

Mobile Ambients in Aspect-Oriented Software Architectures

Nour Ali, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose A. Carsí

Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
{ nourali, jeperez, ccosta, iramos, pcarsi }@dsic.upv.es

Abstract. Nowadays, distributed and mobile systems are acquiring importance and becoming widely extended for supporting ubiquitous computing. In order to develop such systems in a technology-independent way, it is important to have a formalism that describes distribution and mobility at a high abstraction level. Ambient Calculus is a formalism that allows the representation of boundaries where computation occurs. Also, distributed and mobile systems are usually difficult to develop as they need to take into account functional and non-functional requirements and reusability and adaptability mechanisms. In order to achieve these needs it is necessary to separate the distribution and mobility concerns from the rest of the concerns. PRISMA is an approach that integrates the advantages of Component-Based Software Development and Aspect-Oriented Software Development for specifying software architectures. In this paper, we describe how our work combines Ambient Calculus with PRISMA to develop distributed and mobile systems gaining their advantages.

1 Introduction

In the last few decades, the information society has undergone important changes. New technologies have become part of our daily life and the Internet has been established as a framework for global knowledge. For these reasons, two important ideas have been arisen: the world is considered as a whole unit with no boundaries, and people work in a collaborative way without meeting physically. These ideas have created the need for current software development processes to deal with complex structures, new non-functional requirements, dynamic adaptation, and new technologies. In addition, most software systems require the capability to work with different devices (PCs, laptops, PDAs, smart phones, etc) through communication networks in a distributed and secure way. As a result, software development processes must also take into account the distributed, ubiquitous and mobile nature of software systems.

The development of distributed, ubiquitous and mobile software systems is a difficult task, especially if these characteristics are to be considered from the beginning of the software life cycle. Currently, decisions about these characteristics are usually postponed to late stages of the software life cycle (design and implementation). As a

result, there is a loss of traceability, and the system is subject to a specific technological platform (such as CORBA [1] or .NET Remoting [2]). As a result, the development of systems of this kind introduces important challenges such as: how to specify distribution and mobility features in a technology-independent way, how to consistently manage a distributed state, how to support non-functional requirements such as security or fault tolerance.

Software Architecture is considered to be the bridge between the requirements and implementation phases of the software life cycle. Software Architectures describe the structure of software systems in terms of computational (components) and coordination (connectors) units of software. Architecture Description Languages (ADLs) specify the functional and coordination properties of these software units in a formal way. However, current ADLs do not provide constructs for describing distribution or mobility features in an abstract way.

A formalism that provides mechanisms to describe distribution and mobility properties is Ambient Calculus (AC) [3]. AC introduces the concept of ambient, which represents boundaries where computation occurs. Ambients can model the location hierarchy encountered in distributed systems and model the mobility as the crossing of the locations boundaries.

PRISMA [4] is an approach that integrates the advantages of Component-Based Software Development (CBSD) [5] and Aspect-Oriented Software Development (AOSD) [6] to specify software architectures. This approach has a meta-model [4], formal Aspect-Oriented Architecture Description Language (AOADL) [7], and a framework [8].

In this paper, we combine the PRISMA approach and the AC in order to deal with the specification of distributed and mobile features from the beginning of the software life cycle in a technology-independent way. In this work, ambients are specified as architectural elements that use separation of concerns (aspects) to describe their functionalities.

The paper is structured as follows: Section 2 presents related works performed in the area of distribution at an architectural level. Section 3 presents the PRISMA approach and the motivation for the work presented in this paper. Section 4, gives an overview of AC. Section 5, introduces how the PRISMA approach combines ambients. Finally, Section 6 presents conclusions and further works.

2 Related Work

One of the reasons why software architectures emerged was to simplify the construction of dynamic distributed systems. However, at the present time, few ADLs support the specification of distributed systems properties. The first research that provided significant results in distributed software architectures was carried out in the Darwin ADL [9]. Darwin uses π -calculus [10] to define the semantics of distributed message-passing. It builds architectures by defining composite components that are bound and given locations at instantiation time. Darwin has also been used in the CORBA environment to specify the overall architecture of component-based applications [11]. However, in the literature, we have not found new advances to Darwin in constructing

software architectures with mobile and replicable components. As Darwin is based on π -calculus only, mobility can only be simulated by the movement of channels. It lacks primitives to express the movement of crossing boundaries.

The work in [12] states that an ADL should consider features such as composition, reusability, and configuration in order to specify dynamic distributed software architectures. It presents a configuration language that describes a method for a reconfiguration model at run-time. However, the reconfiguration model is not formal. Moreover, it neglects a distribution model for specifying distributed message-passing among components and connectors.

The works of Mascolo and Ciancarini [13,14] introduce MobiS, which is a specification language that is based on a tuple-space model that specifies coordination by multiset rewriting. MobiS can also be used to specify architectures containing mobile components. However, it does not specify the mobility concern separately from the rest of the functionalities of the software architectures, reducing reusability and adaptability to changes.

The ADL C2Sadel has adapted a style to support both distribution and mobility. The style [15] provides software connectors that are able to move components. It also has an implementation infrastructure to support this architectural style. However, this approach has the drawback that there is no separation between coordination and distribution. Therefore, the components are the only architectural elements that are mobile while the connectors are static.

The work of Lopes in [16] describes the semantics of externalizing a distribution dimension in order 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 allowing the specification of security and authentication.

3 PRISMA Distribution and Mobile Model

The PRISMA model [4] allows the definition of software architectures of complex software systems by integrating the AOSD and the CBSD. PRISMA uses the AOSD to separate the crosscutting concerns (distribution, security, context-aware, coordination, etc.) of architectures in aspects. The PRISMA architectural elements are specified using aspects that define their behaviour. As a result, an architectural element (components and connectors) can be viewed as a prism where each side of the prism is an aspect (*white box view*). In addition, an architectural element encapsulates its functionality and publishes a set of services that it offers to the rest of the architectural elements (*black box view*) (see Figure 1).

There are two kinds of architectural elements: components and connectors. A component is an architectural element that captures the functionality of software systems and a connector is an architectural element that acts as a coordinator among other ar-

chitectural elements. Components and connectors are formed by a set of aspects, the weaving relationships among these aspects, and the ports that offer and request services.

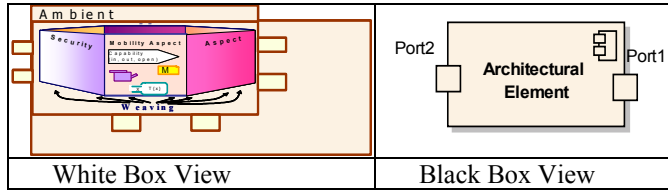


Fig. 1. Views of an Architectural Element

Weavings indicate that the execution of an aspect service can trigger the execution of services in other aspects. Weavings are the glue of the aspects of an architectural element. This glue is defined using temporal operations called weaving operators. Initially, the weaving operators that PRISMA provides are *after*, *before*, *around*, *afterif*, *beforeif*, and *insteadif*. 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.

It is important to emphasize that connectors do not have the references of the components that they connect and vice versa. Thus, architectural elements are reusable and unaware of each other. This is possible due to the fact that the channels (*attachments*) defined between components and connectors have their references, instead of architectural elements. Attachments are the channels that enable the communication between components and connectors. Each attachment is defined by attaching a component port with a connector port (represented as lines in Figure 2).

However, when we applied PRISMA to a real case study such as the tele-operated *TeachMover* robot local communication was a limitation. Tele-operation systems are control systems that depend on software to perform their operations. They are usually robots that perform high-risk activities. For this reason, they must be controlled by operators from safe areas. As a result, the need to locate components in different places (nodes) as well as to communicate the distributed software components of the operator and the robot emerged.

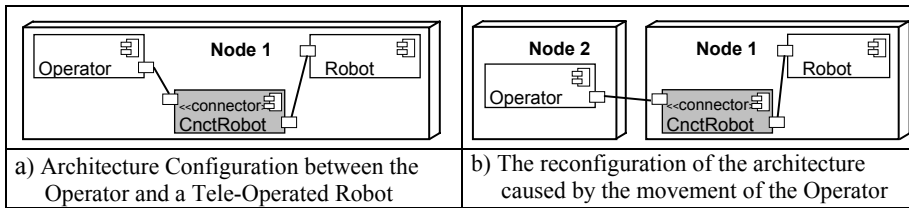


Fig. 2. Mobility of the Operator in a Tele-Operated System

Mobility is also a characteristic that is fundamental in distributed and dynamic systems, where the topology of the architecture can change at run-time. For example, in the tele-operated system, the mobility requirement emerges to be able to move the operator to different places (nodes). This mobility is necessary to allow the operator

send commands to the robot from different places, maintaining the information of the operator component consistent (see Figure 2).

As Figure 2 illustrates, mobility is the process of transferring a component instance from one node to another one. Moreover, the transfer process must ensure that the transferred component instance continues its execution at the target node, conserving its state and maintaining the same execution point.

PRISMA has been adapted to support distribution and mobility properties [17] in order to be applied to real case studies. Distribution is supported in PRISMA by introducing the following properties into the model:

1. The use of attachments: Attachments store, not only the references of the architectural elements that they are connecting, but also the locations of these architectural elements (nodes). In this way, the reusability of architectural elements is preserved, and distributed communication is enabled. As a result, architectural elements are unaware of the distributed or local nature of the others.
2. The use of a Distribution Aspect: The distribution aspect specifies the features and strategies that are related to the distributed behaviour of a PRISMA architectural element. It specifies the site where the architectural element is located and indicates when an element needs to be moved.

This distribution model was initially implemented in the PRISMANET middleware [8] and has been validated using case studies where distribution properties are required. However, a model that includes an explicit primitive for supporting locations as boundaries, describes the location hierarchies and supports the mobility of elements by the crossing of boundaries is richer. Therefore, we have combined the PRISMA model and AC.

4 Ambient Calculus

Ambient Calculus [3] (AC) is a process algebra that extends π -calculus [11] in order to introduce the concept of ambient. An ambient is a bounded place where computation occurs. Thus, an ambient can be anything with a boundary such as a laptop, a web page, a folder, etc. Each ambient has a set of running computations that can control it. These are responsible for moving an ambient. In addition, an ambient can contain other subambients that have running computations.

Thus, mobility is performed at an ambient level, i.e. ambients are mobile. Also, mobility is performed by crossing boundaries of ambients. AC provides mobility and local communication primitives. These primitives can be expressed in a textual syntax and in a graphical syntax which is called Folder Calculus [20] (see Figure 3). Folder Calculus is a graphical metaphor for AC where ambients are visually represented as folders.

AC uses some of the constructs inherited from π -calculus such as naming, restriction, parallel processes, inactive process and replication. However, the names in AC are names of ambients instead of names of channels as in π -calculus. Therefore, in order to syntactically write that an ambient with name n has process P , it is written as $n[P]$.










Textual Syntax	Visual Syntax	Comments	Textual Syntax	Visual Syntax	Comments
$(\nu n)P$		New name n in a scope P .	0		Inactive process (often omitted).
$n[P]$		Folder (ambient) of name n and contents P .	$!P$		Replication of P .
$M.P$		Action M followed by P .	(λM)		Output M .
$P \mid Q$		Two processes in parallel. (Visually: contiguously placed in 2D.)	$(n).P$		Input n followed by P .
			(P)		Grouping

Fig. 3. The Textual and Visual Syntax of Ambient Calculus constructs

Some of the primitives that AC provides are called capabilities. Capabilities are actions that can be performed on ambients. There are three main types of capabilities: enter, exit and open capabilities. The enter capability orders an ambient to enter another ambient on its same hierarchy level (see Figure 4). The exit capability orders an ambient to exit its parent ambient. The open capability dissolves an ambient leaving the processes that were in it.

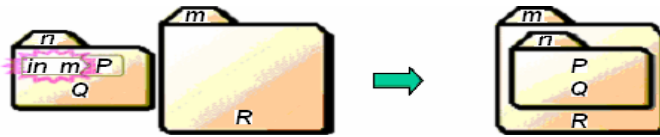


Fig. 4. Applying the enter capability to the ambient n

4 AMBIENT-PRISMA: Combining Ambient Calculus and the PRISMA Approach

This section presents how the AC concepts are integrated to the PRISMA approach in order to describe distributed and mobile systems. To allow PRISMA architectural elements to make use of the ambient concept of AC, the ambient construct must be included in the PRISMA meta-model. Therefore, some mappings between the AC meta-model and the PRISMA meta-model have been identified.

In [18], it is discussed that an ambient can be seen as a software component that offers mobility and that it has a proper identity at run-time so that it can be maintainable. In our model, this corresponds to a PRISMA architectural element. Since PRISMA architectural elements are components and connectors, an ambient cannot be a PRISMA Component because a PRISMA component performs the computations. Nor can the ambient be a PRISMA Connector because a PRISMA Connector coordinates computations. Therefore, in the PRISMA meta-model an ambient is introduced as a new type of architectural element (see Figure 5) that is responsible for providing mobility services to distributed architectural elements. As a result, an ambient inherits all the characteristics of a PRISMA architectural element (its CBSD view and its AOSD view) and provides its proper semantics. Figure 6 shows the graphical repre-

sensation of a PRISMA ambient. The graphical representation preserves the folder calculus representation of an ambient. The Ambient CBSD view describes it as a black box where it communicates with others by using ports that send and receive invocations of services.

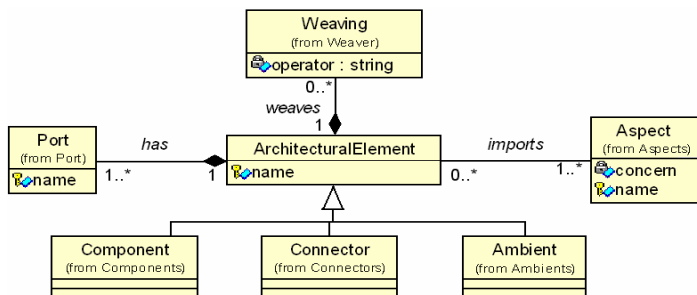


Fig. 5. Including the Ambient as another architectural element

An ambient has a collection of local agents and can also have other subambients [18]. In PRISMA, the local agents correspond to components that are coordinated using connectors. Ambients in PRISMA are complex architectural elements that represent the places where components, connectors and other ambients are located. In addition, by allowing an ambient to have other ambients inside it, the hierarchy of distributed and mobile systems can be modelled in PRISMA.

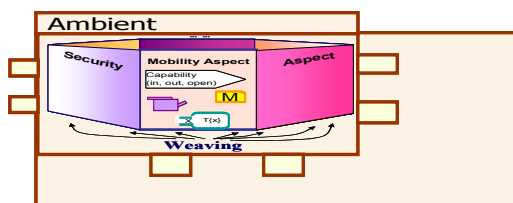


Fig. 6 A PRISMA Ambient with CBSD and AOSD views

The AOSD view describes the PRISMA ambient with a set of aspects that can be weaved. The ambient uses different aspects to specify the services it offers and requests. As ambients are responsible for the mobility concern, all ambients must have the Mobility Aspect to provide mobility services to their local architectural elements.

The Mobility Aspect specifies the following ambient functionalities:

- It allows an ambient to offer the exit service to its subambients that need to exit from it. (The specification of the AC exit capability).
- It allows an ambient to offer the enter service to its subambients that need to enter other subambients. (The specification of the AC enter capability).
- It allows an ambient to create subambients. (The specification of the AC restriction).
- It allows an ambient to accept a new ambient in it from external ambients.
- It allows an ambient to execute the open service. The open service allows a subambient to be destroyed a local ambient without destroying its local architec-

tural elements. As a result, the architectural elements of the destroyed subambient become to form part of the ambient. (The specification of the AC open capability)

The separation of the Mobility Aspect concern from the rest of concerns provides a better maintainability of these functionalities because they are not scattered through the ambient specification. In addition, this is a generic aspect that must be reused by all PRISMA ambients. As a result, ambients are defined by importing the generic mobility aspect and adapting it to the software system needs through weavings. For example, a LAN ambient may need some security policies that are different from a PC ambient inside of the LAN. Therefore, both the LAN and PC ambient import the same Mobility Aspect, but the Mobility Aspect is weaved with different security aspects.

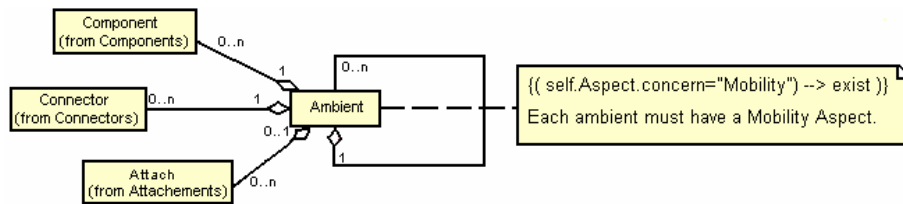


Fig. 7. The Ambient Package in the PRISMA meta-model

In order to introduce the concepts that describe a PRISMA ambient, an Ambient Package has been defined in the PRISMA meta-model(see Figure 7). This package contains the relationships and constraints that an ambient has with other meta-model concepts. An Ambient can contain Components, Connectors, Attachments, and other ambients. A constraint is specified in the Object Constraint Language in order to indicate that an Ambient must have a Mobility Aspect.

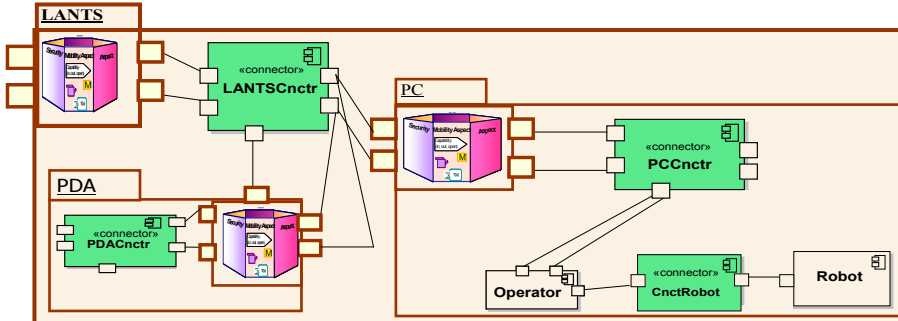


Fig. 8. The Initial Configuration of the Robot software architecture modeled with Ambients.

We are going to use the tele-operation system example to illustrate how a distributed and mobile system is specified in PRISMA after introducing the ambient architectural element. In the example, the operator is a mobile component that can move from a PC to a PDA to be closer to the robot. Figure 8 shows the distributed hierarchy of the tele-operation software architecture. It shows that the LANTS ambient (the LAN of the Tele-operation System) consists of a PDA ambient and a PC ambient. The Operator component, Robot component and their connector (CnctRobot) are initially

located in the *PC* ambient. Using the ambient calculus syntax, this is written as $LANTS[PDA[]] PC[Operator[out PC, in PDA], CnctRobot, Robot]$.

Every ambient has services that are offered to its local architectural elements and services that are offered to the exterior. As a result, some of the ambient ports in Fig. 8 are only internally connected through attachments to a connector (e.g. the *PCCnctr* of the *PC*) that synchronizes the ambient with its local architectural elements.

Figure 9 (b) shows how the *Operator* Component is specified in the PRISMA ADL. As the *Operator* is a distributed component, it imports a predefined Distribution Aspect *OpDistribution*, specified in Figure 9(a), which defines a distributed behaviour. The *Operator* has three ports: *ExitingPort* to request an *exit*, *EnteringPort* to request an *enter*, and *FunctPort* to send commands to the robot. The *OpDistribution* specifies the *move*. *Move* indicates that the movement of the element that imports this aspect needs to *exit* from its parent ambient and then *enter* to another ambient. For this reason, requests for *enter* and *exit* are made to other architectural elements (*out* = client behaviour). For example in Figure 8, to move the *Operator* from the *PC* to the *PDA*, the *Operator* makes a request to *exit* the *PC* from the *PC*. Figure 8 also shows that the ports *ExitigPort* and *EnteringPort* are connected to the *PCCnctr* in order to be synchronized with the *PC* ambient. *FunctPort* is connected to *CnctRobot* to be synchronized with the *Robot*.

<pre> Distribution Aspect OpDistribution using IExiting, IEntering Services out exit (MyName: String); out enter (MyName: String, NewAmbient: loc); Transactions move (NewAmbient: loc) Exiting= out exit(MyName).Entering; Entering= out enter(MyName, NewAmbient); End Distribution Aspect OpDistribution (a) </pre>	<pre> Component_type Operator Import Distribution Aspect OpDistribution; Import Functional Aspect OpFunct; Port ExitingPort: IExiting; EnteringPort: IEntering; FunctPort: IRobotCommands;; End_Port End Component_type Operator; (b) </pre>
---	--

Fig. 9. The Operator Distribution Aspect and Component specified in the ADL

<pre> Mobility Aspect Mobile using IExiting, IEntering, IAccepting Transactions in exit (Requested: String, NewAmbient: loc): EXIT ::= out isSon(input Requested, output isSonOK) → EXIT1; EXIT1 ::= {isSonOK==true} out checkTypeAmbient(input Requested, output isTypeAmbient) → EXIT2; EXIT2 ::= if(isTypeAmbient==false) then createAmbientFor (input Requested, output RequestedAmbient → EXIT3 else EXIT3 ; EXIT3 ::= out movingInf(input RequestedAmbient, output Type, output MobileInstance, output AttachmentList[]) → EXIT4; EXIT4 ::= out accept(input Type, input MobileIntstance, input AttachmentsList, output Acceptance) → EXIT5; EXIT5 ::= {Acceptance==true} out modifyAttachment(Requested) → EXIT6; EXIT6 ::= out destroy(RequestedAmbient) → EXIT7; EXIT7 ::= out removeAttachments(requestedAmbient); End_Mobility Aspect Mobile (a) </pre>	<pre> Ambient_type PC Import Mobility Aspect Mobile; Import Security Aspect Sec; Weavings Sec.CheckSecurity() before Moile.exit(Requested, Ne- wAmbient); End_Weavings Ports AcceptancePort: IAccept; DistServicesPort: ICall; ServicesPort: ICall; CapabilitiesPort: ICapability End_Ports End Ambient_type PC ; (b) </pre>
--	--

Fig. 10. The Mobility Aspect and the PC Ambient specified in the ADL

Figure 10(a) shows a fragment of the Mobility Aspect *Mobile* that all ambients import. It shows how the *exit* capability is mapped in PRISMA. Figure 10(b) shows the specification of the *PC* ambient. It shows that the *PC* imports the behavior that the *Mobile* aspect defines. It also shows that it imports a Security Aspect *Sec*. In the **Weavings** section, a weaving is specified to indicate that a security rule must be checked in the *Sec* aspect *before* the *exit* is executed in the *Mobile* aspect.

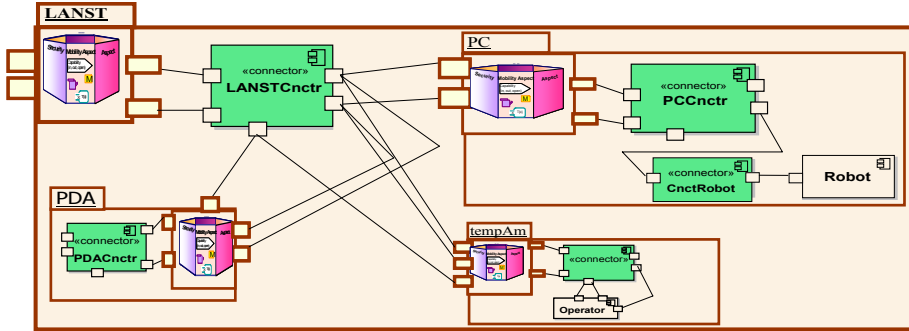


Fig. 11. The new configuration of Figure 8 after the execution of the *exit*

The *exit* in Figure 10(a) is specified as a transaction in the **Transactions** section. The *exit* has a server behaviour in the ambient that imports the aspect, that is, other architectural elements are going to request it (*in*=server behaviour). Using the example in Figure 8, the *Operator* would be the element that makes an *exit* request to the ambient. Then the *exit Transaction* consists of a set of services. First, it checks if the requested element (*Operator*) is one of the PCs children (is one of its local elements). If it is, then the *exit* checks if the requester is an ambient or not. If it is not an ambient, then an ambient is created to encapsulate the element. The creation of an ambient is necessary due to the fact that ambients are the only architectural elements that can be mobile. Then the information needed for the *exit* of the *Operator* is collected: the state of the *Operator* and its attachments. The *exit* transaction then asks the parent ambient (*LANTS*) if it can accept the *Operator's* ambient (*tempAm*) and sends the needed information. If *LANTS* accepts the *tempAm*, for each attachment between the *Operator* and other architectural elements, a new attachment is created between the *PCCnctr* and those local architectural elements that are connected to *Operator*. Figure 11 shows the new attachment that is created between *CnctRobot* and *PCCnctr* in place of the attachment between *CnctRobot* and *Operator*. Then the *PC* ambient deletes the *Operator* and all its attachments. Figure 11 shows the result of the software architecture configuration after executing the *exit* transaction.

Finally, Figure 12 shows the *Operator* component in the *PDA*. This is possible after the *Operator* in *tempAm* in Figure 11 requests the *LANTS* to enter *PDA*. Then, the *LANTS* checks if the *PDA* is local to it and requests the *PDA* to accept the *tempAm*. The *PDA* accepts *tempAm* and opens it, leaving the *Operator* in *PDA*.

The tele-operation system specification shows how its distributed and mobile properties can be described. The previous specification benefits from the concepts introduced in AC; thus the mobility of the *Operator* is specified in a formal way thanks to

the AC capabilities. Also, the AC primitives can be completely specified by the PRISMA ADL in a technology-independent way. In this way, the ambient functionalities can benefit from the reusability and maintainability that the AOSD and CBSD provide.

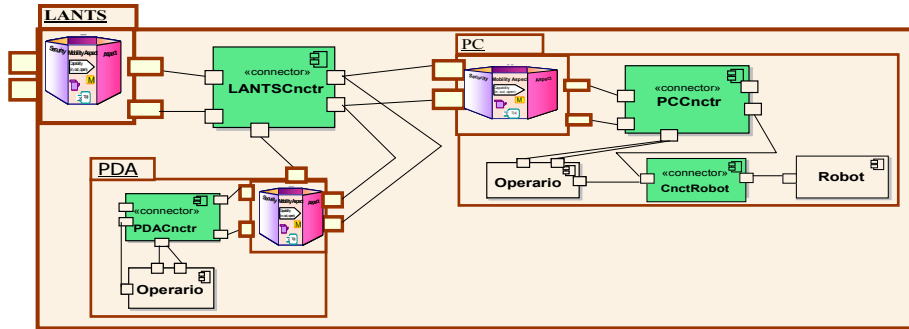


Fig. 12. The configuration of the architecture when the *Operator* reaches *PDA*.

5. Conclusions and Further Work

In this work, we have presented an approach to represent complex, distributed and mobile systems in a technology-independent. Our model combines the PRISMA approach with the AC formalism, which provides the following advantages: 1) It can describe a complex system in terms of computational, coordination and distribution and mobility units on different levels of abstraction. In this way, a system is built by reusing and adapting these separated units achieving a higher level of maintainability. 2) It can also describe the specific issues of current distributed systems such as the network topology and security.

We have introduced the ambient concept in the PRISMA meta-model as a new architectural element that can contain several computation and coordination processes (components and connectors) or other subambients. The capabilities provided by an AC ambient are offered in PRISMA by an ambient-specific aspect called the Mobility Aspect. Another aspect, the Distribution Aspect, manages the location of an architectural element and defines how and when ambient capabilities can be executed. A Security Aspect can be added to an ambient in order to provide security mechanisms. Mobility, distribution and security concerns are specified separately from other functional and non-functional requirements, thereby increasing reusability and adaptability to changes.

In the near future, we are going to introduce these concepts into the PRISMA tool to be able to model and execute mobile distributed software architectures. This will be done in three stages: first, ambients will be introduced in the PRISMANET middleware [8] to execute these concepts; second, ambient graphical metaphor and code templates will be introduced in the modelling framework; third, the implementation will be validated by modelling and executing a complex, distributed, and mobile case study.

References

1. CORBA Official Web Site of the OMG Group, <http://www.corba.org/>
2. Microsoft .Net Remoting: A Technical Overview, <http://msdn.microsoft.com/library/default.asp?url=/library/enus/dndotnet/html/hawkremotimg.asp>
3. Cardelli, L., Gordon, A. D. "Mobile Ambients", Foundations of Software Science and Computational Structures: First International Conference, FOSSACS '98, LNCS 1378, Springer, 1998, pp. 140-155.
4. Perez, J., Ali, N., Carsi, J.A., Ramos, I. "Dynamic Evolution in Aspect-Oriented Architectural Models", European Workshop on Software Architecture, Pisa, June 2005 © Springer LNCS vol n.3527.
5. Szyperki, C., *Component Software: Beyond Object Oriented programming*, ACM Press and Addison Wesley, New York, USA, 2002.
6. Aspect-Oriented Software Development, <http://aosd.net>
7. Pérez, J., Ali, N., Carsi, J.A., Ramos, I. "Designing Software Architectures with an Aspect-Oriented Architecture Description Language", 9th International Symposium on Component-Based Software Engineering (CBSE 2006), Mälardalen University, Västerås near Stockholm, Sweden, June 29th -1st July 2006 (accepted to appear)
8. Perez, J., Ali, N., Costa, C., Carsi, J.A., Ramos, I. "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology", 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005 , 2005
9. Magee, J., Dulay, N., Eisenbach, S., Krammer, J. "Specifying Distributed Software Architectures". 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, pp 137-153.
10. Milner, R., Parrow, J., Walker, D. "A calculus of mobile processes", Parts 1-2. Information and Computation, 100(1), 1992, pp. 1-77.
11. Magee, J., Tseng, A, Kramer, J. "Composing Distributed Objects in CORBA", Third International Symposium on Autonomous Decentralized Systems, Berlin Germany, 1997, pp 257-263.
12. Virginia C. de Paula, G.R., Justo, Cunha, Ribeiro, P. R. F. "Specifying Dynamic Distributed Software Architectures", XII Brazilian Symposium on Software Engineering, BCS Press, October, 1998.
13. Ciancarini, P., Mascolo, C. "Software Architecture and Mobility", 3rd Int. Software Architecture Workshop (ISAW-3), November, 1998.
14. Mascolo, C. "MobiS: A Specification Language for Mobile Systems". 3rd International Conference on Coordination Models and Languages, 1999.
15. Medvidovic, N., Rakic, M. "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility". Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.
16. Lopes, A. Fiadeiro, J.L., Wermelinger, M. "Architectural Primitives for Distribution and Mobility", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, pp. 41-50.
17. Ali, N., Ramos, I., Carsi, J.A. "A Conceptual Model for Distributed Aspect Oriented Software Architectures", International Conference on Information Technology (ITCC 2005), IEEE Computer Society, ISBN 0-7695-2315-3, April 2005, pp 422-427.
18. Cardelli, L. "Abstractions for Mobile Computation." In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, volume 1603 of LNCS, Springer Verlag, pp. 51-94.