Workshop on Aspect-Oriented Software Development – WASP'04, SBES'04.
- Lopes, A., Fiadeiro, J. L., & Wermelinger, M. (2002). Architectural primitives for distribution and mobility. *SIGSOFT Software Engineering Notes, 27*(6), 41–50. doi:10.1145/605466.605473
- Lopes, C. V. (1997). *D: A language framework for distributed programming*. Boston, USA: Northeastern University.
- Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). *Specifying distributed software architectures*. Paper presented at the 5th European Software Engineering Conference.
- Medvidovic, N., & Rakic, M. (2001). *Exploiting software architecture implementation infrastructure in facilitating component mobility*. Paper presented at the Software Engineering and Mobility Workshop.

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), 70–93. doi:10.1109/32.825767

Pérez, J., Ali, N., Carsi, J. A., Ramos, I., Álvarez, B., & Sanchez, P. (2008). Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Information and Software Technology*, 50(9-10), 969–990. doi:10.1016/j.infsof.2007.08.007

Schäfer, C. (2006). *Modeling and analyzing mobile software architectures* (pp. 175–188). Software Architecture.

Shaw, M., & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Prentice Hall.

Singh, M. P., & Huhns, M. N. (2004). *Service-oriented computing semantics, processes, agents*. John Wiley & Sons. doi:10.1002/0470091509

Soares, S., & Borba, P. (2002). *PaDA: A pattern for distribution aspects*. Paper presented at the Second Latin American Conference on Pattern Languages Programming

Xu, D., Yin, J., Deng, Y., & Ding, J. (2003). A formal architectural model for logical agent mobility. *IEEE Transactions on Software Engineering*, 29(1), 31–45. doi:10.1109/TSE.2003.1166587

KEY TERMS AND DEFINITIONS

Ambient: An ambient is a place that is delimited by a boundary and where computation can happen.

Architecture Description Language (ADL): A language that allows the description of software architectures in a formal way. An ADL should allow a description of a software architecture in terms of components, connectors and configurations.

Aspect: A software entity that encapsulates crosscutting concerns.

Aspect-Oriented Software Development (AOSD): A software development paradigm that proposes the separation of crosscutting concerns throughout the different stages of the software life cycle: requirements, architecture, design and implementation.

Component: A unit of decomposition of a system which is reusable, does not have dependencies with other elements of a system and should be easily integrated.

Component-Based Software Development (CBSD): A software development paradigm that proposes to construct the system by connecting entities that provide and require services.

Connector: an architectural element that describes the interactions among components.

Mobile System: Is a computing system with mobile entities. These entities can be either software or hardware.

Software Architecture: Is a design level where the structure of a system is described through elements, and the interconnections among them.

Weavings: Weaving is the process that combines concerns of the system (which can be modularized in aspects and objects) following weaving rules.