High Level Specification of Distributed and Mobile Information Systems

Nour H. Ali, Jennifer Perez, Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera s/n
E-46071 Valencia – Spain
{nourali, jeperez, iramos}@dsic.upv.es

Abstract. Nowadays, information systems are large and complex to develop. An important factor that influences in this complexity is that information systems are tending to be distributed with mobile components. Many technologies have emerged in dealing with distribution issues at an implementation level. However, few approaches have considered distribution from the beginning of the life cycle of software development and have dealt with distribution at a high abstraction level. In this paper, we focus on specifying distributed and mobile information systems at a high abstraction level using an approach called PRISMA. PRISMA is an architectural model which combines the Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD) to describe software architectures. PRISMA has associated an Architectural Description Language (ADL) that is separated into a Type Definition Language and a Configuration Language. We use the Type Definition Language to specify the distribution aspect as a first order citizen of the language. Furthermore, we use the Configuration Language to configure the location of the instances of the software architecture.

1 Introduction

Nowadays, information systems are large and complex to develop. An important factor that influences in this complexity is that information systems are tending to be distributed with mobile components. Many technologies have emerged in dealing with distribution issues at an implementation level. On the other hand, few approaches have dealt with distribution at a high abstraction level. Nevertheless, considering distribution in the whole life cycle of software development minimizes time and costs. Thus, efforts are reduced in the development process by taking into account distribution at an early phase, instead of only introducing it at the implementation phase.

Moreover, considering distribution at analysis and design phases of software development generates high quality distributed applications. This increment of quality is due to taking into account distribution from the beginning of software development

so applications are prepared to support its non-functional requirements. However, if distribution is not considered from the beginning, applications should be changed when they arrive to an implementation phase because they are not prepared to support distribution and the traceability between phases of the life cycle is usually lost.

As a consequence of the poor capability of the object-oriented model to describe complex structures of information systems, the Component-Based Software Development (CBSD) and the Aspect-Oriented Software Development (AOSD) have emerged. The CBSD [20] promises to control the complexity of system construction by coupling entities that provide specific services. The AOSD [4] allows separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. Aspects can be reused and manipulated independently of the rest of properties of the system.

PRISMA [17] is an approach that defines software architectures integrating the CBSD and AOSD in order to take advantage of both approaches. The integration is done by defining types of software architectures via a set of aspects. These aspects can evolve and be reused independently of the rest of aspects. In addition, PRISMA has an associated Architectural Description Language (ADL). Its ADL is separated into a Type Definition Language and a Configuration Language. The Type Definition Language defines architectural elements at a type level of the software architectures by different aspects. The Configuration Language designs the topology of the software architecture by creating and interconnecting instances of the defined types.

In this paper, our aim is to specify distribution and mobile properties at a high abstraction level using the PRISMA approach. Distribution and mobility are addressed by introducing the distribution aspect which is in charge of the dynamic locations of the instances. The distribution aspect is integrated by specifying an associated ADL. As the PRISMA ADL is separated into the Type Definition Language and the Configuration Language, distribution takes advantage of this separation. In this way, the distribution can be configured at an instance level allowing the reusability and independence of the aspect. Furthermore, communication between the elements of the architecture is independent of the location of the instances.

The paper is organized as follows. Section 2 presents some related work to ours that can be found in the literature. In section 3, we present the PRISMA architectural model and how it combines the AOSD and the CBSD. In section 4, the distribution aspect is proposed and showing how communication is established between the distributed components and connectors. Furthermore, mobility of elements is introduced. In addition, an example of a distributed bank system is specified in PRISMA. Finally, section 5 draws some conclusions and highlights some future work.

2 Related Work

Different programming languages have given rise to support distributed systems. Approaches as CORBA [6] and JAVA RMI [11] make distribution partly transparent for the developer. CORBA provides the applications with an intermediate layer or

"middleware" (e.g. the Object Request Broker (ORB)) that hides distribution, whereas the objects have to be registered initially by calls to the ORB. On the other hand, programmers have to know which objects will be distributed and develop interfaces to generate stubs and include them in the ORB. These approaches enable the programmer to define its components without taking into account distribution details as name servers. However, the problem of enabling the specification of an object's behaviour against events occurred in the ORB (e.g. indicating that a component has to be cloned when it receives a certain load of messages per a unit of time) is not supported. This type of problem has to be resolved by the programmer.

The CLIP group of the Madrid University of Technology [7] proposed an extension to the Ciao Prolog language for specifying the distribution of its modules. In this system, all modules are specified as if they were local and the distribution is configured with a new specification in the same language (with the extension proposed). Next, the compiler knows which modules should be distributed and these are compiled in a particular manner. It is important to take into account that the programmer separates the functionality from the distribution and only uses a single language.

Work has also been done in the conceptual modelling of mobile object systems using language constructs which can clearly distinguish between the mobile and stationary parts of a complex system. There is an extension of the TROLL language [1] in order to specify these mobile object systems. However, the work does not incorporate the flexibility of evolving the stationary parts into mobile parts.

Many architectural models with architecture description languages (ADL) have emerged such as LEDA [5], Acme [8] and Darwin [12, 15]. Acme is a well recognized ADL for defining software architectures; however, it is a static ADL that does not support evolution and reconfiguration. Darwin is an ADL that supports distributed message-passing; however, this language does not support mobility of the components.

The works in [11] describe the semantics in externalising a distribution dimension which achieves a separation between computation, coordination and distribution. Our approach is quite similar, but we use the aspect oriented approach [4] which encapsulates distribution issues of components in entities called aspects. We use the aspect-oriented approach because it supports the independence of the aspects from the functionalities thereby, achieving reusability, adaptability and evolution. In addition, it is a considerably mature and tested approach at the implementation level.

The rise of aspect-oriented programming languages introduced the encapsulation of the distribution code in the distribution aspect separating it from the rest of the code. This is done by extending known programming languages as AspectJ [18, 19] in the case of JAVA. Whilst an inconvenient of these low level languages is the definition of the pointcuts inside the aspects decreasing reusability. Therefore, to solve these issues the aspect-oriented software development approach emerged. It incorporates aspect concepts at early stages of development, as in the design phase, and supports code generation to AspectJ. In works as in [14] distribution issues as load balancing, caching, migration are not contemplated. Moreover, it does not use a general purpose language (i.e. the language used to specify the distribution behaviour is different from the one used to specify the functional behaviour of the architecture).In fact, when describing the functional entity (the object) Java is used,

whereas for defining the distribution aspect another language is used and for composing the aspect with the object a third language is used.

3 PRISMA Architectural Model

PRISMA [17] is a model to define architectures of complex software systems. The PRISMA architectural elements are defined by different aspects: functional, quality [16], coordination, context-awareness [10] and many others. The types of aspects forming an architectural element may vary depending on the architectural element and on the information system under consideration.

An architectural element (component, connector, system) of our architectural model can be seen as a prism. This prism has one side for each aspect (see figure 1). The most common aspects an architectural element can contain are the following:

- **Functional Aspect:** The functional aspect captures the semantics of the information system defining its structure and behaviour.
- **Coordination Aspect:** The coordination aspect defines the business rules and the synchronisation between types during their communication. Moreover, the coordination aspect includes the choreography extending the "contract" concept [3].
- **Quality Aspect:** The quality aspect specifies the non-functional requirements.
- **Replication Aspect:** The Replication Aspect specifies the features for producing replications.

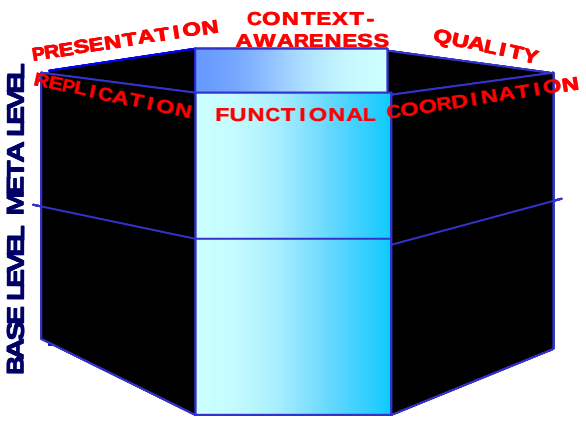

Fig. 1. A PRISMA architectural element

Our architectural model consists of different types:

- **Component:** A component is a type of the information system which does not act like a coordinator between other instances. It is formed by a set of aspects (functional, context-awareness, quality, etc), one or more

ports (**inports** and **outports**), whose type is a specific interface, and a coordination of aspects. Components interact with other architectural elements by means of their ports.

- **Connector**: A connector is a type of the information system which acts as a coordinator between instances. It is formed by aspects (coordination, context-awareness, quality, etc), a set of inroles and outroles, whose type is a specific interface, and a coordination of aspects. Connectors link and synchronise components, systems and other connectors by means of their roles.
- **System**: A system is a component that includes a set of connectors, components and other systems correctly connected. From this composition, emergent properties may arise. Moreover, a system has its own ports where binding links between its ports and the ports of its composite components are defined.

The PRISMA ADL is divided into two different levels of abstraction, the types and the interconnected instances. For this reason the ADL is presented as two languages, whose specification uses the same syntax. These languages are: the component definition language and the configuration language, respectively. Both languages are based on the OASIS 3.0 language [13] to define the semantics of architectural models. The PRISMA ADL is based on this language in order to preserve its advantages. OASIS is a formal language to define conceptual models of object-oriented information systems which allows the verification and automatic code generation of its models.

The Type Definition Language defines the types of the architecture. This language has as first order citizens: interfaces, aspects, components and connectors. The fact that interfaces and aspects are first-order citizens increases reusability because an interface may be used by several aspects and an aspect may be used by several architectural elements. Moreover, components, connectors and systems may be reused in different architectural models.

In addition, in the Type Definition Language a type is specified with the set of ports/roles, the set of aspects it is composed and the inter-aspects relationships. Each one of these relationships weaves the service of an aspect with the service of another aspect that will be executed as a consequence of the first one (crosscutting). In addition, inter-aspects relationships specify the method used to execute the service (**after**, **before** or **around**).

The configuration language is used to define instances and the topology of the architectural model. In order to do so, all needed connectors, components and systems types must be imported. Then, all the necessary instances of the imported types must be defined and, finally, the topology of the model is expressed by interconnecting such instances. Such interconnection process is achieved by creating instances of connector's synchronizations and system's bindings.

The use of different languages to specify types and topologies provides more advantages than the use of only one language. One of these advantages is the improvement of reusability and the maintenance of the types' libraries due to the independent management of types and topologies. Another advantage is the clear difference between the type evolution (Component Definition Language) and the architecture evolution (Configuration Language). In this way, there are different

evolution services in each language: evolution services to evolve types and evolution services to evolve the connections between instances to achieve dynamic reconfiguration.

4 Distribution Support in the PRISMA ADL

So far, we have described the PRISMA architectural model without considering distribution or mobility properties. In this section we aim at extending PRISMA to be able to describe systems with elements that are location-aware conserving reusability, context independence of the components (components shouldn't know to whom they are connected with neither where there connectors are located) and mobility. In this section we introduce how PRISMA supports distributed software architectures. Then, we will introduce how elements can be specified to be mobile. Furthermore, we will show how a small part of a distributed bank system is specified using the PRISMA ADL with its distribution properties.

4.1 Distribution in PRISMA

Surely, to do so in PRISMA a distribution aspect should be added to the set of aspects of the model (see figure 2). The distribution aspect specifies the features that define dynamic location of the instances. Therefore, if the system is local a distribution aspect is not added to the set of aspects of the architectural types.

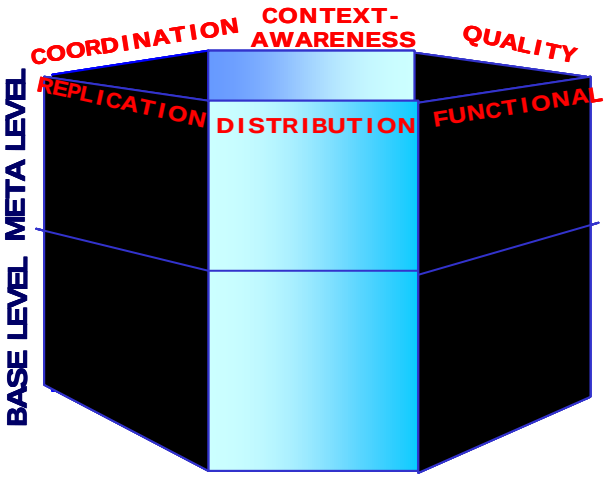

Fig. 2. A PRISMA type with distribution aspect.

The distribution aspect should specify the locations of the instances. However, if the location is specified in the distribution aspect, the same distribution aspect cannot be reused in different types if instances are distributed in different locations. This

problem is solved using the PRISMA ADL due to that it is separated into the Type Definition Language and Configuration Language. Thus, the Type Definition Language specifies that a type has a location without assigning it a value. The value of the location is assigned when types are instantiated in the Configuration Language.

The distribution aspect template will intend to specify a set of patterns detected in [2]. The patterns describe causes for the mobility and the replication of instances when faced with distribution issues such as saturation, unbalance load, latency, etc and changes in properties of the system.

```
01 Distribution Aspect name [specifies Interfaces ];

02 Identification
03    Id: (Id);

04 Attributes
     05    Id:nat;
     06    Location : LOC NOT NULL;

 07 Services
08    InService();
        Valuations
09    OutService();

10 Constraints…
11 Preconditions…
12 Triggers…
13 Operations [transactions]…
14 Protocols…

15 End Distribution Aspect name;
```

Fig. 3. Distribution Type Definition Template.

The template of the Type Definition Language to specify the distribution aspect (see Figure 3) contains the following parts:

- The aspect specifies the semantics of the services of a set of interfaces (see line 01 in Figure 3). The services of these interfaces can be invoked by the environment.
- Each aspect has an identification mechanism that distinguishes it from the other aspects in the PRISMA library (see line 02-03 in Figure 3). This is to enable the reusability of the aspects.
- The aspect specifies its set of attributes with their valuations to specify how service invocations affect attribute values (see lines 04 -06 in Figure 3). The attribute *Location* has as an abstract data type called *LOC*. This data type hides the different mechanisms of locations of an architectural element at a physical level, e.g. it can be a

URL, an IP, etc. The *Location* attribute should have a value when an architectural element is instantiated, this is indicated by the *NOT NULL*.

- Services that allow a change in the attributes values including the interface services must be specified (see line 07-09 in Figure 3). In addition, when it is necessary to differentiate between the client and server behaviour of the same service, the service is typed as ***out*** to specify that it is a service to be requested by the architectural element type and the service is typed as ***in*** to specify that it is a service to be offered by the type. Each *inService* should specify how its execution (invocation) changes the values of the attributes by the reserved word *Valuations.*
- The constraints specify integrity restrictions that always have to be satisfied.
- The preconditions are a set of conditions that have to be satisfied in order to invoke a service.
- The triggers, as the name indicates, specify the triggers that occur.
- The operations specify a set of services considered as one unit; they could be transactional or not.
- The protocols specify how the invocation of services can be organized through time.

Another issue left to overcome is how can components communicate with each other through connectors if components and connectors are distributed in different locations preserving context independence. Therefore, we solve this problem by defining a type in the Type Definition Language called Attachments. This type contains the services that perform attachments between ports and roles of components and connectors respectively. At the Configuration language the Attachment services are used at an instance level. The Attachment type is as follows:

Type Attachment Name
Identification
Name:(name);
Attributes
name: string;
Services
create(name)**new;**
attach(Inport.Component, Outrol.Connector);
attach(Outport.Component,InRol.Connector);
End_Attachment Type;

4.2 Mobility

An instance can be mobile, if the service **move(LOC)** with its corresponding **valuation** (defining that the actual location changes to a new one) is specified in the distribution aspect that defines the architectural element type which it is an_instance_of. If the distribution aspect specifies an interface containing the service **move(LOC),** the environment can control the mobility of the instance because the **move** service is published by the interface and it can be invoked by another instance. Otherwise, the mobility is under control of the proper instance.

The distribution aspect is as follows for specifying mobility:

Distribution Aspect name [**specifies** interfaces];

Identification
Id: (Id);

Attributes
Id:nat;
Location**: LOC;**

Services
move(newLoc:**LOC**);
Valuations
[move(newLoc**:LOC**)] Location:= newLoc**;**
….
….
End Distribution Aspect name**;**

4.3 Example: Bank System

In this section, we will show how to specify using our ADL a distributed bank system. An administrator of a bank system is a mobile user. She/he determines where to move her/himself i.e. she/he controls from where to access to the bank system. Whereas, the administrator can also move a bank account to her/his location. Therefore, the mobility of the account can be controlled by the environment. As well, when the account detects that its principal bank office has changed, the account moves itself to the location of this bank office.

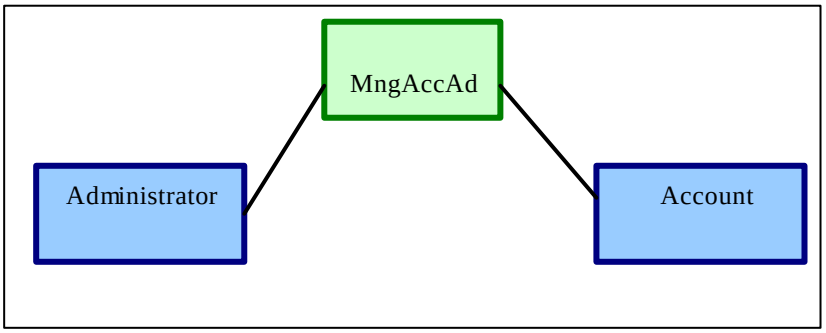

Fig. 3. The types of elements in the Bank system.

First of all, the interfaces that contain services that are shared between the different components of the software architecture are defined. The interface Mblty declares the service move(LOC) which will be shared between more than one component, in our case between the Administrator and the Account. The interface MdfyAccount declares another set of services that are shared between componets (the set of services

that the Administrator can invoke from the account). These interfaces are shown as follows:

Interface Mblty
  Services
    Move(LOC),
End Interface Mblty;

Interface MdfyAccount
  Services
    Change_Name;
    Change_PBankOffice;
    Change_AccountNumber;
End Interface MdfyAccount;

Furthermore, to specify the different distribution behaviours of the components, the distribution aspects are specified. The distribution aspect ExtMbile specifies **Mblty** interface indicating that the environment can control the location of an instance because the **move** service is published by this interface.

Distribution Aspect ExtMbile **specifies** Mblty;

Identification
  Id: (Id);

Attributes
  Id:nat;
  Location**: LOC;**
Services
  move**(**newLoc**:LOC);**

Valuations
move(newLoc**:LOC**) Location:= newLoc**;**

End Distribution Aspect name**;**

The distribution aspect Mbile specifies that it has a mobile behaviour but the service *move* is not accessible to the environment meaning that the mobility is under the type control. In addition, two *move(LOC)* services have been declared in the *Services*. The *move* declared **In** represents the server behaviour of the service i.e. declaring that **In*move*** is specified in the aspect. Whereas, the **Out*move*** specifies the client behaviour of the move i.e. that the type with this distribution aspect can control the mobility of a type in the environment.

Distribution Aspect Mbile;

Identification
  Id: (Id);

Attributes
Id:nat;
Location: **LOC;**

Services
Inmove(newLoc:**LOC**);
Valuations
[**In**move(newLoc**:LOC**)] Location:= newLoc**;**

Outmove(Location:**LOC**);
Triggers
[**Out**move(location:LOC)] **when {In** move(newLoc**:LOC**) **}**

End Distribution Aspect name**;**

After declaring the different distribution behaviours architectural elements types can have, we specify for each type a distribution behaviour. As the Administrator is independent in specifying its mobility and also can influence on the mobility of other components, it imports the distribution aspect Mbile into its definition.

Component_type Administrator
Inport … …
End_Inport;
Outport
Update_Account: MdfyAcc;
Move_Account: Mblty;
End Outport;

Import Distribution Aspect: Mbile;
….
End_Component Administrator;

The Account imports the distribution aspect ExtMbile due to that an Administrator can invoke its service move. In addition, the distribution aspect is weaved with the functional aspect through the service move and change_PBankOffice. This is to specify that after the principle bank office of the account changes (a functional property) move the account to the bank office location (a distribution property).

Component_type Account
Inport
Updates_Acc: MdfyAcc;
Moves_Acc: Mblty;
End_Inport;
Outport
…
End_Outport;

Functional Aspect
Import BankFunctionality: Functional Aspect;
Weaving
Import Distribution Aspect : ExtMbile
move(**LOC**) **after** Change_PBankOffice;
End_ Distribution Aspect
End_Weaving;
End_Functional Aspect
End_Distribution Aspect;
End_Component;

In the Configuration language the instances of the administrator and account are created specifying their location. Then the attachments between the components ports and connectors roles are created as follows:

Architectural Model BankSystem
Import Types
Components
Account, Administrator;
Connectors
MngAccAd;
Instances
Administrator1(*value_of_location*): Administrador;
Account1(*value_of_location*): Account;
MngAccAd1(*value_of_location*): MngAccAd;
End_Instances;

Attachment a.attach(Update_Account.Administrator,Update_Account.MngAccAd1);
Attachment b.attach(Move_Account.Administrator, Move_Account.MngAccAd1);
Attachment c.attach(Updates_Acc.Account, Updates_Acc. MngAccAd1);
Attachment d.attach(Moves_Acc, Moves_Acc. MngAccAd1);

End_Arquitectural Model BankSystem;

Through the following section and example we have shown how the distribution aspect is specified independently of the other concerns of the software architecture. This enables the distribution aspect to evolve and to be reused independently of the other concerns of the system.

5 Conclusions and Future Works

In this paper, we have demonstrated that distribution can be specified at high abstraction level using PRISMA model. PRISMA has been presented as an aspect-oriented component-based approach for describing distributed software architectures.

The distribution properties are separated in the distribution aspect allowing the manipulation and reusability independently of the rest of properties. This huge advantage is provided by the aspect oriented approach of PRISMA model. Moreover, types are reusable and instances are mobile and context-independent. This is due to the separation of the ADL into the Type Definition Language and the Configuration Language. This separation allows us to specify the distribution aspect as a first-order citizen and to instantiate the same type in different locations.

In the medium term many works have to be explored. We will intend incorporating these patterns defined in [2] in the PRISMA ADL. Moreover, one of our objectives is to be able to have a graphical notation in UML for the distribution properties of PRISMA ADL. Furthermore, the implementation of a model compiler is necessary to be able to automatically generate distributed applications from the ADL.

References

[1] Ahlbrecht, P., Eckstein, S. and Neumann, K. Language Constructs for Conceptual Modelling of Mobile Object Systems*: In Proc. 4th Int. Symp. on Collaborative Technologies and Systems, Simulation Series*, Orlando, USA,2003, p 121-126.

[2] Ali, N.H., Silva J., Jaen, J.,Ramos I., Carsi, J.A., and Perez J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proceedings of 15th IASTED , Parallel and Distributed Systems, Acta Press(Marina del Rey, C.A.,USA, November 2003), p 820-826.

[3] Andrade, L.,and Fiadeiro, J., "Interconnecting Objects via Contracts". OOPSLA'99. Tutorial 56.

[4] Aspect-Oriented Software Development, http://aosd.net

[5] Canal, C. *A Language for the Specification and Validation of Software Architectures*, (In Spanish) P.H.D Thesis, Malaga, Spain, 2000.

[6] CORBA Official Web Site of the OMG Group: http://www.corba.org/

[7] Correas, J. and Bueno, F. A Configuration Framework to Develop and Deploy Distributed Logic Applications: *ICLP01 Colloquium on Implementation of Constraint and Logic Programming Systems*, Cyprus, 2001.

[8] Garlan, D. "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events." Third International School on Formal Methods 2003, LNCS, Bertinoro, Italy, September 2003, pp 1-24.

[9] Herrero, J.L. *Proposal of a Platform, Language and Design, for the Development of Aspect Oriented Applications.* (In Spanish)P.H.D. Thesis, Extremadura, Spain, 2003.

[10] Jaén, J. and Ramos, I. A Conceptual Model for Context-Aware Dynamic Architectures: *23rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with ICDCS2003*, Providence, Rhode Island, USA, 2003, p 138-146.

[11] JAVA Remote Method Invocation (RMI) Specification of SUN Microsistems: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html

[12] Kramer, J., Magee J. and Sloman, M. Configuring distributed systems: *In Proceedings of the 5th ACM SIGOPS European Workshop*, Mont St Michel,September 1992.

[13] Letelier, P., Sánchez, P., Ramos I., and Pastor,O. OASIS 3.0, A formal focus for the object oriented conceptual modelling(In Spanish)(Universidad Politécnica de Valencia, SPUPV-98.4011, ISBN 84-7721-663-0, 1998).

[14] Lopes, A., Fiadeiro J.L. and Wermelinger, M. "*Architectural Primitives for Distribution and Mobility*", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002: 41-50.

[15] Magee, J., Dulay, N., Eisenbach S. and Krammer, J. Specifying Distributed Software Architectures: *Proc. of the 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, 1995, 137-153.

[16] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures: *3rd IEEE International Conference on Quality Software (QSIC 2003)*, Dallas, Texas, USA, November 2003, p 59-66.

[17] Pérez, J., Ramos, I., Lorenzo, A., Letelier, P. and Jaén, J. PRISMA: OASIS Platform for Architectural Models(In Spanish), In Proceedings of VII Jornadas de Ingeniería del Software y Bases de Datos, JISBD,(Escorial Madrid, Spain, 2002), 349-360.

[18] Soares, S. and Borba, P. PaDA: A Pattern for Distribution Aspects: *In Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, 2002, p 87-99.

[19] Soares, S., Laureano, E. and Borba, P. Implementing Distribution and Persistence Aspects with AspectJ: *Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02*, Seattle, WA, USA,2002, p 174-190

[20] Szyperski, C., *Component software: beyond object-oriented programming*, (New York, USA: ACM Press and Addison Wesley, 1998).