AMBIENT-PRISMA: Distribution and Mobility in Aspect-Oriented Software Architectures

Nour Ali Irshaid

Department of Information Systems and Computation
Polytechnic University of Valencia

A thesis submitted in partial fulfilment of the requirements
for the
degree of Master in Software Engineering, Formal Methods
and Information Systems

Supervisor: Prof. Isidro Ramos Salavert

July, 2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

This thesis presents an approach for supporting the development of distributed and mobile software systems from an early stage of the software life cycle. Specifically, the approach describes the characteristics of distributed and mobile software systems at the software architecture stage combining Aspect-Oriented Software Development (AOSD) and Component-Based Software Development (CBSD) techniques. Maintenance and reusability are enhanced by this combination. For the construction of aspect-oriented software architectures our approach is based on PRISMA. PRISMA is an approach that combines AOSD and CBSD for describing complex software architectures. To describe distribution and mobile characteristics our approach is based on Ambient Calculus. Ambient Calculus is a process algebra that describes mobility at a high abstraction level. We call the resulting combination Ambient-PRISMA.

Ambient-PRISMA is supported by a framework that includes a metamodel, a language and a middleware for developing Ambient-PRISMA aspect-oriented software architectures. The methodology that we propose adopts automatic prototyping as proposed by the Model Driven Engineering (MDE). The metamodel is allows the specification of Ambient- PRISMA architectures using its Aspect-Oriented Architecture Description Language (AOADL), both in a textual and a graphical way. The AOADL is independent of development platforms and is based on a formal language in order to preserve non-ambiguity in the code generation process. Code generation is supported by the middleware available in the CASE tool prototype.

1.1 MOTIVATION AND RATIONALE

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 risen: 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 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) and 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 software systems with existing characteristics such as distribution and mobility is a difficult task. Currently, decisions about these characteristics are usually postponed to late stages of the software life cycle (design and implementation). As a result, traceability is lost, and the system becomes bounded to a fixed technological platform. Developers of distributed systems also have to spend more time programming than solving and considering problems raised by distribution. Given the complexity of this type of systems, one should require to software engineering techniques that provide reusability and maintainability, specify distribution and mobility features in a technology-independent way and support non-functional requirements such as security or fault tolerance related to distributed systems.

The specific methods and techniques that we have in mind are:

- Software Architecture [Sha96] 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 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, few ADLs provide the needed constructs for describing distribution or mobility features in an abstract way.
- Component Based Software Development (CBSD) [Szy02] and Aspect-Oriented Software Development (AOSD) [Fil04] are two techniques that promise increased lends of reusability, flexibility, and maintainability through the software development process. CBSD proposes to construct the system by connecting entities that provide and require services. AOSD allows the separation of concerns by modularizing crosscutting concerns in separate entities called aspects. Distribution has been clearly identified as a crosscutting concern; separating it from the other concerns of the software system, will decrease cost and efforts to maintain and reuse software.
- Model Driven Engineering (MDE) [Sch06] is a software development approach based on transformations between models. MDE proposes the development of software by using models that describe a system in a technology-independent way. These models can be transformed to technology dependent models, which describe a system based on a specific technology.
- A formalism that provides mechanisms to describe distribution and mobility properties is Ambient Calculus (AC) [Car98a]. 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 [Per05b] is an approach that integrates the advantages of Component-Based Software Development (CBSD) [Szy02] and Aspect-Oriented Software Development (AOSD) [Fil04] to specify software

architectures. This approach has a meta-model [Per05b], formal Aspect-Oriented Architecture Description Language (AOADL) [Per06a], and a framework.

An important challenge in the software engineering area is the integration of software architectures, CBSD, AOSD, and automatic code generation in a unique approach in order to support the development and maintenance of distributed and mobile software systems in an efficient way. The framework that we are proposing integrates all these software engineering techniques to decrease the complexity of developing distributed and mobile software systems as well as to improve their quality, reusability and maintainability.

1.2 OBJECTIVES OF THE THESIS

The main goal of this thesis is to investigate how the combination of the techniques and approaches of CBSD, AOSD, the Paradigm of Automatic Prototyping, and Ambient Calculus support the development of distribution and mobile software systems. The result of this combination is a framework that provides the definition of distributed and mobile software systems improving the reusability, the automatic code generation and the maintainability of the defined systems.

The main goal of the thesis can be decomposed in several specific objectives:

- To study the related works of distribution and mobility, and how they have been supported in Architecture Description Languages (ADLs), Aspect-Oriented Languages, and aspect-oriented ADLs.
- To define a model that integrates PRISMA and Ambient Calculus for the definition of aspect-oriented architectural models of distributed and mobile software systems.
- To incorporate primitives to the Aspect-Oriented ADL (AOADL) of PRISMA for the specification of software architectures based on the defined model. This language must provide the needed expressiveness to specify the characteristics of distributed and mobile software systems and must be

based on formalisms that ensure the non-ambiguity and correctness of the specifications.

- To provide a graphical support for the distribution and mobility characteristics incorporated to the AOADL in order to make a friendly analysis and design.
- To develop a middleware and a catalogue of code generation patterns. The middleware must permit the execution of the distributed and mobile software architectures based on the proposed model and the code generation patterns must provides the rules to generate automatically the source code of a specific programming language and distributed platform from the primitives incorporated to the proposed AOADL.
- To validate the model, the expressiveness of the language and the tool by developing case studies of distributed and mobile software system.
-

1.3 Context

The context of this thesis is the appliance of software engineering techniques for distributed and mobile software systems. Specifically, the software architecture and AOSD techniques are integrated with concepts introduced by AC for specifying distributed and mobile software systems. In this section, a brief explanation to distribution, mobility and AC is given. Also, the case study used to illustrate the work presented in this thesis is presented.

1.3.1 Distributed Systems

<<*A distributed system is a collection of autonomous* <u>*hosts*</u> *that are connected through a computer* <u>*network*</u>*. Each host executes* <u>*components*</u> *and operates a distribution* <u>*middleware*</u>*, which enables the components to coordinate their activities in such a way that users perceive the system as a single, integrated computing facility.*>>

Zahir Tari in [Tar01]

Nowadays, distributed systems are built using distributed object or component middleware. The role of middleware is to ease the task of programming and managing distributed applications. It is a distributed software layer, or 'platform' which abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages.

1.3.2 Mobility

Mobility is the capability of moving a process, object or component instances from one computing node to another during the runtime in a distributed system.

Code mobility is the dynamic change of the bindings between code fragments and locations where they are executed. Code mobility can de defined as the capability of a distributed application to relocate its components at run-time. A detailed overview of existing code mobility techniques is given by Fuggetta et al. [Fug98]. They describe three code mobility paradigms:

- remote evaluation allows the proactive shipping of code to a remote host in order to be executed;
- Mobile agents are autonomous objects that carry their state and code, and proac-tively move across the network
- code-on-demand, in which the client owns the resources (e.g., data) needed for the execution of a service, but lacks the functionality needed to perform the service.

In this thesis, mobile agents is supported.

Mobility is classified by Picco [Fug98] into weak and strong mobility. Weak mobility happens in systems where the migrant is a data object which starts execution from the beginning after migration. Weak mobility transfers the code which may be accompanied by some initialization data, however the state is not involved. This kind of migration is well known in commercial systems. Strong mobility occurs in mobile objects which their execution is interrupted for the migration and once migrated on the destination carries forward executing from the interrupted point. This form of mobility allows migration of both the code and the state of the object before interrupting it.

Strong Mobility is supported by two mechanisms: migration and cloning. The migration mechanism destroys the executing object and transmits it to the destination. Migration can be proactive and reactive. In proactive migration, the decision of moving the object is done by itself determining the time and destination. While in reactive migration the migration decision is determined by another executing object. The cloning mechanism creates a copy of the executing object at the new destination without destroying the executing object. As in migration, cloning can be proactive and reactive.

Weak mobility's mechanisms are influenced on the direction of code transfer, the type of code and the time the code is executed at the destination. The code can be migrated as a standalone or as a code fragment. Standalone code is self-contained and will be used to instantiate a new object on the destination. A code fragment must be connected to an already running code. Mechanisms that support weak mobility can be either synchronous or asynchronous. Figure 1, summarizes Picco's classification of migration.

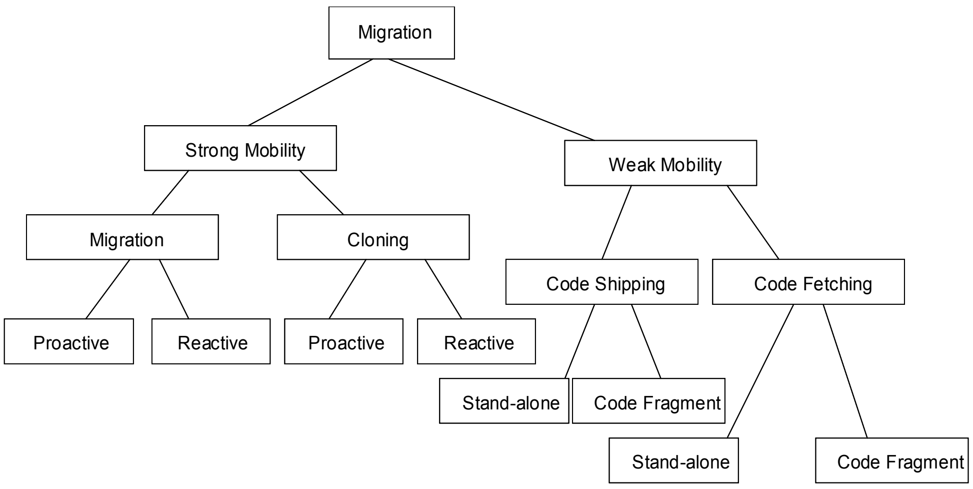


Figure 1 Picco's classification of Migration

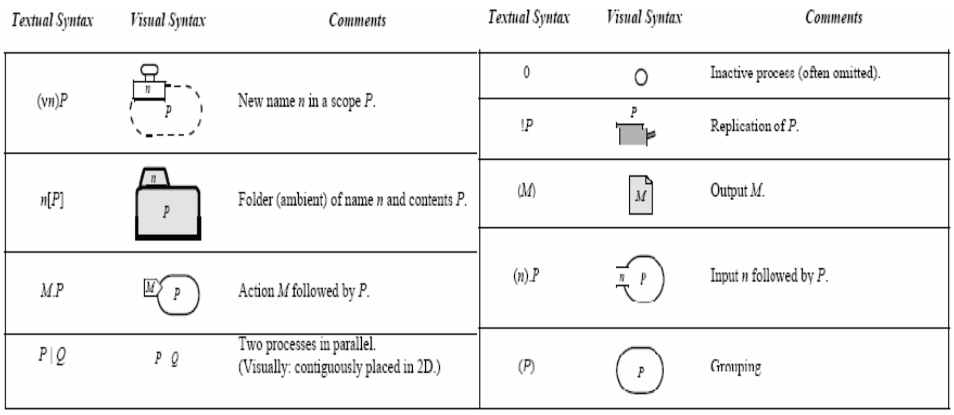
Migration is not totally supported by today's middleware technologies. Therefore, different proposals have been done to extend the frameworks.

1.3.3 Ambient Calculus

Ambient Calculus [Car98a] (AC) is a process algebra that extends π-calculus [Mil92] 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 [Car98b](see Figure 2). 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
$(\nu n)P$	n P	New name *n* in a scope *P*.	0		Inactive process (often omitted).
			!P	P	Replication of *P*.
n[*P*]	n P	Folder (ambient) of name *n* and contents *P*.	$\langle M\rangle$	M	Output *M*.
M.P	M P	Action *M* followed by *P*.	$(n).P$	n P	Input *n* followed by *P*.
$P \mid Q$	P Q	Two processes in parallel. (Visually: contiguously placed in 2D.)	(P)	P	Grouping

Figure 2. **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 3). The exit capability orders an ambient to exit its parent ambient. The open capability dissolves an ambient leaving the processes that were in it.

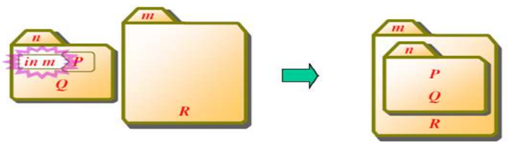


Figure 3. **Applying the enter capability to the ambient n**

1.3.4 Case Study

In order to show how Ambient-PRISMA works, we present an example of an auction site based on mobile agents. A customer of the auction site is interested in buying a specific product at a maximum price limit. To keep track of new auctions offered on the site, the customer designs two mobile agents that take charge of the purchase. The *Procurement* agent is responsible for seeking an appropriate product, and the *Bidder* agent is in charge of bidding for the product. The two agents have to collaborate with each other. The customer sends the agents to the auction site where they act on behalf of the customer. When the purchase is performed, the agents return to their original location.

1.4 STRUCTURE OF THE THESIS

This work is divided into six chapters. In the following the content of each of the chapters is briefly described:

Chapter 2 provides an introduction to software architectures and how distribution and mobility have beed dealt at an architectural level. First the chapter presents the main concepts and properties of software architecture. Then, different Architecture Description Languages that have dealt with distribution and mobility are presented. Finally, a comparison of these approaches is presented and discussed in order to analyze the state of art of distribution and mobility in software architectures.

Chapter 3 introduces Aspect-Oriented Software Development (AOSD) and how AOSD has dealt with distribution and mobility. First the chapter presents the main concepts and properties of AOSD. Then, different approaches that have implemented distributed and mobile software systems using Aspect-Oriented Programming are presented. Then, different approaches that have dealt with distribution and mobility at the the design level are also presented. Finally, a comparison of the aspect-oriented approaches that have dealt with distribution and mobility at the design level is discussed.

Chapter 4 presents PRISMA. PRISMA is an approach that integrates AOSD and software architectures. This chapter provides an overview of the PRISMA model by presenting its metamodel. Also, the software architecture of the Auction System case study is specified in PRISMA using the Aspect-Oriented Architecture Description Language (AOADL). Finally, the PRISMA CASE tool of PRISMA is presented.

Chapter 5 presents Ambient-PRISMA. Ambient-PRISMA enriches the PRISMA approach with primitives in order to specify aspect-oriented software architectures of distributed and mobile software systems. This chapter enriches the PRISMA metamodel and the AOADL. Also, the case study of the AuctionSite is used to demonstrate Ambient-PRISMA properties.

Chapter 6 presents Ambient-PRISMANET. Ambient-PRISMANET is a middleware that implements Ambient-PRISMA in .NET in order to execute Ambient-PRISMA models. This chapter presents how Ambient-PRISMA primitives have been implemented.

Chapter 7 sums up the main contributions of the work and suggest some future works.

CHAPTER 2
SOFTWARE ARCHITECTURES FOR DISTRIBUTION AND MOBILITY

2.1 Introduction

The origin of software architecture evolved from Parnas proposal of simplifying the construction of complex and large systems by modularization [Hof00]. Software Architecture is a discipline that focuses on the design and specification of overall system structure. A system structure is organized from elements that are composed into more complex ones until the overall system structure is obtained. Software architecture also deals with structural issues related to its elements such as their communication protocols, their data access, their functional assignment, and their physical distribution [Sha96].

Software architecture bridges the gap between the requirements phase and the implementation phase of the software development process. It is the first step for designing a system that needs to fulfil a collection of desired properties which can be functional and non-functional. Software architecture is an abstract representation of the system that hides implementation, algorithms or data structure details [Bas03]. At the architectural level requirements are reasoned to satisfy certain properties such as throughput, consistency, and capacity need to be satisfied [Sha96].

The objective of this chapter is to present how distributed and mobile software systems have been described at the software architectural level. In section 2.2, the basic concepts of software architectures are introduced. The objective of this section is not to provide the reader with an analysis or a summary of the state of art of the concepts and properties of software architectures. These issues have been already treated in other works such as in [Per97], [Sar97], [Kog95] and [Med00]. The objective is to give the reader an introduction to the basic concepts of software architectures that support his/her understanding in the next sections and chapters of this thesis. Section 2.3, gives an overview of the available Architecture Description Languages (ADLs) found in the literature that have addressed distributed and mobile software systems. Section 2.4, presents a comparison among the ADLs presented in section 2.3. Finally, section 4.5 presents conclusions of the comparison performed in 2.4.

2.2 Basic Concepts of Software Architectures

Over the past few years several definitions of Software architecture have been proposed. Some of them are listed below:

<<A set of architectural elements that have a particular form. The elements may be processing elements, data elements or connecting elements.>>

Perry and Wolf in [Per92]

<<Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.>>

Shaw and Garlan in [Sha96]

<< Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.>>

ANSI/IEEE Std 1471-2000 [IEE00]

<<The software architecture of a program or a computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.>>

Bass, Clements,and Kazman in [Bas03]

It can be noticed, that each definition presents different issues. However, it can be concluded that each definition is concerned with structure and behaviour. Structure describes how the system is made up of interconnected units called components. Behaviour is referred to the visible behaviour caused by the interaction of the systems components to achieve the overall functionality of the system.

In the following the basic concepts used in software architecture are described.

2.2.1 Architecture Description Languages (ADLs)

Most box-and-line diagrams of architectural designs are not precise enough for analyzing completeness, consistency or correctness. As a result ADLs emerged. An ADL is a language that describes software architectures by providing a textual syntax and usually a graphical support. The purpose of ADLs is to aid the understanding and communication of software systems. Many ADLs can be found in the literature, each emerged for different purposes and domains. Examples of ADLs are Rapide [Luc95] which allows architectural modelling, simulation, analysis and code generation capabilities, Wright [All97] which supports the formal specification and analysis of interactions between components, and SADL [Mor95] which focuses on formal architectural refinement. A comparison of different ADLs is provided in [Med00].

In [Sha94b] and [Sha96], Shaw and Garlan, elaborate six properties that characterize an ideal ADL from other languages:

- **Composition:** The language should allow its user to divide a complex system hierarchically into smaller parts or compose a system from independent elements.
- **Abstraction:** The language should permit both the identification of elements of a high level structure and their roles in a system.
- **Reusability**: A fundamental objective of an ADL is to allow the reusability of components, connectors, and architectural patterns of software architecture, even in a different context of the architectural system they were originally developed for.
- **Configuration**: The language should clearly separate the description of elements from the structures (or composite elements) which they participate in. Also, an ADL should support dynamic reconfiguration. Medvidovic argues that configuration fundamentally characterizes ADLs from other languages [Med00].
- **Heterogeneity**: ADL should be independent of the language used for implementing each component. The ADL should also permit the combination of multiple architectural patterns.
- **Analysis**: It should be possible to perform analysis of the architectural descriptions. This is related to the use of formal methods in order to define semantics properties without ambiguity.

Medvidovic and Taylor [Med00] explain that an ADL provides the description of the building blocks of software architectures. These building blocks are components (see section 2.2.2), connectors (see section 2.2.3) and configurations (see section 2.2.4). Also, it is important to provide tool support for an ADL in order to make it more useful.

2.2.2 Components

Components are the loci of computation and state [Sha96]. Examples of components are clients, servers, databases, and filters. Each component has an interface which specifies the set of services (messages or operations) that it provides and the services it requires of other components. In ADLs, the points of interaction between a component and its environment are called ports. Ports segment the interface of a component. A port can be a service or a set of services of a component interface. The ports of a component provide a black box view of a component where only its interface is visible to the environment.

A typical definition of components is the following:

<< Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system. Composite systems composed of software components are called component software >>

Clemens Szyperski [Szy02]

Meyer presents seven criteria for describing a component:

<<A component is a software element that:

1. May be used by other software elements (its clients).

2. May be used by clients without the intervention of the component's developers.

3. Includes a specification of all dependencies (hardware and software platform, versions, other components).

4. Includes a precise specification of the functionalities it offers.

5. Is usable on the sole basis of that specification.

6. Is composable with other components.

7. Can be integrated into a system quickly and smoothly. >>

Bertran Meyer in [Szy00]

Mainly, a component is the unit of decomposition of a system. It should be a reusable unit that does not have dependencies with other elements of a system. Also, a component should be easily integrated in a system.

2.2.3 Connectors

Connectors describe the interactions among components. Connectors were first introduced by Mary Shaw in [Sha94a] to separate computation from coordination. In this way, a separation of concerns is achieved. Examples of connectors are procedure call, event broadcast, database protocols, and pipes. Connectors imply a runtime mechanism for transferring control and data around a system [Bas03].

Most ADLs explicitly provide connectors as first-class entities that are independent of the components they connect and allow communication protocols among components to be reused. Some ADLs, such as Darwin [Mag95], Leda [Can01] and Rapide [Luc95] do not consider connectors as first-class citizens and as a result they cannot be named nor reused.

In [Sha96], Shaw and Garlan explain the importance of separating connectors from components for the following reasons:

- Connectors can be quite sophisticated elaborating complex definitions.
- Connectors' definition should be localized. Having connectors localized supports the maintenance and an improve design of interactions.
- Connectors are abstract. Connectors may be parameterizable and may define different kinds of interactions that are adapted to the components they coordinate at instantiation time. A connector can be instantiated as many times as necessary.
- Connectors may require distributed support.
- Components should be independent. An interface of a component is defined independently of which components will use it or how they use it.
- Connectors should be independent. Connectors should be able to mediate components that can be dynamically changing.
- Relations among components are not fixed. A component can interact with many components and its interaction with each component is different.
- Systems reuse patterns of composition.

2.2.4 Configurations

Configurations or topologies describe architectural structures which consist of connected components and connectors. A configuration is a specific structure of a concrete system. In some ADLs such as Acme these are called systems [Gar03]. Configurations are defined by connecting components and connectors (if supported by an ADL) or components and components by connections (or sometimes called attachments). Configurations may also be hierarchical i.e. a configuration can be a single component that is part of another configuration.

Configurations enable assessment of concurrent and distributed properties of architecture such as deadlocks, performance, reliability, and security [Med00]. A configuration follows constraints and patterns described in an architectural style. Mary Shaw and Garlan define an architectural style as:

<<An architectural style defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints, having to do with execution semantics, might also be part of the style definition>>

Shaw and Garlan in [Sha96]

2.2.5 Views

Mostly, software architectures describe the structures of large and complex software systems. To ease the complexity and address large systems, the structure of software architectures can be separated into different views. A view is a representation of a set of architectural elements and the relations among them [Cle05]. As a result, each view allows stakeholders to examine and analyze a specific set of concerns of the software architecture.

Kruchten proposes the 4+1 model of views which has been adapted to UML [Kru95]. This model proposes the organization of an architecture using the Logical View, the Process View, the Development View, and the Physical View. Bass et. al propose three views: the Module View, the Component-Connector View and the Allocation View [Bas03].

A representation of the software architecture which all view models agree for their importance is the distribution view. In [Bas03], it is called the Deployment Style which is part of the Allocation View and in [Kru95] it is called the Physical View. In this view, elements of other views are allocated to different physical units (hardware). This view of the architecture allows analyzing how requirements are met by characteristics of hardware such as CPU properties, memory properties and bandwidth.

2.3 Architecture Description Languages for Distribution and Mobility

An important characteristic of distributed systems is location-transparency. This means that elements of a distributed system should be provided with the same functionalities as if they were in a centralized system. This implies that:

- Clients are not aware of where their servers are located in order to perform communications.
- Resources are accessible even though they are distributed
- Elements can move without others noticing it i.e. they are accessible and are able to communicate with the elements and resources in their new location.

In software architectures, components are defined independently of with whom they communicate with and the coordination of interactions is performed by connections or connectors. In this way, software architecture is a technique that can be used to provide location transparency to elements of a distributed system.

Describing software architectures of distributed systems is a mechanism for analyzing non-functional properties of a system such as performance, reliability, and security. Different architectural decisions can be made in allocating architectural elements to the nodes of a network in order to fulfil properties such as response time or latency.

Software architecture is a useful approach for modelling mobile systems since they are a typical example of complex and dynamic systems that need to be adapted and reconfigured to resources, network, non-functional requirements such as fault tolerance, security, and performance. Also, mobile architectural elements can be used in order to improve the efficiency and flexibility of software architectures of distributed systems.

Basically, ADLs that support the description of mobile software architectures should provide mechanisms for describing dynamic systems. Surveys have been performed on different dynamic ADLs such as in [Cue02] and [Bra04]. However, ADLs for mobile distributed systems should introduce other characteristics. In this section, an overview of the ADLs that have addressed distributed and mobile software systems are presented.

2.3.1 Darwin

Magee, Dulay and Kramer at the Imperial College in London were the creators of Darwin, one of the first ADLs that has focused on the specification of software architectures for distributed systems [Mag95]. Darwin has a textual and a graphical notation. It describes configurations by binding components which provide and require services. Also, Darwin allows the construction of composite components by binding other components with their composite components.

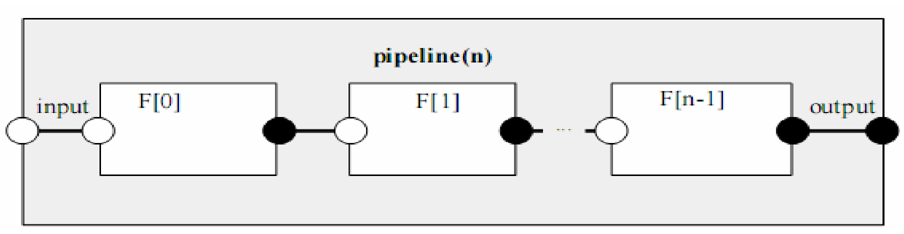


Figure 4. Graphical notation of a composite component pipeline in Darwin taken from [Mag95]

In Figure 22, a pipeline that is composed of filter instances is defined using the graphical notation of Darwin. It can be observed that the filled circles are the services that components provide and the empty circles are the services that components require. A binding between two subcomponents is performed by connecting a required service of a component with a provided service of another one. Bindings between a composite component and a subcomponent can also be performed in order to allow a required or a provided service to be visible at the composite component level. For example, between *F[0]* and the pipeline and the *F[n-1]* and the pipeline.

As it can be noticed, Darwin does not provide connectors as first-class entities for modelling interactions among components. The authors of Darwin argue that a connector is not needed as a first-class entity. If complex interactions need to be modelled, a component can perform this functionality without the necessity of having two different types of architectural elements.

Darwin supports the configuration of distributed component instances and their remote communication. Component instances are distributed on different machines by using a Darwin expression that assigns an integer to each instance. For example, to assign that each filter instance of Figure 22 is distributed the following Darwin expression is used *F[k]@k+1*. These integer expressions are then mapped by the runtime system to real machine addresses and the component instances are configured on those machines.

To allow remote communication among components, the authors in [Mag97a] and [Mag97b], model a component called LOCATE which stores the locations of instances. This component allows other components connected to it, to query the locations of other components. However, this component is an artefact of the modelling technique and not a primitive of the architecture.

Darwin has used π-calculus to specify a formal semantics of its bindings. π-calculus is elegantly used to formalize remote communication between components by transmitting references between processes (components) in messages. Although Darwin only supports a constrained dynamic manipulation of the structure, since runtime changes must be known a priori, π-calculus is appropriate in order to describe its dynamic structures.

Darwin has been used in developing distributed programmes in C++ using an environment called Regis [Mag94]. Regis provides components of a system to be instantiated at configuration time as well as at runtime. Regis also supports a parser and a compiler for Darwin bindings. However, the implementations of the components are not generated by the compiler but are implemented in a traditional programming language. Darwin has also been used in the CORBA environment to specify the overall architecture of component-based applications [Mag97b].

However, in the literature, new advances to Darwin in constructing software architectures with mobile components cannot be found. Since 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 components that cross boundaries.

2.3.2 C2Sadel

C2Sadel is an event-based ADL which was originally designed for supporting the description of user interface systems [Tay95]. C2Sadel only supports an architectural style called C2. This style is able of describing dynamic and distributed software architectures thanks to its potential usage of connectors. The C2 style follows a layered approach. C2Sadel has a framework and code generator called DRADEL for generating skeletons of applications [Med99].

Architectures in C2Sadel are described in terms of components, connectors and topologies. Components maintain a state and perform computations. Each component has an interface which consists of a set of messages that may be requested (top interface) or notified (bottom interface). Connectors bind components together to form architectures. Connectors are responsible of routing, broadcasting and filtering messages. The unique feature of C2 connectors is that they do not have a particular defined interface. Their interface is a function of the interfaces of components attached to it. This is called context reflective interfaces. The topology of the architecture is defined by connecting the tops of components to bottoms of connectors and bottoms of components to tops of connectors. Also, in C2, connectors can be connected and a component can only be connected to a single connector.

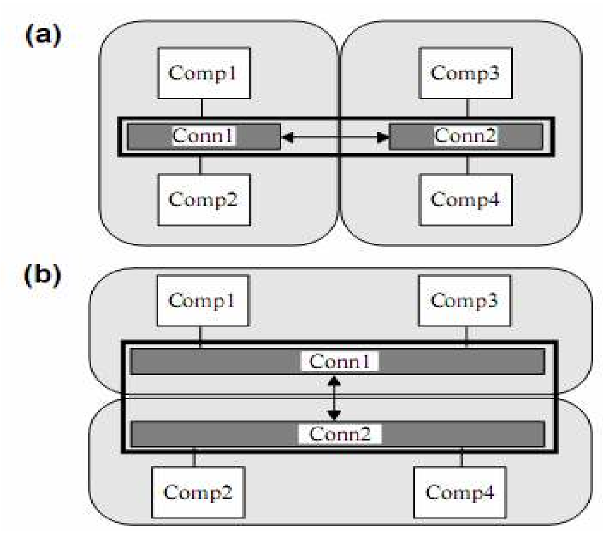


Figure 5. Connectors intermediating between distributed components taken from [Dos99]

C2 provides distributed components to communicate through encapsulating middleware technology access through connectors [Dos99]. The approach consists of implementing a single virtual software connector using a set of segmented connectors. Segments of the virtual connector are linked across the network. Figure 5 shows how connector segments connect distributed components. Shaded ovals represent network boundaries (e.g. hosts). In Figure 5 (a), messages sent to any segment of a virtual connector are broadcast to other segments. In Figure 5 (b), the virtual connector is separated into a top segment and a bottom segment. The connector segments are connected by a middleware. As a conclusion, each host of the distributed system has a connector segment. The connector segments shown in Figure 5 are implemented by the C2 framework.

C2 provides an implementation infrastructure for providing mobility [Med01]. Basically, mobility is provided by evolving the configuration of the distributed software architecture. Evolution is provided by a component called Admin that is connected to each segment connector of the distributed system. To support mobility of a component the Admin component invokes methods for disconnecting and deleting a mobile component connected to its segment connector. Then, the Admin component connected to another segment (on another host) adds the component and attaches it to its segment.

C2Sadel is an ADL that has been designed focusing on the real development of applications from its software architectures. However, it does not provide explicit primitives for modelling mobility in a platform independent way. For example, the mobility provided in C2 highly depends on the implementation infrastructure. In this way, distribution and mobility properties cannot be automatically generated from C2Sadel specifications. Neither, automatic deployment of components can be performed by the tools. Also the mobility that C2Sadel provides is restricted to software components, i.e. connectors in C2Sadel are not mobile.

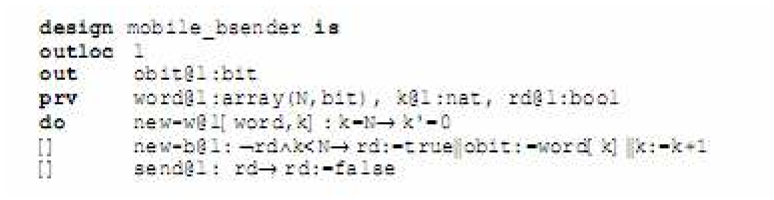
2.3.3 Community

Community [Fia03] is an ADL that is based on category theory [Fia04] and on parallel design languages. An advantage of Community in comparison with other ADLs is that it separates distribution as well as computation, coordination and configurations [Lop02]. The Community Workbench [Oli05] is a tool that provides an environment for defining Community configurations, verifying them and probe specific scenarios.

In Community, components and connectors are called designs. Designs consist of input, output and private channels as well as private and shared actions. Input channels read data from the environment, output channels produce data that can be read by the environment and private channels produce data that cannot be read by the environment. Private actions represent internal computations and their execution is under control of the design. Shared actions represent interactions between the design and the environment.

In Community, space is modelled in an abstract way through a abstract data type called loc. Loc models the positions of space depending on the notion that the modelled system needs. In this way, Community can model different kinds of mobility. A position (location) is assigned to any constitute of a design i.e., channels and actions are location-aware.

In Community, the smallest unit that can be mobile is any constitute of a component design. A design is extended with location variables that are similar to channels. Thus there are input locations and output locations. Input locations are under the control of the environment whereas output locations are controlled by the design. In Community, distribution connectors synchronize the location variables of different component designs. This allows the ability to define different mobility patterns. For example, a distribution connector specifies which component can change the location of another component.



```
design mobile_bsender is
outloc  l
out     obit@l:bit
prv     word@l:array(N,bit), k@l:nat, rd@l:bool
do      new-w@l[ word,k] : k=N→ k'=0
[]      new-b@l: ¬rd∧k<N→ rd:=true‖obit:=word[ k] ‖k:=k+1
[]      send@l: rd→ rd:=false
```

Figure 6. A mobile component specified in Community taken from [Lop04]

Figure 6 shows a mobile component specified in Community. It can be observed that the design has the *outloc* channel. This models that the location of the mobile component is controlled by the environment. Also, it can be observed that each channel and action is assigned with a location using @ expression. In this case, all constitutes of the component are located in l.

Currently, Community is one of the most expressive ADLs for modelling and analyzing distributed and mobile systems. It allows an analyst to simulate different mobility behaviours using the Community Workbench. However, Community models the change of location as a change of value without considering dynamic reconfiguration of the mobile entity with the environment. Community also does not focus on developing mobile and distributed software systems.

2.3.4 MobiS

MobiS [Cia98] is a specification language that extends PoliS [Cia99] for mobility. PoliS is a coordination language that is based on a multiple tuple-space and multi-set rewriting model. PoliS has been declared to be also an ADL because it separates coordination from computation.

MobiS introduces spaces as first class entities that can move and are hierarchically structured i.e. tuple spaces can be nested. MobiS considers spaces as software architecture components that can be composite. A MobiS space can contain three types of tuples: ordinary which are ordered sequences of values, program tuples, which represent agents, and space tuples, which contain subspaces. A program tuple can change its parent space.

Communication between two components is performed by putting tuples that represent messages in the same space. Mobility in MobiS is modelled by the consumption and production of space tuples by rules. A movement is performed step by step in a tree hierarchy of spaces. MobiS defines an architectural style for different units of mobility [Cia99]. MobiS uses spaces to represent both components and multicast channels that support communication among components.

MobiS is expressive enough to fine-grained mobility. However, MobiS does not provide an explicit primitive for locations. It uses spaces both to model components and locations. MobiS also focuses on model checking, although in the literature no model checker has been found for MobiS.

2.3.5 LAM Model

LAM [Xu03] is an architectural model that is formalized with Prt net (a high level formalism of Petri nets) [Gen87]. It does not explicitly provide a proper textual syntax since it only defines a model basing on Prt nets.

In LAM, components represent locations of mobile agents. A component is made up of an environmental part and an internal connector. Internal connectors connect agents with components. Both components and connectors are represented as PrT nets. Agent mobility is represented by the transition firing at runtime which moves an agent from a component to another through the connector.

LAM models in a realistic way logical mobility, since the movement of an agent is performed by deactivating and disconnecting it from an environment and then by reactivating it and connecting it to an arrival environment. Also, mobility is performed taking into account the tree structure of components. However the mobility in LAM is restricted to agents. In LAM, agents can have dynamic connections whereas components are immobile. Thus, LAM models differently components and agents.

2.3.6 π-ADL

π-ADL is a formal ADL that defines the behaviour and structure of software architectures [Oqu04]. It is based on π-calculus for defining its semantics. π-ADL supports a textual notation as well as a UML profile for providing a graphical notation [Oqu06]. In π-ADL, an architecture is described in terms of components, connectors, and their composition. A composite component is defined by connecting components and connectors and by connecting the composite component and its internal architectural elements.

π-ADL provides the definition of dynamic software architectures through parameterization. In this way, the number of components and connectors of architectures is assigned at runtime using a parameter. π-ADL simulates mobility of software architectural elements by dynamically deleting a subcomponent from a composite component and adding it to another component.

However, the dynamic adaptability that π-ADL provides is limited since changes have to be anticipated. For example, the composite component has to have a component type defined in its specification in order to allow that component to move to it. Also, π-ADL does not explicitly support location and mobility which are essential for modelling distributed and mobile software systems instead it simulates them by using components and composite components. In this way, the analysis and generation of a software system does not deal with specific properties of distributed systems.

2.3.7 Con Moto

Con Moto is one of the most recent ADLs that have been developed to address mobile and distributed systems [Gru04][Gru05][Sch06] . The architectural model is specified in a behavioural model and a structural model. In Con Moto, the behavioural model uses polyadic π-calculus for expressing non-functional properties and message passing of components in a distributed system. In the structural model, Con Moto provides primitives to describe the structure of a distributed and mobile system. Con Moto also has both textual language which is an XML dialect and a graphical notation (see Figure 7).

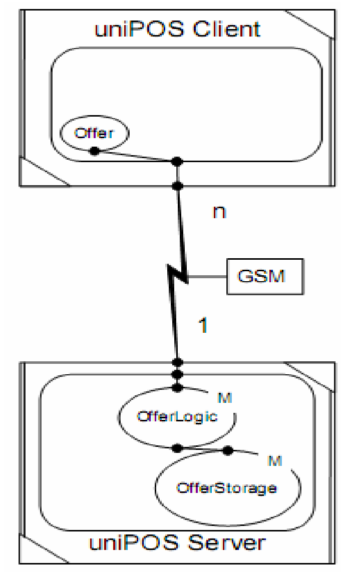


Figure 7. A structural model in Con Moto taken from [Gru04]

Con Moto distinguishes between logical and physical components. Physical components are devices such as PDAs or servers and logical components model software components. The great differences between logical and physical components are that physical components act as execution environments for logical components and physical components have resource constraints. For example in Figure 7, the *uniPOS Server* and the *uniPOS Client* are physical components and the *Offer*, *OfferLogic* and the *OfferStorage* are logical components. Also, Con Moto distinguishes between physical and logical connectors. Physical connectors connect physical components such as the one between *uniPOS Server* and *uniPOS Client* in Figure 2. Logical connectors allow logical components to communicate. Logical connectors can be embedded in many physical connectors. For example in Figure 2, *Offer* can communicate with *OfferLogic*.

In Con Moto, components are the smallest entity of mobility. Components that are marked with an M, as observed in Figure 2, indicate that a component is mobile. However, no information is available for knowing how to specify that a component is mobile or not in the textual language. Also, Con Moto does not provide a precise dynamic model in order to specify how mobility causes the reconfiguration of the structure of the software system.

In conclusion, although Con Moto is a recent approach that is still in development, it is a step forward in the state of art for mobile software systems. This is because it includes explicit primitives for deploying components and supports non-functional properties. Also, it focuses on simulating the behaviour of distributed systems. However, Con Moto needs to define a formal model for mobility in order to be applied to more realistic examples. It does not provide an expressive language for specifying when components move or how. Also, Con Moto does not provide code generation of its specified software architectures to technological platforms.

2.4 Comparison

There are many features that can be used in order to compare and analyze the ADLs presented in section 2.3. However, the features that are of interest in the scope of this thesis are the ones related with distribution and mobility. The paper of Roman et.al [Rom00] is an appropriate starting point for discovering features that are essential for models that support mobility. In this paper, a framework for viewing mobility is described. They propose that space and coordination are the most important dimensions to be considered for dealing mobility and that models which provide mobility are differentiated on how they assume the unit of mobility, location and context which highly depends on the coordination mechanisms that a model provides. In the following, the features that have been considered in the comparison are explained:

- **Location:** Locations represent the different positions where a mobile entity can move in space. Locations, in a model, have to be explicitly dealt as first-class entities and be distinguished from other entities of the model. Locations as first-class entities of a model enable the modelling of space and specify where a mobile entity can and cannot move.

- **Mobility Support**: This feature describes how the architectural model supports the movement of a mobile entity. Basically, mobility support determines what implications are caused when an entity is moved.
- **Unit of Mobility**: Represents the smallest entity of a model that is allowed to move. This feature is important because it represents which entities a model enables to move.
- **Location-Awareness**: This feature determines whether an entity can be aware of its current location or not. Being aware of the current location of a entity is important because it allows an entity to take decisions depending on his current location.
- **Migration Decision**: This feature specifies when and what causes an entity to move. This feature is associated with the terms passive and autonomous mobility. Passive mobility is the movement of an entity caused by the environment. Autonomous mobility is the movement of an entity caused by the same entity.
- **Coordination**: The coordination mechanisms that a model supports determine the context that is seen by each entity. The coordination mechanisms should be specified separately from the functionality of the entity. Since a basic characteristic of ADLs is the separation between coordination and computation all ADLs support this characteristic. Therefore, what differentiates the coordination feature in ADLs is whether they support connectors or not.
- **Formalism**: Models have to be formal enough in order to enable a precise description of the semantics of the distribution and mobility properties that ADLs provide. The formalism chosen should provide the primitives needed to specify mobility features.
- **Graphical Support**: ADLs provide graphical support in order to be usable. Most ADLs do provide one. However, the objective of this comparison is to

discover which ADLs provide a graphical notation for specifically describing the distribution and mobility primitives. A basic graphical notation for distributed systems is the ability to provide a deployment notation, i.e., a graphical notation for specifying where entities are distributed.

- **Middleware**: Middleware support the software development task by implementing models in a specific technology. Middleware implement the distribution and mobility primitives that a model provides. In this comparison the objective is to discover to what extend ADLs have been used to develop distributed and mobile systems.
- **Tool Support**: A tool supports a developer during the development process through providing facilities and guides. Some of the facilities can be modelling support, verification, code generation, code execution, and simulation. The objective of this comparison is to discover what facilities do ADL tools provide for distribution and mobility.

A comparison table has been developed from the features and the approaches analyzed in the above section. This table is divided into two separate tables (

Table 1 and Table 4) due to the limitation of the page dimensions.

Table 1. Comparison of ADLs that address distribution and mobility (Part 1)

	Location	**Mobility Support**	**Unit of Mobility**	**Location-awareness**	**Migration Decision**
Darwin	An integer value that represents a machine	No support	Not defined	No Location-awareness	No Support
C2Sadel	A border connector on each host.	Reconfiguring the architecture	Components mobile, all connectors static	No Location-awareness	Passive

	Location	**Mobility Support**	**Unit of Mobility**	**Location-awareness**	**Migration Decision**
Community	A value of an Abstract Data Type	Change in a value	Fine grained mobility of a component design	Location-aware	Both autonomous and passive mobility
MobiS	Spaces that are composite components	Consuming and producing spaces tuples by rules	Spaces	Not explicitly	Both autonomous and passive
LAM Model	Components nested in a tree structure	Reconfiguring the architecture	Mobile Agents, components and connectors are static	Not explicitly	Both autonomous and passive mobility
π-**ADL**	composite components represent locations	Reconfiguring the architecture	sub-components are mobile	Not explicitly	Both autonomous and passive mobility
Con-Moto	Physical Components	Reconfiguring the architecture	Components are mobile	No Location-awareness	Not clear

Table 2. Comparison of ADLs that address distribution and mobility (Part 2)

	Coordination	**Formalism**	**Graphical Support**	**Middleware**	**Tool Support**
Darwin	Binding among components, No explicit notion of connector	π-calculus	Support but not for distribution	Implemented on CORBA for remote invocations	Compiler generates C++ code and automatic distributed configuration
C2Sadel	Explicit support for connectors that provide distribution and mobility	Semi-formal, first order logic.	Support but not for distribution	Supports remote invocations and code on demand mobility	Generates application skeletons
Community	Explicit support for connectors that provide distribution coordination	Category theory	Support but not for distribution	No support	Modelling, verification and simulation tool
MobiS	No explicit notion of connectors	multiple tuple-space based on PoliS	No graphical Notation	No support	No support
LAM Model	Explicit support for connectors	PrT Nets	PrT Nets notation	No support	No support

	Coordination	**Formalism**	**Graphical Support**	**Middleware**	**Tool Support**
π-**ADL**	Explicit support for connectors	π-calculus	UML profile but without deployment notation	Unavailable information	Code generation to Java, visual modelling, verification tools. Unavailable information for distribution support.
Con-Moto	No explicit notion of connector. Logical and physical connections.	Semi formal, Behaviour based on π-calculus but structure is not formal	Own graphical notation for deployment	No support	Simulation Tool

2.5 Conclusions

From the features that have been taken into account in the comparison, the state of art of the ADLs that have dealt with distributed and mobile systems can be summarized as follows:

- The notion of location has been introduced as: a new type of connector, a new type of component, a value of a data type, and a composite component. In the case when it has been introduced as a composite component the notion of location is quite ambiguous since the same concept is used for both representing locations and hierarchical compositions.
- The support of mobility highly depends on how the notion of location is introduced in the ADL. When location is represented by a value, mobility is supported by the change of this value. When location is supported by

architectural element (component, connector, composite component) mobility is supported by reconfiguring the software architecture.

- Components are the units of mobility provided by ADLs. Connectors have not been considered to be mobile entities.
- Few ADLs allow there elements to be location aware.
- Most ADLs provide passive and autonomous mobility.
- Most ADLs provide connectors as first-class entities in order to provide complex coordination mechanisms. However, the interesting approaches are those that provide special coordination mechanisms for distribution such as Community and C2Sadel.
- ADLs have been formalized by using formalisms that do not explicitly provide first-class entities for distribution and mobility.
- Most ADLs provide a graphical support to model components, connectors and configurations. However, an ADL that provides the graphical support for distributing architectural elements is needed. Providing a graphical support for describing the distribution and mobility issues allow the ADL to be more usable. Con-Moto has a friendly notation to distinguish between logical components and the locations where these components are assigned.
- The only ADLs that have been implemented on a specific middleware have been Darwin and C2Sadel. However, Darwin does not provide mobility and the mobility that C2Sadel allows is restricted.
- Each ADL provides tool support with a different focus. For example, Darwin allows the automatic configuration of components to locations. C2Sadel provides skeleton code generation. Community and Con-Moto focus on simulation. π-ADL provides many tools however, there was not any information available to analyze to what extend does it support code generation of distributed and mobile code.

As a conclusion, the state of art needs to be updated with an ADL that provides:

- Location as a first-class entity

- Location as an architectural element in order to support mobility through reconfiguring the architecture.
- Location-awareness to its distributed elements.
- All architectural elements to be mobile entities.
- The modelling of both passive and autonomous mobility.
- Distributed coordination mechanisms.
- A formal basis with an appropriate formalism that enables locations and mobility as primitives.
- A graphical notation of the distribution and mobility primitives of its ADL.
- An implementation to a specific technological middleware.
- Tool support for the modelling, verification and code generation of the distributed and mobile code.

CHAPTER 3
ASPECT-ORIENTED SOFTWARE DEVELOPMENT FOR DISTRIBUTED SYSTEMS

3.1 Introduction

Since the term “software crises” emerged in the late 1960s, the decrease of software complexity was a matter to be dealt in software engineering techniques and the software development processes. During the 1970s, Parnas proposed the term modularization as a criterion to simplify software development and improve software understanding and quality [Par72]. Modularization decomposes complex software systems into smaller parts called modules. The practice of dividing software into different areas of interest is widely referred to as Separation of Concerns (SoC) [Dij74]:

<<*We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day. . . But nothing is gained—on the contrary—by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns."* >>

Edsger Dijkstra in [Dij74]

SoC is a technique where software can be built in an incremental way. Different developers specialized on specific areas can develop modules of specific concerns. In this way, SoC provides maintainability, traceability, improves comprehension, and makes software evolve in a flexible way.

Object-Oriented Software Development (OOSD) [Boo04] is an approach that follows SoC. OOSD provides a flexible way of building software from core concerns that are modularized (classes) and by offering mechanisms where these concerns can be reused and adapted (inheritance and association). However, as nowadays software systems have become more sophisticated new concerns which mainly deal with non-functional requirements have to be taken into account. These concerns usually intermix (crosscut) the systems functionality. OOSD comes short in modularizing crosscutting concerns and as a consequence software becomes less reusable, adaptable and maintainable.

Aspect-Oriented Programming (AOP) [Kic97] supports separation of concerns by modularizing code that crosscuts the software system in separate entities called aspects. In this way, the code is not tangled nor scattered in multiple entities. As a result, the code is better localized, maintained and reused. Aspect-Oriented Software Development (AOSD) [Fil04] emerged in order to apply techniques of separating crosscutting concerns to all phases of the software development process.

This chapter provides an introduction to aspect-oriented software development, shows how the development of distributed software systems benefit from AOSD techniques and how different AOSD approaches have addressed distributed software.

3.2 Aspect-Oriented Software Development

AOSD emerged from AOP. The early contributors to AOP [Kic97] were Cristina Lopes and Gregor Kiczales at the Palo Alto Research Centre (PARC) of the Xerox Cooperation. Kiczales was the leader of the team that created AspectJ [Kic01]

[Lad03] [Mil04], the first practical implementation of AOP and, currently, the most extended one. In December 2002, the AspectJ project was transferred from Xerox to the open source community at eclipse.org [ASP07].

After the great success of AOP, the concepts and techniques that AOP provides have been taken to earlier phases of the software life cycle such as requirements [Bri02] [Ras02], and analysis and design phases [Suz99] [Aks94]. In this way, AOSD emerged in order to improve the level of modularity, reusability, evolution and maintainability in software development. In this section the basic concepts of AOSD are going to be presented.

3.2.1 Crosscutting Concerns

A crosscutting concern is behaviour or data that is spread in different modules. Crosscutting concerns can be non-functional requirements such as security, logging, authentication, etc or functional concerns that are spread among different modules. Crosscutting concerns cause the following:

- Tangling of Concerns: Tangling of Concerns is caused when a module contains multiple concerns at the same time (see Figure 22(a)). As a result, the module is less maintainable, reusable, and comprehensible.
- Scattering of Concerns: Scattering of concerns is caused when a single concern is defined in many modules such as performance concerns. Scattering of concerns provoke identical definitions to be repeated in multiple modules.

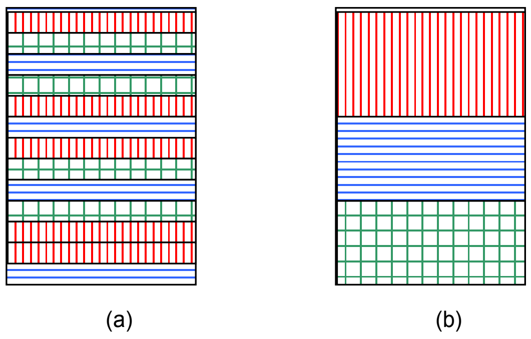


Figure 8. (a) The tangling concerns in the traditional applications (b) the separated concerns in AOSD taken from [Lad03].

3.2.2 Aspect

An aspect is a software entity that provides mechanisms for encapsulating crosscutting concerns. Figure 22(b) shows the elegancy that is achieved through aspects in the modularization and the localization of the crosscutting concerns of Figure 22(a). An aspect normally contains declarations similar to the ones of a class and can contain, depending on the aspect language, the mechanisms that specify the interactions of an aspect with the underlying system. For example, in AspectJ aspects are the following (see section 3.2.3 for definition of pointcuts and advice):

<<*Aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations.*>>

Gregor Kizcales et al. [Kiz01]

3.2.3 Weaving

Weaving is the process that combines the concerns of the system (which can be modularized in aspects and objects) following the weaving rules. The weaving rules specify how the aspects are integrated with the rest of the system. Weaving rules must not be specified in the core entities (e.g. classes) since a basic concept in AOSD is *obliviousness* [Fil00] i.e. the core entities are unaware of the woven aspects. In this way, the system can be changed only by changing the weaving rules.

In AspectJ, the weaving rules are defined in the aspect entity. In order to define the weaving rules it introduces three concepts:

- Join points: Join points are well-defined points in the structure of a program where aspect behaviour can be attached. The most common elements of a join point are method calls.
- Pointcuts: A pointcut selects a set of join points and collects context about these join points. For example, which class a join point belongs to or the arguments of a method. A pointcut allows the aspect to do something with a single statement in many places (this is called *quantification* [Fil00])
- Advice: Advice is the behaviour to be executed at a join point that has been selected in a pointcut. An advice can execute before, after or around a join point. The advice makes AOSD *oblivious* since the join point is unaware that the advice is executed.

In AspectJ, aspect reusability is lost, since aspects are defined for a specific context. In other approaches such as JAsCo [Suv03] or MINOS [Mez01] weaving rules are defined in external entities to aspects. For example, in JAsCo, these entities are called hooks. Hooks are generic, reusable, and can be considered a combination of AspectJ's pointcuts and advice. The initialization of a hook with a specific context is done by making use of connectors.

Two weaving models are distinguished:

- Static Weaving: Static models are those in which the aspects and non aspect entities are declared as separate entities, however at compilation time the two entities are combined into one. This model has the drawback that at execution time the aspects cannot be manipulated. As a result, aspects fail to gain a high level of evolution, maintenance and reusability. Examples to this model are AspectJ [Kic01] and Composition Filters [Aks94] (see section 3.3.2.2).
- Dynamic Weaving: In this model the separation of the aspects and non-aspects entities is conserved at all moments even at execution time. Aspects

can be woven and unwoven at run-time. Examples to this model are the PROgrammable extenSions of sErvices (PROSE) platform that provides dynamic AOP for Java [Pop02] (see section 3.3.1.4), JAsCo [Suv03], the Disguises Model [San98] (see section 3.3.2.4) and the Dynamic Aspect-Oriented middleware Framework (DAOF) [Pin02].

3.3 Aspect-Oriented Software Development for Distributed Systems

Distributed applications are inherently more complex to develop than centralized ones because of the additional requirements caused by partitioning the software system across the network (e.g., handling of communication, replication, naming and synchronization between system components, network failures, management of load balancing, etc). The development of distributed systems can be facilitated by achieving a level of transparency where all issues related with distribution are hidden. Middleware technology such as CORBA [COR07], Java Remote Method Invocation (RMI) [RMI07] and .NET Remoting [Ram02] have been used to simplify the development of distributed applications and provide some type of transparency.

The different technological middleware offer constructs that solve a number of problems such as remote access, fault tolerance and security. However, applications that need these kinds of primitives have to introduce them in an inelegant way. For example, Figure 9 shows the code of a C# class that its objects offer and request services to and from other remote objects using .NET Remoting. The code in italics shows the code related to distribution.

As it can be observed, the same module (in this case the class) has code that is concerned with functionality and remote access. Thus distribution-related concerns such as remote access (distributed communication) crosscut the code of a distributed application. This tangled code (sometimes referred to as spaghetti code) affects the implementation of the methods (as it can be observed in the *Start()* method) and also in the class hierarchy (the class has to derive from

MarshalByRefObject). As a result, the code of the class *SearchAgent* cannot be reused because it has the tangled distribution code. Also, the class *SearchAgent* cannot benefit from inheritance since it needs to inherit from *MarshalByRefObject* and C# does not allow multiple inheritance.

When AOSD techniques are used, all the code related to remote access can be separated into an aspect and then it can be weaved with all the existing classes that need remote access. In this way, the distributed application has a higher quality since maintenance, reusability and understanding of the remote access concern is improved. AOSD is a promising technique for developing distributed systems since it separates concerns that crosscut them. In addition to the remote access concern depicted in Figure 9, [Sub05] detects other crosscutting concerns found in distributed applications such as synchronization, fault tolerance and security.

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.IO;

namespace DistributedSearchAgents {
     public class SearchAgent : MarshalByRefObject, INotification {

               ...
        public SearchAgent(string[] keywords, string origin, ArrayList
locationsToVisit){
                  ......
            }

        private void Search(){
                  ... ...
            }

        public void Start(){
       Console.WriteLine("Agent " + this.ToString() + " starting");
       SearchAgent pal = (SearchAgent)
             Activator.GetObject( typeof(SearchAgent),

"tcp://dofi.dsic.upv.es:39876/SearchAgent.rem");
                  .... ....
            }
       static void Main(string[] args){

              ......
               SearchAgent myAgent = new   SearchAgent(keywords,

"tcp://klaatu.dsic.upv.es",locationsToVisit);
          // Remoting
          TcpChannel channel = new TcpChannel(39876);
          ChannelServices.RegisterChannel(channel);
          // Published Object
          RemotingServices.Marshal(myAgent,"SearchAgent.rem");
          Console.WriteLine("SIMULATION START");
          Console.WriteLine("");
          myAgent.Start();
          Console.WriteLine("Simulation finished. Keypress to exit.");
          Console.ReadLine();
            }
      }//end SearchAgent

}//end namespace DistributedSearchAgents
```

Figure 9. Code to handle a distributed client/server object in .NET Remoting

In the following, an overview of the different approaches that have addressed AOSD for distributed systems is going to be presented. The objective is to provide a notion of the research works that have been developed to deal with AOSD and distribution at the implementation and at the analysis and design phases of the software life cycle.

3.3.1 Implementation

This section presents how AOP languages and platforms have been used to implement distributed and mobile applications.

3.3.1.1 AspectJ

AspectJ [Kic97] is a general purpose language that extends the Java object-oriented programming language in order to incorporate aspects. In AspectJ, the base code is programmed using Java objects and in order to define aspects, AspectJ is used. AspectJ provides the aspect construct that contains: pointcuts, advice, and Java constructs. The aspect, pointcuts and advice concepts have been previously introduced in sections 3.2.2 and 3.2.3.

As previously stated, AspectJ is one of the first AOP language as well as the most extended. Although AspectJ does not explicitly provide any support for distribution and mobility constructs, it has been combined with existing middleware frameworks to implement distributed and mobile applications. These applications have been developed by implementing the functional requirements in Java and by implementing the code related to distribution middleware such as CORBA and RMI in aspects.

One of the first versions of AspectJ was used, specifically AspectJ0.3beta1, to implement a simple remote client/server application using CORBA [Pul99]. The objective was to work out if aspects could be used to improve the understanding, coding and reuse of applications using CORBA.

Two aspects were defined for the client: an aspect to be woven with the client when it was in a local version and another aspect to be woven when the client required to be executed on the CORBA environment for distributed communication. Also, two aspects were defined for the server: an aspect in order to allow the server to provide a name server and another aspect in order to allow the server to provide a file based functionality.

Mostly, the resulting benefit of this experience was that the structure of the application was improved. This improves the transparency of distributed applications as the programmer can implement the objects without getting confused with distribution issues.

However, some conflicts may occur with this implementation, as both aspects for client and server maybe woven at the same time. This problem can be solved by defining another aspect that takes the responsibility of controlling the priority. Another problem with this implementation is that there is no way to ensure that when a client is woven with the aspect for distribution issues, the server is also woven with the aspect for a name server. This can cause an inconsistent state where the application is not really prepared for distribution. Also, since AspectJ provides static weaving, the final code is intermixed.

However, more recent versions of AspectJ have been used to implement distributed software systems. As a consequence, two patterns have been defined: a Pattern for Distribution Aspects (PaDA) [Soa02a], and a Mobility Aspect Pattern [Gar04]. In the following these patterns are explained:

3.1.1.1.1 A Pattern for Distribution Aspects (PaDA)

Soares [Soa04] proposes to use AspectJ as a general aspect-oriented language to implement distributed applications. He believes that using a general language is easier for users because they do not have to learn different languages.

In [Soa02b], Soares reports his experience in restructuring a web-based health complaint system which was initially implemented in plain Java and RMI into AspectJ. Soares removed the RMI specific code from the initial version in pure Java into a set of aspects. He noticed that all the distribution code was tangled both in the server-side and in the client-side. As a consequence of this experience, Soares proposes a pattern called PaDA [Soa02a] that provides a structure for implementing distribution using AOP (see Figure 10). The pattern consists of implementing three distribution aspects: a client-side aspect to call remotely to the server-side, a server-side aspect to enable the reception of remote calls and an exception handling aspect. The client-side aspect is weaved with the client, the server-side aspect is weaved with the server and the exception handling aspect is weaved with both client and server.

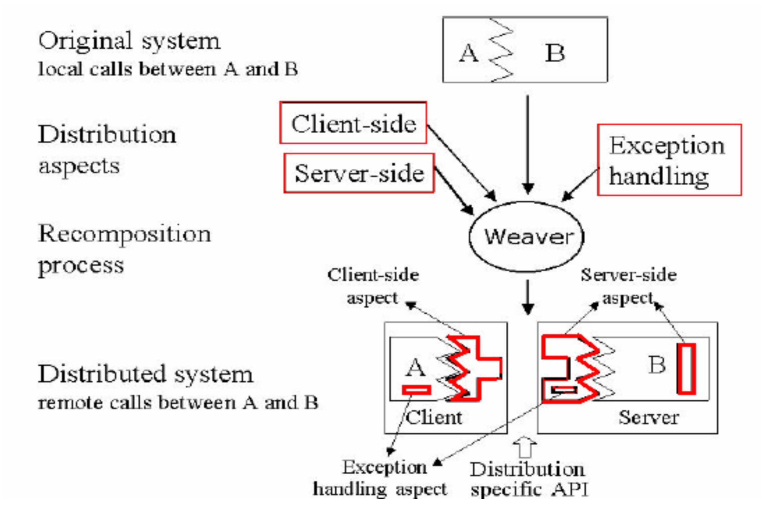


Figure 10. PaDA's Structure taken from [Soa02a]

The results that Soares encountered after his experience were that the AspectJ versions of the system using PaDA are superior to the plain Java versions. This is because the distribution code can be added in a progressive and incremental way without affecting on the original application. Also, the testing of the application can be facilitated using AOP because the functionality can be tested without distribution. Another advantage is that separating the distribution concerns in aspects can easily facilitate the change of the distribution middleware without affecting on the implementation of the other aspects of the application. In addition, distribution aspects can be easily reused and extended for other application domains.

Other techniques can be used in order to separate the distribution code by using factories or adapters' patterns [Gam95], however, the code of the version of the application where these patterns are used is more than the code in the AspectJ version. Also, the code of the server and the client has to be changed in order to incorporate these patterns to the application. In this way, the AspectJ version is more maintainable and adaptable.

On the other hand, Soares detects drawbacks in implementing distribution aspects with AspectJ. The most essential drawback is that the definition of a pointcut is defined inside an aspect. This makes the defined aspects specific for a system, or for systems adopting the same naming conventions, decreasing the possibilities of reusability. Soares suggests that either aspect parameterization or code generation tools need to be used when applications with aspects in AspectJ are developed.

3.1.1.1.2 The Mobility Aspect Pattern

AOP has been used in Multi-Agent Systems (MASs) in order to separate the code that deals with agents' mobility from the core classes, since mobility is a crosscutting concern of a mobile software system. In [Gar04], Garcia et al. present a pattern that proposes to use mobility aspects to separate the code that specifies how and when an agent should move. The pattern proposes to define abstract aspects in AspectJ for mobility. An abstract aspect [Lad03] can define abstract pointcuts and methods in order to increase the reusability of aspects. In this way, concrete mobility aspects extend the abstract ones implementing the exact details and the weaving rules. Also, mobility aspects are associated with the mobility framework. In this way, if the mobility framework should be changed, the mobility aspect is the only code affected.

This pattern has been implemented using a framework called AspectM [Lob04] that provides mobility. AspectM provides a set of abstract mobility aspects where the user extends them for its application. In this way, the mobile agents' development is performed in a flexible way since the mobility strategies are independent of the other concerns and can be easily changed. However, since aspects define pointcuts, as with all AspectJ applications, the reusability of mobility aspects are decreased. Also, the same authors of the mobility pattern suggest in [Gar06] that research has to be done in order to improve the traceability of aspects and provide tools or wizards that automate code generation of aspects.

3.3.1.2 General Object to EJB Conversion Helper (GOTECH) Framework

GOTECH is a framework [Til03] that automates the distribution of an existing Java application following the PaDA pattern (see section 3.1.1.1.1). The framework provides the programmer with a tagged language that the user introduces in the source code in order to convert the classes into Enterprise Java Beans (EJBs) and indicates where they need to be deployed. The framework generates the aspect code in AspectJ, converts Java classes into EJBs and deploys them, and compiles the AspectJ code with the EJBs.

In GOTECH, the user has to learn to use the tagged language provided by the framework. Also, this language is neither an aspect-oriented language nor a formal one. GOTECH does not weave the aspect code to the source code. It generates new EJB code and then weaves the AspectJ aspects. This is an inconvenient since the source code is not reusable and maintained. Also, the framework has to change the client code in order to allow it to interact with EJBs instead of plain Java. An inconvenient of GOTECH is that the approach assumes that the server site never calls back to the client site.

3.3.1.3 DJCutter

In a distributed software system, software entities are located in different hosts. These software entities communicate and collaborate. A software entity that is at a host shares characteristics with entities that are located at other hosts. The same may be applied to crosscutting concerns. The crosscutting concerns or an aspect may be reused by several entities that are distributed in different hosts. The crosscutting concerns maybe spread over entities that are distributed.

DJCutter [Nis04] extends AspectJ in order to address crosscutting concerns that are scattered in several objects on several hosts. Mainly, DJCutter extends AspectJ remote pointcuts. Remote pointcuts identify join points on remote hosts i.e. the advice of an aspect is executed in a host different from the join points of the object. In this way, developers do not have to include code for remote method calls in Java RMI. As a result, the development is simpler and more transparent. For example, a remote pointcut that identifies join points that are calls to the setX() method in the Point class on the hosts with the names hostId1 or hostId2, is defined as follows:

```
pointcut sample(): call(void Point.setX(int)) &&
                   hosts(hostId1, hostId2)
```

DJCutter provides an aspect server where aspects with remote pointcuts execute the advice body as well as a class loader from where the classes on each host have to be loaded. This class loader automatically weaves the aspects with the class at loading-time. Whereas in AspectJ, the user has to manually deploy the woven aspect and classes to every host. Although DJCutter has an advantage over AspectJ, having an aspect server centralizes all aspects that have remote pointcuts. This is not efficient as it may cause a bottleneck.

A shortcoming of DJCutter is that remote pointcuts are dependent on the hosts of the join points defined at compilation time. As a result, DJCutter cannot be used for defining mobile objects where their hosts change at runtime. Another, shortcoming is that DJCutter, as AspectJ, provides weaving at load time, i.e. static weaving. Also, pointcuts or remote poincuts are defined in aspects reducing aspect reusability.

3.3.1.4 PROgrammable extenSions of sErvices (PROSE)

PROSE is a platform that provides dynamic weaving of aspects [Pop02]. PROSE consists of the Java Virtual Machine (JVM) and a set of libraries. PROSE does not define a new AOP language. It provides aspect constructs that are implemented in pure Java. In order to use PROSE, one has to extend the constructs it provides, e.g. in order to define a specific aspect the base class aspect has to be extended. To provide dynamic weaving of aspects, PROSE uses the Java Virtual Machine Debugger Interface (JVMDI) to stop the execution of the JVM at join points and then calls the advice behaviour.

To support distributed systems, PROSE allows objects and aspects to be instantiated at different nodes of the network and weave aspects on different nodes with an object [Pop01]. PROSE has been used with Jini to support services that join the community dynamically. When a service joins the community it registers a proxy at a lookup service. A service then can be dynamically weaved with aspects that are stored in a central repository of aspects.

Currently, PROSE does not provide any explicit language support for mobile objects, although it can be extended. Also, PROSE aspects define pointcuts and advices. This reduces the reusability of aspects in PROSE.

3.3.1.5 AspectIX

AspectIX [Hau98] is a middleware that supports AOP for CORBA distributed objects. The middleware offers the ability to add new aspects dynamically, i.e. it provides dynamic weaving. In AspectIX, an object consists of a set of fragments (see Figure 11). Each fragment has a specific behaviour of an object. For example, a fragment may hold the constraints on the communication channel to another fragment. Aspects are specified in order to determine which fragment represents a distributed object.

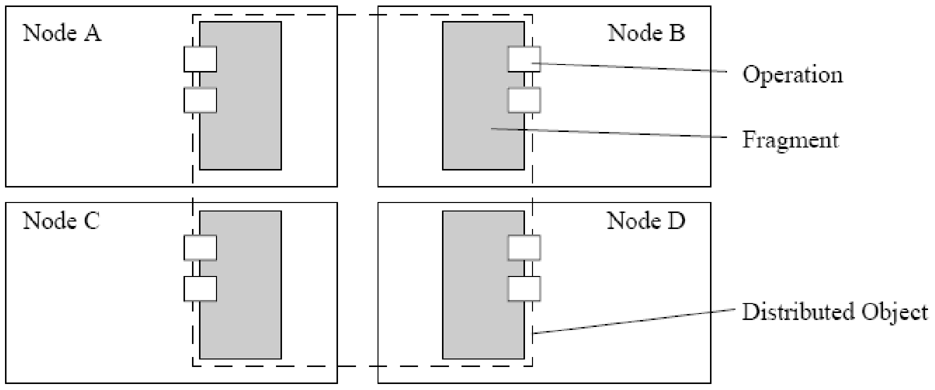


Figure 11. Fragments of a distributed object in AspectIX taken from [Hau98]

AspectIX provides mobility and replication of its objects [Gei98]. New fragments are added in order to support them. In the case of replication, a fragment is created locally, in the case of mobility the fragment is created in the destination site and the state of the previous one is transferred to it. The decision whether an object has to be replicated or migrated is specified in an aspect.

3.3.2 Design

The above approaches have proposed languages and platforms for the implementation of distributed applications. Although the applications have increased their quality considerably, the implementation does not follow the software design. In this way, software traceability is lost. As a result, there is more effort spent in the implementation, and the maintenance and evolution of the system becomes difficult [Cle04]. In order to solve this problem and in order to make AOP more reusable, appropriate design techniques should support aspect-orientation. In the following, different approaches that have considered distributed systems using aspect-oriented design techniques are presented. Some of these approaches also support the implementation stage from their design.

3.3.2.1 The D-Framework

Lopes is the first to explicitly study how distributed applications needed aspect-oriented support and the fact that distribution is a concern that crosscuts an application. Lopes designed a language framework called D for distributed programming [Lop97]. D is based on separating the tangled code of a distributed object-oriented application in aspects. The framework considers thread synchronization and remote access as clear *aspects* that should be treated separately throughout the development phases of a distributed application. Therefore, in order to support each of these aspects, the D framework consists of two declarative languages: the Coordination Aspect Language (COOL), for supporting thread synchronization and the Remote Interface Aspect Language (RIDL) for supporting interactions between remote components.

Aspects written in COOL are called coordinators. Coordinators allow objects to have concurrent threads by controlling mutual exclusion of threads, synchronization state, guarded suspension and notification. Aspects written in RIDL are called portals. Portals allow objects to be distributed by describing which methods of a class can be accessed remotely and what data can be passed: both by copy or by reference.

Aspects in D (coordinators and portals) are not types and cannot be instantiated. Aspects specify the classes that they are weaved to and the methods they can access to. An aspect is automatically associated with an object when a class is instantiated i.e. the associations between objects and aspects is one-to-one. Aspects do not have any relationships among them as well as they do not have inheritance.

DJ is the implementation of D in Java and RMI. A tool called the Aspect Weaver automates the transformations of D constructs (coordinators and portals) into Java classes as well as weaves the aspects to the base code in Java. The Aspect Weaver extends the base code with hooks (meta-data and new code) that transfer the control to the aspects at the beginning and end of methods. Also, a layer called RIDL has been implemented above RMI in order to support D portals. Figure 12 shows the run-time architecture for a client object (*aObj* in *Space 1*) calling a remote object (*aObj* in *Space 2*) that is associated with a D portal. The client object refers to a proxy of the portal (*aObjPP*) that checks if there are illegal remote calls. If there aren't any illegal calls, the portal proxy redirects the call to the portal object (*aObjP*) by using the RMI layer, and as a consequence the portal object redirects the call to the real object.

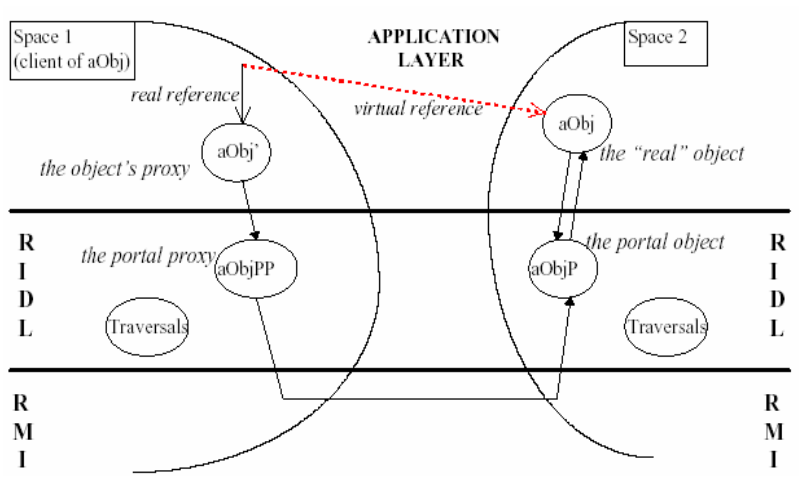


Figure 12. Run-time Architecture for D's remote objects taken from [Lop97]

Lopes has validated the D framework by applying it to a set of case studies [Lop97]. The lines of code obtained when implementing the case studies with DJ aspects were compared with the ones obtained when implementing them with plain Java. Also, the tangled code reduced by DJ was measured. Despite the fact that the validation has been performed on small and academic case studies, it gives us an idea of the advantages of the aspect-oriented approach. The validation was performed on 10 case studies. In all the case studies, the tangled code was reduced and localized in the aspect modules. The results obtained with respect to the comparison with the lines of code obtained were the following:

- Four of the case studies had the number of lines achieved in plain Java was the same as the ones achieved in DJ.

- ➢ Five of the case studies the number of lines achieved in DJ was reduced in compared with plain Java.
- ➢ One case study the result was not available.

The validation of the D framework demonstrates that distributed applications benefit from AOSD, as the lines of code are reduced and also the distribution concerns are well localized and can be properly maintained.

Although the languages of D are designed to be independent of the object-oriented programming language that the classes are implemented in (the implementation of the base code), D has been designed so that aspects are weaved to objects that have Java-like characteristics. This limits D from being applied to other object-oriented programming languages. Another important drawback of D is that aspect definitions are dependent on classes i.e. once aspects are defined, the classes that these aspects are weaved to, have to be indicated. This prevents the reusability of aspects behaviour by many classes. Also, D should be extended to support mobility and replication.

As a conclusion, the D approach is the first to provide declarative languages that support aspects. The D languages are domain-specific languages as each of them deals with a specific problem (concern). The study of the crosscutting concerns performed in order to define D and the implementation of DJ were the base for the birth of the actual versions of AspectJ. However, currently, aspect-oriented languages are general purpose languages instead of domain specific. As Cristina Lopes describes in [Lop02]:

<<The first version of AspectJ, made public in March of 1998, was a reimplementation of DJava. It supported only COOL. Another release followed soon; I believe it was AspectJ 0.1. It included RIDL>>

<<Past the transition from concern-specific to general-purpose aspect language, which happened in 1998, AspectJ evolved considerable.>>

Cristina Videira Lopes in [Lop02]

3.3.2.2 Composition Filters

Composition Filters [Aks94] is a platform independent model that extends the object-oriented model by introducing modules that can manipulate the messages that an object receives or sends. These modules are called filter modules. In composition filters objects are the concerns that can be implemented in any technology and the filter modules represent the crosscutting concerns.

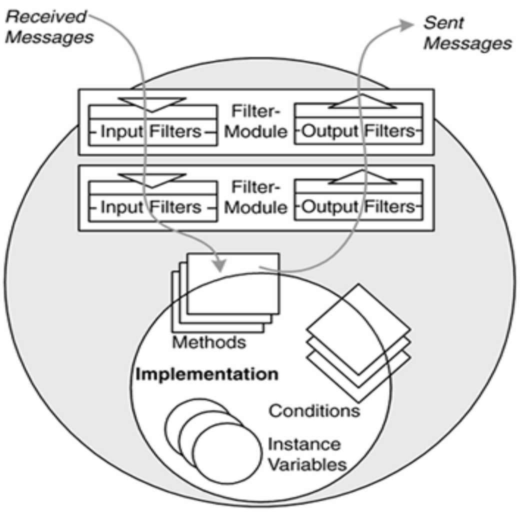


Figure 13. Filters in the composition filter model taken from [Ber04]

Filter modules are composed of a set of filter elements that determine whether a message is either accepted or rejected and what action to be performed in either case (see Figure 13). Filter modules are separated into Input Filters that filter object received messages and Output Filters that filter object sent messages. A filter element consists of:

- A condition: which specifies a necessary condition to be fulfilled in order to continue evaluating a filter
- A matching part which matches the message against a defined pattern
- substituting part which replaces parts of the message.

The Composition Filters model is implemented for several languages such as Smalltalk, C++, and Java. These implementations extend the languages syntax to include keywords to support filters attached to classes.

In the area of distributed systems, the authors of Composition Filters have applied composition filters model to abstract communications among objects [Aks94]. Abstract Communication Types (ACTs) are used to hide the interaction details among objects and thereby improve the reusability of objects. ACT classes can represent distributed algorithms, coordinated behaviour or inter-object constraints. In the distributed system design, ACTs can model layered architectures, distributed concurrency control mechanisms and security protocols.

ACTs in composition filters are classes that can manipulate messages. This is possible thanks to special types of Filters called Meta filters. These filters reify messages and pass them to ACTs as arguments of class Message. In this way, an ACT can make use of the methods of class Message such as changing the receiver, sender, server and changing the arguments of the message.

However, composition filters does not support any explicit notion for distribution in its model neither it supports a model for supporting mobility.

3.3.2.3 UML All pUrpose Transformer (UMLAUT)

UMLAUT is a framework that allows the user to weave aspects at the level of the UML meta-model [Ho02]. The weaving of aspect-oriented designs is handled at the meta-model level of UML. Thus, a weaving operation is described as a transformation from an initial UML model to a final UML model. The final UML model is called an implementation model as it contains enough information for implementing the model. The UMLAUT framework provides a library that has a set of transformation operators. The designer defines the weaving composing a set of operators.

To support distribution aspects, a stereotype called remote is used in order to indicate that two classes are related by a physical distribution medium. However, from this stereotype there is not sufficient information to generate an implementation model. Also, using the stereotype called remote on classes is quite restrictive, since normally instances are deployed in different machines and not classes.

3.3.2.4 The Disguises Model Approach

The disguises model [Her03] is a model that separates synchronization, concurrency control, distribution [San00], and replication from the functional behaviour of a system in aspects. An aspect in this model is a non-functional property and a non-functional property is represented by an aspect. Also, the model provides dynamic weaving of aspects.

In the disguise model, a standard language such as Java is used to implement the functional behaviour, a specification language is defined for each aspect (disguise) that the model supports, and a specification language is defined for specifying the composition rules of the disguises with the functional behaviour. Also, a UML profile is defined with stereotypes for each of the concepts of the model. Therefore, the developer can design graphically its specification and then the specification languages that the model supports are automatically generated. Also, code generation is supported from the specification languages to Java.

The disguises model establishes a distinction between objects and aspects by computation reflection techniques [Mae87]. It defines auxiliary objects that establish a connection between the functional object and its aspects. The auxiliary objects are separated into Input and Output objects (see Figure 14). Input Objects intercept intercept input messages to objects and send them to aspects. Output Objects intercept output messages from objects and send them to aspects. Aspects are separated into input and output aspects. Input aspects are activated when a method in the functional object is invoked such as the synchronization and replication aspects. Output aspects are activated when a functional object invokes a method, such as the distribution aspect.

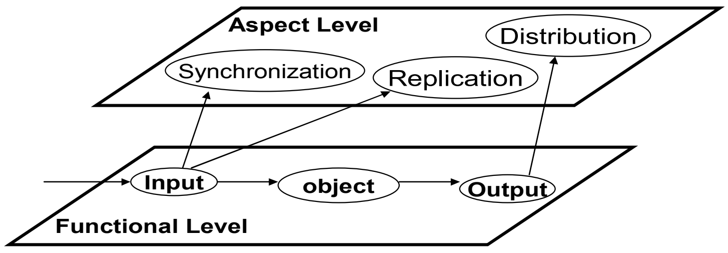


Figure 14. Structure of the Disguises Model taken from [Her03]

The distribution disguise performs the tasks depicted in Figure 15. The aspect publishes the functional object, looks in a reference table, and is in charge of sending remote messages using the platforms. The distribution aspect receives an output message of an object, and then it looks in the reference table in order to search for the receptor object of the message. When it has the remote reference of the receptor, it transforms the message in order to send it through a communication platform to the receptor.

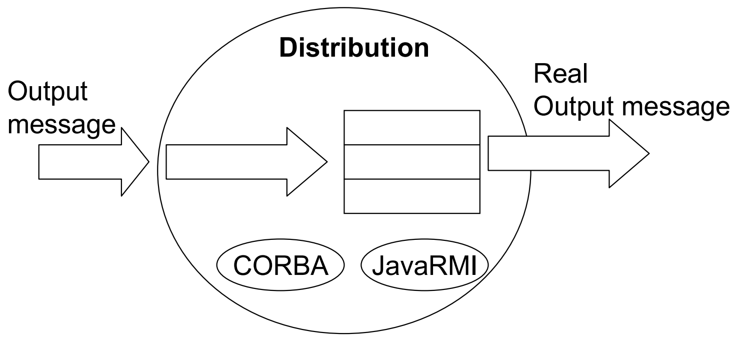


Figure 15. Structure of the distribution aspect taken from [Her03]

The replication disguise specifies either two techniques for replicating objects: the Active Replication and the Passive Replication. In the active replication all the replicas are equal and act in the same form. In the passive replication a main replica exists that is in charge to manage all the others.

The aspects of the disguises model are platform independent; however, the object code is in Java. Also, the languages of the disguises model are domain specific since each is for a concrete purpose. This makes it difficult to be used. However, the UML profile may facilitate its usability. On the other hand, the disguises model does include any notion for software mobility.

3.3.2.5 AWED

AWED [Ben06] is an aspect-oriented language that includes constructs for supporting the distribution of the aspect-oriented application. The language provides constructs for specifying remote pointcuts that match join points at remote hosts (similarly to DJCutter in section 3.3.1.3) and can determine where remote advices can be executed. Hosts in AWED can be grouped and a pointcut can match on a group of hosts. An aspect in AWED contains a set of fields as well as pointcuts and advices. An aspect construct determines whether it is dynamically deployed on all hosts or only on the local one. Also, AWED provides constructs for specifying how a distributed aspect shares its state with other aspects basing on the type of the aspect, its group or the host. AWED has been implemented by extending JAsCo [Suv03] and by using RMI.

Currently the AWED language does not provide any explicit notion for object or aspect mobility. Also, the AWED language specifies weaving rules inside aspects reducing aspect reusability.

3.4 Conclusions

In this chapter, AOSD concepts have been introduced. Also, this chapter shows that the development of distributed and mobile software systems can greatly benefit from the reusability, maintainability and comprehension that AOSD provides to their characteristics.

It can be concluded that at the implementation level of software development, the AOP languages and platforms have improved the maintenance and complexity of the code of distributed systems. Also, AOP has been applied for remote communication more than on mobility. This fact is observable because mobility needs a platform that provides dynamicity, including dynamic weaving.

It is important to take into account the design of distribution and mobile aspects in order to preserve the traceability of the software development process. Also, the code of distribution and mobility in most cases is repetitive. As a result, using automatic code generation tools would decrease the effort in developing distributed and mobile applications.

Table 3 Comparison of Aspect-Oriented Models that support Distribution

	Distribution	**Mobility**	**Weaving**	**Implementation**	**Graphical Support**
D	Explicitly in the language	No support	Static Weaving	Automatic code generation framework	No support
Composition Filters	No explicit support, but ACTs can be used	No support	Static Weaving	Implementations exists but are not automatic	No support
UMLAUT	Explicitly with a UML stereotype	No support	Static Weaving	Partial Implementation models	Supports with a UML stereotype
Disguises Model	Explicitly in language	No support	Dynamic Weaving	Automatic code generation framework	Support with a UML profile
AWED	Explicitly in language	No support	Dynamic Weaving	Implementation exits but not automatic	No support

Table 1 shows a comparison for the approaches that take into account distribution aspects at the design level. As it can be noticed, most of the approaches take into account distribution (remote access), however, none take into account mobility. As previously stated, dynamic weaving is important for distributed and mobile systems, however, only two of the approaches support it. It can also be noticed, that all of the approaches support implementations for their models however, only two of them really support automatic code generation. Also, in order to make a design approach more reusable, the graphical support is important. However, only two of them support a graphical notation. It can be noticed that the Disguises model mainly supports all the comparative features excluding mobility.

CHAPTER 4
PRISMA

4.1 Introduction

PRISMA is an approach that integrates AOSD and software architecture approaches in order to specify software architectures of complex systems. PRISMA is based on its models, metamodel [Per05b], Aspect-Oriented Architecture Description Language (AOADL) [Per06a], a methodology, and a CASE tool called PRISMA CASE. The metamodel defines the concepts and constraints needed for defining PRISMA architectural models. The AOADL provides the primitives needed to specify PRISMA software architectures. The PRISMA CASE tool allows the development of aspect-oriented software architectures following the PRISMA approach using graphical modeling tools, verification mechanisms, model compilers to automatically generate code and tools for executing the generated code.

Since this thesis presents an extension to PRISMA, the objective of this chapter is to provide the reader with a presentation to the PRISMA approach in order to permit the comprehension of the coming chapters of this thesis. The reader is referred to [Per06b] for a more detail description of PRISMA. In addition, this chapter illustrates the primitives that the AOADL provides by specifying the aspect-oriented software architecture of the Auction system case study previously presented without distribution and mobility characteristics.

This chapter is organized as follows: Section 4.2 presents an overview of the PRISMA model. The primitives that PRISMA provides are illustrated by the metamodel and the Aspect-Oriented Architecture Description Language (AOADL). Section 4.3 explains the methodology that PRISMA proposes for modelling aspect-oriented software architectures. Section 4.4 presents how the PRISMA CASE tool has been developed and how it supports the development of PRISMA software architectures. Finally, section 4.5 concludes the chapter.

4.2 PRISMA Model Overview

PRISMA [4] is a model that integrates Aspect-Oriented Software Development (AOSD) and Component Based Software Development (CBSD) in order to describe software architectures of complex software systems. In PRISMA, architectural elements (components and connectors) are specified by a set of aspects, which are first-class entities of software architectures. Aspects represent specific *concerns* (safety, coordination, etc) that crosscut the software architecture.

PRISMA uses a symmetrical aspect-oriented model [Har02] because it does not consider functionality as a kernel entity that is different from aspects, and it does not constrain aspects to specify non-functional requirements. A concern can be specified by several aspects of a software architecture, whereas a PRISMA aspect represents a concern that crosscuts the software architecture. This crosscutting is due to the fact that the same aspect can be imported by more than one architectural element of a software architecture. In this sense, aspects crosscut those elements of the architecture that import their behaviour (see Figure 16).

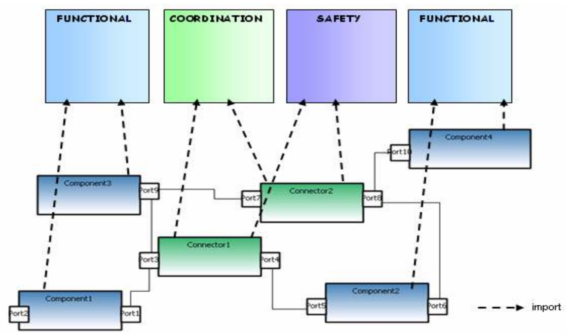


Figure 16. Crosscutting Concerns in PRISMA software architectures taken from [Per06b]

An aspect defines the state and behaviour of a specific *concern* of the software system. Examples of concerns are functionality, coordination, safety, and distribution among others. The state of an aspect at any given moment is determined by the value of its attributes. An aspect declares a number of interfaces and defines a behaviour for the services that these interfaces publish. This behaviour specifies whether or not services can be executed, when they are executed, how the execution of services changes the state of the aspect, and the order in which they can be executed. The behaviour of an aspect is defined by means of a protocol. The protocol describes how the different services are coordinated.

A PRISMA architectural element can be seen from two different views: the internal and the external. In the external view (Black box view), architectural elements encapsulate their functionality as black boxes and publish a set of services that they offer to other architectural elements (see Figure 17 A). These services are grouped into interfaces that are published through ports of architectural elements. As a result, ports are the interaction points of architectural elements.

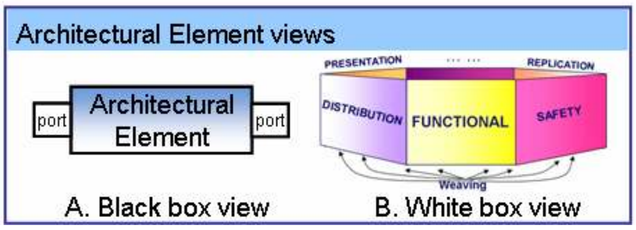


Figure 17.Views of a PRISMA architectural element taken from [Per06b]

The internal view (white box view) shows an architectural element as a prism. Each side of the prism is an aspect that the architectural element imports. In this way, architectural elements are represented as a set of aspects (see Figure 17 B) and the weaving relationships among them. Weavings allow the execution of an aspect service to trigger the execution of services in other aspects. A weaving is defined by means of operators that describe the order in which services are executed. From the AOSD point of view PRISMA weavings can be defined as follows: every service of an aspect is a join point, the services that trigger a weaving are the pointcuts, and the services that are executed as a consequence of weavings are the advices. In PRISMA, weavings are specified outside of aspects and inside of architectural elements in order to preserve the independence of the aspect specification from other aspects and weavings. As a result, aspects can be reused.

The white box and the black box views are connected by means of interfaces which are associated to ports and are used by aspects. Consequently, a request for a service that arrives to a port of an architectural element is processed by an aspect that uses the same interface that is used by this port.

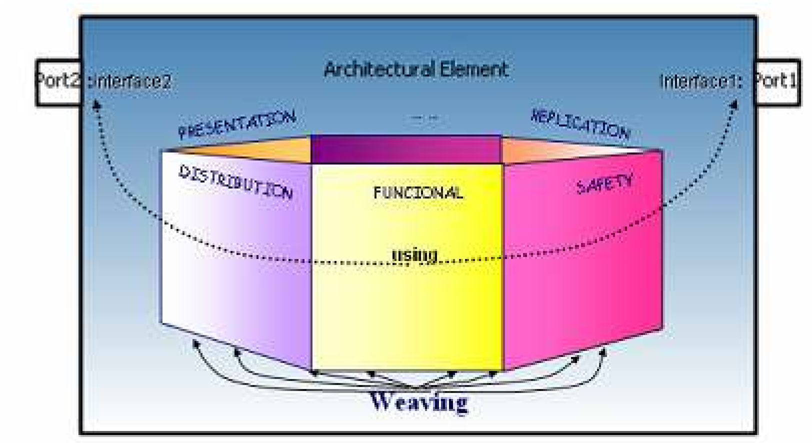


Figure 18. Communication between the white box and the black box views taken from [Per06b]

PRISMA has three kinds of architectural elements: components, connectors, and systems. Components and connectors are simple, but systems are complex components. A component is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements; whereas, a connector is an architectural element that acts as a coordinator among other architectural elements.

In software architectures, components are connected with connectors. As a result, 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. In Figure 16, the lines between component ports and connector ports are attachments.

PRISMA components can be simple or complex. The complex ones are called systems. A PRISMA system is a component that includes a set of architectural elements (connectors, components and other systems) that are correctly attached. In addition, a system can have its own aspects and weavings as components and connectors. Since a system is composed by other architectural elements, the composition relationships among them must be defined. These composition relationships are called bindings. Bindings establish the connection among the ports of the complex component (the system) and the ports of the architectural elements that a system contains (see Figure 19).

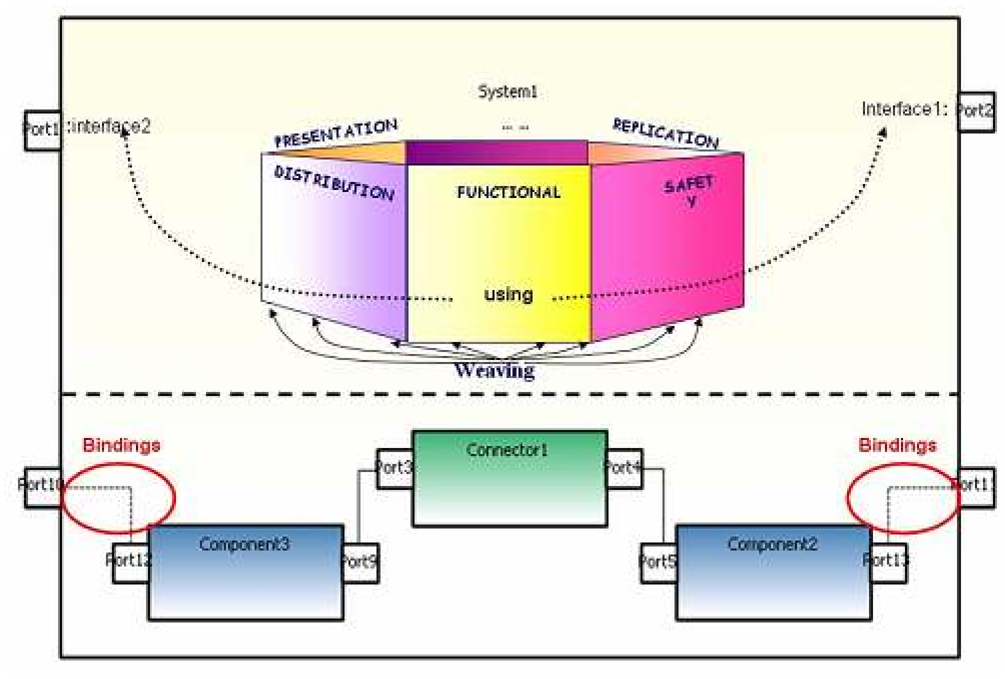


Figure 19. Systems taken from [Per06b]

PRISMA provides a metamodel in order to define properties of PRISMA models in a precise way [Per05b]. The PRISMA metamodel is defined using the UML version 1.5 class diagram and the constraints are specified using OCL [UML07]. PRISMA also provides an AOADL [Per06a] that allows the description of architectural models based on its metamodel.

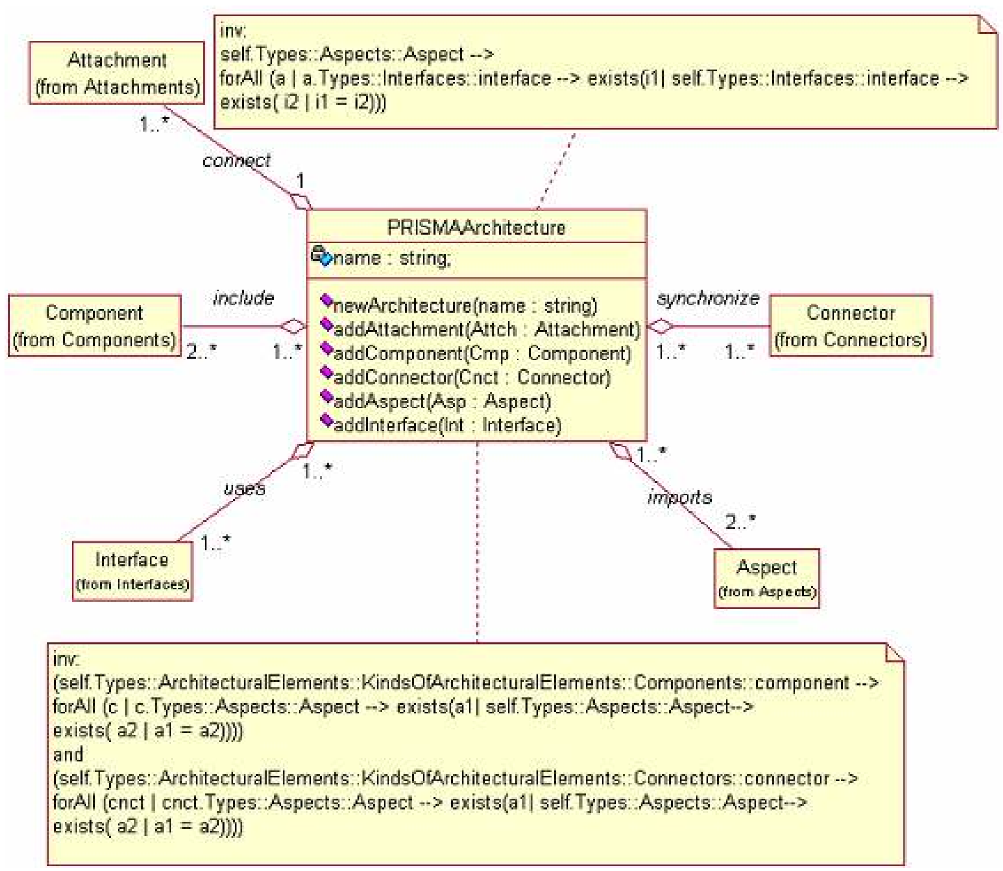
The PRISMA AOADL is an extension of the OASIS language [Let98]. OASIS is a formal language that defines conceptual models. PRISMA AOADL uses some of OASIS syntactical constructions which are based on Modal Logic of Actions [Har84] in order to specify state and a dialect of the Polyadic Pi-Calculus [Mil93] to define its behaviour. The grammar of the AOADL is presented in APPENDIX A.

In the following, the concepts of PRISMA are explained using the metamodel and the AOADL.

4.2.1 Architectural Model

An architectural model defines a software architecture from the first-class entities of the type definition level. The first-class entities are components, connectors, aspects, interfaces, and attachments. An architectural model in the metamodel is represented by the *PRISMAArchitecture* metaclass (see Figure 20). The *PRISMAArchitecture* metaclass has five aggregation relationships with each one of the classes that represent the first-class entities of the PRISMA model. Since components, connectors, interfaces, and aspects are reusable, they can be used by more than one architectural model.

The *PRISMAArchitecture* metaclass has an attribute called name and six methods: one for creating a new software architecture and the other five for creating the first-class entities of a software architecture.



```
inv:
self.Types::Aspects::Aspect -->
forAll (a | a.Types::Interfaces::interface --> exists(i1| self.Types::Interfaces::interface -->
exists( i2 | i1 = i2)))
```

```
inv:
(self.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::component -->
forAll (c | c.Types::Aspects::Aspect --> exists(a1| self.Types::Aspects::Aspect-->
exists( a2 | a1 = a2))))
and
(self.Types::ArchitecturalElements::KindsOfArchitecturalElements::Connectors::connector -->
forAll (cnct | cnct.Types::Aspects::Aspect --> exists(a1| self.Types::Aspects::Aspect-->
exists( a2 | a1 = a2))))
```

Figure 20. The package *ArchitectureSpecification* of the PRISMA metamodel taken from [Per06b]

Figure 21 shows the syntax of the architectural model in the AOADL. The specification of an architectural model starts with the reserved word *Architectural_Model* and ends with the reserved word *End_Architectural_Model*. Then, a name is given to the architectural model. In the case of the architectural model of the case study it is called *AuctionAgents*. Afterwards, each first-class entity is specified. Systems (complex components) can be also specified in the architectural model.

```
Architectural_Model AuctionAgents
<interface_block>
<aspect_block>
<component_block>
<connector_block>
<attachments_block>
[<system_block>]
End_Architectural_Model AuctionAgents;
```

Figure 21. Syntax of the architectural model in the AOADL

In the next sections each one of the blocks of the specification are explained.

4.2.2 Interfaces

An interface publishes a set of services. It describes the signature of the services that can be invoked or requested through that interface (see Figure 22). The signature of a service specifies its name and parameters (see Figure 23). Parameters are declared by specifying their *kind* (input/output), name and data type.

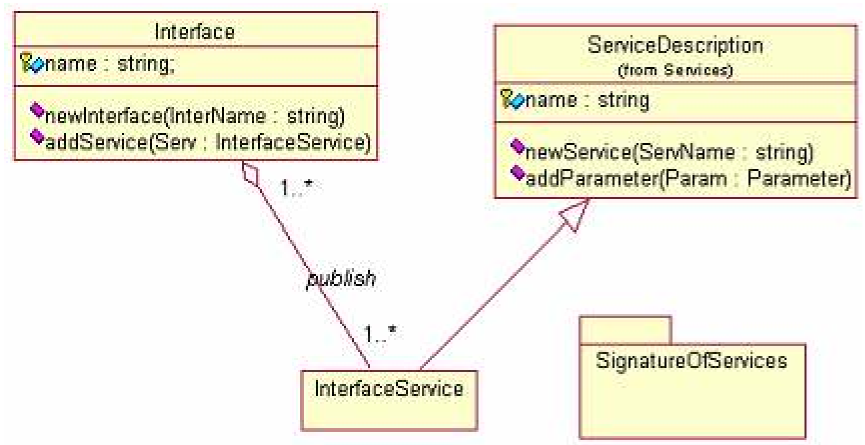

Interface
name : string;
newInterface(InterName : string)
addService(Serv : InterfaceService)
ServiceDescription
(from Services)
name : string
newService(ServName : string)
addParameter(Param : Parameter)
1..*
publish
1..*
InterfaceService
SignatureOfServices

Figure 22. The package *Interfaces* of the PRISMA metamodel taken from [Per06b]

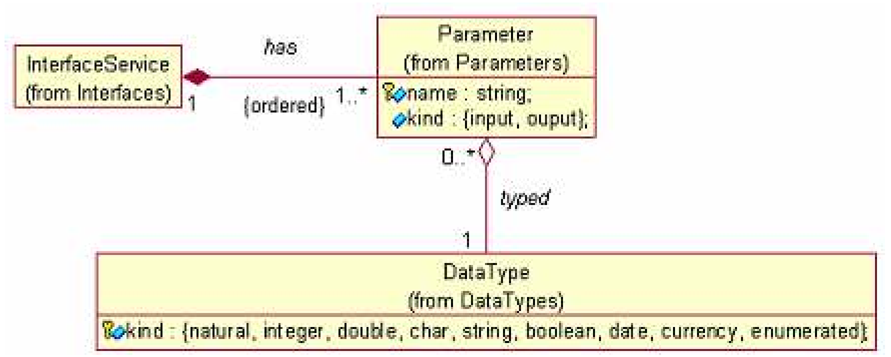


Figure 23. The package *SignatureOfService* of the PRISMA metamodel [Per06b]

An example of an interface in the Auction case study is the interface *ICustProc,* which publishes the service *notifyProdInterest* (see Figure 24). The *notifyProdInterest* service has four input parameters and an output parameter. Input parameters are those that are required for executing the service; whereas output parameters are those that are generated by the execution of the service. For example, the *notifyProdInterest* service is used by the *Procurement* component and the *Customer* component. The *Procurement* component sends to the *Customer* component the information of a product: a saleroom (*Saleroom*), a sale number (*SaleNum*), a date of an auction (*DateOfAuction*) and a lot description (*Lotdescrip*) and the Customer returns if it is interested or not (*Interested*).

```
Interface ICustProc
 notifyProdInterest(input Saleroom:string, input SaleNum,
                    input DateOfAuction:string,
                    input Lotdescrip: string,
                    output Interested:bool);

End_Interface ICustProc
```

Figure 24. Specification of the interface *IProcurAuction*

4.2.3 Aspects

An aspect defines the behaviour of a specific concern of the software system which can be functional, coordination, safety, etc. An aspect specification consists of several sections each represented in the metamodel as a metaclass (see Figure 25).

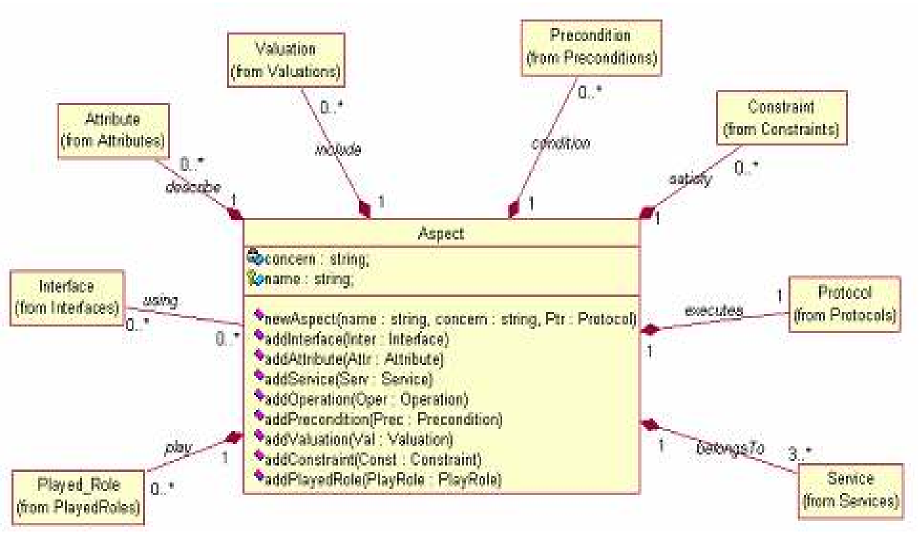

Valuation (from Valuations)
Precondition (from Preconditions)
Attribute (from Attributes)
Constraint (from Constraints)
include
condition
describe
satisfy
Aspect
concern : string;
name : string;
newAspect(name : string, concern : string, Ptr : Protocol)
addInterface(Inter : Interface)
addAttribute(Attr : Attribute)
addService(Serv : Service)
addOperation(Oper : Operation)
addPrecondition(Prec : Precondition)
addValuation(Val : Valuation)
addConstraint(Const : Constraint)
addPlayedRole(PlayRole : PlayRole)
Interface (from Interfaces)
using
Protocol (from Protocols)
executes
play
Played_Role (from PlayedRoles)
belongsTo
Service (from Services)
0..*
1
3..*

Figure 25. The metaclass Aspect of the package Aspects of the PRISMA metamodel taken from [Per06b]

Each aspect specifies the kind of concern that it defines, its name, and the interfaces it uses to specify its behaviour. Figure 26 shows the header of the aspect called *ProcurFunct* in the AOADL. *ProcurFunct* aspect specifies a concern of kind functional, and it uses the interfaces *IProcurAuction* and *ICustProc*.

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
   … …
End_Aspect ProcurFunct;
```

Figure 26. Specification of the header of an aspect with interfaces (*ProcurFunct*)

4.2.3.1 Attributes

Attributes store information about the characteristics of an aspect. Each attribute has a name and a data type. The data type defines the kind of values that the attribute can store. There are three kinds of attributes:

- *Constant:* The stored values cannot change
- V*ariable:* The stored values can be changed
- *Derived:* The value is calculated on demand applying its derivation rule.

Constant and variable attributes can specify that they must always store a value by specifying the reserved word *NOT NULL* after their data type specification. In addition, they can store a value by default, which can be only modified when the attribute is variable.

PRISMA proposes that the first letter of an attribute is written in small letter in order to clearly distinguish attributes from parameters, which start in capital letter.

Figure 27 shows the specification of the *ProcurFunct* aspect attributes. In the AOADL, the attributes section is preceded by the reserved word *Attributes.* The *ProcurFunct* aspect specifies the variable attributes *keywords, saleroom,* and *saleNum* whose data type is string, the variable attributes *limitDate* and *dateOfAuction* whose data type is Date and the variable attributes *keepSearching* and *finishedSearching* whose data type is boolean. The attributes *keywords* and *limitDate* cannot have an empty value because they are *NOT NULL.*

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
 Attributes
  Variables
    keywords: string NOT NULL;
    limitDate: Date NOT NULL;
    saleroom: string;
    saleNum: string;
    dateOfAuction: Date;
    keepSearching: boolean;
    finishedSearching: boolean;
  ……
End_Aspect ProcurFunct;
```

Figure 27. Specification of variable attributes of the aspect *ProcurFunct* aspect

4.2.3.2 Services

Services specify the behaviour of an aspect. An aspect defines public and private services. Public services define behaviour of services published in interfaces which an aspect uses. Private services define internal behaviour that an aspect needs.

Each service has a name and can have a set of parameters (as described in section 4.2.2) whose type can be *input* or *output*. A service can be *in*, *out* and *in/out*. An *in* service has a server behaviour and receives results. An *out* service has a client behaviour and sends results. An *in/out* service has both a client and a server behaviour.

Every aspect must specify the *begin* service, the *end* service and the interface services that the aspect defines. The *begin* service executes when an aspect starts its execution and the *end* service is executed when an aspect stops. The services *begin* and *end* can only be requested from the creation and destruction services of the architectural elements that import the aspect because aspects can only be instantiated in a context of an architectural element.

For example, Figure 28 shows a segment of the services section of the *ProcurFunct* aspect. It can be observed that the *begin* service is specified with parameters. It can be observed that these parameters are specified for initializing the attributes that need a value at instantiation time, i.e. the attributes that are *NOT NULL* (see Figure 27). The *end* service is also specified. Since the aspect uses the *IProcurAuction* interface and the *ICustProc* interface (see Figure 24), their services *searchforlot* and *notifyProdInterest* have to be specified. For example, *searchforlot* service is indicated with an *in/out* meaning that the aspect requests the service (*out*) and the aspect receives the results of the service execution (*in*).

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
  Services
   begin(input Keywords: string, LimitDate: Date);
   ……
  finishedSearchingWithoutResults();
   in/out searchforlot(input keywords:string, output Saleroom:string,
                       output SaleNum, output DateOfAuction:string,
                       output Lotdescrip:string);
    in/out notifyProdInterest(input Saleroom:string,input SaleNum:string,
                              input DateOfAuction:Date,
                              input Lotdescrip:string,
                              output Interested: boolean);
     … …
    end();

End_Aspect ProcurFunct;
```

Figure 28. Specification of some services of *ProcurFunct aspect*

4.2.3.3 Valuations

Valuations define the change of state of an aspect when one of its services is executed. A service can have one or more valuations associated to it. The specification of a valuation consists of three sections: condition, service and postcondition. A condition is validated in the aspect state before its execution, and its specification is optional (meaning “true” when missing).

The meaning of a valuation depends on whether a service is *in* or *out* and its kind of parameters. In the following, the meaning of the postcondition of a valuation is explained:

➢ A service without output parameters
   - ***in***: The service is provided and executed by the aspect.

- ***Valuation in***: The postcondition must be satisfied after the service execution

  An example of this case is the *changeMaximumBid* service of the *BidderFunct* aspect (see Figure 29). *changeMaximumBid* service is executed by *BidderFunct* aspect when the Customer requests to change the maximum bid of a product. *changeMaximumBid* is of kind *in* and the valuation assigns the *NewMaximumBid* input parameter to *lotMaximumBid* attribute.

  .

```
Functional Aspect BidderFunct using ICustBidder, IBidderAuct
 Services
……
  in changeMaximumBid(input NewMaximumBid:double)
     Valuations
         [in changeMaximumBid(input NewMaximumBid)]
           lotMaximumBid:= NewMaximumBid;
…….

End_Aspect BidderFunct;
```

Figure 29. Specification of a valuation of in `changeMaximumBid` of the *BidderFunct aspect*

- ***out***: The service is requested by the aspect.
  - ***Valuation out***: The postcondition must be satisfied after the service request.
- ***in/out***: The service is provided and executed by the aspect, and the service is also requested by the aspect. In this case, two kinds of valuations can be defined:
  - ***Valuation in***: The postcondition must be satisfied after the service execution.
  - ***Valuation out***: The postcondition must be satisfied after the service invocation.

➢ A service with output parameters: A service with output parameters always has an in/out behaviour. The meaning of the valuation varies depending if the service is provided or requested as presented below.

- The service is provided and executed by the aspect, and the aspect sends the results of the service execution
    - ***Valuation in**:* The postcondition must be satisfied after the service execution. The output parameters of the service cannot be used as a right term of the valuation in any case, neither in the condition nor in the postcondition, due to the fact that the service has not been executed and its output parameters do not have value. In fact, output parameters are usually used as a left term of the postcondition in order to satisfy the condition that requires output parameters to have a value after the service execution. must be satisfied after the service execution.

Figure 30 shows the specification of the *searchforlot* service of the *AuctFunct* aspect. The *searchforlot* service is provided and executed by the *AuctFunct* aspect, and the aspect sends the results of the service execution. In this case, a condition checks whether the *Keywords* input parameter matches with the value stored in the *lotDescrip* attribute of the *AuctFunct* aspect. If the condition is satisfied, the postcondition of the valuation assigns the values of the output parameters with values stored in attributes of the *AuctFunct* aspect.

```
Functional Aspect AuctFunct using IProcurAuction, IBidderAuct, ICustAuct
 Services
 … ….
   in/out searchforlot(input Keywords:string, output Saleroom:string,
                    output SaleNum:string, output DateOfAuction:string,
                    output Lotdescrip:string)
      Valuations
        {Keywords==lotDescrip}[in searchforlot(input keywords,
                                             output SaleRoom,
                                             output SaleNum,
                                             output DateOfAuction,
                                             output Lotdescrip)]
          Saleroom := saleroom1, SaleNum:= saleNum1,
          DateOfAuction:= dateOfAuction1, Lotdescrip:=lotdescrip;
    … …
   end();
End_Functional Aspect AuctFunct;
```

Figure 30. Specification of a valuation of in/out `searchforlot` of the *AuctFunct* aspect

- o ***Valuation out***: The postcondition must be satisfied after sending the result. The semantics of the out valuation is conditioned by the semantics of the in valuation. If the in obtains a result, the out is related to sending the result. Since the service has already been executed and the output parameters have value, they can be used in the condition and in the right terms of the postcondition.

- The service is requested by the aspect and the aspect receives the result of the service.

- o ***Valuation in***: The postcondition must be satisfied after the reception of the service result. Since the service has already been executed and the output parameters have value, they can be used in the condition and in the right terms of postcondition.

Figure 31 shows the specification of *searchforlot* service which is first requested by the *ProcurFunct* aspect (*out*) and then the aspect receives the result of the service (*in*). The valuation specifies how the state of *ProcurFunct* aspect changes when the aspect receives the result (the output parameters). Since the service has already been executed and the output parameters have value, they can be used in the condition and in the right terms of the postcondition. In this case, a condition checks whether the result received in *DateOfAuction* is before the date stored in the *limitDate* attribute (see Figure 27) of the *ProcurFunct* aspect. If the *DateOfAuction* is before the date stored in the *limitDate* attribute, the postcondition of the valuation assigns the results returned in the output parameters to the values of the attributes of the *ProcurFunct* aspect.

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
 Services
 … ….
   in/out searchforlot(input Keywords:string, output Saleroom:string,
                        output SaleNum:string, output DateOfAuction:string,
                        output Lotdescrip:string)
      Valuations
       {DateOfAuction<= limitDate}[in searchforlot(input keywords,
                                                   output SaleRoom,
                                                   output SaleNum,
                                                   output DateOfAuction,
                                                   output Lotdescrip)]
          saleroom := Saleroom, saleNum:= SaleNum,
          dateOfAuction:= DateOfauction, lotdescrip:=Lotdescrip;

    … …
   end();

End_Aspect ProcurFunct;
```

Figure 31. Specification of a valuation of in/out `searchforlot` of the *ProcurFunct aspect*

- *Valuation out:* The postcondition defines the state of the aspect after the service request. The output parameters of the service cannot be used as a right term of the valuation in any case, neither in the condition nor in the postcondition due to the fact that the service has not been executed and its output parameters do not have value.

4.2.3.4 Preconditions

Preconditions establish the conditions that have to be satisfied in order to execute an aspect service. In the AOADL, a precondition is specified by indicating the service and the condition for its execution separated by the reserved word *if*. An example of a precondition is in the *BidderFunct* aspect (see Figure 32). The precondition indicates that the *bid* service cannot be executed unless the stop attribute is not false. This indicates that the bidder cannot bid if the customer does not want it to bid.

```
Functional Aspect BidderFunct using ICustBidder, IBidderAuct
… …
   Preconditions
    bid(input currentBiddingAmount, output Situation) if (stop = false);
… …
End_Aspect BidderFunct;
```

Figure 32. Specification of a precondition in the *BidderFunct* aspect

4.2.3.5 Constraints

Constraints condition the value of attributes of an aspect. Constraints have to be satisfied each throughout the entire execution process of an aspect. As a result, each time that a service execution is finished, the value of each attribute must satisfy the aspect constraints.

A constraint specification consists of defining a static or a dynamic condition. Static constraints make reference to one state of the aspect whereas dynamic constraints make reference to several states of the aspect. If the condition is dynamic, it uses one of the temporal operators: *always, never, since, until,* and their possible combinations.

An example of a possible constraint can be in the *ProcurFunct* aspect. A constraint for the *dateOfAuction* attribute can be specified (see Figure 33). The constraint specifies that the value stored in the *dateOfAuction* attribute is always less than the value stored in the *limitDate* attribute.

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
 Attributes
  Variables
    ……
    dateOfAuction: Date;
 ……
 Constraints
    ……
    always {dateOfAuction<= limitDate}
  ……
End_Aspect ProcurFunct;
```

Figure 33. Specification of a constraint in the *ProcurFunct aspect*

4.2.3.6 Transactions

Transactions are complex services that are composed of other services. The services that compose a transaction are atomically executed (all or none). As a result, if the execution of a transaction service fails, the services that have been already executed are roll backed.

An example of a transaction is the transaction *NOTIFYPRODINTEREST* of the *CustFunct* aspect (see Figure 34). This transaction receives the information of a product that is going to be auctioned in the input parameters. The transaction is composed of the *setInterested* service and the *saveItem* service. First the transaction invokes the *setInterested* service (indicated with "?"). This service is needed to allow the customer to introduce if he is interested or not in the product. Then, the transaction invokes the *saveItem* service. The *saveItem* service has two valuations. In one valuation, the postcondition is satisfied if the customer is interested in a product and as a result, the product information is saved. In the other valuation, the postcondition does not save any values for the product information when the customer is not interested in the product. The *NOTIFYPRODINTEREST* transaction also has a valuation associated to it. The valuation assigns a value to the *Interested* output parameter.

```
Functional Aspect CustFunct using ICustProc, ICustBidder

   Services      … …

    setInterested(input CustomerInterest:boolean)
      Valuations
        [setInterested(input CustomerInterest)]
                                     interested:=CustomerInterested;

    saveItem(SaleRoom, SaleNum,DateOfAuction,Lotdescrip)
       Valuations
         {interested==true}[saveItem(SaleRoom, SaleNum, DateOfAuction,
                                  LotNumber)]
                     iSaleRoom:=saleroom, iSaleNum:=SaleNum,
                     iDateOfAuction:=DateAuction,
                     iLotdescrip:=Lotdescrip;
         {interested==false}[saveItem(SaleRoom, SaleNum, DateOfAuction,
                                    LotNumber)]
                     iSaleRoom:=NULL, iSaleNum:=NULL,
                     iDateOfAuction:=NULL,
                     iLotdescrip:=NULL;


  Transactions
   in/out NOTIFYPRODINTEREST(input Saleroom:string, input SaleNum:string,
                             input DateOfAuction:string,
                             input Lotdescrip: string,
                             output Interested:boolean);

     notifyProdInterest = setInterested?(CustomerInterested)→ SAVEITEM;
     SAVEITEM = saveItem?(SaleRoom, SaleNum, DateOfAuction,LotNumber);
         Valuations
           [NOTIFYPRODINTEREST(SaleRoom, SaleNum,DateOfAuction,Lotdescrip,
Interested)]
           Interested:=interested;

End_Aspect CustFunct;
```

Figure 34. Specification of a transaction in the *CustFunct aspect*

4.2.3.7 Played_Roles

Played_Roles define the roles that an aspect can play. A played role establishes how and when the services of an interface can be required or provided. As a result, played_roles are used for specifying how public services execute and private services are not specified in played_roles.

In Figure 35, the played_roles of the *ProcurFunct* aspect are specified. *ProcurFunct* aspect has two played_roles: *CUSTPROC* and *PROCURAUCT*. *CUSTPROC* is a played_role that defines how the service of the interface *ICustProc* executes. The name of the interface that the played_role defines is preceded after the reserved word *for*. A process is then specified using a dialect of π-calculus. The process of *CUSTPROC* specifies that the service *notifyProdInterest* has a client behaviour (indicated with "!") and after the service must have a server behaviour (indicated with "?"). *PROCURAUCT* is a played_role that defines how the *searchforlot* service of the interface *IProcurAuction* executes.

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc

 CUSTPROC for ICustProc ::= notifyProdInterest!(Saleroom, SaleNum,
                                               DateOfAuction, Lotdescrip,
                                               Interested)
                            →
                            notifyProdInterest?( Saleroom, SaleNum,
                                               DateOfAuction, Lotdescrip,
                                               Interested);
 PROCURAUCT for IProcurAuction ::= searchforlot!(Keywords, SaleRoom,
                                                   SaleNum, DateOfAuction,
                                                   Lotdescrip)
                                   →
                                   searchforlot?(Keywords, SaleRoom,
                                                   SaleNum, DateOfAuction,
                                                   Lotdescrip);

End_Aspect ProcurFunct;
```

Figure 35. Specification of played_roles of *ProcurFunct aspect*

4.2.3.8 Protocols

Protocols glue the set of services of an aspect. The services of an aspect can be private services and services of the different played_roles. As a result, a protocol defines a process that coordinates the private and public services of an aspect. The protocols section has to be included in an aspect definition.

Figure 36 shows the specification of the protocols of the *ProcurFunct* aspect. The protocols glue the begin service, the end service, the private services of the *ProcurFunct* aspect: *setKeepSearchingToTrue*, and *finishedSearchingWithoutResult*, and the services of the played_roles: *PROCURACUCT* and *CUSTPROC*. The protocol specifies that when the *begin* service is executed, the *setKeepSearchingToTrue* service, the end service, or the *PROCURFUNCT2* process is executed (indicated with +). The *PROCURFUNCT2* process specifies that the *searchforlot* service of the played_role *PROCURACUCT* has to be requested and then received, then the aspect can keep requesting and receiving the *searchforlot* service, execute the *PROCURFUNCT3* process, or execute the *finishedSearchingWithoutResult*. This is needed in order to specify that once the *ProcurFunct* aspect searches for a product: it can keep searching for another product, it can notify the customer about the found product or it can finish searching.

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc

Protocol

  PROCURFUNCT:= begin(Keywords, LimitDate)→ PROCURFUNCT1;
  PROCURFUNCT1:= setKeepSearchingToTrue() + end()+ PROCURFUNCT2;
  PROCURFUNCT2:= PROCURACUCT_searchforlot!(Keywords, SaleRoom,
                                           SaleNum, DateOfAuction,
                                           Lotdescrip)
                 →
                 PROCURACUCT_searchforlot?(Keywords, SaleRoom,
                                           SaleNum, DateOfAuction,
                                           Lotdescrip)
                 →
                 (PROCURFUNCT2 + PROCURFUNCT3
                 + finishedSearchingWithoutResult?()) ;
  PROCURFUNCT3:= CUSTPROC_notifyProdInterest!(Saleroom, SaleNum,
                                    DateOfAuction, Lotdescrip,
                                    Interested)
                  →
                 CUSTPROC_notifyProdInterest?(Saleroom, SaleNum,
                                    DateOfAuction, Lotdescrip,
                                    Interested)
                  →
                 (PROCURFUNCT1 + PROCURFUNCT2);

End_Functional Aspect ProcurFunct
```

Figure 36. Specification of protocols of *ProcurFunct aspect*

4.2.4 Simple Architectural Elements: Components and Connectors

An architectural element is specified by its set of ports, the aspects that form it, and the aspect weavings. An architectural element is represented in the PRISMA metamodel by the *ArchitecturalElement* metaclass (see Figure 37). *ArchitecturalElement* metaclass is related with Port, *Aspect* and *Weaving* metaclasses. The *Port* and *Weaving* metaclasses have a composition relationship with the *ArchitecturalElement* metaclass. This is due to the fact that ports and weavings are not reused by many architectural elements, i.e., each architectural element has its own ports and weavings. The *Aspect* metaclass has an association relationship with the *ArchitecturalElement* metaclass because an aspect can be reused by many architectural elements.

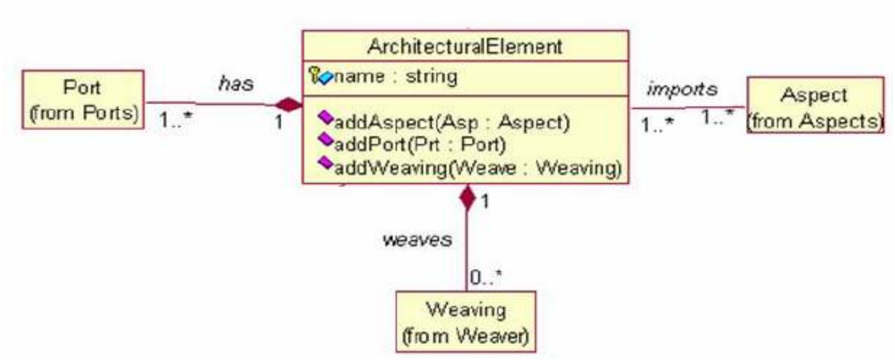


Figure 37. The package *ArchitecturalElements* of the PRISMA metamodel taken from [Per06b]

Ports of an architectural element represent interaction points that an architectural element has with other architectural elements. Each port publishes public services of an aspect played_role. An architectural element has at least one port. Each port is specified by defining its name, the interface that it publishes, and the played_role that it associates to the interface.

A weaving specification defines how the execution of a service of an aspect can trigger the execution of a service of another aspect. An aspect weaving is specified by determining the aspects that participate in the weaving, the services of the aspects where they are weaved, and the weaving operators.

The AOADL allows specifying a weaving between service s1 of aspect A1 and service s2 of aspect A2 using one of the following weaving operators:

- A2.s2 **after** A1.s1: A2.s2 is executed after A1.s1
- A2.s2 **before** A1.s1: A2.s2 is executed before A1.s1
- A2.s2 **instead** A1.s1: A2.s2 is executed in place of A1.s1
- A2.s2 **afterif (Boolean condition)** A1.s1: A2.s2 is executed after A1.s1 if the condition is satisfied.
- A2.s2 **beforeif (Boolean condition)** A1.s1: if the condition is satisfied, A2.s2 is executed followed by A1.s1; otherwise, only A2.s2 is executed.
- A2.s2 **insteadif (Boolean condition)** A1.s1: A2.s2 is executed in place of A1.s1 if the condition is satisfied.

As it can be noticed from section 4.2.3 an aspect definition does not include any references to other aspects. This independence of the aspect specification from other aspects and weavings allows aspects to be reusable. In addition, the fact that the specification of weavings is inside architectural elements provides the flexibility of specifying different behaviours of an architectural element by importing the same aspects and defining different weavings. Therefore, architectural elements definitions import aspects and define the needed weavings.

A simple architectural element in PRISMA can be a component and a connector. A component is an architectural element that captures a given functionality of a software system. A connector is an architectural element that acts as a coordinator between other architectural elements. The difference between a component and a connector is that a connector must import a coordination aspect and a component cannot. As a result, a *Component* metaclass and a *Connector* metaclass inherit the properties of the *ArchitecturalElement* metaclass in the metamodel (see Figure 38). For this reason, both components and connectors are specified by aspects, ports and weavings in the AOADL.

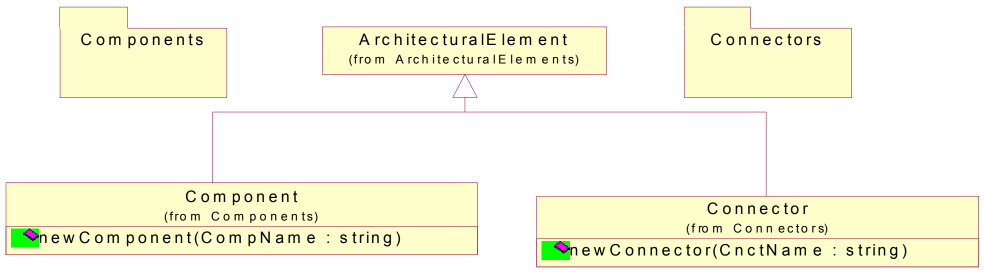
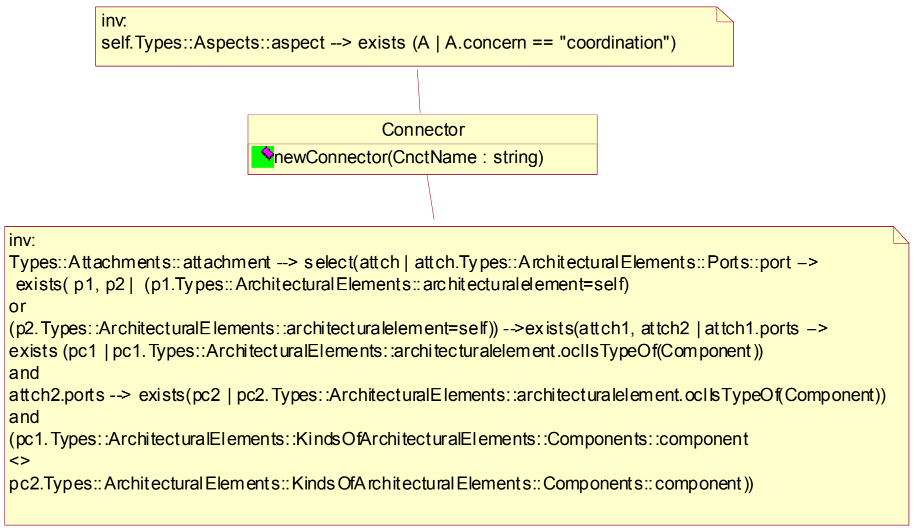


Figure 38. The package *KindsOfArchitecturalElements* of the PRISMA metamodel taken from [Per06b]

The Connectors package of Figure 38 is shown in detail in Figure 39. The *Connector* metaclass has an associated constraint that specifies that a connector must import an aspect whose concern is coordination. Moreover, the *Connector* metaclass has another constraint associated to it that specifies that a connector must have at least two attachments (see section 4.2.5) associated to it, and each attachment must connect the connector to two different components.



```
inv:
self.Types::Aspects::aspect --> exists (A | A.concern == "coordination")
```

```
inv:
Types::Attachments::attachment --> select(attch | attch.Types::ArchitecturalElements::Ports::port –>
 exists( p1, p2 |  (p1.Types::ArchitecturalElements::architecturalelement=self)
or
(p2.Types::ArchitecturalElements::architecturalelement=self)) -->exists(attch1, attch2 | attch1.ports –>
exists (pc1 | pc1.Types::ArchitecturalElements::architecturalelement.oclIsTypeOf(Component))
and
attch2.ports --> exists(pc2 | pc2.Types::ArchitecturalElements::architecturalelement.oclIsTypeOf(Component))
and
(pc1.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::component
<>
pc2.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::component))
```

Figure 39. The package *Connectors* of the PRISMA metamodel taken from [Per06b]

An example of an architectural element specification is the *Customer* component specification (see Figure 40). A component specification starts with the reserved word

Component and ends with the reserved word *End_Component*. The specification consists of a name (*Customer*), the importation of needed aspects (*CustDist*, *CustFunct*), the weavings among the aspects (a weaving between the *move* service of the *CustDist* aspect and the *biddingInf* service of the *CustFunct* aspect), a set of ports: *MOVEProcPort*, *MOVEBidderPort*, *CUSTPROCPort*, *CUSTBIDDERPORT*, and CUSTAUCTPort and the two sections that allow the creation and destruction of the architectural element.

```
Component Customer

 Import Distribution Aspect CustDist;
 Import Functional Aspect CustFunct;
 Weavings
  CustDist.move(NewAmbient) after CustFunct.biddingInf(iSaleRoom,
                                                        iSaleNum,
                                                        iDateofAuction,
                                                        iLotNumber,
                                                        maximumBid);
 End_Weavings

 Ports
     MOVEProcPort: IMobility Played_Role CustDist.MOVEProc;
     MOVEBidderPort: IMobility Played_Role CustDist.MOVEBidder;

     CUSTPROCPort: ICustProc Played_Role CustFunct.CUSTPROC;
     CUSTBIDDERPORT: ICustBidder Played_Role CustFunct.CUSTBIDDER
     CUSTAUCTPort: ICustAuct Played_Role CustFunct.CUSTAUCT;
 End_Ports
   new(input ParentAmbient: string)
       {
        CustDist. begin(ParentAmbient);
        CustFunct.begin();
        }
        {
             CustDist.end();
             CustFunct.end();
         }


End Component Customer;
```

Figure 40. Specification of the *Customer* component type

Each port is specified by defining its name, the interface that it publishes and the played_role that it associates to the interface. For example in Figure 40, the

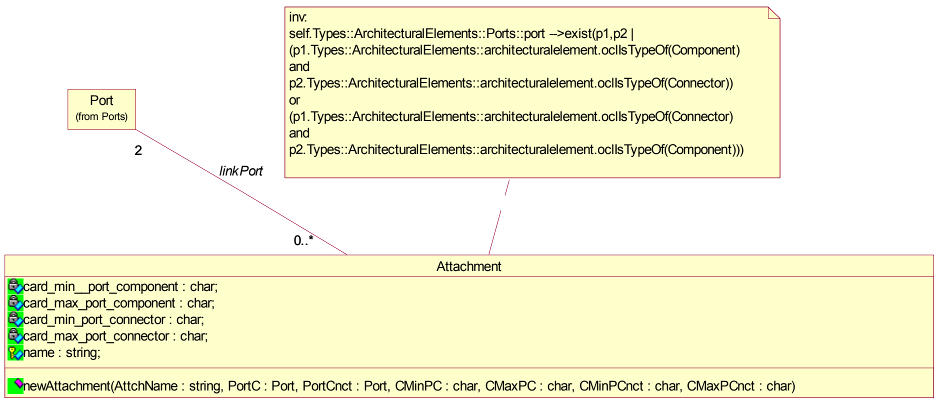
MOVEProcPort port publishes the interface called *IMobility* and the played_role *MOVEProc* of the *CustDist* aspect.

The constructor section is preceded by the reserved word *new* and the needed list of parameters. Next, the invocations of the begin services of the imported aspects are specified in curly brackets. On the other hand, the destruction section is preceded by the reserved word *destroy()*. Next, the invocations of the end services of the imported aspects are specified in curly brackets.

A connector architectural element is specified in the same way as a component. The only difference between a component and a connector specification is that the connector type is specified starting with the reserved word *Connector* and ending with the reserved word *End_Connector* instead of starting with *Component* and ending with *End_Component*.

4.2.5 Attachments

An attachment establishes a connection between a port of a component and a port of a connector. In the PRISMA metamodel, the *Attachment* metaclass is associated with the Port metaclass (see Figure 41). A constraint is specified in order to restrict that an attachment can only connect a component port and a connector port. Also, the *Attachment* metaclass has four properties in order to define the maximum and minimum cardinalities of an attachment. These properties constrain the instances of an attachment connected to a port of a component instance or a connector instance.



```
inv:
self.Types::ArchitecturalElements::Ports::port ->exist(p1,p2 |
(p1.Types::ArchitecturalElements::architecturalelement.oclIsTypeOf(Component)
and
p2.Types::ArchitecturalElements::architecturalelement.oclIsTypeOf(Connector))
or
(p1.Types::ArchitecturalElements::architecturalelement.oclIsTypeOf(Connector)
and
p2.Types::ArchitecturalElements::architecturalelement.oclIsTypeOf(Component)))
```

```
card_min__port_component : char;
card_max_port_component : char;
card_min_port_connector : char;
card_max_port_connector : char;
name : string;
newAttachment(AttchName : string, PortC : Port, PortCnct : Port, CMinPC : char, CMaxPC : char, CMinPCnct : char, CMaxPCnct : char)
```

Figure 41. The package *Attachments* of the PRISMA metamodel taken from [Per06b]

Figure 42 specifies attachments of the *AuctionAgents* architectural model. A type of attachment called *AttchAuctCnct* is specified to connect the *CnctAuctPortBidder* port of *AuctionCnct* connector and the *BidderAuctPort* port of the *AuctionHouse* component. In addition, the minimum and maximum cardinalities of the attachment are specified. The *AuctionCnct.CnctAuctPortBidder(1,1)* means that only one instance of the *AttchAuctCnct* can be connected to the *CnctAuctPortBidder* port of a *AuctionCnct* instance. The *Auction.BidderAuctPort(1,1)* means that only one instance of the *AttchAuctCnct* can be connected to the *BidderAuctPort* port of an *AuctionHouse* instance. *AttchAuctCnct* is graphically shown in Figure 43.

```
Attachments
 AttchAuctCnct:
     AuctionCnct.CnctAuctPortBidder(1,1)←→ AuctionHouse.BidderAuctPort(1,1);
 AttchCustAuc:
     Customer.CUSTAUCTPort(1,n)←→ AuctionCnct.CustPortAuct(1,n);
 ......
End_Attachements
```

Figure 42. Specification of attachments in the auction software architecture

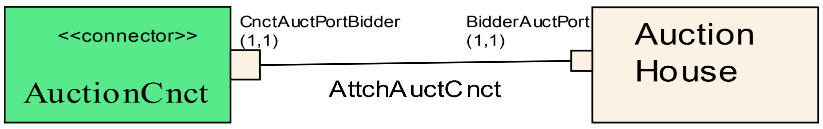


Figure 43. ***AttchAuctCnct*** **attachment that connects** ***AuctionCnct*** **and** ***Auction***

Another attachment specified in Figure 42 is the *AttchCustAuc* attachment type that connects the *CUSTAUCTPort* port of the *Customer* to the *CustPortAuct* port of the *AuctionCnct*. It is also specified that at least one to many *(1,n) AttchCustAuc* attachment instances can be connected to the *CUSTAUCTPort* port and that at least one to many *(1,n) AttchCustAuc* attachment instances can be connected to the *CustPortAuct* port. Figure 44 shows a possible configuration of AuctionCnct instances called *eBayCnct* and *ChristiesCnct* instances connected to Customer instances called *Client1* and *Client2*. The figure shows that

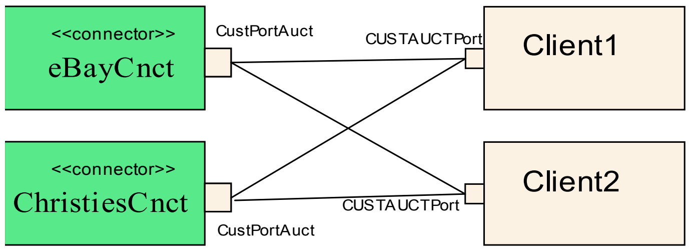


Figure 44. A possible configuration of ***AttchCustAuc*** **attachment that connects** ***AuctionCnct*** **and** ***Customer*** **instances**

4.2.6 Systems

Systems are complex components. A PRISMA system is a component that is composed of a set of components (simple components or other systems) and connectors that are connected through attachments. A system specifies an architectural pattern that can constrain how architectural elements are connected and the number of its architectural element instances.

The composition between a system and one of its architectural elements is made between a port of a system and a port of its architectural elements. This composition relationship is called a binding. A binding is required to resend the provided and requested services of the architectural element through a system port. As a result, in the metamodel the *Binding* metaclass is associated with the *Port* metaclass. In addition, an OCL constraint is specified in order to indicate that a binding can connect a port of a system and a port of either a component (or system) or a connector that belong to a system.

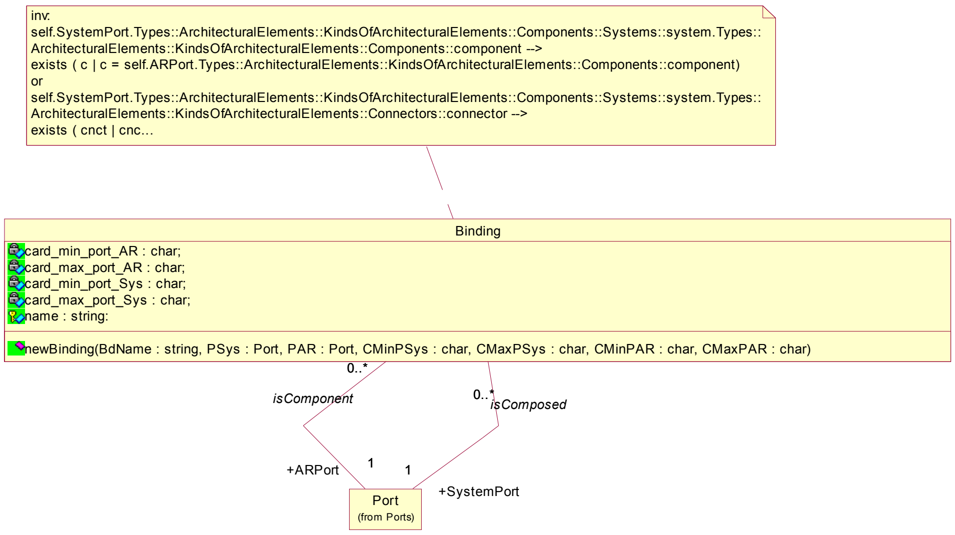

inv:
self.SystemPort.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::Systems::system.Types::
ArchitecturalElements::KindsOfArchitecturalElements::Components::component -->
exists (c | c = self.ARPort.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::component)
or
self.SystemPort.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::Systems::system.Types::
ArchitecturalElements::KindsOfArchitecturalElements::Connectors::connector -->
exists (cnct | cnc...
Binding
card_min_port_AR : char;
card_max_port_AR : char;
card_min_port_Sys : char;
card_max_port_Sys : char;
name : string:
newBinding(BdName : string, PSys : Port, PAR : Port, CMinPSys : char, CMaxPSys : char, CMinPAR : char, CMaxPAR : char)
0..*
isComponent
0..*
isComposed
+ARPort
1
1
+SystemPort
Port
(from Ports)

Figure 45. The package Bindings of the PRISMA metamodel taken from [Per06b]

A system captures functionality of a software architecture and does not act as a coordinator. A system as all architectural elements of PRISMA can have aspects and weavings relationships. As a result, in the metamodel the *System* metaclass inherits from the *Component* metaclass (see Figure 46). In this way, a system is also a component and cannot have a coordination aspect in its set of aspects (indicated by the OCL constraint).

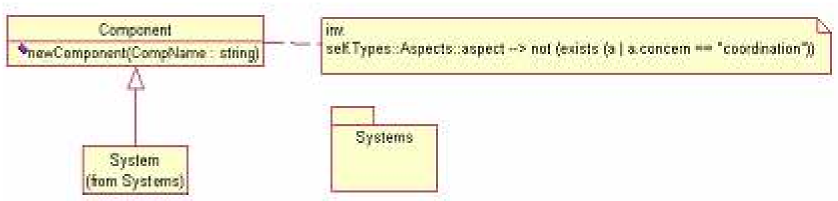

Component
newComponent(CompName : string)
inv:
self.Types::Aspects::aspect --> not (exists (a | a.concern == "coordination"))
System
(from Systems)
Systems

Figure 46. The package Systems of the PRISMA metamodel taken from [Per06b]

The *System* metaclass has a composition relationship with the *Attachment* and the *Binding* metaclasses and an aggregation relationship with the *Component* and the *Connector* metaclasses (see Figure 47).

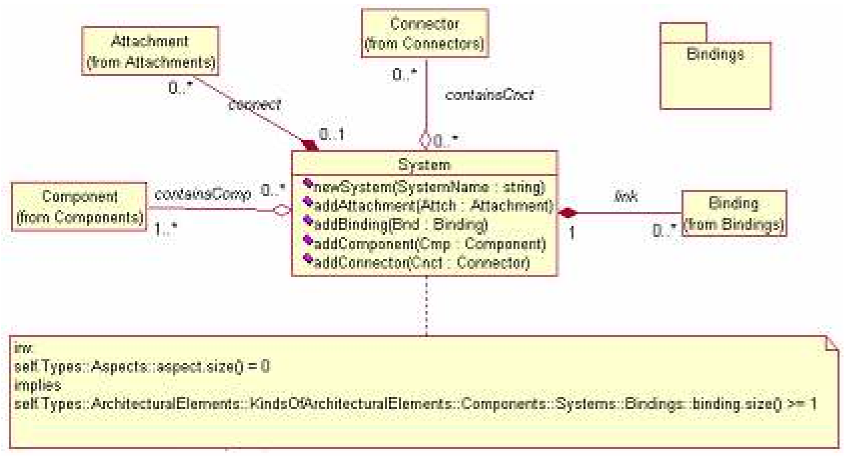

Attachment
(from Attachments)
0..*
connect
0..1
Connector
(from Connectors)
0..*
containsCnct
0..*
Bindings
System
newSystem(SystemName : string)
addAttachment(Attch : Attachment)
addBinding(Bnd : Binding)
addComponent(Cmp : Component)
addConnector(Cnct : Connector)
Component
(from Components)
containsComp
0..*
1..*
link
1
0..*
Binding
(from Bindings)
inv:
self.Types::Aspects::aspect.size() = 0
implies
self.Types::ArchitecturalElements::KindsOfArchitecturalElements::Components::Systems::Bindings::binding.size() >= 1

Figure 47. The Systems package of the PRISMA metamodel taken from [Per06b]

A system specification is preceded and ended by the reserved words *System* and *End_System,* respectively (see Figure 48). The specification of a system consists of a name, the importation of needed aspects, the weavings among aspects, a set of ports, the importation of architectural elements, the attachments among architectural elements, the bindings between the system and the architectural elements, and the two sections that allow the creation and destruction of systems. It is important to keep in mind that a system can be defined without aspects and weavings, attachments or bindings. As a result, these sections of the system specification are optional.

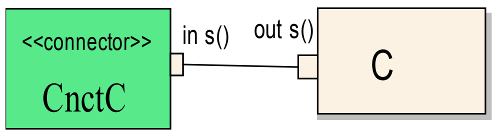
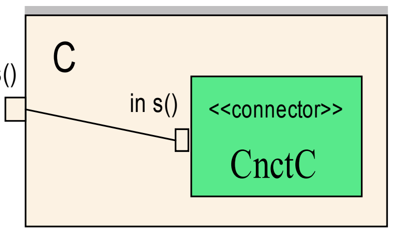
```
<system> ::= System <system_name>
                    [<aspects_importation_seq>]
                    [<weavings>]
                    <ports>
                    <architectural_element_importations>
                     [<attachments>]
                     [<bindings>]
                     <system_creation>
                    <system_destruction>
             End_System <system_name>';'

<architectural_element_importations> ::= Import Architectural Elements
                                          <architectural_element_import_list>';'
<architectural_element_import> ::= <architectural_element> '('<min_number_value>
                                    ',' <max_number_value>')'
<architectural_element > ::= <component_name> | <connector_name> |
                              <system_name>
```

Figure 48. Specification of systems

Table 4 shows the difference between an attachment and a binding in terms of their functionality in sending and receiving services among the architectural element ports they connect. When an attachment connects a component *C* that requests a service *s* of kind *out* using its port, a connector *CnctC* receives an *in* invocation of the service *s* through its port (see Table 4(a)). When a binding connects a system *C* that receives a service *s* invocation of kind *in* using its port, the connector *CnctC* receives also an *in* invocation of the service through its port.

Table 4. Comparison between Attachments and Bindings

	
(a) Attachment	(b) Binding

4.2.7 Configuration of an Architectural Model

A configuration defines a specific architectural model for a software system. At the configuration level, the needed architectural element types are instantiated and the instances are connected by attachments or binding instances. The instances are created by executing the new services of their types and the constraints are validated to ensure that the instantiations are performed following the defined architectural pattern.

An example of an architectural model at configuration is the *ConfigAuctionAgents* (see Figure 49). The architectural model specification is preceded by the reserved word *Architectural_ModelConfiguration* and the name of the architectural model configuration. This configuration consists of instantiating the architectural elements by providing the needed values of the constructors. In addition, the attachments relationships are also instantiated by connecting the instances.

```
Architectural_Model_Configuration
       ConfigAuctionAgents = new AuctionAgents {
         Auction1 = new AuctionHouse (“London, King street”, 1876”,
                                      “1 Jun”, “Spanish painting”, “800”);
         Customer1 = new Customer();
         Procurement1 = new Procurement ();
         Bidder1 = new Bidder(“painting”, “3 Jul”);
         AgentCustCnct1 = new AgentCustCnct();
         AuctionCnct1 = new AuctionCnct();
         AttchAuct1Cnct = new AttchAuctCnct (AuctionCnct1,
                                             CnctAuctPortBidder,
                                             AuctionHouse1,
                                             BidderAuctPort);
         AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                                      AuctionCnct1, CustPortAuct);
    … …
    };
```

Figure 49. Specification of the Auction Configuration

4.3 PRISMA Methodology

PRISMA proposes a methodology for modeling software architectures using the primitives it provides. The methodology is divided into five steps: Interfaces, Aspects, Architectural Elements, Systems and Configurations (see Figure 50). The methodology is supported by modeling a software architecture using the PRISMA AOADL which defines the architectural elements at two levels of abstraction: at the type definition level and at the configuration level. The type definition level defines architectural types which are stored in a repository in order to be reused by other types or specific architectures. The configuration level designs the topology of a specific architectural model by creating and interconnecting instances of the defined architectural elements in the type definition level.

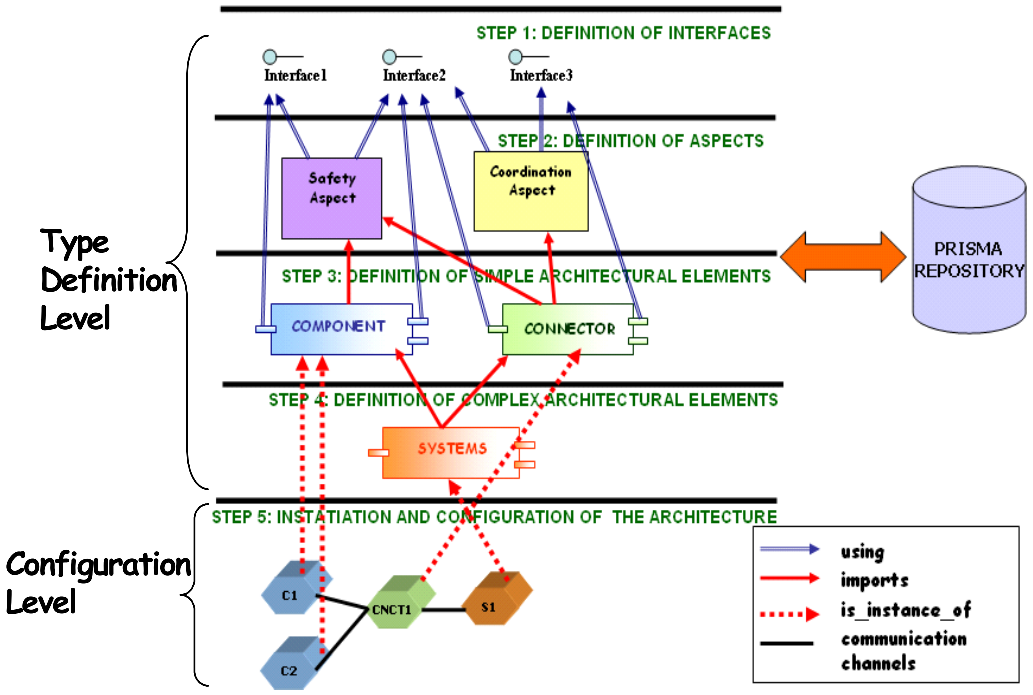

STEP 1: DEFINITION OF INTERFACES
Interface1
Interface2
Interface3
STEP 2: DEFINITION OF ASPECTS
Safety Aspect
Coordination Aspect
Type Definition Level
STEP 3: DEFINITION OF SIMPLE ARCHITECTURAL ELEMENTS
COMPONENT
CONNECTOR
PRISMA REPOSITORY
STEP 4: DEFINITION OF COMPLEX ARCHITECTURAL ELEMENTS
SYSTEMS
STEP 5: INSTATIATION AND CONFIGURATION OF THE ARCHITECTURE
Configuration Level
C1
C2
CNCT1
S1
using
imports
is_instance_of
communication channels

Figure 50. The methodology of the PRISMA approach

Steps 1-4 are modelled by using the AOADL at the type definition level and Step 5 is modelled by using the AOADL at the configuration level. These steps are explained below:

- STEP 1: Interfaces are the first to be specified due to the fact that it is not necessary to previously define other elements of the model. Interfaces are stored in a PRISMA repository for reuse.
- STEP 2: Aspects can use interfaces to define their behaviour. The same aspect can be used by many aspects. As a result, aspects are defined in the second step. Aspects are reusable entities that define a specific behaviour of a crosscutting concern. The number of aspects for the same concern is decided by the analyst, taking into account the software system and criteria such as reusability and/or understanding. An aspect for a concern or several aspects for the same concern can be defined. Aspects are reusable

entities and are stored in the repository. As a result, aspects can be reused in different software architectures.

- STEP 3: The aspects that are defined in step 2 are imported by architectural elements. An architectural element imports the aspects that define the concerns that it requires. An aspect can be imported by many architectural elements (see steps 2 and 3 of Figure 50). Architectural elements types which need to be stored in a repository implies that the storage of their aspects.
- STEP 4: To completely specify a system, the architectural elements that the system is composed of should be previously defined. In addition, the communication channels that permit the communication among them are defined. Systems are defined as patterns or architectural styles that can be reused in any software architecture whenever they are needed. For this reason, they are stored in a repository.
- STEP 5: Architectural elements types that have been stored in a repository are instantiated. As a consequence, architectural element instances have the properties and behaviours of the aspects that their architectural elements types import. The attachments and bindings relationships are instantiated in order to connect the architectural element instances. In this way, a specific software architecture is designed in PRISMA.

It is important to keep in mind that the enumeration of these steps is not a restrictive order. The enumeration simply indicates the dependencies between the different concepts that arise when the architectural model is being modelled. These dependencies are the following:

- To configure an architectural model, the concepts that are instantiated during the configuration process must have been previously defined
- To completely define a complex architectural element, the architectural elements that it consists of must have been previously defined
- To completely define an architectural element, the aspects that it imports and the interfaces that their ports use must have been previously defined

- To completely define an aspect that uses interfaces, the interfaces must have been previously defined

4.4 PRISMA Case Tool

The objective of the PRISMA Case tool is to allow a user of the PRISMA approach to model architectural models using an intuitive and friendly graphical AOADL, verify that the models are correctly built, and automatically generate executable code.

Model-Driven Engineering is a software development paradigm that is based on models that are transformed and generated in order to obtain software products [Sch06]. PRISMA follows MDE in order to develop applications from its technology-independent aspect-oriented software architectural models. Currently, there are two main approaches that apply MDE: the Model Driven Architecture (MDA) [MDA07], proposed by the Object Management Group (OMG), and the Software Factories approach, proposed by Microsoft [Gre04].

PRISMA uses Domain-Specific Languages Tools (DSL Tools) [DSL07] as an integrated framework for developing the PRISMA CASE tool. Figure 51 shows the architecture of the PRISMA CASE and how the parts that compose it allow us to automatically generate executable C# code from architectural models that have been modelled using the modelling tool. This generation is possible thanks to the code generation templates (model compiler), which isolate the specification from the source code preserving their independence. Until now, the tool generates PRISMA aspect-oriented C# code that is executable in .NET framework thanks to PRISMANET middleware, which gives support to aspect execution over .NET technology.

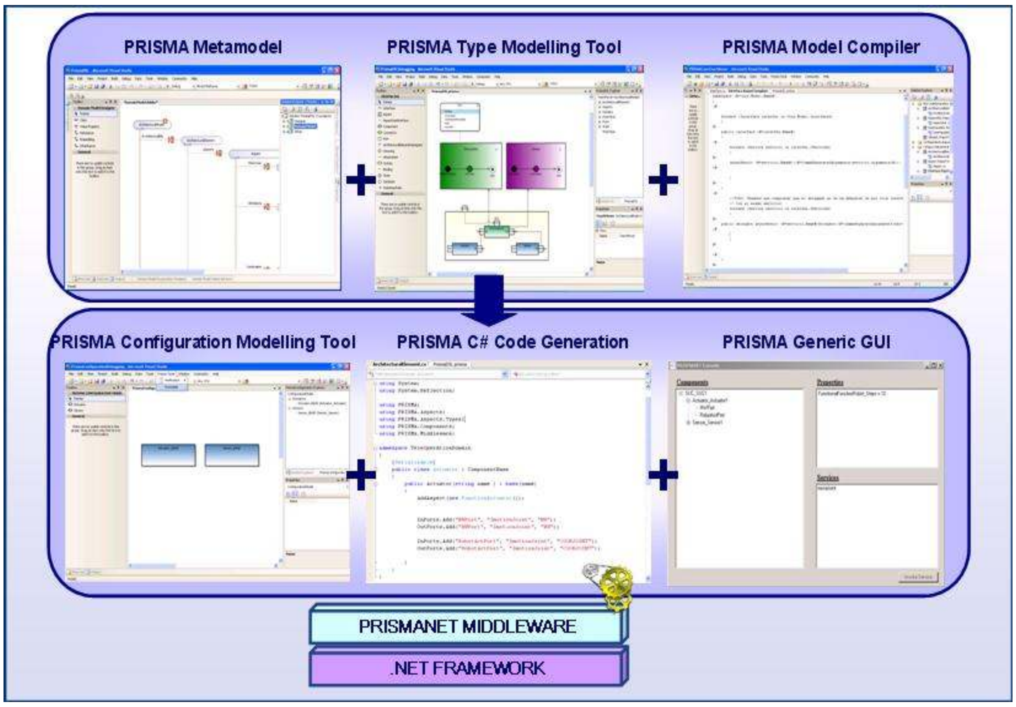


Figure 51. PRISMA CASE PARTS

The *PRISMA Type Modelling Tool* defines all the reusable types (interfaces, aspects, simple architectural elements and systems) and the architectural model of the software system are modelled in a graphical way by dragging and dropping the PRISMA modelling primitives. The *PRISMA Configuration Modelling Tool* models the configuration of the initial architecture of a specific system by instantiating the types and the architectural model that has been defined in the *PRISMA Type Modelling Tool* . Finally, the *PRISMA Model Compile* generates the code of the types and its instances. The code of the types is generated by executing the *PRISMA Model Compiler* from the *PRISMA Type Modelling Tool,* and the code of the instances is generated by executing the *PRISMA Model Compiler* from the *PRISMA Configuration Modelling Tool.* Next, the execution of the generated code joint the PRISMANET middleware can be launched from the *PRISMA Configuration Modelling Tool* .Once the aspect-oriented software architecture is executed the user can interact with it using the *PRISMA Generic GUI,* which allows the user to execute services, query the value of attributes and validate the correct behaviour of each of the architectural elements that compose the architecture.

In the following, the parts of the PRISMA CASE are briefly explained.

4.4.1 PRISMA Metamodel in DSL

DSL Tools provides the Domain Model Tool. The Domain Model tool provides a class diagram toolbox for allowing the user to define a domain model. The user represents concepts of its metamodel by using classes and relationships. In this way, each metaclass and relationship of the PRISMA metamodel has been introduced in the Domain Model (see Figure 52). Also, the OCL constraints of the metamodel (some have been presented in section 4.2) are implemented. These constraints have been implemented in order to check whether or not certain rules are satisfied during the modelling process.

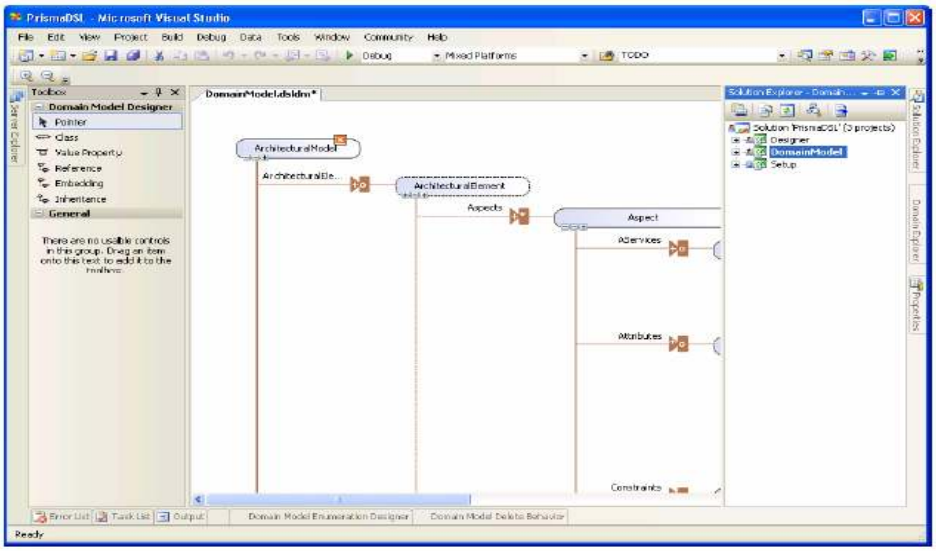


Figure 52. DSL Tools Framework: Domain Model of PRISMA taken from [Per06b]

DSL Tools also provides the Model Designer Tool. This tool allows associating a graphical metaphor to each concept defined using the Domain Model. As a result, The *Designer* project of PRISMA associates a graphical metaphor to each PRISMA metaclass of the domain model that it requires.

4.4.2 PRISMA Modelling Tool

The PRISMA Modelling tool is supported by the *PRISMA Type Modelling Tool* and the *PRISMA Configuration Modelling Tool*. The *PRISMA Type Modelling Tool* is generated from the projects defined in section 4.4.1. The modelling tool is composed of a toolbox, a drawing sheet, a *model explorer,* a *window of properties* and a PRISMA menu (see Figure 53).

The user of the PRISMA Modelling tool graphically models PRISMA types by dragging icons from the toolbox and dropping them on the drawing sheet. During the modelling process constraints are checked. Some constraints which are called *hardconstraints* do not allow the user to model something that is not allowed. Other constraints are only checked when the model is saved or when requested by the user.

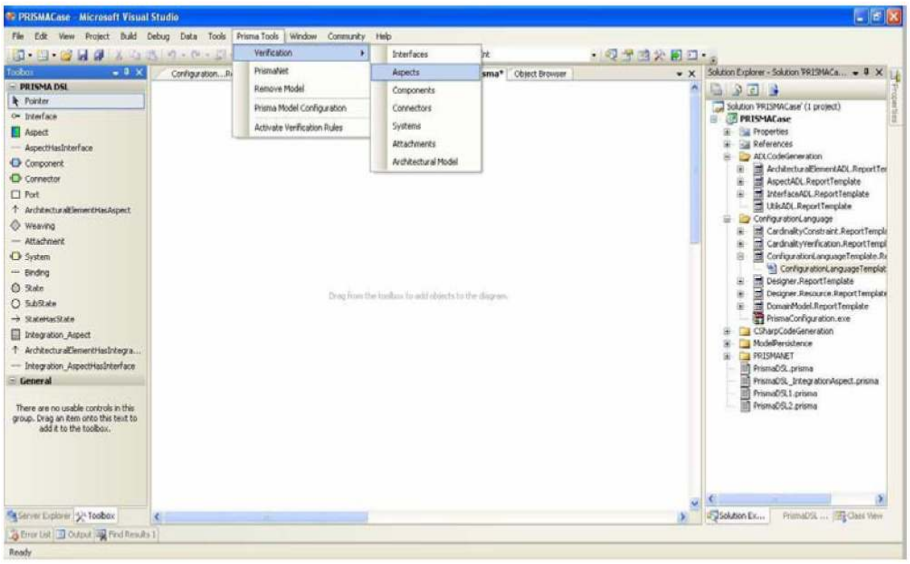


Figure 53. PRISMA Type Modelling Tool taken from [Per06b]

The *PRISMA Configuration Modelling Tool* is generated for each software architecture specified in the *PRISMA Type Modelling Tool*. The configuration modelling tool is used to develop specific software architectures using the PRISMA types defined in the PRISMA type modelling tools as modelling primitives. As a result, the toolbox of the *Configuration Modelling Tool* includes icons of the software architecture specified in the *Type Modelling Tool* instead of the PRISMA metamodel concepts. As a result, the PRISMA graphical modelling is compliant with the PRISMA AOADL, which is also divided into types and configuration.

4.4.3 PRISMANET Middleware

The .NET platform does not directly support all the PRISMA primitives. As a result, a middleware called PRISMANET [Per05a] has been implemented in C# in order to provide PRISMA constructs in the .NET platform. PRISMANET is the platform-dependent model of PRISMA in .NET, which permits PRISMA software architectures to be developed and executed on the .NET platform.

PRISMANET allows the execution of aspects, the concurrent execution of aspects and architectural elements, the loading of architectural elements, the creation of execution threads, and the management of the local components. The implementation of PRISMANET has been performed using the standard constructs of .NET and without extending the development platform. In this way, PRISMANET is an abstract middleware that sits above the .NET platform (see Figure 51). This is an advantage because PRISMANET does not have to be modified in order to be compatible with future versions of the .NET platform.

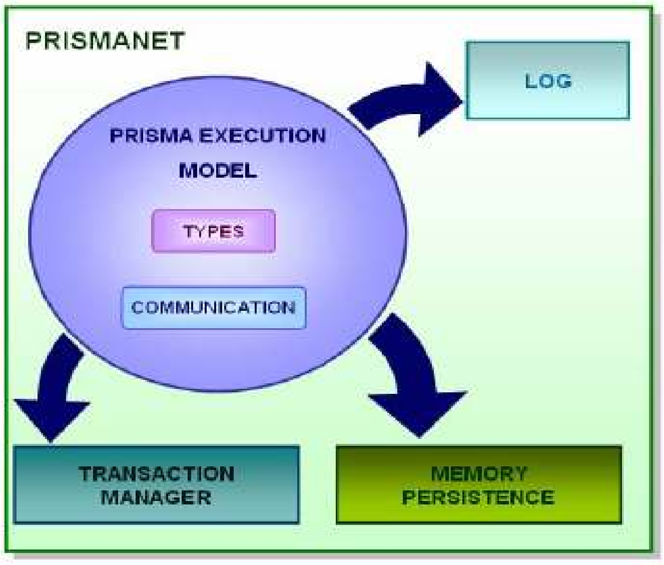


Figure 54. PRISMANET middleware architecure taken from [Per06b]

The PRISMANET architecture is constituted by four main modules (see Figure 54):

- **PRISMA Execution Model:** This module implements the basic functionality of the PRISMA types. This implementation is divided into two modules that contain the classes that implement this functionality: the Types and Communications modules. The Types module implements aspects and architectural elements, and the Communications module implements attachments and bindings. As a result, the implementation of the PRISMA application is achieved by extending these classes. The Types and Communication modules have been grouped in namespaces as shown in Figure 55.
- **Memory Persistence**: This module provides services to manage and maintain the instances of architectural elements that are stored in the main memory during their execution. The middleware manages the instances that are locally executed. Some of these services are the loading of architectural

elements instances, the creation of execution threads, and the management of architectural element lists.

- **Transaction Manager**: This module provides services to suitably execute transactions.
- **Log**: This module logs every operation that it is performed by the middleware in order to register the execution history of software architectures.

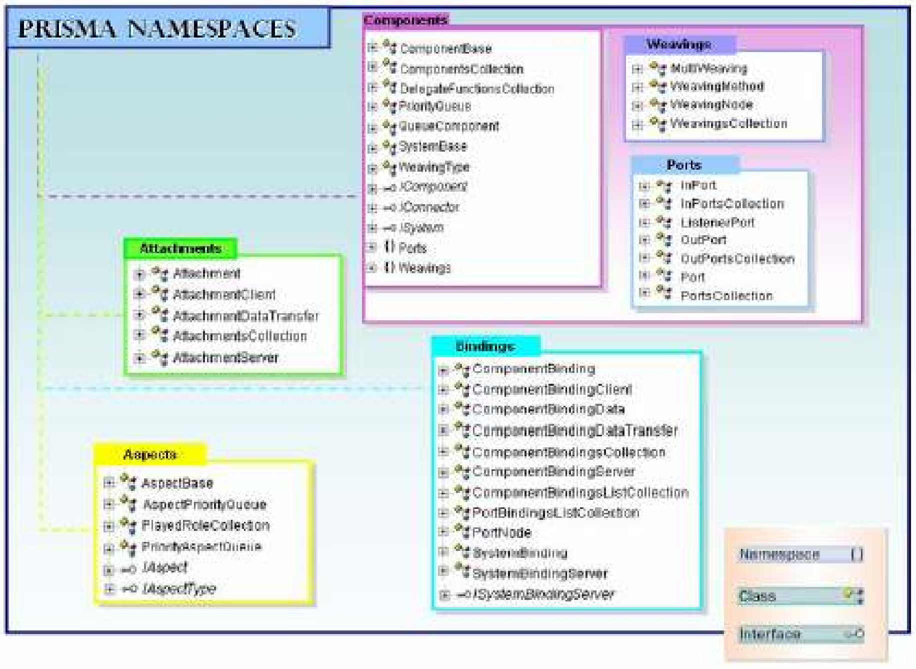


Figure 55. Namespaces of the module *PRISMA Execution Model* taken from [Per06b]

4.4.4 PRISMA Model Compiler

DSL Tools provides the capacity to implement templates that transform models specified by a graphical notation to any language. The PRISMA CASE implements a set of templates to transform each graphically specified concept in the PRISMA Modelling Tool into the textual language of PRISMA.

The PRISMA CASE also contains implemented templates to transform graphical specifications specified in the PRISMA Modelling Tool into C# code. The templates for generating C# code extend classes of the PRISMANET middleware. In order to develop these code generation templates, a set of patterns have been identified and defined to generate the C# code for each one of the PRISMA concepts to be executed over PRISMANET.

4.5 Conclusions

This chapter presents PRISMA. PRISMA is an approach that integrates AOSD and CBSD in order to describe software architectures of software systems. The PRISMA approach integrates an aspect-oriented symmetric model with an architectural model that has the notion of connector. In this way, PRISMA architectural models gain the advantage of separation of concerns techniques such as reusability and maintainability.

PRISMA follows MDE in order to provide the development of applications from models. As a result, PRISMA provides a metamodel that defines the properties of its first-class entities: interfaces, aspects, components, connectors, systems and attachments that allow the specification of architectural models. This metamodel has facilitated the automation and maintenance tasks of PRISMA software architectures since modelling tools are based on metamodels to support these tasks. In this case, the metamodel has been introduced in DSL Tools in order to develop the PRISMA framework.

An AOADL, which is based on Modal Logic of Actions and π-calculus, supports the specification of PRISMA aspect-oriented architectural models in a technology independent way. The language defines architectural models at two levels of abstraction: the type definition level and the configuration level. This permits types defined at the type definition level to be reused by the configuration level in order to define a specific software architecture.

PRISMA is supported by the PRISMA CASE. PRISMA CASE is a tool that supports the PRISMA approach by integrating the PRISMA metamodel, AOADL (graphical and textual), model compiler, and middleware. Therefore, PRISMA is a well supported approach for developing software systems.

PRISMA allows the modelling of aspect-oriented software architectures and their code generation. However, it does not support the modelling of a software architecture with properties of distributed and mobile software systems. Neither, it provides the generation of the code needed for executing systems of this kind.

The work related to PRISMA has produced a set of results that are published in the following publications:

- Cristobal Costa, **Nour Ali**, Jennifer Perez, Jose Angel Carsi, Isidro Ramos, "Towards Dynamic Reconfiguration of Aspect-Oriented Software Architectures", First International Conference on Software Architecture (ECSA 2007), **LNCS** Springer Verlang, Madrid, September, 2007 (poster).
- Jennifer Pérez**, Nour Ali**, Jose A. Carsí, Isidro Ramos, "Designing Software Architectures with an Aspect-Oriented Software Architectures", The 9th International Symposium on Component-Based Software Engineering (CBSE), Lecture Notes Computer Science, Springer Verlang, LNCS 4063, pp. 123-138, ISBN: 0302-9743, Västeras, Sweden, June-July, 2006.
- Jennifer Pérez, **Nour Ali**, Jose A. Carsí, Isidro Ramos, "Dynamic Evolution in Aspect- Oriented Architectural Models", Second European Workshop on Software Architecture, Springer LNCS 3527, pp.59-16, ISSN: 0302-9743, ISBN: 3-540-26275-X , Pisa, Italy, June 2005.
- M^a Eugenia, **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí**,** **"**DIAGMED: An Architectural model for a Medical Diagnosis", IV workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 1-7, Archena, Murcia, November, 2005. (In Spanish)
- Rafael Cabedo, Jennifer Pérez, **Nour Ali**, Isidro Ramos, Jose A. Carsí, Aspect-Oriented C# Implementation of a Tele-Operated Robotic System, III Workshop on Aspect-Oriented Software Development (DSOA), X Conference

on Software Engineering and Databases (JISBD), pp. 53-59, ISBN: 84-7723-670-4, Granada, September, 2005. (In Spanish)

- Cristóbal Costa, Jennifer Pérez, **Nour Ali**, Jose Angel Carsí, Isidro Ramos, "PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures"**,** X Conference on Software Engineering and Databases (JISBD), pp. 27-34, ISBN: 84-9732-434-X, Granada, September, 2005. (In Spanish)
- Jennifer Pérez, **Nour Ali** , Jose A. Carsí, Isidro Ramos, "PRISMA Architecture of the Robot 4U4 Case Study", Technical Report DSIC-II/13/04, pp. 72, Polytechnic University of Valencia, 2004. (In Spanish)
- **Nour Ali**, Jennifer Pérez, Cristobal Costa, Jose A. Carsí, Isidro Ramos, "Implementation of the PRISMA Model in the .Net Platform", II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- **Nour H. Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsi, "Aspect Reusability in Software Architectures", The 8th International conference of Software Reuse (ICSR), July, 2004.(poster)
- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, "Development of a Tele-Operation System using the PRISMA Approach", VIII Conference on Software Engineering and Databases (JISBD), pp. 411-420, ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)
- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Jose A. Carsí, "PRISMA: Aspect-Oriented and Component-Based Software Architectures", Workshop on Aspect-Oriented Software Development (DSOA), Conference on Software Engineering and Databases (JISBD), Technical Report TR-20/2003 of the Polytechnic School of the University of Extremadura, pp. 27-36, Alicante, November, 2003. (In Spanish)

CHAPTER 5
Ambient-PRISMA

Ambient-PRISMA enriches PRISMA with Ambient Calculus (AC) concepts in order to define aspect-oriented software architectures of distributed and mobile systems. This combination enables PRISMA to address in an abstract way, a notion of location and mobility of its architectural elements.

This chapter motivates and discusses details of this combination. This chapter is structured as follows: Section 5.1 presents how the ambient concept has been introduced in PRISMA to define Ambient-PRISMA. Section 5.2, the characteristics that distinguish an ambient architectural element from other architectural elements. Section 5.3enriches the PRISMA metamodel with the needed concepts. Section 5.4 presents the extension of the AOADL that includes the constructs needed to specify ambients.

5.1 Introduction

Ambient-PRISMA is the result of integrating AC concepts in PRISMA for describing distributed and mobile systems. This integration is achieved by enriching PRISMA with an ambient construct. The characteristics of an ambient as originally proposed by AC (see section 1.3.3) and the first-order concepts of PRISMA (see CHAPTER 4) have to be reviewed in order to support the extension.

In AC [Car98a], an ambient is defined as a bounded place where computation happens. The boundary separates what is in the ambient from what is not. In

PRISMA, software architectures are described by means of architectural elements. An architectural element is either a component or a connector. Components are the entities that are responsible for capturing computations. Connectors are the entities that coordinate computations. Neither of them captures the notion of boundary or location that we could possibly use to capture what ambients are meant to provide.

For example, in the Mobile Agent Auction case study, the Customer and the Procurement are components and the AgentCustCnctr connector coordinates them (see Figure 56). There is no primitive that represents where these components and connectors are located nor a primitive responsible for the way these move. The objective is to allow PRISMA components to be distributed across several locations and that PRISMA connectors coordinate components regardless of where they reside and they are not responsible for moving them around.

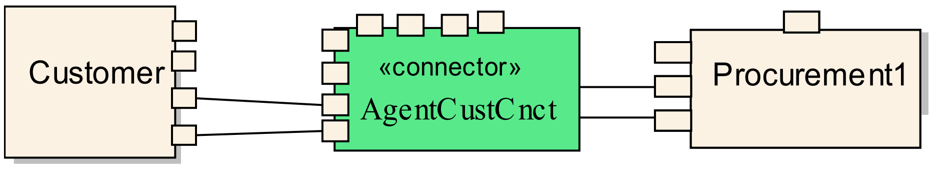


Figure 56. A Customer and Procurement connected through a AgentCustConct

A PRISMA component has the following characteristics: It defines the computation of a software system. It cannot have a coordination aspect. A component can be composed by other components (Systems) to define its functionality. An ambient in PRISMA, cannot be a component because it does not represent the computation of the software system. Also, an ambient cannot be composed of other ambients in order to form more complex ones. As a result, an ambient is not a component. An ambient coordinates what is in its boundary from what is out and needs a coordination aspect to perform this coordination.

A PRISMA system has the following characteristics: It defines complex computations of a software system that needs other components to perform them.

Components can be directly connected to other connectors or components that do not form part of the system they are part of. An ambient is an element that locates elements, however the functionality of an ambient does not depend on the functionality of the elements it contains. Also, elements that are in an ambient cannot be connected directly to elements of other ambients. Elements that need to be connected to elements of other ambients have to be managed by an ambient in order to control security.

A PRISMA connector has the following characteristics: It defines coordination mechanisms among components (or systems). It needs a coordination aspect. Ambients also coordinate elements that are contained in it with elements that are outside it. As a result, ambients need coordination mechanisms in order to control what can enter or exit its boundary. However, in PRISMA connectors cannot locate and manage other architectural elements and their attachments.

Therefore, ambients are introduced in PRISMA as a specialized kind of connector that extends connectors in order to be able to coordinate a boundary that models the notion of location and provides mobility support to other architectural elements. An ambient architectural element represents the boundary of a place where PRISMA components and their connectors are located. For example, if the Customer1 and the Procurement1 are in the same site, they are located in the same ambient. However, if they are in different sites, then they are in different ambients. This boundary needs coordination mechanisms in order to determine what can pass in or out of it. If the Procurement1 component and the AgentCustCnctr connector are in different ambients and they need to communicate, the ambient of each architectural element needs to synchronize their services so that their services pass out and in the boundaries of the other ambient. Also, if the Procurement1 component needs to move from a site to another, it has to cross the boundary of the Site ambient where it resides and enter the boundary of another ambient. Procurement1 crosses the boundary of its Site ambient (movement) when its Site ambient coordinates it with its new destination ambient.

Ambients have a CBSD view, an AOSD view and a boundary. Figure 57 shows the graphical representation of a PRISMA ambient. The Ambient CBSD view describes it as a black box where it communicates with other architectural elements by using ports through which invocations of services are sent and received. The AOSD view describes an ambient as an architectural element that is composed of different aspects weaved together. A boundary defines a notion of space. The dotted line of an ambient, represents its boundary. The graphical representation preserves the folder calculus representation typical of AC (see Ambient Calculus), however, the boundary is represented with a dotted line instead of a straight line. This is to clarify that a PRISMA ambient is not composed of the architectural elements in its boundary. It can be noticed that the ambient has ports that are outside the boundary and ports that are inside the boundary. This is to indicate that elements of its boundary are connected to the ports that are in the boundary and the ports outside its boundary are for connecting the ambient with elements that are not in its boundary.

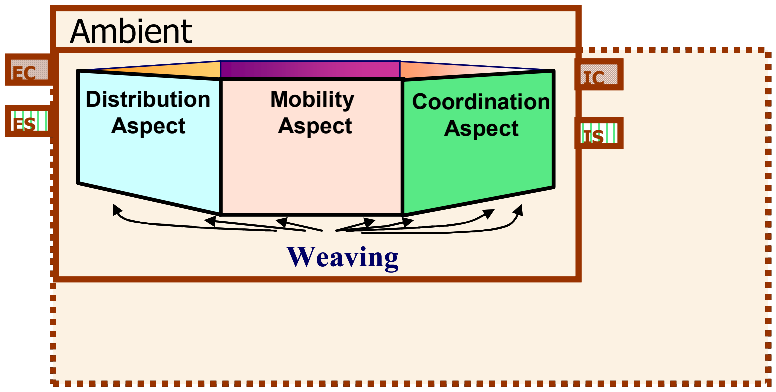


Figure 57. A PRISMA Ambient with CBSD and AOSD views

Ports of architectural elements of a boundary are connected to their ambient ports by attachments. For example, Procurement1 is connected to its parent ambient called AuctionSite by means of attachments. Poner figura

In AC, an ambient can be nested within other ambients [Car98a]. Ambients in PRISMA can be located inside the boundary of other ambients. This allows Ambient-PRISMA software architectures to describe topologies of locations where hierarchies of distributed and mobile systems can be modelled. For example, Figure 58 shows that the Customer1 component and the AgentCustCnctr connector are located in the ClientSite ambient, whereas the Procurement1 component, the Auction1 component and the AuctionCnctr1 connector are located in the AuctionSite ambient. Furthermore, the ClientSite and the AuctionSite ambients are located in the Root ambient.

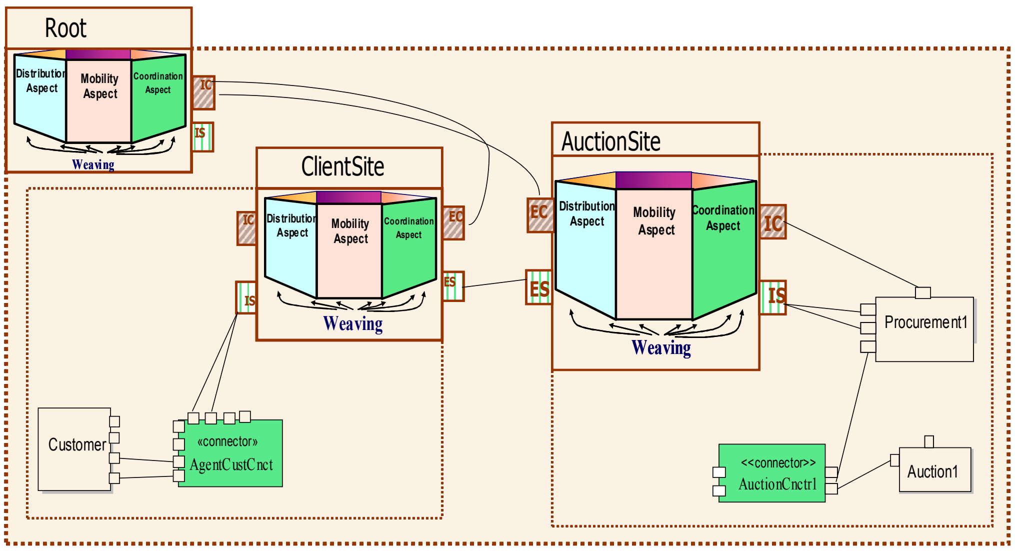


Figure 58. A possible Configuration of the Auction Mobile Agent Case Study where the Procurement instance is in AuctionSite

As a result, Ambient-PRISMA extends PRISMA with a new type of connector called ambient. Ambient-PRISMA preserves the PRISMA connector which coordinates computation and as a result coordinates at least two components. An ambient coordinates components with connectors that are in different ambients for communication purposes. Also, an ambient coordinates components and connectors with other ambients for mobility purposes. For example, when the Procurment1 component moves from AuctionSite to ClientSite it needs to be coordinated: First, the AuctionSite ambient coordinates the Procurement with the Root ambient and then the Root ambient coordinates the Procurement1 with the ClientSite.

Elements of different ambients do not have attachments between each other. Attachments only connect elements of the same ambient which can be ambients, components or connectors. For example, Clientsite and AuctionSite are connected because they are siblings (share the same parent). Also, Customer1 has an attachment with *AgentCustCnctr*. However, Procurement1 does not have an attachment with AgentCustCnctr. The connection between Procurement1 and AgentCustCnctr is performed by three attachments: an attachment between Procuremet1 and AuctionSite, an attachment between AuctionSite and ClientSite, and an attachment between ClientSite and AgentCustCnctr.

When an architectural element moves from an ambient to another, the attachments associated to an ambient are reconfigured in order to provide the mobile architectural element to be connected to the architectural elements it communicates with. For example,

5.2 Ambient Characteristics in Ambient-PRISMA

An ambient is introduced in PRISMA as a new kind of connector. However, it has its proper characteristics that distinguish it from connectors and other PRISMA architectural elements (components and systems). In the following, the characteristics of an ambient are explained:

5.2.1 Interfaces

The interfaces of an ambient publish a set of services. An interface describes the signature of the services that can be invoked or requested through that interface. The signature of a service specifies its name and parameters. The data type and the kind (input/output) of parameters are also declared.

An ambient has at least two predefined interfaces. They are the *ICall* interface and *ICapabilities* interface. In the following, these interfaces are explained:

5.2.1.1 ICall Interface

An ambient needs to control the invocations of requested and provided services of architectural elements located in an ambient. This implies that every ambient must specify all the interfaces of its internal elements. However, this is not efficient because the set of interfaces of an ambient would change each time its internal elements change. For example, when an architectural element moves from an ambient to a new ambient, the first ambient has to remove the interfaces of the architectural element and the new ambient has to add these to its set of interfaces. Therefore, a generic interface has been defined that abstracts to an ambient the interfaces of its internal elements. This interface is called *ICall* which permits the redirection of service requests independently of their interfaces.

The *ICall* interface describes a service called *call* (see Figure 59). The *call* service encapsulates the signature of a requested or a provided service of one of the architectural elements located in an ambient as its parameters. The call service has three parameters: *Name* parameter which stores the name of the original service, *ParamsList* parameter which is a list that stores the parameters of the original service and the *ParamType* parameter which is a list that stores whether the parameters of the original service are input or output by storing *true* or *false*, respectively.

```
Interface ICall

   call(input Name: string, input ParamsList[]: Parameters,
        input ParamType[]: boolean);

End_Interface ICall
```

Figure 59. Specification of the ICall Interface

The call service is the transformation of a requested or a required service of an architectural element in an ambient. For example, the *Procurment1* component of Figure 58 requires the *notifyProdInterest(**input** Saleroom:string, **input** SaleNum:string, **input** DateOfAuction:Date, **input** Lotdescrip:string, **output** Interested: boolean)* service from the Customer1 component through the AgentCustCnctr connector. As a result, the *notifyProdInterest* service is transformed into a call invocation which the *AuctionSite* ambient receives with the value of the Name parameter is equal to *notifyProdInterest*, the *ParamsList* stores [*Saleroom, SaleNum, DateOfAuction, Lotdescrip, Interested*], and the *ParamType* stores [true, true, true, true, false].

5.2.1.2 ICapability Interface

An ambient offers a set of services for allowing architectural elements in it to be able to move. The interface that publishes the set of services for allowing architectural elements in it to move is called *ICapability* (see Figure 60). Some of these services are *exit*, *enter*, and *finishMovement*. The *exit* service has two parameters: *Name* and *Parent*. The *Name* parameter indicates the name of the architectural element that needs to exit. The *Parent* parameter indicates the name of the ambient that an architectural element needs to exit from.

```
Interface ICapability

   ….
   exit (input Name: string, input Parent:string );
   enter (input Name: string, input NewAmbient: string);
   finishMovement(input Name, input CommunicationList[]: Attachment);
    …

End_Interface ICapability
```

Figure 60. Partial specification of the ICapability Interface

For example, when *Procurement1* component of Figure 58 needs to exit from the *AuctionSite* ambient, it invokes the *exit (Procurement1, AuctionSite)* service which is published through the *AuctionSite* ambient *ICapability* interface.

5.2.1.3 *IGetLocation Interface*

An ambient publishes in an interface called *IGetLocation* a service called *getLocation* in order to inform the name of its parent ambient (see Figure 61). The *getLocation* service has one parameter called *Location*. The *Location* parameter is an output parameter that returns the name of the parent ambient.

```
Interface IGetLocation
 getLocation(output Location: string);
End_Interface IGetLocation
```

Figure 61. Specification of the IGetLocation Interface

For example, if the *Procurement1* component of Figure 58 invokes the *getLocation* of the the *AuctionSite* ambient, the AuctionSite would return the value “Root” in order to indicate that *AuctionSite* is located in the *Root* ambient.

5.2.2 Aspects

Aspects of an ambient define its state and behaviour from a specific concern of the software system. The different aspects of an ambient specify the services it offers and requests. Since ambients provide coordination for providing distribution and mobility support to the architectural elements inside their boundary, an ambient at least has the following aspects:

- **Coordination Aspect**: This aspect coordinates services required and provided by architectural elements of an ambient with the exterior. This aspect is unique in Ambient-PRISMA and it is called *ACoordination*.

- **Mobility Aspect**: This aspect specifies the ambient calculus primitives as services that are offered to the architectural elements of an ambient to allow them to move. The services of this aspect differentiate an ambient from other PRISMA connectors. This aspect is unique in Ambient-PRISMA and it is called *MobilityAspect*.

- **Distribution Aspect**: This aspect allows an ambient to be aware of its parent, except for the Root ambient that does not have one. It also specifies whether the ambient is mobile or not, and what its mobility strategies are i.e. how and when it needs to move.

The Mobility Aspect and the Coordination Aspect are predefined, i..e. the behaviour that these aspects provide are always the same. Therefore, these two aspects are generic and are reused by all ambients. On the other hand, the distribution aspect may specify different distribution behaviours depending on the requirements that the ambient needs to achieve. Therefore, the distribution aspect is not predefined. An ambient can also have other kinds of aspects such as a security aspect depending on the requirements of the architectural model.

The aspects of an ambient distinguish it from components (or systems) and connectors. Components cannot import a coordination aspect. Both an ambient and a typical PRISMA connector have a coordination aspect. However, a PRISMA connector cannot import a Mobility Aspect.

In the following, the three different kinds of aspects of an ambient are explained in detail:

5.2.2.1 ACoordination Coordination Aspect

An ambient is responsible for controlling the exchange of services across its boundary i.e. the communication that architectural elements inside its boundary have with architectural elements outside it. This is achieved by the coordination aspect of the ambient called *ACoordination*. The *ACoordination* aspect coordinates the sending of messages from elements that are located inside an ambients boundary to elements that are outside its boundary and vice versa. In this way, communication also respects the ambient hierarchy, since services have to pass through all the coordination aspects of all the ambients until they reach their destination.

For example, in Figure 58 the *Procurement1* component needs to communicate with the Customer1 component through the *AgentCustCnctr* connector in order to send to the *Customer1* component the product description of a possible candidate product that the Customer maybe interested in. Also, the *Customer1* component has to reply to the *Procurement1* component by telling it if he/she is interested or not. Since the *Customer1* component is in an ambient different from that of the *Procurement1* component, the services that they interchange must pass through their ambients until reaching them. In this case, this is achieved through the *ACoordination* aspect of both the *ClientSite* ambient and the *AuctionSite* ambient.

Since all services of the internal and external elements of an ambient must pass through its *ACoordination* aspect, an ambient must control all the interfaces of its internal elements. Therefore, the *ACoordination* aspect uses the *ICall* interface which is specified in Figure 59 in order not to change its interface each time an architectural element exits or enters an ambient. The *ACoordination* aspect specifies the behaviour of the call service (see Figure 62). It specifies that the *call* service is an in/out i.e. it is required and provided. The *ACoordination* coordination aspect is a generic aspect that is reused by all ambients.

```
Coordination Aspect ACoordination using ICall

Services
   …
   in/out call(input Name:string, input ParamsList[]: Parameters,
               input ParamType[]: bool);
   …


End_Coordination Aspect ACoordination
```

Figure 62. Partial specification of the ACoordination Aspect

5.2.2.2 Mobility Aspect

Since an ambient controls what moves in and out of its boundary, one of its responsibilities is to offer services that allow architectural elements inside its boundary to move. These mobility services are specified in the mobility aspect. As the Mobility Aspect gives a behaviour that is common to all ambients, this behaviour is defined in an aspect that is imported and reused by all ambients. The unique Mobility Aspect is called *MobilityAspect*.

The mobility services that we consider in Ambient-PRISMA are primitives of AC: the capabilities (except for the open capability), the ambient creation and replication. Other services needed for preserving a consistent state of the architecture for mobility are also offered. The mobility aspect of an ambient provides the following services:

- It allows an ambient to offer the exit service to the elements that need to exit from it. (The specification of the AC exit capability).
- It allows an ambient to offer the enter service to the elements 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 element from external ambients.
- It allows an ambient to create copies of architectural elements. (The specification of the replication).
- It allows an ambient to offer the startMovement service to the elements that are going to make a sequence of exits and enter requests. This service allows an ambient to prepare an element to move.
- It allows an ambient to offer the finishMovement service to the elements that have finished making requests to a sequence of enters and exits. This service provides an ambient to reconfigure attachments once a mobile architectural element reaches destination.

For example, the Procurement1 component in Figure 58 is a mobile component. In order to be able to move, it requests the mobility services that the *MobilityAspect* aspect of its parent ambient, in this case the *AuctionSite* ambient. The *MobilityAspect* of the *AuctionSite* requests from the *MobilityAspect* aspect of the destination ambient of the *Procurement1* component to accept it. In this way, the *AuctionSite* acts as a coordinator between the *Procurment1* component and its destination ambient.

The *MobilityAspect* aspect uses the *ICapability* interface which is specified in Figure 60 in order to specify the behaviour of the services which it publishes (see Figure 63). For example, the *MobiltyAspect* aspect specifies that the *exit* service is a transactional service. It specifies that the exit transaction is ***in*** i.e., it is provided. The transaction checks whether the architectural element that needs to exit is a child of the aspects ambient or not. This check is performed through the *isChild* service. The *ischild* service returns true through the *isChildOK* output parameter if the requested architectural element is a child of the ambient. If *ischild* returns true, and the architectural elements needs to exit the ambient which is currently its parent then the transaction proceeds to execute *getParent* service. The *getParent* service returns in the *Parent* parameter the name of the parent ambient of the *MobilityAspect* ambient. This is needed in order to indicate the destination of the exiting architectural element. Then, the moving process is performed through the *moving* service[1]. The *moving* service has two parameters: *Requested* and *Parent*. The *Requested* parameter indicates the name of the exiting (or moving) architectural element and the *Parent* parameter indicates the destination of the moving architectural element.

```
Mobility Aspect MobilityAspect using ICapability

  …
  TRANSACTIONS in EXIT(input Requested: string, input Ambient: string):
   exit  = isChild! (input Requested, output isChildOK)→
          EXIT1;
   EXIT1 = {isChildOK==true & self.Name==Ambient}
           getParent(output Parent)→EXIT2;
   EXIT2 = moving! (Requested,Parent);



End_Mobility Aspect MobilityAspect
```

[1] The moving service is a transactional service. This is explained in detail in section 5.4.6

Figure 63. Partial specification of the *MobilityAspect* aspect

Most of the services that the *MobilityAspect* aspect provides are transactional services of reconfiguration services, i.e., services that change the configuration of the software architecture. For example, the exit service is a transactional service that consists of removing an element from the boundaries of an ambient, removing attachments between the element and the ambient, removing attachments between the element and the elements of the same ambient and creating attachments between elements that were connected to the element and the ambient.

Ambient-PRISMA does not provide the open capability of Ambient Calculus as a primitive. This is due to the fact that Ambient-PRISMA follows previous works such as Boxed Ambients [Bug01] in considering the open capability as a threat to security. The open capability allows opening ambients that can have malicious contents which cannot be known in advance. Therefore, it is preferable not to open ambients. As a result, Ambient-PRISMA does not provide opening of ambients.

5.2.2.3 Distribution Aspect

A distribution aspect is responsible for specifying the location of an ambient. An ambient is located in a hierarchy of other ambients, therefore the distribution aspect has to specify its parent ambient. The distribution aspect gives location-awareness to an ambient and it allows it to take different decisions or strategies depending on its current location. It also specifies the mobility strategies of an ambient, thus an ambient can be a mobile architectural element. The distribution aspect specifies if the ambient is mobile or not and if it is, it specifies when and how the ambient should move. Therefore, using the distribution aspect an ambient can request mobility services from its current parent ambient, and therefore the ambient can change its position in the hierarchy changing its parent ambient.

Each distribution aspect must save the parent ambient of the ambient that imports it and provides a service in order to allow other aspects of the ambient to consult its current location. Therefore, a distribution aspect uses the *IGetLocation* interface specified in Figure 61. Figure 64 shows the specification of a possible a distribution aspect called *ADist*. *ADist* aspect is an aspect of an ambient. It stores the name of the parent ambient of the ambient that imports it through the *location* attribute. The *ADist* aspect specifies that the *getLocation* service is an 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*).

```
Distribution Aspect ADist using IGetLocation
 Attributes
  …
  location : string NOT NULL;
  …
 Services
  …
  in/out getLocation( output Location:string)
     Valuations
         [in getLocation(output Location)] Location := location;
  ….
End_Distribution Aspect ADist
```

Figure 64. Partial specification of an aspect of kind Distribution

Although an ambient must have a distribution aspect, a distribution aspect is not a generic aspect that is predefined. In this way, a distribution aspect is defined differently depending on the systems requirements. For example, the distribution aspect of the *ClientSite* and the *AuctionSite* specifies that their parent ambient is the Root. Also, these two ambients are not mobile ambients. Therefore, both can reuse the same Distribution Aspect. However, in Figure 65 *MobileAmb* is an ambient that can move. As a result, its distribution aspect specifies that it is mobile and when it can move.

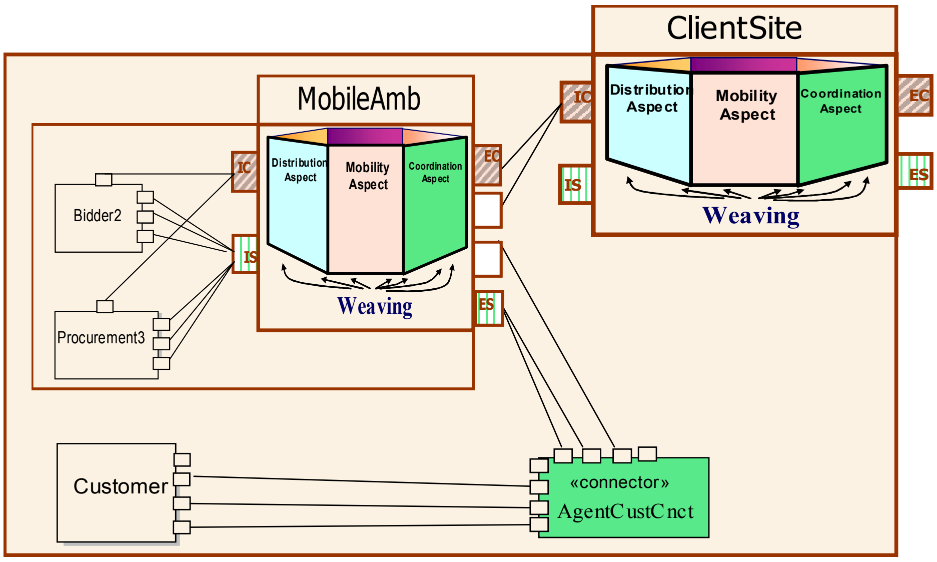


Figure 65. MobileAmb is a mobile ambient

5.2.3 Weavings

Aspects of an ambient can be weaved together. The services of two aspects can be synchronized by the weavings in order to define the overall behaviour of an ambient.

A weaving is needed because when an ambient serves the exit transaction, the ambient needs to know its parent ambient. This is due to the fact that an architectural element that needs to exit, it does not specify its destination ambient. Since an architectural element that exits an ambient moves to the parent of its ambient, a weaving needs to trigger the *getLocation* service of the distribution aspect of the same ambient. As a result, all ambients have a predefined weaving between the *MobilityAspect* aspect and the distribution aspect of an ambient.

The predefined weaving of each ambient specifies that the *getParent* service of the *MobilityAspect* aspect triggers the *getLocation* service of a distribution aspect (see Figure 66). The *getLocation* service is executed instead of the *getParent* service.

```
Weavings
     ADist.getLocation(Location)instead
                                MobilityAspect.getParent(Parent);
```

Figure 66. Predefined weaving between the MobilityAspect aspect and a distribution aspect called ADist

In addition, an ambient can have other weavings depending on the aspects that it imports. Although an ambient has three obligatory aspects, an ambient can also have more aspects. For example, an ambient can have a security aspect that constrains the messages that can be received and sent by the architectural elements inside its boundary. The security aspect can also define constraints on what type of architectural elements can be accepted to enter its boundary. The proper behaviour of such a security aspect is captured in the way it is weaved with the coordination aspect or with the mobility aspect.

Additional constraints can be added to the communication provided by the coordination aspect by adding aspects to the ambient. For example, in order to define certain security politics to constrain the type of services that the elements of an ambient can interchange with elements from the exterior, a security aspect can be imported and weaved with the coordination aspect.

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 *MobilityAspect* aspect, but the *MobilityAspect* aspect is weaved with different security aspects.

5.2.4 Ports

The services that the *MobilityAspect* aspect and the *ACoordination* aspect specify, are offered to other architectural elements through the ports. Therefore, an ambient also has some predefined ports.

As a consequence, each ambient must have at least four ports: two for the *MobilityAspect* aspect and two for the *ACoordination* aspect. The two ports for providing/requesting services of the *MobilityAspect* are *InCapabilitiesPort* port and *ECapabilitiesPort* port. The *InCapabilitiesPort* port offers the mobility services of the parent ambient to its architectural elements (Marked with IC in Figure 58 and Figure 65). The *ECapabilitiesPort* port allows an ambient to request or receive to/from its parent ambient to accept an architectural element (Marked with EC in Figure 58 and Figure 65).

The two ports for providing/requesting services of the *ACoordination* aspect are *InServicesPort* port and *EServicesPort* port. These ports allow architectural elements of an ambient to be in contact with the exterior boundary of an ambient. The port of an ambient that is marked with IS in Figure 58 and Figure 65, is the *InServicesPort* port and it is connected to internal architectural elements. The port of an ambient that is marked with ES in Figure 58 and Figure 65, is the EServicesPort port and it is connected to exterior architectural elements.

Figure 67 shows the specification of the ports section in an ambient specification. It can be observed that the *InCapabilitiesPort* port and the *ECapabilitiesPort* are of type *ICapability* interface i.e., they publish the services of the *ICapability* interface. It can also be observed that the *EServicesPort* port and the *InServicesPort* port is of type *ICall* inetraface.

```
Ports
     InCapabilitiesPort: ICapability …
     ECapabilitiesPort: ICapability …
     EServicesPort: ICall ….
     InServicesPort: ICall ….
….
 End_Ports
```

Figure 67. Partial specification of ambients ports

5.2.5 Attachments

The Component-Based view of an ambient describes it as a black box that can only communicate with the other architectural elements through ports that send and receive services. The parent ambient communicates with the architectural elements that are inside its boundary through *attachments* that connect its ports with the ports of the architectural elements inside its boundary (see Figure 58 and

Figure 65). An ambient communicates with architectural elements that are its siblings through its attachments. For example, the *ClientSite* ambient and the *AuctionSite* ambient of Figure 58 are siblings and are connected through attachments.

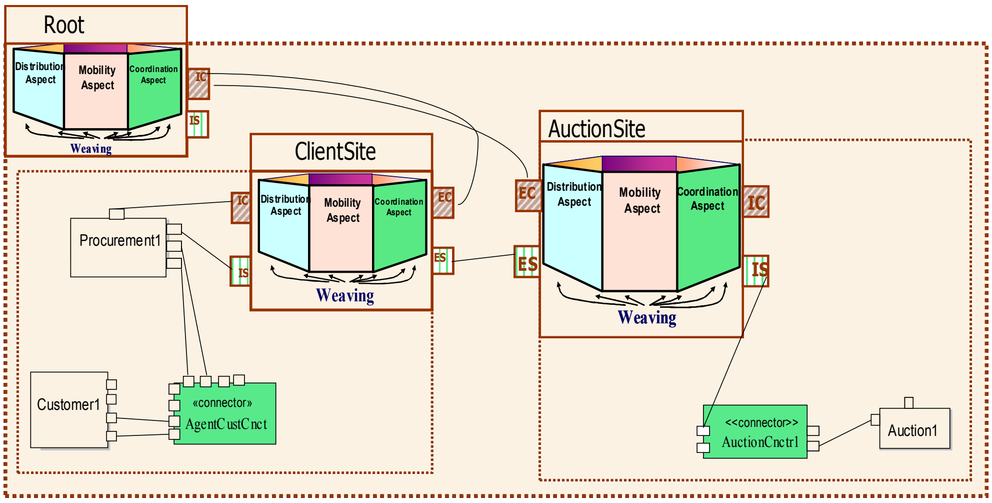


Figure 68. Configuration of the software architecture when Procurement1 is located in ClientSite ambient

An architectural element in an ambient cannot be directly connected through an attachment to another architectural element in another ambient. An attachment that connects two architectural elements in different ambient is broken down into many attachments. For example in Figure 58, the *Procurement1* component and the *AgentCustCnctr* connector have an attachment that connects them in order to communicate. This attachment is broken down into an attachment that connects the *InServicesPort* port of the *AuctionSite* ambient and a port of the *Procurement1*, the an attachment that connects the *EServicesPort* port of the *AuctionSite* ambient and the *EServicesPort* port of the *ClientSite* ambient, and an attachment that connects the *AgentCustCnctr* connector to the *InServicesPort* port of the *ClientSite* ambient. When the *Procurement1* component moves from the *AuctionSite* ambient to the *ClientSite* ambient (see Figure 68), the attachment between the *Procurment1* component and the AgentCustCnctr connector is directly connects them.

An ambient is responsible of managing the attachments that are located in its boundary i.e., the attachments between its architectural elements and its ports and the attachments between its architectural elements. As a result, when an architectural element moves from an ambient, the ambient is responsible of reconfiguring the attachments.

The attachment that connects a mobile architectural element to the InCapabilities port of its parent ambient always has to be created in the new ambient where the architectural moves. For example, the attachment that connects the *Procurement1* component to the *AuctionSite* ambient, when the *Procurement1* component is located in AuctionSite ambient (see Figure 58) is substituted to an attachment between the *Procurement1* component and the *InCapabilitiesPort* port of the *ClientSite* ambient (, when the *Procurement1* component moves to the *ClientSite* ambient (see Figure 68). As a result, the *Procurement1* component can always request mobility services because it is connected by an attachment to its *InCapabilitiesPort* port parent ambient.

5.2.6 Kinds of Ambients

Although the previous characteristics are in common to all ambients, however, it is necessary to separate different types of ambients as each can have its own constraints and semantics. In Ambient-PRISMA, we have identified two kinds of ambients:

- **Site Ambients**: Sites are ambients that represent physical locations; that is, they 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. For example, the ClientSite and the AuctionSite of Figure 58 are of type Site.
- Non-Site Ambients: Non-Site Ambients do not model a physical location. They are separated into two kinds :
  - **Group Ambients**: Group ambients represent a collection of architectural elements. There utility can be different depending on the domain. They can

be used in order to group a set of architectural elements that share specific security politics or in order to move architectural elements at the same time. The Group ambient can lodge other Group ambients, Components and Connectors. For example, *MobileAmb* in Figure 65 is a Group Ambient that moves architectural elements together.

- **Virtual Ambients**: They are virtual ambients that can lodge other Virtual Ambients or Sites, but not both. Virtaul Ambients allow to model networks or vitual spaces where Sites are located. A Virtual Ambient that consists of a collection of Sites represents the boundary of a network (e.g. a Local Area Network). The modelling of a Virtual Ambient hierarchy can represent the hierarchy of LANs to form a Wide Area Network (WAN). Different Virtual Ambient hierarchy levels can be represented in the software architecture depending on the software architect and on the distributed requirements. In Ambient-PRISMA there is always a default Virtual Ambient that represents the root of an ambient hierarchy called Root(see Figure 58).

In summary, the Root ambient (Virtual Ambient) can either contain another set of Virtual Ambients or a set of Sites. Sites can contain a combination of Components, Connectors or Group ambients. Finally, Group ambients can contain also Components, Connectors and other Group ambients. The difference between a Site and a Group ambient is that a Site ambient must have a Virtual ambient as its parent and a Group ambient cannot have a Virtual Ambient as a parent ambient.

Figure 69 shows an example of how two WANs formed by two LANs have been modelled using the different kinds of ambients. Also, the LANs are formed from hosts that represent devices. Also, each host can contain other ambients. In the Figure, the kind of each ambient is indicated.

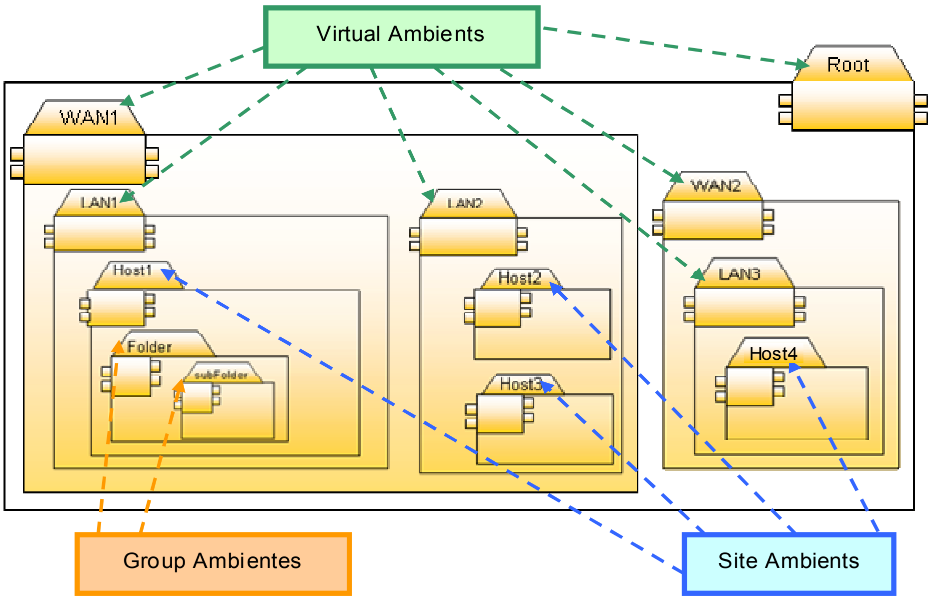


Figure 69. Kinds of ambients in a possible network

5.3 Ambient-PRISMA Metamodel

PRISMA provides a metamodel that includes a set of metaclasses and their relationships. Each concept considered in PRISMA is represented by a metaclass that defines a set of properties and services. The metamodel has been defined using the class diagram of the Unified Modelling Language (UML) 1.5. and the Object Constraint Language (OCL) 2.0 [UML07]. Section 4.2 presented part of the PRISMA metamodel. Since Ambient-PRISMA extends PRISMA with new concepts related to ambients, the metamodel of PRISMA has to be enriched.

In this section, the PRISMA metamodel is extended with concepts and properties that Ambient-PRISMA. These concepts have been included by adding new metaclasses and relating them with the ones which PRISMA provides. The metaclasses and relationships define the structure and the information that is needed to describe Ambient-PRISMA architectural models. In addition, Ambient-PRISMA metamodel includes OCL expressions [War03] in order to define constraints that must be satisfied by an Ambient-PRISMA architectural model. These constraints guide the methodology for modelling Ambient-PRISMA architectural models. At the end of the modelling process, all of them have to be satisfied in order to ensure that an architectural model is correct.

In the following, the different packages that are altered for Ambient-PRISMA are presented:

5.3.1 Connector Package

The *Connector* package is extended in order to define an ambient architectural element as a specialized type of connector (see Figure 70). As a result, the *Ambient* metaclass is introduced by inheriting from the *Connector* metaclass. The connector inherits all the properties of a connector. As a result, it inherits the invariant called *coordinationAspect* that specifies that a connector must import an aspect whose concern is *coordination*.

Moreover, all ambients have a unique Coordination Aspect. The Coordination Aspect of ambients is called ACoordination. As a result, the *Ambient* metaclass has an invariant (constraint) called *ambientCoordination* that specifies that an ambient only has a Coordination Aspect and its name is *ACoordination*.

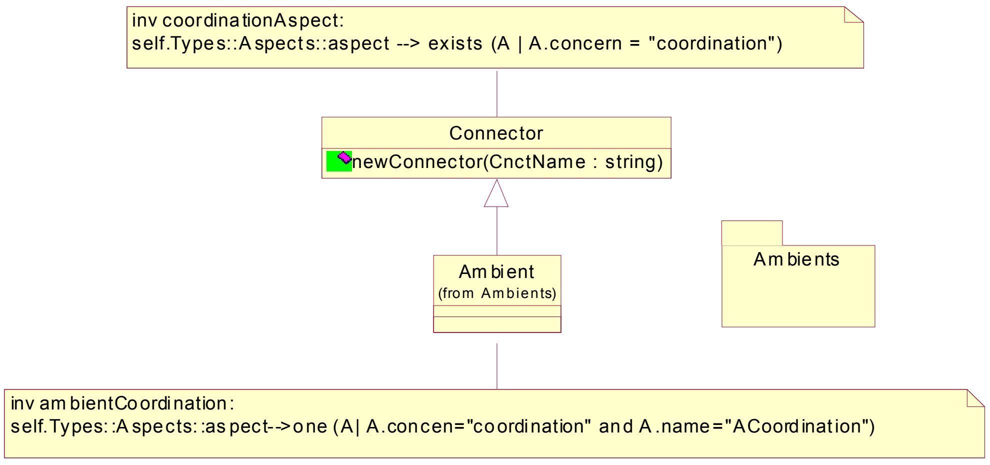

inv coordinationAspect:
self.Types::Aspects::aspect --> exists (A | A.concern = "coordination")
Connector
newConnector(CnctName : string)
Ambient
(from Ambients)
Ambients
inv ambientCoordination:
self.Types::Aspects::aspect-->one (A| A.concen="coordination" and A.name="ACoordination")

Figure 70. Connector Package of the Ambient-PRISMA metamodel

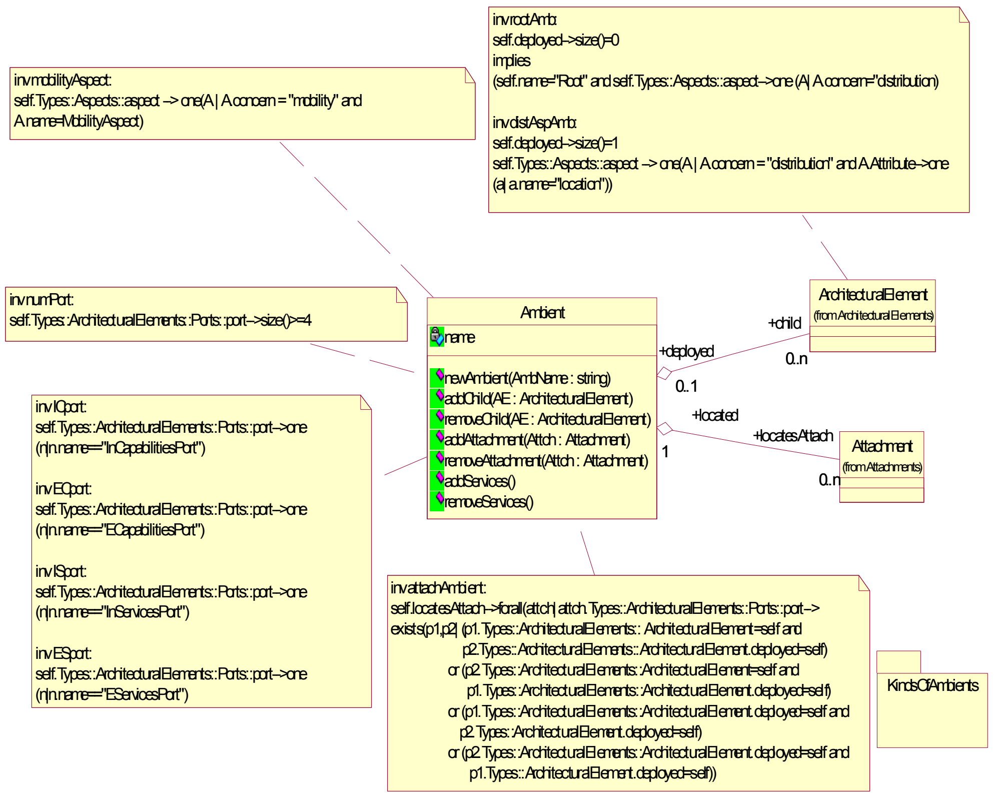
In the PRISMA metamodel, connectors were constrained to only be attached to two different components (see section 4.2.4). However, this constraint cannot be applied in Ambient-PRISMA due to the fact that ambients can be attached to a connector, a component, or another ambient. Since ambients are connectors that can be attached to other connectors this constrain is not valid in Ambient-PRISMA. As a result, this constraint has been eliminated.

The package that defines the proper properties of an ambient is a subpackage of the *Connector* package. This subpackage is called *Ambients* and its details are explained in the following.

5.3.2 Ambient Package

Ambients are connectors that separate what is in a boundary from what is out. The *Ambient* package defines the commonalities and unique properties of the three kinds of ambients. It also defines how an ambient is related to the other concepts of the metamodel.

An ambient locates architectural elements (which can be components, connectors or other ambients), attachments that connect an ambient with architectural elements located in it, and attachments that connect architectural elements of an ambient. As a result, the *Ambient* metaclass has two aggregation relationships: one with the *ArchitecturalElement* metaclass and the other with the *Attachments* metaclass (see Figure 71). The aggregation between the *Ambient* metaclass and the *ArchitecturalElement* metaclass has the *child* role and the *deployed* role. The *child* role indicates that an ambient can have from 0 to many architectural elements located in it. The *deployed* role indicates that an architectural element can be located in 0 or in only one ambient. The aggregation between the *Ambient* metaclass and the *Attachments* metaclass has the *located* role and the *locatesAttach* role. The *locatesAttach* role indicates that an ambient can have 0 or many attachments located in it. The *located* role indicates that an attachment can only be located in one ambient.



```
inv mobilityAspect:
self.Types::Aspects::aspect -> one(A | A.concern = "mobility" and
A.name=MobilityAspect)

inv rootAmb:
self.deployed->size()=0
implies
(self.name="Root" and self.Types::Aspects::aspect->one (A| A.concern="distribution)

inv distAspAmb:
self.deployed->size()=1
self.Types::Aspects::aspect -> one(A | A.concern = "distribution" and A.Attribute->one
(a| a.name="location"))

inv numPort:
self.Types::ArchitecturalElements::Ports::port->size()>=4

inv ICport:
self.Types::ArchitecturalElements::Ports::port->one
(n|n.name=="InCapabilitiesPort")

inv ECport:
self.Types::ArchitecturalElements::Ports::port->one
(n|n.name=="ECapabilitiesPort")

inv ISport:
self.Types::ArchitecturalElements::Ports::port->one
(n|n.name=="InServicesPort")

inv ESport:
self.Types::ArchitecturalElements::Ports::port->one
(n|n.name=="EServicesPort")

inv attachAmbient:
self.locatesAttach->forall(attch| attch.Types::ArchitecturalElements::Ports::port->
exists(p1,p2| (p1.Types::ArchitecturalElements:: ArchitecturalElement=self and
                 p2.Types::ArchitecturalElements::ArchitecturalElement.deployed=self)
             or (p2.Types::ArchitecturalElements::ArchitecturalElement=self and
                  p1.Types::ArchitecturalElements::ArchitecturalElement.deployed=self)
             or (p1.Types::ArchitecturalElements::ArchitecturalElement.deployed=self and
                p2.Types::ArchitecturalElement.deployed=self)
             or (p2.Types::ArchitecturalElements::ArchitecturalElement.deployed=self and
                  p1.Types::ArchitecturalElement.deployed=self))
```

Ambient
name
newAmbient(AmbName : string)
addChild(AE : ArchitecturalElement)
removeChild(AE : ArchitecturalElement)
addAttachment(Attch : Attachment)
removeAttachment(Attch : Attachment)
addServices()
removeServices()
+deployed
0..1
+child
0..n
ArchitecturalElement
(from ArchitecturalElements)
+located
1
+locatesAttach
0..n
Attachment
(from Attachments)
KindsOfAmbients

Figure 71. Ambient Package of the Ambient-PRISMA metamodel

The *Ambient* metaclass has associated a set of OCL invariants (constraints) to completely model the properties of ambients. In the following they are explained:

- ➢ The *mobilityAspect* invariant specifies that an ambient must import an aspect whose concern is mobility and the name of this aspect is *MobilityAspect*.
- ➢ The *numPort* invariant specifies that an ambient must have at least four ports.
- ➢ The *ICport* invariant specifies that an ambient must have a port called InCapabilitiesPort.

- The *ECport* invariant specifies that an ambient must have a port called ECapabilitiesPort.
- The *ISport* invariant specifies that an ambient must have a port called InServicesPort.
- The *ESport* invariant specifies that an ambient must have a port called EServicesPort
- The *attachAmbient* invariant specifies that attachments that are located in an ambient are those that connect the ambient with the architectural elements located in its boundary or those that connect two architectural elements that are located in its boundary. Elements of different ambients cannot have attachments.

In addition, the *Ambient* package includes some OCL invariants associated to the *ArchitecturalElement* metaclass. These invariants specify the following:

- The *rootAmb* invariant specifies that the only architectural element that is not deployed (or located) in any ambient (the deployed role is equal to 0) is called the Root and that Root must import an aspect whose concern is distribution
- The *distAspAmb* invariant specifies that all architectural elements that are deployed in an ambient must import an aspect whose concern is distribution and that this aspect must have an attribute called location.

The *Ambient* metaclass has an attribute called name which specifies the name of an ambient and seven services. The *newAmbient* service creates a new ambient by providing the name of the ambient as a parameter. The *addChild* service adds a new architectural element in the boundary of an ambient. The *removeChild* service removes an architectural element that is located in the boundary of an ambient. The *addAttachment* service adds an attachment that connects an ambient to one of its children or an attachment between two children of the same ambient. The *removeAttachments* removes an attachment of located in the boundary of an ambient. The *addServices* updates the set of services of an ambient *InServicesPort* and *EServicesPort* ports when a new child is added to an ambient. The removeServices removes the services of an ambient InServicesPort and EServicesPort port when a child is removed.

In addition, the *Ambient* package includes the *KindsOfAmbients* subpackage that defines the concepts required for the three kinds of ambients.

5.3.3 KindsOfAmbient Package

In Ambient-PRISMA, there are three kinds of ambients: sites, virtuals, and groups. The KindsOfAmbient package defines that the three kinds of ambients inherit the properties of the *Ambient* metaclass (see Figure 72). Each kind of ambient is represented by a metaclass and each one has a new service to create new kinds of ambients. Also, each metaclass has OCL invariants associated to it in order to differentiate each kind of ambient from the other.

The *Site* metaclass has the following OCL invariants:

- The *distSite* invariant specifies that the distribution aspect of a site ambient must have an attribute called *physicalLocation* and that its data type is *loc*.
- The *childSite* invariant specifies that a site ambient must not have a virtual ambient as one of its children.

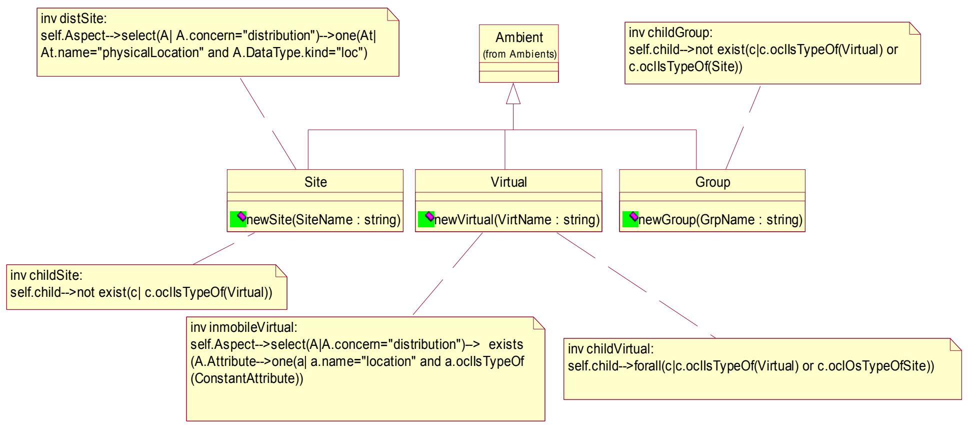

inv distSite:
self.Aspect-->select(A| A.concern="distribution")-->one(At| At.name="physicalLocation" and A.DataType.kind="loc")
Ambient
(from Ambients)
inv childGroup:
self.child-->not exist(c|c.oclIsTypeOf(Virtual) or c.oclIsTypeOf(Site))
Site
newSite(SiteName : string)
Virtual
newVirtual(VirtName : string)
Group
newGroup(GrpName : string)
inv childSite:
self.child-->not exist(c| c.oclIsTypeOf(Virtual))
inv inmobileVirtual:
self.Aspect-->select(A|A.concern="distribution")--> exists (A.Attribute-->one(a| a.name="location" and a.oclIsTypeOf (ConstantAttribute))
inv childVirtual:
self.child-->forall(c|c.oclIsTypeOf(Virtual) or c.oclOsTypeOfSite))

Figure 72. The *KindsOfAmbients* package of the Ambient-PRISMA metamodel

The *Virtual* metaclass has the following OCL invariants:

- The *inmobileVirtual* invariant specifies that the distribution aspect of a virtual ambient must have a constant attribute called *Location*. This invariant indicates that virtual ambients are not mobile architectural elements (their location does not change).
- The *childVirtual* invariant specifies that a virtual ambient must only have site ambients or virtual ambients as children.

The *Group* metaclass has the following OCL invariants:

- The *childGroup* invariant specifies that a group ambient must not have a virtual ambient or a site ambient as one of its children.

5.3.4 Systems Package

All ports of a system have to be in the same ambient. Ports of a system are those that have a binding associated to them. As a result, there is a constraint that all ports of architectural elements associated to the same system must have in common the same ambient.

5.3.5 Attachments Package

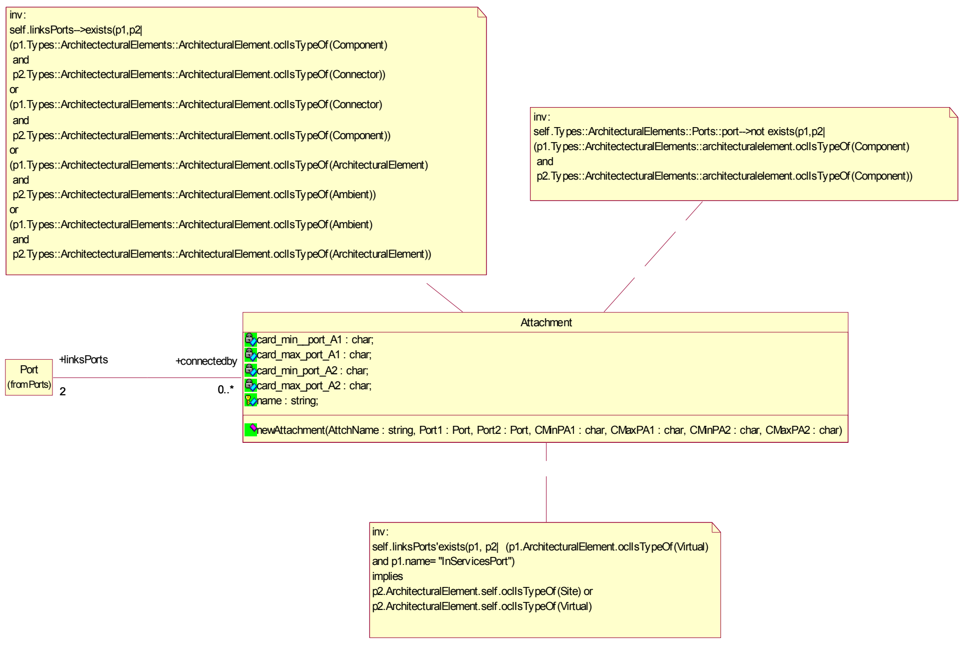
The *Attachments* package of the PRISMA metamodel defines an attachment as a communication channel between a port of a component and a port of a connector (see section 4.2.5). The *Attachments* package in Ambient-PRISMA is extended in order to include attachments that can also connect ambients with other architectural elements. This extension has been performed extending an OCL constraint and including three new ones. In addition, the names of the attributes of the *Attachment* metaclass have been changed. As a result, the *Attachment* package becomes as shown in Figure 73.

The association between the *Attachment* metaclass and the *Port* metaclass establishes that an attachment must connect two ports. However, the *Attachment* metaclass has two OCL constraints that determine which architectural elements ports can be attached. The OCL constraints specify the following:

<< One of the ports of an attachment must belong to a component and that the other must belong to a connector OR one of the ports must belong to an ambient and that the other port must belong to any architectural element>>

<<An attachment cannot connect two ports of a component >>

<<The *InServicesPort* port of an ambient can only be attached to a port of a Virtual ambient or a Site ambient >>


Port
(from Ports)
+linksPorts
2
+connectedby
0..*
Attachment
card_min__port_A1 : char;
card_max_port_A1 : char;
card_min_port_A2 : char;
card_max_port_A2 : char;
name : string;
newAttachment(AttchName : string, Port1 : Port, Port2 : Port, CMinPA1 : char, CMaxPA1 : char, CMinPA2 : char, CMaxPA2 : char)

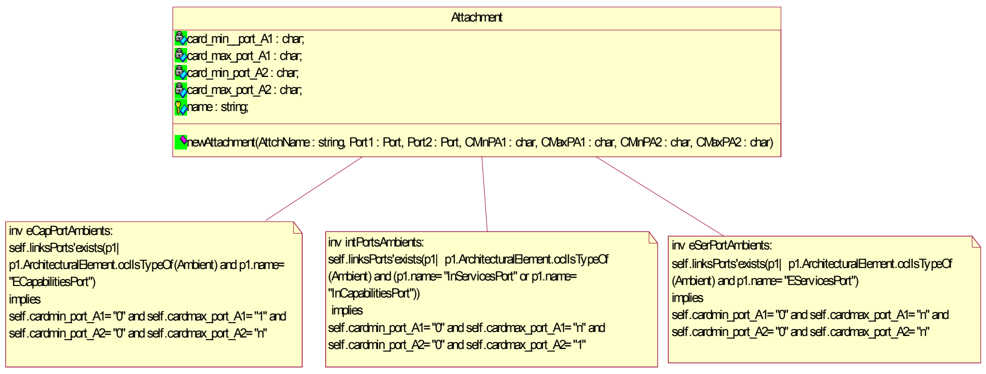
```
inv:
self.linksPorts-->exists(p1,p2|
(p1.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(Component)
 and
 p2.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(Connector))
or
(p1.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(Connector)
 and
 p2.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(Component))
or
(p1.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(ArchitecturalElement)
 and
 p2.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(Ambient))
or
(p1.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(Ambient)
 and
 p2.Types::ArchitectecturalElements::ArchitecturalElement.oclIsTypeOf(ArchitecturalElement))
```

```
inv:
self.Types::ArchitecturalElements::Ports::port-->not exists(p1,p2|
(p1.Types::ArchitectecturalElements::architecturalelement.oclIsTypeOf(Component)
 and
 p2.Types::ArchitectecturalElements::architecturalelement.oclIsTypeOf(Component))
```

```
inv:
self.linksPorts'exists(p1, p2|  (p1.ArchitecturalElement.oclIsTypeOf(Virtual)
and p1.name= "InServicesPort")
implies
p2.ArchitecturalElement.self.oclIsTypeOf(Site) or
p2.ArchitecturalElement.self.oclIsTypeOf(Virtual)
```

Figure 73. The *Attachments* package of the Ambient-PRISMA metamodel

The *Attachment* metaclass has an attribute called name for storing the name of an attachment. The *Attachment* metaclass also has four more attributes to specify the attachment communication pattern i.e., the instantiation pattern of the attachment. It is necessary to constrain how many instances of the attachment can be attached to the port of each architectural element instance that an attachment connects. The *card_min_port_A1* and *card_min_port_A2* attributes specify the minimum number of attachment instances that must be connected to one instance of A1 and A2 architectural elements through their ports, respectively. The *card_max_port_A1* and *card_max_port_A2* attributes specify the maximum number of attachment instances that must be connected to one instance of A1 and A2 architectural elements through their ports, respectively.


Attachment
card_min__port_A1 : char;
card_max_port_A1 : char;
card_min_port_A2 : char;
card_max_port_A2 : char;
name : string;
newAttachment(AttchName : string, Port1 : Port, Port2 : Port, CMnPA1 : char, CMaxPA1 : char, CMnPA2 : char, CMaxPA2 : char)

```
inv eCapPortAmbients:
self.linksPorts'exists(p1|
p1.ArchitecturalElement.oclIsTypeOf(Ambient) and p1.name=
"ECapabilitiesPort")
implies
self.cardmin_port_A1= "0" and self.cardmax_port_A1= "1" and
self.cardmin_port_A2= "0" and self.cardmax_port_A2= "n"
```

```
inv intPortsAmbients:
self.linksPorts'exists(p1|  p1.ArchitecturalElement.oclIsTypeOf
(Ambient) and (p1.name= "InServicesPort" or p1.name=
"InCapabilitiesPort"))
 implies
self.cardmin_port_A1= "0" and self.cardmax_port_A1= "n" and
self.cardmin_port_A2= "0" and self.cardmax_port_A2= "1"
```

```
inv eSerPortAmbients:
self.linksPorts'exists(p1|  p1.ArchitecturalElement.oclIsTypeOf
(Ambient) and p1.name= "EServicesPort")
implies
self.cardmin_port_A1= "0" and self.cardmax_port_A1= "n" and
self.cardmin_port_A2= "0" and self.cardmax_port_A2= "n"
```

Figure 74. OCL constraints for the Attachment metaclass

In Ambient-PRISMA, an architectural element is deployed in only an ambient. An ambient also has four predefined ports. Therefore, the attachment communication pattern of an ambient are predefined. As a result, the *Attachment* metaclass has three OCL invariants to predefine the values of the cardinality attributes. In the following, these are explained (see Figure 74):

- The *intPortsAmbients* invariant specifies that the attachment that connects the *InServicesPort* port or *InCapabilitiesPort* port of an ambient to a port of a child architectural element, the values of the cardinalities to cardmin_port_A1=0, cardmax_port_A1=n, cardmin_port_A2=0, and cardmax_port_A2=1.
- The *eSerPortAmbients* invariant specifies that the attachment that connects the *EServicesPort* port of an ambient to a port of a child architectural element, the values of the cardinalities to cardmin_port_A1=0, cardmax_port_A1=n, cardmin_port_A2=0, and cardmax_port_A2=n.
- The *eCapPortAmbients* invariant specifies that the attachment that connects the *ECapabilitiesPort* port of an ambient to a port of a child architectural element, the values of the cardinalities to cardmin_port_A1=0, cardmax_port_A1=1, cardmin_port_A2=0, and cardmax_port_A2=n.

5.3.6 AmbientPRISMAArchitecture Package

The Ambient PRISMA Architecture package defines how a software architecture with ambients is defined. The *PRISMAArchitecture* metaclass that represents a software architecture (see section 4.2.7) has an aggregation relationship with the *Ambient* metaclass (see Figure 75). An architecture in Ambient-PRISMA is defined by at least an ambient. Since an ambient is reusable, it can be used by more than one software architecture. In this way, an ambient is a first-class entity of an Ambient-PRISMA software architecture

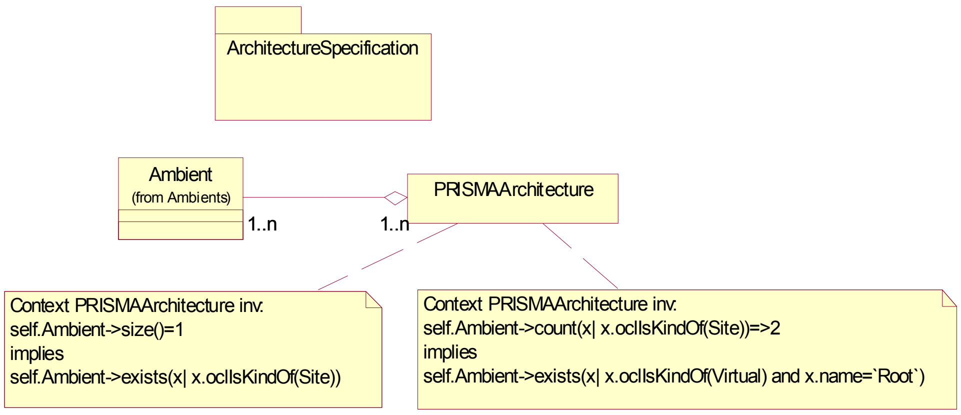

ArchitectureSpecification
Ambient
(from Ambients)
1..n
1..n
PRISMAArchitecture
Context PRISMAArchitecture inv.
self.Ambient->size()=1
implies
self.Ambient->exists(x| x.oclIsKindOf(Site))
Context PRISMAArchitecture inv.
self.Ambient->count(x| x.oclIsKindOf(Site))=>2
implies
self.Ambient->exists(x| x.oclIsKindOf(Virtual) and x.name=`Root`)

Figure 75. The *AmbientPRISMAArchitecture* package of the Ambient-PRISMA metamodel

The *PRISMAArchitecture* metaclass has two constraints associated to it in order to ensure that a model is correctly defined. Their meaning is the following:

<< If an architectural model only defines one ambient, then this ambient must be a Site ambient. >>

<<An architectural model that has two or more Site ambients, must have a Virtual ambient called Root. >>

5.3.7 Data Types Package

The different data types that PRISMA supports are included in the *DataTypes* package (see Figure 76). The *DataType* metaclass has an attribute called *kind*. The *kind* attribute can have different values, e.g., natural, integer, double, etc. A new kind of data type called *loc* is included in the Ambient-PRISMA *DataType* metaclass. *loc* data type is needed in order to model the different physical locations that form a specific distributed system.

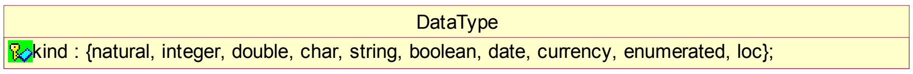

DataType
kind : {natural, integer, double, char, string, boolean, date, currency, enumerated, loc};

Figure 76. The package *DataTypes* of Ambient-PRISMA

5.4 Ambient-PRISMA Language

In this section, the new constructs of an ambient included to the AOADL are explained. These constructs include specifications of predefined, interfaces, aspects, ports, and weavings of ambients that the user of Ambient-PRISMA does not have to specify. New extensions are also included to the AOADL are presented.

5.4.1 Architectural Model

Figure 77 shows the templates of the Ambient-PRISMA architectural model in the AOADL. The specification of an architectural model starts with the reserved word *Architectural_Model* and ends with the reserved word *End_Architectural_Model*. Then, a name is given to the architectural model. In the case of the architectural model Afterwards, each first-class entity is specified. First the virtual ambients block can be specified. A site ambient block must be specified. A group ambient block can be optionally specified. Components and connectors are specified. Systems (complex components) can be optionally specified in the architectural model. The attachments that connect components and connectors are specified.

```
<architectural_model> ::= Architectural_Model <model_name>
```

[<virtual_ambient_block>]

<site_ambient_block>

[<group_ambient_block>]

<component_block>

<connector_block>

[<system_block>]

<attachment_block>

End_Architectural_Model <model_name>**‘;’**

Figure 77. Architectural Model template in Ambient-PRISMA

5.4.2 ICall Interface

The *ICall* interface is an interface for generic services. The interface consists of the *call* service. The call service has three parameters: *Name*, *ParamsList* and *ParamType*. The *Name* parameter stores the name of a service that needs to be invoked or provided by an architectural element located in an ambient. The *ParamsList* parameter is a list that stores the parameters of the service. The *ParamType* parameter is a list that stores the kind of each parameter (if it is input or output). *ParamType* stores the kind of parameters in the same order as the parameters appear in the original service. It stores true if it is an input parameter and false if it is an output parameter.

```
Interface ICall

   call(input Name: string, input ParamsList[]: Parameters,
        input ParamType[]: boolean);

End_Interface ICall
```

Figure 78. Specification of the ICall Interface

The Coordination aspect uses the *ICall* interface. The *call* service signature allows the original service to be reconstructed after it is processed in the Coordination

Aspect. In this way, the Coordination Aspect can be specified in a generic way specifying the *ICall* interface.

5.4.3 ICapability Interface

This interface specifies the services that the Mobility Aspect offers (see Figure 79). These services have been incorporated in order to provide AC capabilities (exit, the in capability of AC and enter, the out capability of AC), services that are needed in order to manage a consistent state of the distributed software architecture (*startMovement*, *finishMovement*, *changeLocation* and *accept*), *newAmbientInst* service which creates new subambients of an ambient and *copy* service that replicates architectural elements of an ambient.

```
Interface ICapability

   startMovement(input Name:string,
                 output CommunicationList[]:Attachment);
   exit (input Name: string, input Parent:string );
   enter (input Name: string, input NewAmbient: string);
   finishMovement(input Name, input CommunicationList[]: Attachment);
   changeLocation(input Name: string, output NewLocation: string);
   accept(input NewAmbient: string, input Caller:string,
          input Child:string, input Type: string,
          input AttachmentsList[]:Attachments,
          output AcceptanceOK: boolean);
   newAmbientInst(input Name: string, input AmbientType: string);
   copy(input Name: string, output Ok:boolean);

End_Interface ICapability
```

Figure 79 .Specification of the ICapability Interface

5.4.4 IGetLocation Interface

An ambient publishes in an interface called *IGetLocation* a service called *getLocation* in order to inform the name of its parent ambient (see Figure 61). The *getLocation* service has one parameter called *Location*. The *Location* parameter is an output parameter that returns the name of the parent ambient.

```
Interface IGetLocation
 getLocation(output Location: string);
End_Interface IGetLocation
```

Figure 80. Specification of the IGetLocation Interface

5.4.5 ACoordination Aspect

The Coordination Aspect of an ambient allows architectural elements located in an ambient boundary to request or provide services from or to architectural elements located in other ambients. The specification of this aspect cannot be modified by the user. Also, this aspect is reused by all ambients.

The Coordination aspect is specified in Figure 81. The specification follows the PRISMA bnf of an aspect that is shown in the APPENDIX A. The Coordination Aspect is called *ACoordination* and uses the *ICall* interface (see Figure 78). As a result, in its *Services* section it specifies the *begin* and the *end* services as well as the *call* service. The call service is specified as an *in* and *out* service. This is due to the fact that the *ACoordination* aspect can receive services that are required (out) or provided (in). The aspect has two played_roles: INTERIOR and EXTERIOR. Both of them specify when and how the service of the *ICall* interface can be required or provided. The *INTERIOR* played_role is for receiving (*?*) and sending (*!*) services of architectural elements located in the boundary of an ambient. The *INTERIOR* played_role specifies that it can receive or send (indicated with + symbol) *call* services. The *EXTERIOR* played_role is for receiving and sending services of architectural elements located outside the boundary of an ambient. Also, the *EXTERIOR* played_role specifies that it can receive or send call services.

```
Coordination Aspect ACoordination using ICall

Services
   begin();
   in/out call(input Name:string, input ParamsList[]: Parameters,
               input ParamType[]: bool);
   end;

Played_Role
  INTERIOR for ICall::= call?(input Name, input ParamsList[],
                              input ParamType[])
                         +
                        call!(input Name, input ParamsList[],
                              input ParamType[]);
  EXTERIOR for ICall::= call?(input Name, input ParamsList[],
                              input ParamType[])
                         +
                         call!(input Name, input ParamsList[],
                               input ParamType[]);

Protocol
  ACOOR ::= begin→ ACOOR1;
  ACOOR1 ::= INTrecEXTsnd+INTsndEXTrec+EXTrecINTsnd+EXTsndINTrec+ end;
  INTrecEXTsnd ::= (INTERIOR.call?(input Name, input ParamsList[],
                                   input ParamType[])
                    →
                    EXTERIOR.call!(input Name, input ParamsList[],
                                   input ParamType[]))→ACOOR1;
  INTsndEXTrec ::= (INTERIOR.call!(input Name, input ParamsList[],
                                   input ParamType[])
                    →
                   EXTERIOR.call?(input Name, input ParamsList[],
                                   input ParamType[]))→ACOOR1;
  EXTrecEXTsnd ::= (EXTERIOR.call?(input Name, input ParamsList[],
                                   input ParamType[])
                     →
                    INTERIOR.call!(input Name, input ParamsList[],
                                   input ParamType[]))→ACOOR1;

  EXTsndINTrec ::= (EXTERIOR.call!(input Name, input ParamsList[],
                                   input ParamType[])
                     →
                    INTERIOR.call?(input Name, input ParamsList[],
                                   input ParamType[]))→ACOOR1;

End_Coordination Aspect ACoordination
```

Figure 81. Specification of the ACoordination Aspect

The *ACoor* protocol of the *ACoordination* aspect glues the *INTERIOR* and the *EXTERIOR* played roles. Basically, the protocol specifies that once the *begin* service is executed, i.e. the aspect is instantiated, the aspect executes the *ACOOR1* process.

The *ACOOR1* process specifies a nondeterministic choice among the execution of four subprocesses: *INTrecEXTsnd*, *INTsndEXTrec*, *EXTrecINTsnd*, and *EXTsndINTrec* and the *end* service. Each subprocess specifies the possible combination of the reception or sending of a call service from a played role and the reception and sending of a call service from another played role. For example, the *INTrecEXTsnd* subprocess specifies that when a *call* service of the *INTERIOR* played role is received, the aspect sends a *call* service by the *EXTERIOR* played role.

5.4.6 MobilityAspect aspect

The *MobilityAspect* aspect of an ambient provides architectural elements located in its boundary the needed services so that they are capable of moving. The specification of this aspect cannot be modified by the user. Also, this aspect is reused by all ambients. The *MobilityAspect* aspect uses the *ICapability* interface which is specified in Figure 79 in order to specify the behaviour of the services which it publishes (see Figure 82).

```
1   Mobility Aspect MobilityAspect using ICapability
       …. …
    Services
        begin();
         in startMovement(input Name:string,
                          output CommunicationList[]: Attachment);
         in finishMovement(input Name:string,
                           input CommunicationList[]: Attachment);
        end;

2   TRANSACTIONS in exit(Requested: string, Ambient: string):
      EXIT := out isChild(input Requested, output isChildOK)
                   →EXIT1;
      EXIT1::= {isChildOK==true & self.Name==Ambient}
                 getParent(output Parent)→EXIT2;
      EXIT2::= moving (input Requested, input Parent);

3   TRANSACTIONS in enter(Requested: string, NewAmbient: string):
      ENTER := out areChildren(input Requested, input Ambient,
                               output areChildrenOK)→ENTER1;
      ENTER1::= {areChildrenOK==true} in moving(Requested,
                                            NewAmbient);

4   TRANSACTIONS in moving(Requested: string, NewAmbient: string):
      MOVING := out movingInf(input Requested, input self.Name,
                               input Requested, output Type,
                               output AttachmentList[])→ MOVING1;
      MOVING1::= out accept(input NewAmbient, input Type,
                            input Requested, input AttachmentsList[],
                            output AcceptanceOK)→ MOVING2;
      MOVING2::= {Acceptance0K==true}
                 out modifyAttachment(Requested)→MOVING3;
      MOVING3::= out removeAttachments(Requested)→MOVING4;
      MOVING4::= out removeChild(Requested);

5  TRANSACTIONS in accept(input NewAmbient:string, input Caller:
    string,
                           input Child:string, input Type:string,
                           input AttachmentsList[]:Attachments,
                           output AcceptanceOK: boolean):
      ACCEPT::= out addChild(input Type, input Child,
                             input AttachmentsList[], output AddedOK)
                    {AcceptanceOK==AddedOK}→ ACCEPT1;
      ACCEPT1::=out changeLocation(Child, self.Name);

6  Preconditions
          in accept(input NewAmbient, input Caller,
                    input Child, input Type,
                    input AttachmentsList[],
                    output AcceptanceOK)
             if
               {self.Name==NewAmbient};

7  Played_Role
      INTERIOR for ICapability::= accept?(input NewAmbient,
                                          input Caller,
                                          input Child,
                                          input Type,
                                          input AttachmentsList[],
                                          output AcceptanceOK)
                                  +
                                  accept!(input NewAmbient,
                                          input Caller,
                                          input Child,
                                          input Type,
                                          input AttachmentsList[],
                                          output AcceptanceOK)
                                  +
                                  enter?(input Requested,
                                         input NewAmbient)
                                  +
                                  exit?(input Requested,
                                        input Ambient)
                                  +
                                  startMovement?(input Name,
                                                 output
    CommunicationList[])
                                  +
                                  finishMovement?(input Name,
                                                  input
   CommunicationList[])
                                  +
                                  changeLocation!(input Child,
                                                  input self.Name);

8  EXTERIOR for ICapability::= accept?(input NewAmbient, input
   Caller,
                                       input Child, input Type,
                                       input AttachmentsList[],
                                       output AcceptanceOK)
                                  +
                                  accept!(input NewAmbient, input
   Caller,
                                          input Child, input Type,
                                          input AttachmentsList[],
                                          output AcceptanceOK) ;

      …   …
   End_Mobility Aspect MobilityAspect
```

Figure 82. Specification of the *MobilityAspect* aspect

The services *startMovement* and *finishMovement* (see Figure 82, section 1) are auxiliary services that the aspect uses in order to make special operations at the beginning and at the end of a mobility process. The *startMovement* service is invoked by an architectural element in order to indicate to an ambient that is going to invoke a chain of capabilities (such as enter, exit). The *finishMovement* service is invoked by an architectural element in order to indicate to an ambient that it has finished invoking the chain of capabilities, i.e., it has finalized the mobility process. In this way, an ambient has services in order to manage the beginning and the end of a mobility process.

The *exit (Requested: string, Ambient: string)* transaction (see Figure 82, section 2) is in charge of serving an exit requested by architectural elements of an ambient. This transaction checks whether the architectural element that needs to exit is a child of the aspects ambient or not. This check is performed through the *isChild* service. The *ischild* service returns true through the *isChildOK* output parameter if the requested architectural element is a child of the ambient. If *ischild* returns true, and the architectural elements needs to exit the ambient which is currently its parent then the transaction proceeds to execute *getParent* service. The *getParent* service returns in the *Parent* parameter the name of the parent ambient of the *MobilityAspect* ambient. This is needed in order to indicate the destination of the exiting architectural element. Then the *moving* transaction is executed in order to move the architectural element.

The *enter (Requested: string, NewAmbient: string)* transaction (see Figure 82, section 3) is in charge of serving an enter requested by an architectural element of an ambient that needs to enter a sibling ambient. This transaction checks whether both the mobile architectural element and the destination ambient are siblings, i.e., both are children of the requested ambient. If this is fulfilled, the moving transaction is executed in order to move the architectural element.

The *moving (Requested: string, NewAmbient: string)* transaction (see Figure 82, section 4) is in charge of performing all steps needed in order to reconfigure the architecture in the origin ambient. Initially, the information about the attachments connected to the mobile architectural element is obtained. This information is encapsulated in a list in order to enable their recreation in the destination ambient. The accept service of the destination ambient is invoked and the information of the attachments is sent. In any case, the origin ambient is attached to the destination ambient. If the destination ambient accepts to receive the mobile architectural element, the origin ambient concludes the mobility process by means of modifying the attachments. Then, the origin ambient removes all the attachments that connect the mobile architectural element to the parent ambient or to other architectural elements in the ambient, creates attachments between its *InServicesPort* port and the elements that were connected to the mobile architectural element and, finally, deletes the architectural element instance.

The accept transaction (see Figure 82, section 5) is invoked by an origin ambient to a destination ambient in order to request to the destination ambient to accept a new architectural element. This transaction adds a new element to its list of elements, correctly creates the needed attachments thanks to the list of information saved about the attachments that it receives, and finally requests the *changeLocation* service of the new element, so that it updates its location stored in the distribution aspect. The accept transaction is in charge of performing all steps needed in order to reconfigure the architecture in the destination ambient. This transaction has a precondition associated to it (see Figure 82, section 6). The precondition indicates that the accept transaction is only executed if the service is directed to the ambient of the *MobilityAspect* aspect.

Finally, the INTERIOR (see Figure 82,, section 7) and the EXTERIOR (see Figure 82, section 8) played_roles are described. The INTERIOR played role allows an ambient to offer the services of the ICapabily interface to the elements located in its boundary. The EXTERIOR played role allows the ambient to offer and request the accept service of the ICapability interface to the elements located outside the boundary of the ambient. The architectural elements located outside the boundary of an ambient cannot request other services such as the exit or the enter.

5.4.7 Distribution Aspect

Each distribution aspect must save the parent ambient of the ambient that imports it and provides a service in order to allow other aspects of the ambient to consult its current location. Therefore, a distribution aspect uses the *IGetLocation* interface specified in Figure 80. Figure 83, Figure 84, and Figure 85 show the specifications of possible distribution aspects. These aspects store the name of the parent ambient of the ambient that imports it through the *location* attribute. The aspects aspect specifies that the *getLocation* service is an 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*).

Figure 83 shows the minimum specification that a distribution aspect of a Site ambient can contain. A distribution aspect of a Site must include an attribute called *physicalLocation* of type *loc*. The *physicalLocation* attribute stores the physical location that the site represents. This attribute is a *NOT NULL* attribute that always has to have a value. As a result, the begin service must have a parameter that gives a value to *phyicalLocation* when the aspect starts executing.

```
Distribution Aspect SiteAmbient using IGetLocation
 Attributes
  …
  location : string NOT NULL;
  physicalLocation: loc;

  …
 Services
  …

  begin(input ParentAmbient: string, input PhysicalLocation: loc)
     Valuations
      [begin (ParentAmbient, PhysicalLocation)]
                          location := ParentAmbient,
                          physicalLocation:=PhysicalLocation;

  in/out getLocation( output Location:string)
     Valuations
         [in getLocation(output Location)] Location := location;

 Protocol
  DIST:= begin→ DIST1;
  DIST1:= (getLocation?(Location)→ getLocation!(Location))+ end;

End_Distribution Aspect SiteDist
```

Figure 83. A distribution aspect of a Site ambient

Figure 84 shows the minimum specification of a distribution aspect that is importd by a Group Site. It can be observed, that the aspect has the location attribute as *NOT NULL*.

```
Distribution Aspect GroupAmbient using IGetLocation
 Attributes
  …
  location : string NOT NULL;

  …
 Services
  …

  begin(input ParentAmbient: string)
     Valuations
```

```
      [begin (ParentAmbient)]
                          location := ParentAmbient,


  in/out getLocation( output Location:string)
     Valuations
         [in getLocation(output Location)] Location := location;

 Protocol
  DIST:= begin→ DIST1;
  DIST1:= (getLocation?(Location)→ getLocation!(Location))+ end;

End_Distribution Aspect GroupDist
```

Figure 84. A distribution aspect of a Group ambient

Figure 85 shows the minimum specification of a distribution aspect imported by the Root ambient. It can be observed that the location attribute always has NULL value. This is due to the fact that a *Root* ambient does not have a parent ambient. As a result, the *begin* service of the aspect can have no parameters and no valuation.

```
Distribution Aspect RootAmbient using IGetLocation
 Attributes
  Constant
  location : string (NULL);


  …
 Services
  …

  begin()


  in/out getLocation( output Location:string)
     Valuations
         [in getLocation(output Location)] Location := location;

 Protocol
  DIST:= begin→ DIST1;
  DIST1:= (getLocation?(Location)→ getLocation!(Location))+ end;

End_Distribution Aspect RootDist
```

Figure 85. A distribution aspect of the Root ambient

5.4.8 Ambients

An ambient is specified with the set of aspects it is formed of, aspect weavings and the ports. Figure 86, shows the template of ambients. All the kinds of ambients are specified in the same way. The only difference is the reserved words that precede and end their specifications. These reserved words are *Virtual…End_Virtual*, *Site…End_Site,* and *Group…End_Group* for the specification of virtual, site and group ambients, respectively.

```
<Virtual> ::= Virtual <virtaul_name>
                         <aspects_importation_seq>
                         [<weavings>]
                         <ports>
                         <creation>
                         <destruction>
              End_Virtual <virtual_name>';'
<Site> ::= Site <site_name>
                         <aspects_importation_seq>
                         [<weavings>]
                         <ports>
                         <creation>
                         <destruction>
           End_Site <site_name>';'


<Group> ::= Group <group_name>
                         <aspects_importation_seq>
                         [<weavings>]
                         <ports>
                         <creation>
                         <destruction>
              End_Group <group_name>';'
```

<aspects_importation> ::= <concern> **Aspect Import** <aspect_name>

<creation> ::= **new**‘**(**‘ [<param_service_list>]‘**)**’ ‘**{**‘ <start_aspects_seq> ‘**}**’

<destruction> ::= destroy‘**(**‘ ‘**)**’ ‘{‘ <stop_aspects_seq> ‘**}**’

<start_aspects> ::= <aspect_name>‘**.**’**begin**‘**(**‘ [<parameter_name_list>]‘**)**’

<stop_aspects> ::= <aspect_name>‘.’end‘(‘ ‘)’

Figure 86. Specification Template of Ambients

Figure 87 shows the specification of a site ambient called *HostSite*. The ambient imports the *MobilityAspect* aspect, the *ACoordination* aspect, and the *ADist* aspect. The predefined ports of the ambient are specified with their played roles. The predefined weaving is also specified.

```
Ambient_Site type HostSite
Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Ports
     InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
     ECapabilitiesPort: ICapability Played_Role Mobile.Child;
     EServicesPort: ICall Played_Role ACoordination.Client;
     InServicesPort: ICall Played_Role ACoordination.Server ;

 End_Ports
 Weavings
     ADist.getLocation(Location)instead
                               MobilityAspect.getParent(Parent);

End Ambient_Site type HostSite;
```

Figure 87. Specification of HostSite ambient

It can be noticed, that the template of the ambient does not include the architectural elements that it locates or the attachments. It is assumed that ambients of an architectural model can locate all architectural elements.

5.4.9 Configuration

The template for specifying configuration is shown in Figure 88. The possible physicalLocations are instantiated by instantiating the loc data type. Then the ambients are instantiated. The root ambient has to be initially instantiated, then the site ambients and finally the group ambients. Each ambient is instantiated indicating their parent ambient with the exception of the Root ambient. Then, components and connectors are instantiated indicating where they are located. Finally, the attachments are instantiated. Figure 89, shows the specification of the configuration of the AuctionConfig architectural model.

```
<architectural_model_configuration> ::=
                Architectural_Model_Configuration  <configuration_name>  '='
                       new  <model_name> '{'<loc_instantiation_seq>
                                              [<virtualAmbient_instantiation_seq>]
                                              <siteAmbient_instantiation_seq>
                                              [<groupAmbient_instantiation_seq>]
                                              <components_instantiation_seq>
                                             [<systems_instantiation_seq>]
                                             <connectors_instantiation_seq>
                                             <attachments_intantiation_seq> '}'
<LOC_instantiation> ::= <loc_name> '=' new
                                    loc'(' [<param_value _list> ] ')
<virtualAmbient_instantiation> ::= <virtualAmbient_instance_name> '=' new
                                    <virtualAmbient_name>'(' [<param_value _list> ] ')
<siteAmbient_instantiation> ::= <siteAmbient_instance_name> '=' new
                                    <siteAmbient_name>'(' [<param_value _list> ] ')
<groupAmbient_instantiation> ::= <groupAmbient_instance_name> '=' new
                                    <groupAmbient_name>'(' [<param_value _list> ] ')
<components_instantiation> ::= <component_instance_name> '=' new
                                    <component_name>'(' [<param_value _list> ] ')
```

<connectors_instantiation> ::= <connector_instance_name> **'=' new** <connector_name> **'('** [<param_value_list>] **')**

<attachments_instantiation> ::= <attachment_instance_name>[1] **'=' new** <attachment_name> **'('**<param_attachment_value>**')'**

<systems_instantiation> ::= <system_instance_name>[1] **'=' new** <system_name> **'('**[<param_service_value_list>**','**] <architectural_element_number_value_list>, [<attachment_number_value_list>'**,'** <binding_number_value_list>] **')'** **'{'** [<start_aspects_seq>] <architectural_elements_instantiation_seq> <attachments_instantiation_seq> <bindings_instantiation_seq> **'}'**

<architectural_element_instantiation > ::= <components_instantiation> | <connectors_instantiation> | <systems_instantiation>

< bindings_instantiation> ::= <binding_instance_name>[1] **'=' new** <binding_name> '(' <param_binding_value> ')'

Figure 88. Specification Template of configurations of ambient-PRISMA architectural models

```
Architectural_Model_Configuration AuctionConf =

New MobileAuction
{
       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)

       Auction1 = new AuctionHouse ("London, King street", 1876",
                                   "1 Jun", "Spanish painting", "800",
                                   "AuctionSite");
         Customer1 = new Customer("ClientSite");
         Procurement1 = new Procurement ("ClientSite");
         Bidder1 = new Bidder("painting", "3 Jul", "ClientSite");
         AgentCustCnct1 = new AgentCustCnct("ClientSite");
         AuctionCnct1 = new AuctionCnct("AuctionSite");
         AttchAuct1Cnct = new AttchAuctCnct (AuctionCnct1,
                                             CnctAuctPortBidder,
                                             AuctionHouse1,
                                              BidderAuctPort);
        AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                                    AuctionCnct1, CustPortAuct);
```
Figure 89. Specification of the AuctionConfig configuration

5.5 Conclusions

This chapter introduces Ambient-PRISMA. It presents how the ambient concept has been introduced in PRISMA in order to define architectural models of distributed and mobile software systems. An extension of the PRISMA metamodel and the AOADL have been needed in order to define how an ambient is related with the other architectural elements of PRISMA.

The Ambient-PRISMA is an approach that provides distribution and mobility with the following characteristics:

- Explicit notation for location using an ambient architectural element
- The unit of Mobility in Ambient-PRISMA are Components, Connectors and ambients.
- Mobility is supported by the reconfiguration of the software architecture.
- The location attribute is only changed by an ambient.
- Migration Decision can be passive and autonomous.
- Support distributed coordination
- A mobile element has to define its mobility strategies in its Distribution Aspect.

The work related to PRISMA has produced a set of results that are published in the following publications:

- **Nour Ali**, Carlos Millán, Isidro Ramos, Developing Mobile Ambients using an Aspect-Oriented Software Architectural Model, 8th International Symposium on Distributed Objects and Applications (DOA 2006), **LNCS** Springer Verlang, Montpellier, France, October, 2006.
- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, Mobile Ambients in Aspect-Oriented Software Architectures, IFIP Working

Conference on Software Engineering Techniques (SET 2006), **Springer Series in Computer Science**, Warsaw, Polond, October 17-20, 2006.

- **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí *Introducing Ambient Calculus in Mobile Aspect-Oriented Software Architectures* , Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), **IEEE Computer Society**, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper).
- **Nour Ali**, Isidro Ramos, Jose A. Carsí, *A Conceptual Model for Distributed Aspect-Oriented Software Architectures* , International Conference on Information Technology Coding and Computing (ITCC 2005), **IEEE Computer Society**, Las Vegas, NV, USA, 2005.
- **Nour Ali**, Jennifer Perez, Isidro Ramos, High Level Specification of Distributed and Mobile Information Systems, Second International Symposium on Innovation in Information & Communication Technology (ISSICT 2004), April, Amman, Jordan,2004.
- **Nour Ali**, Josep Silva, Javier Jaén, Isidro Ramos, José Ángel Carsí, Jennifer Pérez, Mobility and Replication Patterns in Aspect-oriented component Based Software Architectures, 15th IASTED International Conference, PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS~PDCS 2003~, November 3-5, Marina del Rey, CA, USA, 2003.
- **Nour Ali**, Jennifer Pérez,Cristóbal Costa, Isidro Ramos,Jose Ángel Carsí, Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes, XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Sitges, Barcelona, Octubre 3-6, 2006.
- **Nour Ali**, Jose Angel Carsi, Isidro Ramos,"Analysis of a Distribution Dimension for PRISMA", Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004), Málaga, 10-12 Noviembre 2004
- **Nour Ali**, Isidro Ramos, "Ambient-PRISMA: Ambients in Distributed and Mobile Aspect-Oriented Software Architectures", V Jornadas de Trabajo DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, 2006.

- **Nour Ali,** Jennifer Perez, Jose Angel Carsi, Isidro Ramos, “Mobility of Objects in PRISMA” , III Jornadas de DYNMAICA, Al Magro, Ciudad Real, 2005.
- **Nour Ali**, Jose Cercos, Isidro Ramos, Patricio Letelier, Jose Angel Carsi, “Distribution in PRISMA”, workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- **Nour H. Ali**, Josep F. Silva, Javier Jaen, Isidro Ramos, Jose Á. Carsí, Jennifer Pérez, “Distribution Patterns in Aspect-Oriented Component-Based Software Architectures”, IV Workshop Distributed Objects, Languages, Methods and Environments (DOLMEN), pp.74-80, Alicante, November, 2003.

CHAPTER 6
Ambient-PRISMANET

The PRISMANET middleware provides an execution platform for PRISMA aspect-oriented software architecture configurations. For example, it contains classes that map the concepts of aspects, weavings, components, ports, and attachments between local instances.

Since Ambient-PRISMA extends PRISMA, PRISMANET has to be extended in order to support the distribution and mobility characteristics of Ambient-PRISMA. Specifically, Ambient-PRISMANET implements the constructs needed to support the ambient primitive of Ambient-PRISMA. Also, it provides the execution of distributed communication and mobility of architectural elements at runtime. Ambient-PRISMANET provides a distributed runtime environment for applications. Each Ambient-PRISMANET middleware of a host can collaborate with other Ambient-PRISMANET middlewares on other hosts (see Figure 90). The runtime environment has a distributed Domain Name Server (DNS) which all middlewares on hosts can access.

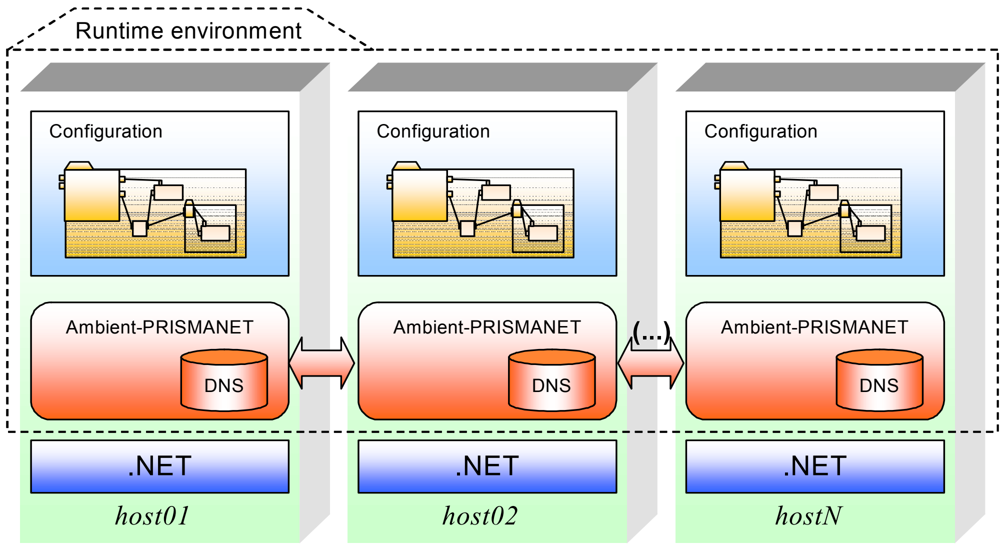


Figure 90 Distributed Run-time Environment of Ambient-PRISMANET

In the following, we present how the ambient concept has been implemented in Ambient-PRISMANET. Then we briefly explain how distributed communication and mobility are supported using the Auction Systems case study.

6.1 Ambient Construct

The PRISMANET middleware includes the `ComponentBase` class. This class is responsible for implementing architectural elements by defining aspects, weavings and ports. Since ambients are also defined in this way, the `Ambient` class has been implemented by extending the `ComponentBase` class (see Figure 91). The `Ambient` class inherits `ComponentBase` properties and methods and also defines its own. A characteristic of all ambients is that they have the predefined Mobility and Coordination aspects as well as their four predefined ports. Figure 11 shows that the predefined aspects have been included by using the `AddAspect()` method in the `Ambient` constructor as well as the predefined ports by the `Add()` method.

```
[Serializable]
 public class Ambient: ComponentBase
  {
  (...)
   public Ambient(string AmbientName): base(AmbientName)
{
(...)
 /**DEFINITION OF ASPECTS OF AN AMBIENT **/
 AddAspect(new AmbientCoordinationAspect());
 AddAspect(new MobilityAspect());
(...)
 /** DEFINITION OF PORTS OF AN AMBIENT **/
 // Ports for generic communication
 InPorts.Add("ICallPort-InOut", "ICall", "INTERN");
 InPorts.Add("ICallPort-OutIn", "ICall", "EXTERN");
 // Ports for mobility
 InPorts.Add("ICapabilityPort-OutIn", "ICapability", "EXTERN");
 InPorts.Add("ICapabilityPort-InOut", "ICapability", "INTERN");
(...)
  }
```

Figure 91. A segment of the `Ambient` class constructor

In the following, the implementations of the Mobility Aspect and the Coordination Aspect are explained.

6.1.1 The Mobility Aspect

The Mobility Aspect allows an ambient to provide AC capabilities to mobile architectural elements as well as other services in order to provide a consistent state of the software architecture during and after a mobility process. Figure 92 shows that the `MobilityAspect` class extends the AspectBase class (the class that represents aspects in PRISMANET) and implements the ICapability interface. The MobilityAspect is a C# sealed class, i.e., the class cannot be extended. This is to emphasize that this aspect is predefined and the user does not have to include extra functionality.

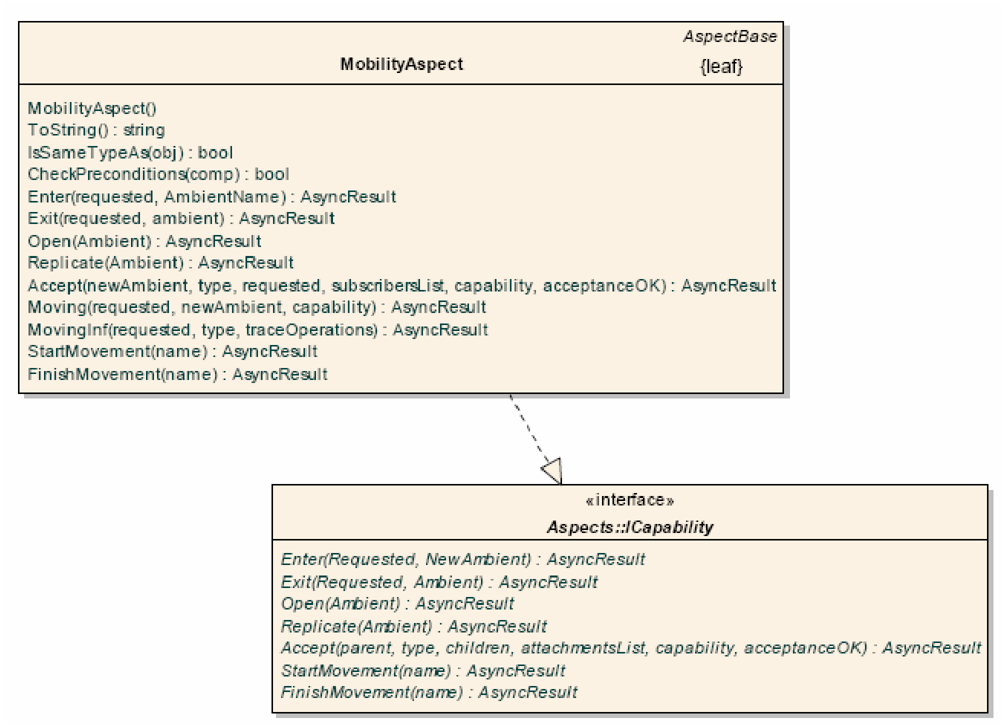

AspectBase
MobilityAspect
{leaf}
MobilityAspect()
ToString() : string
IsSameTypeAs(obj) : bool
CheckPreconditions(comp) : bool
Enter(requested, AmbientName) : AsyncResult
Exit(requested, ambient) : AsyncResult
Open(Ambient) : AsyncResult
Replicate(Ambient) : AsyncResult
Accept(newAmbient, type, requested, subscribersList, capability, acceptanceOK) : AsyncResult
Moving(requested, newAmbient, capability) : AsyncResult
MovingInf(requested, type, traceOperations) : AsyncResult
StartMovement(name) : AsyncResult
FinishMovement(name) : AsyncResult
«interface»
Aspects::ICapability
Enter(Requested, NewAmbient) : AsyncResult
Exit(Requested, Ambient) : AsyncResult
Open(Ambient) : AsyncResult
Replicate(Ambient) : AsyncResult
Accept(parent, type, children, attachmentsList, capability, acceptanceOK) : AsyncResult
StartMovement(name) : AsyncResult
FinishMovement(name) : AsyncResult

Figure 92. **The `MobilityAspect` class and the `ICapability` interface**

In the following, the `MobilityAspect` services are explained. These services have been implemented following their specification in the AOADL:

- StartMovement(name)service is in charge of managing the tasks needed for preparing an architectural element to move given its name. Basically, the service prepares the attachments connected to the mobile element to be reconfigured (i.e., added or removed).
- Enter(requested, newAmbient) service checks whether the mobile architectural element (requested) and the destination ambient (newAmbient) are located in the same parent ambient, i.e., are children. This is done by asking the parent ambient if it contains both the mobile element and the destination ambient. If this is satisfied, the Moving() service is executed in order to begin the mobility process. Since Enter()is implemented as a

transactional service, any satisfaction of an exception would cause the abort of the Enter().

- Exit(requested, Ambient) service checks whether the mobile architectural element (requested) is requesting to exit its parent ambient (Ambient). This is done by asking the ambient serving Exit() to check whether requested is actually its child and that Ambient has the same value as its name. If this is satisfied, the service GetParent() is invoked in order to obtain the destination ambient (the parent of the server ambient). This service has a weaving with the Distribution Aspect where the name of the parent ambient is stored. Then, the Moving() service is executed in order to begin the mobility process. Like the Enter() service, Exit() is implemented as a transactional service.
- Moving(requested, newAmbient) service performs the mobility process. Therefore, its two parameters are the mobile element (requested) and the destination ambient (newAmbient). Moving()is a transactional service that requests the MovingInf service to obtain information about the mobile element. Then, the transaction invokes the Accept() service of the destination ambient. When the destination ambient returns a flag to notify that it has accepted the mobile element, Moving()performs a post process to reconfigure the attachments which were connected to the moved architectural element.
- Accept(newAmbient, Type, requested, subscriberList, capability) service is a transactional service that is in charge of receiving the mobile architectural element in the destination ambient. Accept() checks whether the accepting petition is directed to the receiver by verifying that the newAmbient parameter has the value of the destination ambient name. If the petition is directed to the ambient, Accept() adds the new architectural element using the attachments information that has been passed as arguments. Then, the Mobility Aspect invokes the ChangeLocation() service of the moved architectural element in order to update its location value to the new parent ambient.

- FinishMovement(name) service is in charge of creating and executing attachments that a moved element needs at the destination ambient.

Note that the services of the Mobility Aspect are strictly related to an ambient. As a result, only ambients can import this aspect.

6.1.2 The Coordination Aspect

The Coordination Aspect provides the services of architectural elements that are located in an ambient to architectural elements that are located outside it. The Coordination Aspect also resends requests of architectural elements in an ambient to the parent ambient. The Coordination Aspect has been implemented in the AmbientCoordinationAspect class (Figure 93). This class extends the AspectBase class and implements the ICall interface. Like the MobilityAspect class, the AmbientCoordinationAspect class is a sealed class.

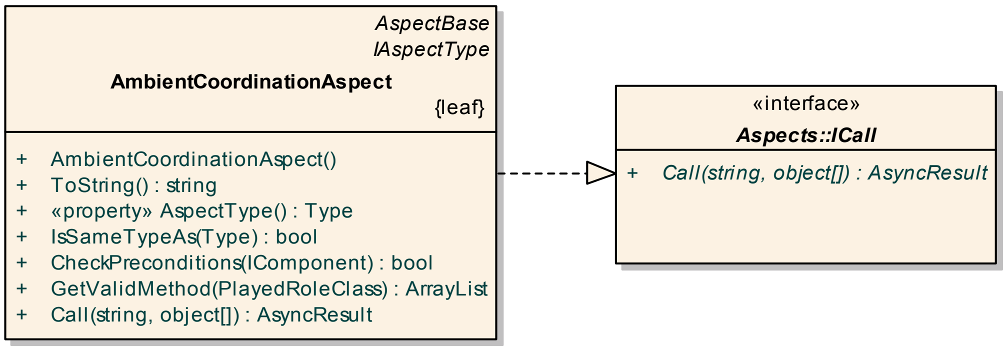

AspectBase
IAspectType
AmbientCoordinationAspect
{leaf}
+ AmbientCoordinationAspect()
+ ToString() : string
+ «property» AspectType() : Type
+ IsSameTypeAs(Type) : bool
+ CheckPreconditions(IComponent) : bool
+ GetValidMethod(PlayedRoleClass) : ArrayList
+ Call(string, object[]) : AsyncResult
«interface»
Aspects::ICall
+ Call(string, object[]) : AsyncResult

Figure 93. The `AmbientCoordinationAspect` class and the `ICall` interface

Since the set of architectural elements located in an ambient can dynamically change, the interface that this aspect implements should dynamically change depending on the services of the architectural elements of an ambient. However, implementing a dynamic interface is not efficient because, once an interface is changed, all the constructs related to the interface must also be changed. Therefore, a generic interface that is in charge of sending messages through the ambient has been defined. This interface encapsulates requests in a generic invocation method called `Call()`. The `Call()` service has two parameters: the name of the service as a `string` and the set of parameters of the service in an array of `objects`. In this way, all the petitions that the Coordination Aspect processes must be extracted in a `Call()` service. These must then be transformed into the original services when the petitions are sent to the architectural element ports.

In PRISMANET, the `InPort` class, which implements the part of a port that receives service invocations and resends them to aspects, only accepts invocation of services that are part of the `ValidMethods` list. This list is obtained through the `GetValidMethod` method of the `AspectBase` class that implements the services of the specific interfaces. Since the `InPort` instance that receives service invocations to the the `AmbientCoordinationAspect` class has to accept services that are dynamically changing, the class has to override `GetValidMethod` method. The `GetValidMethod` method must return the list to the `InPort` of the InServicesPort of the ambient, depending on the architectural elements that are in an ambient. In this way, the services that are in charge of moving an architectural element are also in charge of updating the list of `ValidMethods` in the `InPort`.

6.2 The three kinds of Ambients

The three kinds of ambients: Group, Site, and Virtual all share the characteristics described in section 0. However, each ambient kind has its own characteristics. In Figure 94, shows how the three ambients extend the `Ambient` class.

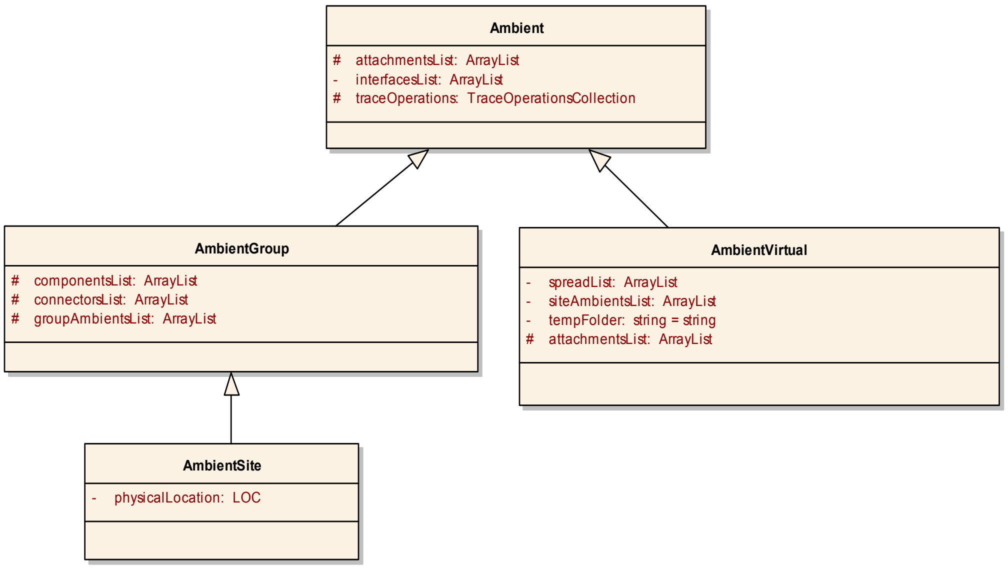

Ambient
# attachmentsList: ArrayList
- interfacesList: ArrayList
# traceOperations: TraceOperationsCollection
AmbientGroup
# componentsList: ArrayList
# connectorsList: ArrayList
# groupAmbientsList: ArrayList
AmbientVirtual
- spreadList: ArrayList
- siteAmbientsList: ArrayList
- tempFolder: string = string
# attachmentsList: ArrayList
AmbientSite
- physicalLocation: LOC

Figure 94. **The three kinds of ambients in Ambient-PRISMANET**

In the following, the implementation of the three kinds of ambients is explained.

6.2.1 Group Ambients

Group Ambients are implemented in the `AmbientGroup` class. For example, the *AgentsGroup* ambient that is specified in Figure 95 must inherit from the `AmbientGroup` class in order to be executed.

```
Ambient_Group type AgentsGroup
 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect AGADist;
 Ports
     InCapabilitiesPort: ICapability;
     ECapabilitiesPort: ICapability;
     EServicesPort: ICall;
     InServicesPort: ICall;
     EDistPort : ICapability;
     InMobilityPort : IMobility;
     EMobilityPort  : IMobility;
 End_Ports
 Weavings
     AGADist.getLocation(Location) instead
                               Mobile.getParent(Parent);
 End Ambient_Group type AgentsGroup;
```
Figure 95. Specification of the AgentsGroup ambient

`AmbientGroup` class has a list for each kind of architectural element that it can contain: components (`componentsList`), connectors (`connectorsList`), and group ambients (`groupAmbientsList`). The methods of this class, mainly override the ones of the `Ambient` class. The methods of the `AmbientGroup` are explained below:

- `AddChild()`: This method adds new architectural elements to the lists of the ambient. The method checks which kind of architectural element is to be added and then adds it to the corresponding lists.
- `RemoveChild()`: This method removes an element from its corresponding list. First, it looks for the element in the three lists and then it removes it.
- `IsChild()`: Given the name of the architectural element, this method returns a boolean value to indicate whether the architectural element is located inside the ambient.
- `AddLocalAttachment()`: This method creates new communication channels in the ambient.
- `SearchAttachment()`: This method returns the attachment that forms part of a complex attachment.
- `GetAmbientType()`: This method returns a string to indicate the kind of ambient. In this case, it returns a string with “AmbientGroup”.
- `RecomposeCommChannels()`:This method reconfigures the attachments needed for an architectural element in an ambient.

Elements that move in a Group ambient originate from either a Site ambient or another Group ambient. Also, the destination of elements that exit a Group ambient is either a Site ambient or a Group ambient. In this way, ambients of the Group ambient kind do not deal with serialization or deserialization since the mobility that is performed is always local to the current host.

6.2.2 Site Ambients

Site Ambients are implemented in the `AmbientSite` class. For example, the *HostSite* ambient that is specified in Figure 96 must inherit from the `AmbientSite` class in order to be executed. This kind of an ambient can contain the same architectural elements as a Group ambient. As a result, the `AmbientSite` class extends the `AmbientGroup` class in order to inherit its lists and the methods that manage them. The only difference is the `GetAmbientType()` method which must be overridden in order to return the string "AmbientSite".

```
Ambient_Site type HostSite
 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect HostDist;
 Ports
     ECapabilitiesPort: ICapability;
     EServicesPort: ICall;
     InServicesPort: ICall;
     InCapabilitiesPort: ICapability;
 End_Ports
 Weavings
     HostDist.getLocation(Location) instead
                                  Mobile.getParent(Parent).
… …
End Ambient_Site type HostSite;
```

Figure 96. . Specification of the HostSite ambient

An important fact that distinguishes a Site ambient from other ambients is that it represents a device. As a result, the physical location (the URL) that the ambient represents is passed in the constructor of the Site ambient. When a Site ambient is instantiated, it notifies the Ambient-PRISMANET middleware that it is going to represent it. In this way, there is only one instance of a Site ambient executing on a host, and this instance is the only element that has the reference to the Ambient-PRISMANET middleware of the host. For this reason, a Site ambient cannot change its physical location during runtime.

In other words, a Site ambient can be considered to be a bridge between an ambient hierarchy and the middleware where other architectural elements execute. Each time an element exits a Site, the element moves into a new host and starts executing on another Ambient-PRISMANET middleware. However, this is transparent to the element that is moving, since it is the Site ambient that is in charge of performing these changes.

6.2.3 Virtual Ambients

Virtual Ambients are implemented in the `AmbientVirtual` class. For example, the Root ambient in Figure 97 is instantiated from the `AmbientVirtual` class in order to be executed. These kinds of ambients are the parents of Site ambients and represent the Ambient-PRISMANET execution environment that is made up of the different middlewares executing on the distributed hosts.

```
Ambient_Virtual type Root

 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect RootDist;

 Ports
     InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
     ECapabilitiesPort: ICapability Played_Role Mobile.Child;
     EServicesPort: ICall Played_Role ACoordination.Client;
     InServicesPort: ICall Played_Role ACoordination.Server ;
 End_Ports

 Weavings
     Dist.getLocation(Location) instead
                                MobilityAspect.getParent(Parent);

End Ambient_Virtual type Root;
```
Figure 97. Specification of the Root ambient

A Virtual Ambient is the representative of a specific execution environment which can be connected to other execution environments. As a result, at implementation, an instance of an `AmbientVirtual` class executing on an Ambient-PRISMANET middleware host represents a gateway for other `AmbientVirtual` instances that contain other Sites. In this way, an execution environment needs a Root that is represented in the host where an `AmbientVirtual` instance executes.

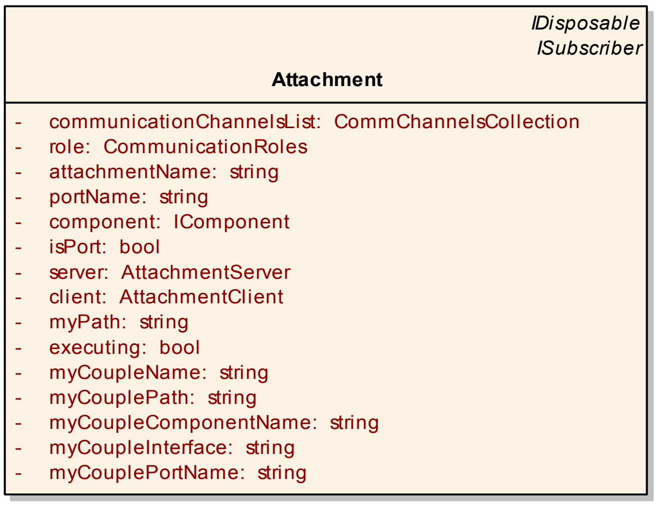
When architectural elements move from Site to Site, Site ambients provide this mobility through the `AmbientVirtual` instance of their execution environment. Thus, architectural elements must enter the `AmbientVirtual` instance so that they can be serialized. However, elements that move among Sites cannot stay in an `AmbientVirtual` instance. If an architectural element enters a Virtual ambient and does not exit it, the mobility transaction fails and the element is sent back to its original ambient.

The `AmbientVirtual` class stores the name of Site ambients that it locates in a list called `siteAmbientsList`. Also, an element that is temporally passing through a Virtual ambient is stored in `tempFolder`. The methods of `AmbientVirtual` also override the methods of `Ambient` in a similar way to `AmbientGroup`. Most of the methods manage the list `siteAmbientsList`. The `AddLocalAttachment()` method of the `AmbientVirtual` class creates: attachments between the ports of architectural elements (that are temporally located in the Virtual ambient) and the Virtual ambient ports or attachments between the ports of Site ambients and the ones of the Virtual ambient. The attachments that connect two Site ambients are distributed. In this case, the location of the distributed ambients must be identified using the Ambient-PRISMA DNS and be connected by .NET Remoting.

6.3 Distributed Communication

In Ambient-PRISMA, communication among architectural elements is performed by the attachments. As explained, the user is not aware of the creation of a complex attachment that is formed by other attachments. Ambient-PRISMANET is responsible for creating the attachments that communicate two distributed architectural element instances.

In PRISMA, a communication channel is formed by an attachment that connects a port of a component instance to a port of a connector instance. In Ambient-PRISMA, the distributed communication channel is not only formed by an attachment but by many attachments. An attachment can also be used by many communication channels. From now on, the term attachment is used to refer to a simple attachment and the term communication channel (a complex attachment) is used to refer to an attachment in PRISMA. For example, the attachments shown in Figure 6 are referred to as attachments and the *attachments* specified in Figure 9 are *communication channels*. Thus, *attachments* can be added or removed when one of the instances they connect moves.



IDisposable
ISubscriber
Attachment
- communicationChannelsList: CommChannelsCollection
- role: CommunicationRoles
- attachmentName: string
- portName: string
- component: IComponent
- isPort: bool
- server: AttachmentServer
- client: AttachmentClient
- myPath: string
- executing: bool
- myCoupleName: string
- myCouplePath: string
- myCoupleComponentName: string
- myCoupleInterface: string
- myCouplePortName: string

Figure 98. The `Attachment` class in Ambient-PRISMANET

To implement an attachment in Ambient-PRISMA, the `Attachment` class (see Figure 98) has a collection that stores the names of the communication channels that use an attachment instance (`communicationChannelsList`). For example, in Figure 99 the attachment between the *AuctionSite* and the *ClientSite* can be used by any communication channel that connects an instance located in *AuctionSite* and an instance located in *ClientSite*. As a result, the `Attachment` class has methods for consulting and modifying the communication channel collection. Also, the `Attachment` has a property called `role` to indicate whether an attachment instance connects two siblings or it connects a parent ambient and an instance located in it. This information is needed to configure attachments.

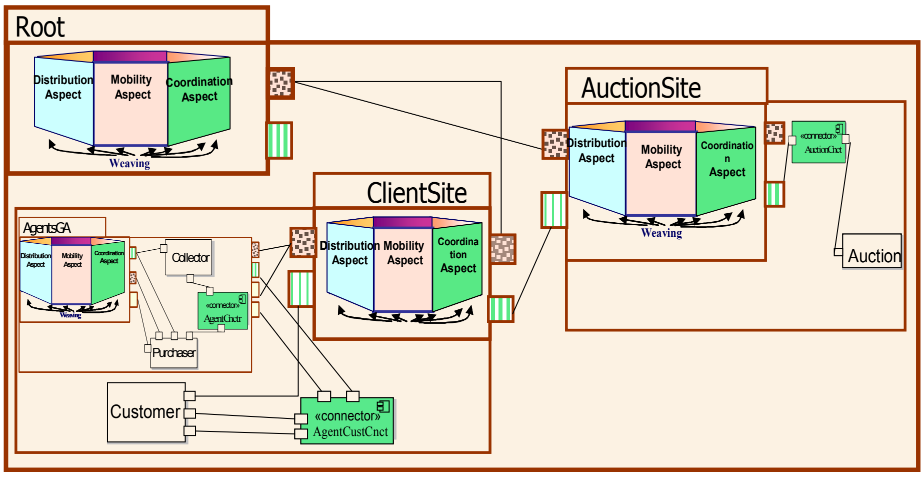


Figure 99. The Initial Configuration of the Software architecture of the Mobile Agent Case Study

The initial configuration of the attachments is obtained as follows: Once the developer specifies the communication channels in the configuration as shown in Figure 9, the Ambient-PRISMANET middleware uses the DNS to know on which host the instances that need to be connected are located. Then a bottom-up search algorithm is applied to obtain the route of each instance in a tree hierarchy of ambients (i.e., from the instance passing through the intermediate ambients until the Site ambient is reached). A bottom-up search is more efficient because an instance knows its direct parent ambient, whereas an ambient has many children. Once the route of both instances is obtained in an ordered list, the ambients common to both routes are identified. An attachment with sibling role is created between the next elements on both lists, and the remaining attachments are created with role parent. For example, to create the communication channel between the *Customer* and the *AuctionCnct* in Figure 99: the route obtained for the *Customer* is Root\ClientSite\Customer, and the route obtained for the *AuctionCnct* is Root\AuctionSite\AuctionCnct. Ambient-PRISMANET creates an attachment between *AuctionSite* and *ClientSite* with a sibling role, an attachment between *Customer* and *ClientSite* with a parent role, and an attachment between *AuctionCnct* and *AuctionSite* with a parent role.

6.4 Mobility

When an instance moves, the communication channels are reconfigured at runtime by adding and deleting attachments, i.e., when an instance invokes the services provided by the Mobility Aspect of its parent ambient. In the following, we show the steps performed by the Ambient-PRISMANET middleware when *AgentsGA* executes the *move(NewAmbient:string)* specified in Figure 100.

```
Distribution Aspect AGADist using IMobility, ICapability
  Attributes
    location : string NOT NULL;
  Services
     in changeLocation(Name: String, NewLocation:loc)
          Valuations
            [changeLocation()] location:=NewLocation;
  ..........
    Transactions in move(NewAmbient:string)
      move::= out startMovement(input Name,
                                output CommunicationList[])
      → MOVE1;
      MOVE1::= out exit(Name,Parent)→ MOVE2;
      MOVE2::= out enter(Name, NewAmbient)→MOVE3;
      MOVE3::= out finishMovement(input Name,
                                  input CommunicationList[]);
  ........
  End_Distribution Aspect AGADist
```

Figure 100 . Specification of the AGADist distribution aspect

The *AgentsGA* invokes the `StartMovement(AgentsGA)` service of the Mobility Aspect of the *ClientSite* through its *IC* port . This is to indicate to the *ClientSite* that it is going to begin a moving process. As a result, the *ClientSite* obtains all the information about the attachments that are connected to the *AgentsGA*. The *ClientSite* removes all the attachments connected to the AgentsGA and creates attachments between its *IS* port and the elements that were connected to the AgentsGA. The only attachment that is not deleted is the one connected to the IC port of the *ClientSite*.

The AgentGA invokes the Exit(AgentsGA, ClientSite)service through the IC port, and the *MobilityAspect* of the ClientSite is executed. As a result of the execution, the ClientSite sends the Accept() service to Root with the information of the attachments and the AgentsGA. The Ambient-PRISMANET deletes the AgentsGA from the ClientSite and deletes the attachment between the AgentsGA and the IC port of the ClientSite. As a result, the Root creates an attachment between the AgentsGA and its IC port. The resulting configuration is the one shown in Figure 101.

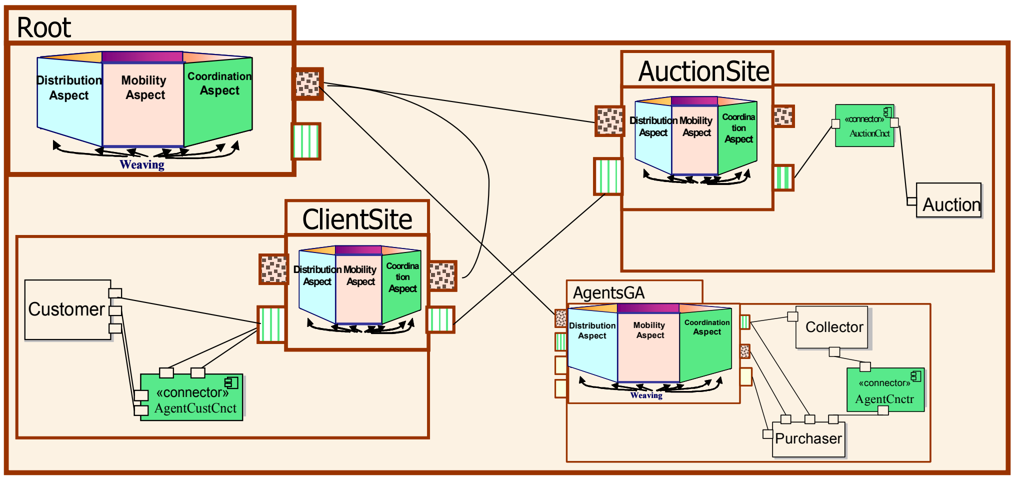


Figure 101. The Software Architecture when the *AgentsGA* executes the exit

Then, the *AgentsGA* requests the `Enter(AgentsGA, AuctionSite)` service from the *Root* through the attachment that connects the AgentsGA and the IC port of the Root. The Root executes its Mobility Aspect. As a result, the Root executes the Accept() method of the AuctionSite and sends the information of the AgentsGA attachments that was passed by the ClientSite to the AuctionSite. The Root serializes the AgentsGA so that it can be sent to the AuctionSite. When the AuctionSite notifies acceptance, the AgentsGA is deleted from the Root and its attachment. As a result, the AuctionSite adds it and connects the AgentsGA to its IC port. The AgentsGA then requests the FinishMovement(AgentsGA) service and the AuctionSite uses the information about the attachments to create the attachments that the AgentsGA needs. The result of the configuration is shown in Figure 102.

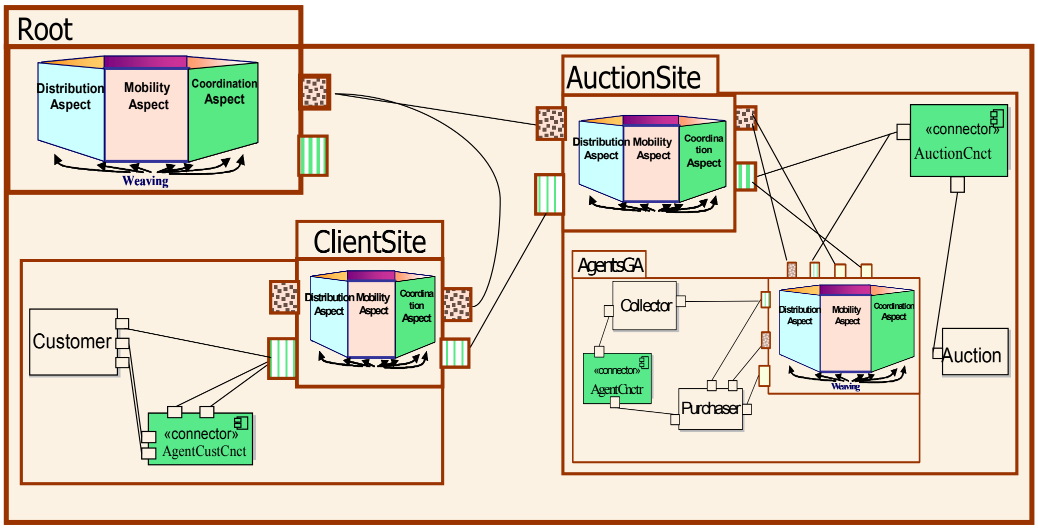


Figure 102. The *AgentsGA* becomes a child of *AuctionSite*

6.5 Conclusions

This chapter has presented Ambient-PRISMANET. Ambient-PRISMANET is a middleware that implements the constructs needed to execute Ambient-PRISMA models. In this way, these models can execute in the .NET platform.

As a result, the code generation patterns have been identified to automatically generate distributed applications from Ambient-PRISMA. The code generation patterns describe how classes of the Ambient-PRISMANET have to be extended to generate distributed and mobile applications.

The work related to Ambient-PRISMANET has produced a set of results that are published in the following publications:

- Cristobal Costa, **Nour Ali,** Carlos Millan, Jose A. Carsi**,** "Transparent Mobility of Distributed Objects using .NET, 4th International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2006.

- Jennifer Pérez**, Nour Ali**, Cristobal Costa, José Á. Carsí, Isidro Ramos, "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology", 3rd International Conference on .NET Technologies, pp. 97-108, Pilsen, Czech Republic, May-June 2005.
- **Nour Ali**, Jennifer Pérez, Cristobal Costa, Jose A. Carsí, Isidro Ramos, "Implementation of the PRISMA Model in the .Net Platform", II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- Carlos Millán, **Nour Ali**, Isidro Ramos, "DESARROLLO DE APLICACIONES DISTRIBUIDAS Y MÓVILES DESDE UN MODELO ARQUITECTÓNICO ORIENTADO A ASPECTOS", Informe Tecnico DSIC, Universidad Politécnica de Valencia, Octubre, 2006.

CHAPTER 7
Conclusions and Further Works

This chapter presents the main contributions of the work presented in this thesis. It also presents some future work that can be done based on the work of this thesis.

7.1 Conclusions

Currently. most software systems have a distributed nature. As a result, the development of software systems has to be taken into account at early stages of the software life cycle. A charcterstic of systems of this kind is that they are dynamic and need mobility support. Software architectures and AOSD are two techniques that promise the improvement of the quality and mantainability of software of complex systems increasing their reusability.

In order to take advantage of both Software architecture and AOSD, the PRISMA approach which integrates them, is enriched with the ambient architectural element.In this way, Ambient-PRISMA is created in order to specify PRISMA architectural models that take into account distribution and mobility properties. The ambient architectural element provides the notion of location in a software architecture. The advantage of including the notion of location as an architectural element of the software architecture provides the modelling of mobility as through the reconfiguration of the software architecture.

Ambient-PRISMA provides architectural elements to be aware of their location. As aresult, architectural elements can be autonomous in taking decisions about their location. Also, the migration decision in Ambient-PRISMA can be modelled as both passive and autonomous. The unit of mobility in Ambient-PRISMA is Components, Connectors and ambients.

Ambient-PRISMA approach enriches the PRISMA metamodel and the AOADL in order to allow the modelling of the network of a distributed software system, the definition of ambients that can be reusable in different distributed architectural models and the specification of different distribution and mobility startegies. Ambient-PRISMA also provides the execution of its specifications by enriching PRISMANET middleware for distribution and mobility. As a result, the code generation patterns have been identified for automatically generating distributed and mobile code.

All these contributions have been published in one national journal, eleven international conferences, one international workshop, five national conferences, eight national workshops, and four technical reports. These contributions are the following:

- **National Journals:**

  - **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose A. Carsí, *Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes*, **Revista IEEE America Latina**, 2007. (Invited: JISBD)

- **Internacional Conferences:**

  - Cristobal Costa, **Nour Ali**, Jennifer Perez, Jose Angel Carsi, Isidro Ramos, "Towards Dynamic Reconfiguration of Aspect-Oriented Software Architectures", First International Conference on Software Architecture (ECSA 2007), **LNCS** Springer Verlang, Madrid, September, 2007 (poster).
  - **Nour Ali**, Carlos Millán, Isidro Ramos, Developing Mobile Ambients using an Aspect-Oriented Software Architectural Model, 8th International Symposium on Distributed Objects

and Applications (DOA 2006), **LNCS** Springer Verlang, Montpellier, France, October, 2006.

- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, Mobile Ambients in Aspect-Oriented Software Architectures, IFIP Working Conference on Software Engineering Techniques (SET 2006), **Springer Series in Computer Science**, Warsaw, Polond, October 17-20, 2006.
- Cristobal Costa, **Nour Ali,** Carlos Millan, Jose A. Carsi, "Transparent Mobility of Distributed Objects using .NET, 4th International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2006.
- Jennifer Pérez, **Nour Ali**, Jose Ángel Carsí, Isidro Ramos, Designing Software Architectures with an Aspect-Oriented Architecture Description Language, 9th Symposium on the Component Based Software Engineering (CBSE), Springer Verlang LNCS 4063 ,pp. 123-138, ISSN: 0302-9743, ISBN: 3-540-35628-2, Vasteras, Suecia, June 29th-July 1st, 2006.
- **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí *Introducing Ambient Calculus in Mobile Aspect-Oriented Software Architectures* , Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), **IEEE Computer Society**, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper).
- **Nour Ali**, Isidro Ramos, Jose A. Carsí, *A Conceptual Model for Distributed Aspect-Oriented Software Architectures* , International Conference on Information Technology Coding and Computing (ITCC 2005), **IEEE Computer Society**, Las Vegas, NV, USA, 2005.
- Jennifer Pérez, **Nour Ali**, Cristobal Costa ,Isidro Ramos, Jose A. Carsí, *Executing Aspect-Oriented Software Architectures in .NET,* 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005.
- **Nour Ali**, Jennifer Pérez Isidro Ramos, Jose A. Carsí , Aspect Reusability in Software Architectures, 8th International conference of Software Reuse (ICSR), July, 2004 (poster)
- **Nour Ali**, Jennifer Perez, Isidro Ramos, High Level Specification of Distributed and Mobile Information Systems, Second International Symposium on Innovation in

Information & Communication Technology (ISSICT 2004), April, Amman, Jordan,2004.

- **Nour Ali**, Josep Silva, Javier Jaén, Isidro Ramos, José Ángel Carsí, Jennifer Pérez, Mobility and Replication Patterns in Aspect-oriented component Based Software Architectures, 15th IASTED International Conference, PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS~PDCS 2003~, November 3-5, Marina del Rey, CA, USA, 2003.

- **International Workshops:**

  - Jennifer Pérez, **Nour Ali**, Jose Ángel Carsí, Isidro Ramos, *Dynamic Evolution in Aspect-Oriented Architectural Models*, European Workshop on Software Architecture, (EWSA), LNCS Springer Verlang , Pisa, Italy, June 2005.

- **Nacional Conferences:**

  - **Nour Ali**, Jennifer Pérez,Cristóbal Costa, Isidro Ramos,Jose Ángel Carsí, Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes, XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Sitges, Barcelona, Octubre 3-6, 2006.
  - Cristóbal Costa, Jennifer Pérez, **Nour Ali**, Jose Angel Carsí, Isidro Ramos, “PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures”**,** X Conference on Software Engineering and Databases (JISBD), pp. 27-34, ISBN: 84-9732-434-X, Granada, September, 2005. (In Spanish)
  - **Nour Ali**, Jose Angel Carsi, Isidro Ramos,“Analysis of a Distribution Dimension for PRISMA”, Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004), Málaga, 10-12 Noviembre 2004
  - Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, “Development of a Tele-Operation System using the PRISMA Approach”, VIII Conference on Software Engineering and Databases (JISBD), pp. 411-420, ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)

    - Josep Silva, **Nour Ali**, Jose Angel Carsi, Isidro Ramos, "El aspecto de distribución de PRISMA", VIII Conference on Software Engineering and Databases (JISBD), ISBN: 84-688-3836-5, Alicante, November, 2003.

- **Nacional Workshops:**
    - **Nour Ali**, Isidro Ramos, "Ambient-PRISMA: Ambients in Distributed and Mobile Aspect-Oriented Software Architectures", V Jornadas de Trabajo DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, 2006.
    - **Nour Ali,** Jennifer Perez, Jose Angel Carsi, Isidro Ramos, "Mobility of Objects in PRISMA" , III Jornadas de DYNMAICA, Al Magro, Ciudad Real, 2005.
    - Mª Eugenia, **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí**, "**DIAGMED: An Architectural model for a Medical Diagnosis"*,* IV workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 1-7, Archena, Murcia, November, 2005. (In Spanish)
    - Rafael Cabedo, Jennifer Pérez, **Nour Ali**, Isidro Ramos, Jose A. Carsí, Aspect-Oriented C# Implementation of a Tele-Operated Robotic System, III Workshop on Aspect-Oriented Software Development (DSOA), X Conference on Software Engineering and Databases (JISBD), pp. 53-59, ISBN: 84-7723-670-4, Granada, September, 2005. (In Spanish)
    - **Nour Ali**, Jennifer Pérez, Cristobal Costa, Jose A. Carsí, Isidro Ramos, "Implementation of the PRISMA Model in the .Net Platform", II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
    - **Nour Ali**, Jose Cercos, Isidro Ramos, Patricio Letelier, Jose Angel Carsi, "Distribution in PRISMA", workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.

- **Nour H. Ali**, Josep F. Silva, Javier Jaen, Isidro Ramos, Jose Á. Carsí, Jennifer Pérez, "Distribution Patterns in Aspect-Oriented Component-Based Software Architectures", IV Workshop Distributed Objects, Languages, Methods and Environments (DOLMEN), pp.74-80, Alicante, November, 2003.
- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Jose A. Carsí, "PRISMA: Aspect-Oriented and Component-Based Software Architectures", Workshop on Aspect-Oriented Software Development (DSOA), Conference on Software Engineering and Databases (JISBD), Technical Report TR-20/2003 of the Polytechnic School of the University of Extremadura, pp. 27-36, Alicante, November, 2003. (In Spanish)

- **Technical Report:**
    - Carlos Millán, **Nour Ali**, Isidro Ramos, "DESARROLLO DE APLICACIONES DISTRIBUIDAS Y MÓVILES DESDE UN MODELO ARQUITECTÓNICO ORIENTADO A ASPECTOS", Technical Report DSIC, Polytechnic University of Valencia, October, 2006.
    - Jennifer Pérez, **Nour Ali** , Jose A. Carsí, Isidro Ramos, "PRISMA Architecture of the Robot 4U4 Case Study", Technical Report DSIC-II/13/04, pp. 72, Polytechnic University of Valencia, 2004. (In Spanish)
    - **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "A Compiler for the Automatic Generation of the Distribution Aspect to Distributed Applications", Technical Report DSIC-II/15/04, Polytechnic University of Valencia, October 2004. - 2004
    - **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "DISTRIBUTION IN AN ASPECT-ORIENTED COMPONENT BASED SOFTWARE ARCHITECTURE THROUGH PRISMA", Technical Report DSIC-II/14/04, Polytechnic University of Valencia, October 2004. - 2004

7.2 Further Work

In the near future, much further work can be performed. In the following, some are listed below:

- The bahaviour of Ambient-PRISMA has to be formalized with Channel Ambient Calculus in order to facilitate the code generation and develop tools for validating the behaviour of models.
- The graphical notation of Ambient-PRISMA has to be included in the PRISMA CASE in order to provide the graphical modelling of Ambient-PRISMA architectural models.
- The code generation patterns have to be implemented in order to automatically generate the code from the Ambient-PRISMA specifications.

In a long term, some possible work is listed below:

- Extend Ambient-PRISMA for physical mobility where sites are mobile. The modelling of physical mobility is possible in the current version of Ambient-PRISMA. This can be done by specifying mobility in the distribution aspect of a site ambient. However, the Ambient-PRISMANET Wireless middleware has to be extended for physical mobility.
- Consider additional non-functional requirements such as resources, security, and fault tolerance. These can be included as primitives in the AOADL, allowing the user to model properties related to them.

BIBLIOGRAPHY

[Aks94]Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, Y., “Abstracting Object Interactions Using Composition Filters”, Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming, pp. 152-184, 1994.

[Ald03]Aldawud O., Elrad T., Bader A., “UML profile for Aspect Oriented Software Development”, The Aspect-Oriented UML Workshop, International Conference on Aspect-Oriented Software Development, Boston, USA March 18, 2003.

[All97] Allen, R., Garlan, D., “A formal basis for architectural connection”, ACM Transactions on Software Engineering and Methodology, July, 1997.

[ASP07]The AspectJ Project Website, http://eclipse.org/aspectj/

[Bas03] Bass, L., Clements, P., Kazman, R., “Software Architecture in Practice”, Second Edition, Addison Wesley, 2003.

[Ben06]Navarro, L.D.B., Südholt, M., Vanderperren, W., De Fraine, B., Suvée, D., “Explicitly distributed AOP using AWED”, In Proceedings of the 5th International Conference on Aspect-Oriented Software Development, ACM Press, p. 51-62, 2006.

[Ber01]Bergmans, L. and Aksit, M., “Composing multiple concerns using composition filters”, Communications of the ACM, Octorber 2001.

[Ber04] Bergmans, L. and Aksit, M., “Principles and Design Rationale of Composition Filters”, In Aspect-Oriented Software Development, Addison Wesley, October, 2004.

[Boo04]Booch, G., Maksimchuk, R.A., Engel, M.W., Young, B.J., Conallen, J., Houston, K. A., Martin, R., Newkirk, J.W., “Object-Oriented Analysis and Design with Applications (3rd Edition)”, Addison-Wesley Professional, 2004

[Bra04] Bradbury, J. S., Cordy, J. R., Dingel, J., Wermelinger, M., "A survey of self-management in dynamic software architecture specifications", Workshop on Self-managed systems, Proceedings of the 1st ACM SIGSOFT, Newport Beach, California, pp. 28-33, 2004.

[Bri02]Brito, I., Moreira, A., "Towards a Composition Process for Aspect-Oriented Requirements", Workshop on Early-Aspects, The 1st International Conference on Aspect-Oriented Software Development (AOSD), Twente, The Netherlands, April 2002.

[Bug01] Buglesi, M., Castagna, G., Crafa, S., " Reasoning about security in mobile ambients", In CONCUR´01, number 2154 in LNCS, pp. 102-120, 2001.

[Can01] Canal, C., Pimentel, E., Troya, J.M., "Compatibility and Inheritance in Software Architectures", Science of Computer Programming, Vol. 41, Nº 2. 2001.

[Car98a] 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.

[Car98b] 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.

[Cia98] Ciancarini, P., Mascolo, C. "Software architecture and mobility", In D. Perry and J. Magee, editors, Proc. 3rd International Software Architecture Workshop (ISAW-3), ACM SIGSOFT Software Engineering Notes, pp. 21-24, Orlando, FL, November 1998.

[Cia99] Ciancarini, P., Mascolo, C., "Specification and Analysis of Component Based Software Architectures", Proc. First IFIP International Working Conf. on Software Architecture, 1999.

[Cla05]Clarke, S., and Baniassad. E., "Aspect-Oriented Analysis and Design: The Theme Approach", Addison Wesley Professional, March 23, 2005.

[Cle04]Clemente, P., Hernandez, Herrero, J.L., Murillo, J.M., Sanchez, F., "Aspect-Orientation in the Software Lifecycle: Fact and Fiction", In Aspect-Oriented Software Development, Addison Wesley, pp. 407-423, October, 2004.

[Cle05] Clements, P., Bachman, F., Bass, L., Garlan, D., et al., "Documenting Software Architectures : Views and Beyond", SEI Series in Software Engineering, Addison Wesley, 2005.

[COR07]CORBA Official Web Site of the OMG Group: http://www.corba.org/

[Cue02] Cuesta, C., "Dynamic Software Architecture based on Reflection", PhD. Thesis, University of Valladolid, Spain, July, 2002. (In Spanish)

[Dij74]Dijkstra, E W., "A Discipline of Programming", EWD. 477, Neuen, The Netherlands, 30 August 1974.

[Dos99] Dashofy, E. M., Medvidovic, N., Taylor, R. N., "Using Off-the Shelf Middleware to Implement Connectors in Distributed Software Architectures, ICSE'99, Los Angeles, CA, May 1999.

[DSL07] Domain-Specific Language (DSL) Tools, http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx

[Fia03] Fiadeiro, J.L., Lopes, A., Wermelinger, M., "A Mathematical Semantics for Architectural Connectors", LNCS 2793, pp. 190-234, 2003.

[Fia04] Fiadeiro, J.L., "Categories for Software Engineering", Springer, 2004.

[Fil00]Filman R., Friedman D., "Aspect-Oriented Programming is Quantification and Obliviousness", Workshop on Advanced Separation of Concerns, OOPSLA, October, 2000.

[Fil04] Filman, R. E., Elrad, T., Clarke, S., Aksit, M., "Aspect-Oriented Software Development", Addison Wesley Professional, October 06, 2004.

[Fug98] Fuggetta, A., Picco, G.P., and Vigna, G. Understanding Code Mobility. In *IEEE Transactions on Software Engineering*, 24(5): 342-361, 1998.

[Gam95]Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[Gar03] Garlan, D., "Formal Modelling and Analysis of Software Architecture: Components, Connectors, and Events. Formal Methods for Software Architectures, Lecture Notes in Computer Science, Springer Verlang, Eds. Marco Bernardo and Paola Inverardi, LNCS 2804, September, 2003

[Gar04]Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C., "The Mobility Aspect Pattern", The 4th Latin-American Conference on Pattern Languages of Programming (SugarLoafPLoP'04), Fortaleza, Brazil, August 2004.

[Gar06]Garcia, A., Kulesza, U., Sant'Anna, C., Chavez, C., Lucena, C., "Aspects in Agent-Oriented Software Engineering: Lessons Learned", In: "Agent-Oriented Software Engineering VI", Joerg Mueller and Franco Zambonelli, LNCS, Springer, May 2006.

[Gei98]Geier, M., Steckermeier, M., Becker, U., Hauck, F, Meier, E., Rastofer, U., "Support for mobility and replication in the AspectIX architecture", Object-Oriented Technology, ECOOP'98 Workshop Reader, pp. 325-326, LNCS 1543, Springer, 1998.

[Gen87] Genrich, H.J., "Predicate/Transition Nets", Petri Nets: Central Models and Their Properties, W. Brauer, W. Resig, and G. Rozenberg, eds., pp. 207-247, 1987.

[Gre04] Greenfield J., Short K, Cook S., and Kent S. Software Factories. Wiley Publising Inc., 2004.

[Gru04] Gruhn,V., Schafer,C., "An Architecture Description Language for Mobile Distributed Systems", In Proceedings of the First European Workshop on Software Architecture (EWSA2004), Springer-Verlag, pp. 212-218, 2004.

[Gru05] Gruhn,V., Schafer,C., "Architecture Description for Mobile Distributed Systems", In Proceedings of the Second European Workshop on Software Architecture (EWSA2005), Springer-Verlag, 2005.

[Har02] Harrison, W.H., Ossher, H.L., Tarr, P.L., “Asymmetrically Vs. Symmetrically Organized Paradigms for Software Composition”, IBM Research Report, RC22685 (W0212-147) Thomas J. Watson Research Center, IBM, December 2002.

[Har84] Harel D., “*Dynamic Logic*. Handbook of Philosphical Logic II”, editors D.M. Gabbay, F. Guenthner, pp. 497-694, Reidel, 1984.

[Hau98]Hauck, F, Becker, U., Geier, M., Meier, E., Rastofer, U., Steckermeier, M., “AspectIX Middleware for Aspect-Oriented Programming”, Tevhnical Report TR-I4-98-06, Univ. of Erlangen-Nuernberg, IMMD IV, 1998.

[Her03]Herrero, J.L., “A proposal of a platform, language and design, for developing Aspect-Oriented Applications” (In Spanish), PhD thesis, Computer Science Department, University of Extremadura, Extremadura, Spain, May, 2003.

[Ho02] Ho, W., Jézéquel, J., Pennaneac'h, F., and Plouzeau, N., “A toolkit for weaving aspect oriented UML designs”, Proceedings of the 1st international conference on Aspect oriented software development, pp. 99 – 105, April 2002.

[Hof00] Hoffman, D., Weiss, D., (eds), “Software Fundamentals: Collected Papers by David L. Parnas”, Addison-Wesley, 2001.

[IEE00] IEEE Recommended Practices for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000, Software Engineering Standards Committee of the IEEE Computer Society, 21 September 2000.

[Kic01]Kiczales G., Hilsdale E., Huguin J., Kersten M., Palm J., Griswold W.G., “An Overview of AspectJ”. The 15th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol.2072, Budapest, Hungary, June 18-22, 2001.

[Kic97]Kizcales G., Lamping J., Mendhekar A., Maeda C., Lopes, C., Loingtier, J., Irwin, J., “Aspect-Oriented Programming”, The 11th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 1241, Jyväskylä, Finland, June 9-13, 1997.

[Kog95] Kogut, P., Clements, P. C., "Feature Analysis of Architecture Description Languages", Proceedings of the Software Technology Conference (STC´95), Salt Lake City, April, 1995.

[Kru95] Kruchten, P., "The 4+1 View Model of Architecture", IEEE Software Vol. 12, Nº 6, pp. 42-50, November 1995.

[Lad03]Laddad, R., "AspectJ in Action: Practical Aspect-Oriented Programming", Manning Publications Co., 2003.

[Let98] Letelier, P., Sánchez, P., Ramos, I. and Pastor, O. "OASIS 3.0, A formal language for the object oriented conceptual modeling", Polytechnic University of Valencia, SPUPV-98.4011, ISBN 84-7721-663-0, 1998.(In Spanish).

[Lob04]Lobato, C., Garcia, A., Romanovksy, A., Sant'Anna, C., Kulesza, U., Lucena, C. , "Mobility as an Aspect: The AspectM Framework.", The 1st Brazilian Workshop on Aspect-Oriented Software Development – WASP'04, SBES'04, Brasília, Brazil, October 2004.

[Lop02] Lopes, A., Fiadeiro, J.L., Wermelinger, M., "Architectural Primitives for Distribution and Mobility", Proceedings of 10th Symposium on Foundations of Software Engineering, ACM Press, pp. 41-50, 2002.

[Lop02]Lopes, C. V., "Aspect-Oriented Programming: An historical perspective (what's in a name?) ", Technical Report, Institute for Software Research, University of California, Irvine, December 2002.

[Lop04] Lopes, A., Fiadeiro, J.L., "Adding Mobility to Software Architectures", ENTCS 97, pp. 241-258, 2004.

[Lop97]Lopes, C. V., "D: A Language Framework for Distributed Programming", Ph.D. Thesis, College of Computer Science, Northeastern University, Boston, USA, 1997.

[Luc95] Luckham, D.C., Kenney, J.J., Augustin, L. M., Vera, J., Bryan, D., Mann, "Specification and Analysis of System Architecture Using Rapide", IEEE

Transactions on Software Engineering, Vol. 21, Nº 4, pp. 336-355, April, 1995.

[Mae87]Maes, P., "Concepts and experiments in computational reflection", Proceedings of OOPSLA'98, Vol.22 of ACM SIGPLAN Notices, pp. 147-155, 1987.

[Mag94] Magee, J., Dulay, N., Krammer, J., "Regis: A constructive Development Environment for Distributed Programs", In IOP/IEE/BCS Distributed Systems Engineering, 1:5, pp. 304-312, 1994.

[Mag95] Magee, J., Dulay, N., Eisenbach S. and Krammer, J., "Specifying Distributed Software Architectures", Proceedings of the 5th European Software Engineering Conference (ESEC 95), pp. 137-153, Sitges, Spain, 1995.

[Mag97a]Magee, J., Kramer, J., Giannakopoulou, D., "Analysing the behaviour of distributed software architectures: a Case Study", Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97), pp. 240-245, 1997.

[Mag97b]Magee, J., Tseng, A., Kramer, J., "Composing Distributed Objects in CORBA", Proceedings of the Third International Symposium on Autonomous Decentralized Systems, pp. 257-263, Berlin, Germany, 1997.

[Mas99] Mascolo, C., "MobiS: A Specification Language for Mobile Systems", Proc. 3rd Int. Conf. on Coordination Models and Languages, 1999.

[MDA07] Object Management Group. Model Driven Architecture Guide, 2003, http://www.omg.org/docs/omg/03-06-01.pdf

[Med00] Medvidovic N., Taylor R.N., "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions of Software Engineering, Vol. 26, Nº 1, January 2000.

[Med01] Medvidovic, N., Rakic, M., “Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility”, Proceedings of the Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

[Med99] Medvidovic, N., Rosenblum, D.S., Taylor, R.N., “A language and environment for architecture-based software development and evolution”, Proceedings of 21st International Conference on Software Engineering, pp. 44-53, Los Angeles, CA, USA, 1999.

[Mez01]Mezini, M., and Ostermann, K., “Object Creation Aspects with Flexible Aspect Deployment”, Technical Report, 2001.

[Mil04] Miles, R., “AspectJ Cookbook”, O'Reilly, December 2004.

[Mil92] Milner, R., Parrow, J., Walker, D. “A calculus of mobile processes”**,** Parts 1-2. Information and Computation, 100(1), 1992, pp. 1-77.

[Mil93] Milner R., “The Polyadic -Calculus: A Tutorial”, Laboratory for Foundations of Computer Science Department, University of Edinburgh, October 1993.

[Mor95] Moricon, M., Qian, X., Riemenschneider, R., “ Correct architecture refinement”, IEEE Transactions on Software Engineering, Special Issue on Software Architecture, Vol. 21, N° 4, pp. 356-372, April, 1995.

[Nis04] Nishizawa, M., Shiba, S. and Tatsubori, M., “Remote pointcut - a language construct for distributed AOP”, Proceedings of the 3rd international conference on Aspect-oriented software development, ACM Press, pp. 7-15, 2004.

[Oli05] Oliveira, C., Wermelinger, M., Fiadeiro, J. L., Lopes, A., “Modelling the GSM handover protocol in CommUnity", Electronic Notes in Theoretical Computer Science, ISSN 1571-066, Vol. 141 N°3, pp. 3-25, 2005.

[Oqu04] Oquendo, F., “π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures”, ACM SIGSOFT Software Engineering Notes, Vol. 29 N°3, May 2004.

[Oqu06] Oquendo, F., “Formally modelling software architectures with the UML 2.0 profile for π-ADL”, ACM SIGSOFT Software Engineering Notes, Vol. 31 Nº 1, January 2006.

[Par72]Parnas, D.L., “On the Criteria To Be Used in Decomposing Systems Into Modules”, Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058, December, 1972.

[Per05a] Perez, N., Ali, N., Costa, C., Carsí, J.A., Ramos, I., “Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology”, 3rd International Conference on .NET Technologies, pp. 97-108, Pilsen, Czech Republic, May-June 2005.

[Per05b]Pérez, J., Ali, N., Carsí, J. A., Ramos, I., “Dynamic Evolution in Aspect-Oriented Architectural Models”, Second European Workshop on Software Architecture, Springer LNCS 3527, pp.59-16, Pisa, Italy, June 2005.

[Per06a] Pérez, J., Ali, N., Carsí, J.A., Ramos, I., “Designing Software Architectures with an Aspect-Oriented Software Architectures”, The 9th International Symposium on Component-Based Software Engineering (CBSE), Lecture Notes Computer Science, Springer Verlang, LNCS 4063, pp. 123-138, Västeras, Sweden, June-July, 2006.

[Per06b] Perez, J., “PRISMA: Aspect-Oriented Software Architectures”, PhD. Thesis, Polytechnic University of Valencia, December, 2006.

[Per92] Perry, D., Wolf, A., “Foundations for the Study of Software Architecture”, ACM Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.

[Per97] Perry, D., “State-of-the-Art: Software Architecture”, .Proceedings of the 19 International Conference on Software Engineering, ACM Press pp. 590-591, 1997.

[Pin02] Pinto, M., Fuentes, L., Fayad, M.E., Troya, “Separation of coordination in a dynamic aspect oriented framework”, Proceedings of the 1st International

Conference on Aspect-oriented Software Development, pp. 134-140, Enschede, The Netherlands, 2002.

[Pop01]Popovici, A., Alonso, G., Gross, T., "AOP Support for Mobile Systems", Presented in Advanced Separation of Concerns in Object-Oriented Systems, (OOPSLA 2001 Workshop), Tampa, USA, October 15, 2001.

[Pop02]Popovici, A., Gross, T. and Alonso, G., "Dynamic Weaving for Aspect-Oriented Programming", Proceedings of the 1st International Conference on Aspect-Oriented Software Development, Enschede The Netherlands, April 2002.

[Pul99]Pulvermueller, E., Klaeren, H., Speck, A., "Aspects in Distributed Environments", Proceedings of the First International Symposium on Generative and Component-Based Software Engineering, (GCSE'99), Lecture Notes In Computer Science; Vol. 1799, p. 37 - 48 , Erfurt, Germany, September, 1999.

[Ram02] Rammer, I., "Advanced .NET Remoting", Apress, 2002.

[Ras02] Rashid, A.., Sawyer, P., Moreira, A., Araujo, J., "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", IEEE Joint Conference on Requirements Engineering, Essen, IEEE Computer Society, p. 199-202, Germany, September 2002.

[RMI07] Remote Method Invocation (RMI) Website of Sun Developer Network: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[Rom00] Roman, G., Picco, G., Murphy, A., "Software Engineering for Mobility: A Roadmap", International Conference on Software Engineering - Future of SE Track, pp. 241-258, Limerick, Ireland, 2000.

[San00] Sanchez, F., Heranandez, J., Murillo, J.M., Rodríguez, R., "Adaptability of object distribution protocols using the disguises model approach", In Proceedings of International Symposium on Distributed Objects and Applications (DOA 00), IEEE Computer Society, pp. 315-322, Antwerp, Belgium, 2000.

[San98]Sanchez, F., Heranandez, J., Murillo, J.M., Pedraza, E., “Runtime adaptability of synchronization constraints in COOLs”, II ECOOP Workshop of Aspect Oriented Programming, 1998.

[Sar97] Sartipi, K., “A Survey on Software Architecture Domain”, PhD. Thesis, University of Waterloo, Ontario, Canada, February, 1997.

[Sch06] Schafer, C., “Modeling and Analyzing Mobile Software Architectures”, In Proceedings of the Third European Workshop on Software Architecture, (EWSA2006), Springer-Verlag, 2006.

[Sch06] Schmidt, D. C., “Model-Driven Engineering”, Computer, Vol. 39, Nº 2, IEEE Computer Society, February, 2006.

[Sha94a] Shaw, M., “Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status”, Proceedings of Workshop on Studies of Software Design, January, 1994.

[Sha94b] Shaw, M., Garlan, D., “Characteristics of Higher-Level Languages for Software Architecture”, Technical Report CMU-CS-94-210, SEI-94-TR-23, ESC-TR-94-023, Software Engineering Institute, Carnegie Mellon University, December, 1994.

[Sha96] Shaw, M. and Garlan, R., “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, April, 1996.

[Soa02a] Soares, S. and Borba, P., “PaDA: A pattern for distribution aspects”, In Second Latin American Conference on Pattern Languages Programming - SugarLoafPLoP 2002, Itaipava, RJ, Brazil. ICMC - University of São Paulo Magazine, p. 87-99, August, 2002.

[Soa02b] Soares, S., Laureano, E. and Borba, P. , “Implementing Distribution and Persistence Aspects with AspectJ”, In Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02, ACM Press, Seattle, WA, USA, p. 174-190, November, 2002.

[Soa04] Soares, S., “An Aspect-Oriented Implementation Method”, PhD thesis, Informatics Centre, Federal Universisty of Pernambuco, Recife, Brazil, October 2004.

[Sub05] Subotic, S. and Bishop, J., “Emergent Behaviour of Aspects in High Performance and Distributed Computing”, Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, ACM International Conference Proceeding Series; Vol. 150, White River, South Africa, p. 111-119, 2005.

[Suv03] Suvée, D., Vanderperren, W, Jonckers, V., “JAsCo.: An aspect-oriented approach tailored for component based software development”, In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, ACM Press, p. 21-29, 2003.

[Suz99] Suzuki, J. and Yamamoto, Y., “Extending UML with aspects: Aspect support in the design phase”, In Int'l Workshop on Aspect-Oriented Programming (ECOOP), Lisbon, 1999.

[Szy00] Szyperski C., Booch G., Meyer B., “Beyond Objects”, Software Development Magazine, March, 2000 (originally BrucePowel Douglass).

[Szy02] Szyperski, C., “Component Software: Beyond Object Oriented Programming”, ACM Press and Addison Wesley, New York, USA, 2002.

[Tar01] Tari, Z., Bukhres, O., “Fundamentals of Distributed Object Systems”, Series: Wiley Series on Parallel and Distributed Computing, 9 Oct 2001.

[Tay95] Taylor, R., N., Medvidovic, N., Anderson, K. M., Whitehead, Jr., E., J., Robbins, J. E., “A Component-and Message-Based Architectural Style for GUI Software”, Proceedings of the 17th international conference on Software Engineering, pp. 295-304, Seattle, Washington, United States, 1995.

[Til03] Tilevich, E., Urbanski, S., Smaragdakis, Y., Fleury, M., “Aspectizing Server-Side Distribution”, In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03), IEEE Computer Society, p. 130-141, 2003.

[UML07] The Unified Modeling Language Website, Object Management Group (OMG), http://www.uml.org/

[War03] Warmer, J., Kleppe, A., “Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition”, Addison Wesley Professional Pub, August 27, 2003

[Xu03] Xu, D., Yin, J., Deng, Y., Ding, J., “A Formal Architectural Model for Logical Agent Mobility”, IEEE Transactions on Software Engineering, Vol. 29, N° 1, January 2003.

APPENDIX A

AMBIENT-PRISMA AOADL

A.1 ARCHITECTURAL MODEL

<architectural_model> ::= **Architectural_Model** <model_name>

[<virtual_ambient_block>]

<site_ambient_block>

[<group_ambient_block>]

<component_block>

<connector_block>

[<system_block>]

<attachment_block>

End_Architectural_Model <model_name>**';'**

A.2 INTERFACES

<iservice> ::= <service_name> **'('** [<param_service_list>] **')'** **';'**

<interface> ::= **Interface** <interface_name>

<iservice_list>

End_ Interface <interface_name> **;**

A.3 ASPECTS

<aspect> ::= <aspect_type> **Aspect** <aspect_name>

[**using** <interface_name_list>]

[<constant_attributes>]

[<variable_attributes>]

<services>

[<preconditions>]

[<transactions>]

[<constraints>]

[<played_roles>]

<protocol>

End_Aspect <aspect_name>**';'**

<aspect_type> ::= **functional** | **coordination** | **distribution** | **replication** | **mobility**

A.4 Attributes

<constant_attributes> ::= **Constant** <var_cons_attribute_seq>

<variable_attributes> ::= **Variable** <var_cons_attribute_seq>

<var_cons_attribute> **::=** <attribute_name>**':'** <data_type>[**','** **NOT NULL**]

A.5 Services

<services> ::= **Services**

begin‘(‘ [<param_service_list>]**‘)’‘;’** [<valuations>]

< service_section_seq> [<valuations>]

end‘(‘ ‘)’‘;’

<service_section> ::= <service> [**as** <service_name>]

<service> ::= <service_type> <service_name>**‘(‘** [<param_service_list>]**‘)’**

<service_type> ::= **in** | **out** | **in/out**

<valuations> ::= **Valuations** <valuation_seq>

<valuation> ::= [**‘{‘** <condition> **‘}’**] **‘[‘** <action> **‘]’** <assignation_list>

<action> ::= <service_type> <service_name>**‘(‘** [<parameter_name_list>]**‘)’**

<assignation> ::= <property> **‘:=’** <formulae>

<property> ::= <attribute_name> | <parameter_name>

A.6 Preconditions

<preconditions> ::= **Preconditions** <precondition_seq>

<precondition> ::= <invocation> **if** **‘{‘** <condition> **‘}’**

<invocation> ::= <service_name>**‘(‘** [<parameter_name_list>]**‘)’**

A.7 Transactions

<transactions> ::= **Transactions** <transaction_seq>

<transaction> ::= <transaction_name>**'('** [<param_service_list>] **')'':'**

<initial_transaction_process> <transaction_process_seq>

<initial_transaction_process> ::= <transaction_name>**'::='** <process> **'→'**

<process_name>**';'**

<transaction_service> ::= ['{' <condition> '}'] <transaction_service_type>

<channel_kind> **'('** [<parameter_name_list>]')'

<transaction_service_type> ::= <trans_public_service> | <private_service>

<trans_public_service> ::= <interface_name>**'.'**<service_name>

<channel_kind> ::= <input_channel> | <output_channel>

<input_channel> ::= **'?'**

<output_channel> ::= **'!'**

<private_service> ::= <service_name>

A.8 Constraints

<constraints> ::= **Constraints** <constraint_seq>

<constraint> ::= **always** **'{'** <condition> **'}'** | **sometimes** **'{'** <condition> **'}'** |

['{' <condition> '}'] **next** **'{'** <condition> **'}'** |

<condition_before> **since** <condition_after> |

<condition_before> **until** <condition_after> |

always <condition_before> **since** <condition_after> |

sometimes <condition_before> **since** <condition_after>

<condition_before> ::= <condition>

<condition_after> ::= <condition>

A.9 Played_Roles

<played_roles> ::= **Played_Roles** <played_roles_seq>

<played_role> ::= <played_role_name> **for** <interface_name> **'::='**
<process>**';'**

<played_role_service> ::= <service_name> <channel_kind>
'(' [<parameter_name_list>]**')'**

A.10 Protocol

<protocol> ::= **Protocol** <initial_process> <protocol_process_seq>

<initial_process> ::= <aspect_name> **'::='** **<**process> **'→'** < process_name>**';'**

<protocol_process> ::= **<**process_name> **'::='** <process>
[**'→'** <process_name>]

<protocol_service> ::= [**'{'** <condition> **'}'**] <protocol_service_type>
<channel_kind> **'('** [**<**parameter_name_list>]**')'':'**
<priority>

<process_service_type> ::= <public_service> | <private_service>

<public_service> ::= <played_role_name>**'.'**<service_name>

<priority> ::= <priority_value>

A.11 PORTS

<ports> ::= **Ports** <port_seq> **End_Ports';'**

<port> **::=** <port_name>**':'** <interface_name>',' **Played_Role**
<aspect_name>**'.'**<played_role_name>

A.12 WEAVINGS

<weavings> ::= **Weavings** <weaving_seq> **End_Weavings';'**

<weaving>::= <aspect_name>**'.'**<service_name>**'('** [<parameter_name_list>]**')'**
<weaving_operator> <aspect_name>**'.'**<service_name>
'(' [<parameter_name_list>]**')'**

<weaving_operator> ::= **after** | **before** | **instead** | **afterif'('** <condition> **')'**
| **beforeif'('** <condition> **')'** | **insteadif'('** <condition> **')'**

A.13 Virtual Ambients

<Virtual> ::= **Virtual** <virtaul_name>

```
                              <aspects_importation_seq>
                              [<weavings>]
                              <ports>
                              <creation>
                              <destruction>
             End_Virtual <virtual_name>';'
```

A.14 Site Ambients

```
<Site> ::= Site <site_name>
                              <aspects_importation_seq>
                              [<weavings>]
                              <ports>
                              <creation>
                              <destruction>
          End_Site <site_name>';'
```

A.15 Group Ambients

```
<Group> ::= Group <group_name>
                              <aspects_importation_seq>
                              [<weavings>]
                              <ports>
                              <creation>
                              <destruction>
              End_Group <group_name>';'
```

<aspects_importation> ::= <concern> **Aspect Import** <aspect_name>

<creation> ::= **new**'(' [<param_service_list>]')' '{' <start_aspects_seq> '}'

<destruction> ::= destroy'**(**' '**)**' '{' <stop_aspects_seq> '}'

<start_aspects> ::= <aspect_name>**'.'begin**'(' [<parameter_name_list>]')'

<stop_aspects> ::= <aspect_name>**'.'end'(' ')'**

A.16 COMPONENT

<Component> ::= **Component_type** <component_name>

<aspects_importation_seq>

<weavings>

<ports>

<creation>

<destruction>

End_Component _type<component_name>**';'**

<aspects_importation> ::= <aspect_type> **Aspect Import** <aspect_name>

<creation> ::= **new'('** [<param_service_list>]**')'** **'{'** <start_aspects_seq> **'}'**

<destruction> ::= **destroy'('** **')'** **'{'** <stop_aspects_seq> **'}'**

<start_aspects> ::= <aspect_name>**'.'begin'('** [<param_service_list>]**')'**

<stop_aspects> ::= <aspect_name>**'.'end'('** **')'**

A.17 CONNECTORS

<connector> ::= **Connector_type** <connector_name>

<aspects_importation_seq>

<weavings>

<ports>

<creation>

<destruction>

End_Connector _type <connector_name>**‘;’**

A.18 ATTACHMENTS

<attachments> ::= **Attachments** <attachment_seq> **End_Attachments’;’**

<attachment> ::= <attachment_name> <component_name>**’.’**<port_name>

‘(‘ <card_min_name> **‘..’** <card_max_name **>‘)’** **‘←→’**

‘(‘ <card_min_name> **‘..’** <card_max_name>**‘)’**

<connector_name>**’.’**<port_name>

<card_min> := **string** [2]

<card_max> ::= **string** [2]

A.19 CONFIGURATION

<architectural_model_configuration> ::=

Architectural_Model_Configuration <configuration_name> ‘=’

new <model_name> ‘{‘<loc_instantiation_seq>

[<virtualAmbient_instantiation_seq>]

<siteAmbient_instantiation_seq>

[<groupAmbient_instantiation_seq>]

[2] They are usually natural numbers, but the letter ‘n’ is also used to specify “many instances” without specifying a specific number of instances

<components_instantiation_seq>
[<systems_instantiation_seq>]
<connectors_instantiation_seq>
<attachments_intantiation_seq> '}'

<LOC_instantiation> ::= <loc_name> **'=' new**
loc'(' [<param_value _list>] **')**

<virtualAmbient_instantiation> ::= <virtualAmbient_instance_name> **'=' new**
<virtualAmbient_name>**'('** [<param_value _list>] **')**

<siteAmbient_instantiation> ::= <siteAmbient_instance_name> **'=' new**
<siteAmbient_name>**'('** [<param_value _list>] **')**

<groupAmbient_instantiation> ::= <groupAmbient_instance_name> **'=' new**
<groupAmbient_name>**'('** [<param_value _list>] **')**

<components_instantiation> ::= <component_instance_name> **'=' new**
<component_name>**'('** [<param_value _list>] **')**

<connectors_instantiation> ::= <connector_instance_name> **'=' new**
<connector_name> **'('** [<param_value_list>] **')**

<attachments_instantiation> ::= <attachment_instance_name>[1] **'=' new**
<attachment_name> **'('**<param_attachment_value>**')'**

<systems_instantiation> ::= <system_instance_name>[1] **'=' new** <system_name>
'('[<param_service_value_list>**','**] <architectural_element_number_value_list>,
[<attachment_number_value_list>',' <binding_number_value_list>] **')'**
'{' [<start_aspects_seq>] <architectural_elements_instantiation_seq>
<attachments_instantiation_seq> <bindings_instantiation_seq> **'}'**

<architectural_element_instantiation > ::= <components_instantiation> |
<connectors_instantiation> |
<systems_instantiation>

< bindings_instantiation> ::= <binding_instance_name>[1] **'=' new**
<binding_name> '(' <param_binding_value> ')'

APPENDIX B

AMBIENT-PRISMA SOFTWARE ARCHITECTURE OF THE MOBILE AUCTION CASE STUDY

This appendix presents the complete specification of the *MOBILE AUCTION* using the Ambient-PRISMA AOADL.

B.1 ***Interfaces***

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

```
Interface IGetLocation
 getLocation(output Location: string);
End_Interface IGetLocation
```

```
Interface ICustProc
 notifyProdInterest(input Saleroom:string, input SaleNum:string,
                   input DateOfAuction:string, input Lotdescrip: string,
                   output Interested:Boolean);

End_Interface ICustProc
```

```
Interface ICustBidder
  biddingInf(input ISaleRoom: string, input ISaleNum:string,
             input IDateofAuction:Date, input ILotNumber:string,
             input MaximumBid: double);
  changeMaximumBid(input NewMaximumBid: double);
  biddingStatus(input Quantity: double, input Situation: string);
End_Interface ICustBidder;
```

```
Interface IProcurAuction
searchforlot(input Keywords:string, output Saleroom:string,
             output SaleNum: string, output DateOfAuction:string,
             output Lotdescrip:string);
End_Interface IProcurAuction
```

```
Interface IBidderAuct

 startAuction(input biddingAmount1:double);
 changeStatus(input NewBid: double);
 finishedAuction(input WonBid: double, input Situation: string);
End_Interface IBidderAuct
```

```
Interface ICustAuct
 register(input UserName: string, input Password: string,
          input SaleRoom: string);
End_Interface ICustAuct
```

B.2 *Aspects*

```
 Distribution Aspect CustDist using IMobility

 Attributes
  Constant
  location : string NOT NULL;
 Services
  begin(input ParentAmbient: string)
     Valuations
      [begin (ParentAmbient)] location := ParentAmbient;

  out move (input NewAmbient:string );
  end;

 Played_Roles
  //There are two played roles with the same interface. This is because one
role is for
    the movement of the procurement agent and the other for the Bidder.
  MOVEProc for IMobility ::= move!(input NewAmbient);
  MOVEBidder for IMobility::=move!(input NewAmbient);
 Protocol
 // The protocol specifies that the customer moves the procurement agent and
then it
    either moves the Bidder or ends.
  CUSTDIST:= begin(ParentAmbient)→ CUSTDIST1;
  CUSTDIST1:= MOVEProc.move!(NewAmbient)→CUSTDIST2;
  CUSTDIST2:= MOVEBidder.move!(NewAmbient)+end;

End_Distribution Aspect CustDist
Distribution Aspect ProcurDist using IMobility, ICapability

 Attributes
  Variable
  location: string NOT NULL;
  nextAuctionSiteLoc: string NOT NULL;
  counter: integer(0);
  Constant
   custLocation: string NOT NULL;
 Services
  begin(input ParentAmbient: string, input CustLocation:string,
        input NextAuctionSiteLoc: string);
     Valuations
      [begin (ParentAmbient, CustLocation, NextAuctionSiteLoc)]
          location := ParentAmbient, CustLocation :=custLocation,
          nextAuctionSiteLoc := NextAuctionSiteLoc;
  initializeCounter()
     Valuations
      [initializeCounter ()]
          counter := 0;
  setCounter2()
     Valuations
      [setCounter2 ()]
          counter := 2;
  addCounter()
     Valuations
      [initializeCounter ()]
          counter := counter +1;


  //These services are concerned with mobility in order to interact with the
parent
    ambient.
  in changeLocation(Name: string, NewLocation:string)
        Valuations
          [changeLocation()] location:=NewLocation;
  out startMovement(input Name:string, output CommunicationList[]:
Attachment)
     Valuations
         [startMovement(Name, CommunicationList[])] Name:= self.Name;
  out exit (Name: string, Parent:string )
     Valuations
         [exit(Name, Parent)] Name:= self.Name & Parent:=location;

  out finishMovement(input Name, input CommunicationList[]);
     Valuations
         [finishMovement(Name, CommunicationList[])] Name:= self.Name;

   out enter (Name: string, NewAmbient: string);
     Valuations
         [exit(Name, NewAmbient)] Name:= self.Name
  end;
//
 TRANSACTIONS in move(NewAmbient:string)
      move::= out startMovement(input Name, output CommunicationList[])→
MOVE1;
```

```
      MOVE1::= out exit(Name,Parent)→ MOVE2;
      MOVE2::= out enter(Name, NewAmbient)→MOVE3;
      MOVE3::= out finishMovement(input Name, input CommunicationList[]);
      MOVE4:: in changeLocation(Name, NewLocation);
 Preconditions
 //This precondition is concerned with the interaction with the ambient
      in  changeLocation(Name, NewLocation)
                           if {self.Name==Name};

 triggers
       initializeCounter()  when custLocation==location;
       move(nextAuctionSiteLoc)   when counter==1;
       move(custLocation)   when counter==2;



 Played_Roles
  CapParent for ICapability ::=startMovement!(Name, CommunicationList)→
                                exit!(Name,Parent)→ enter!(Name,
NewAmbient)→
                                finishMovement!(Name,CommunicationList) →
                                changeLocation?(Name, NewLocation);
  CUSTMOVESPROC for IMobility::= move?(NewAmbient);


 Protocol
 // The mobility should be first executed by the customer and then it can
decide.
  PROCURDIST:= begin→ PROCURDIST1;
  PROCURDIST1:= MOVEProc.move!(NewAmbient)→ PROCURDIST2;
  PRDIST2:= move(NewAmbient)+end;

End_Distribution Aspect ProcurDist
```

```
Distribution Aspect BidderDist using IMobility, ICapability

 Attributes
  Variable
  location: string NOT NULL;

  Constant
   custLocation: string NOT NULL;
 Services
  begin(input ParentAmbient: string, input CustLocation: string)
     Valuations
      [begin (ParentAmbient, CustLocation)]
          location := ParentAmbient, CustLocation :=custLocation;


  //These services are concerned with mobility in order to interact with the
parent ambient.
  in changeLocation(Name: String, NewLocation: string)
        Valuations
          [changeLocation()] location:=NewLocation;
  out startMovement(input Name:string, output CommunicationList[]:
Attachment)
     Valuations
         [startMovement(Name, CommunicationList[])] Name:= self.Name;
  out exit (Name: string, Parent:string )
     Valuations
         [exit(Name, Parent)] Name:= self.Name & Parent:=location;

  out finishMovement(input Name, input CommunicationList[]);
     Valuations
         [finishMovement(Name, CommunicationList[])] Name:= self.Name;

   out enter (Name: string, NewAmbient: string);
     Valuations
         [exit(Name, NewAmbient)] Name:= self.Name
  end;
//
 TRANSACTIONS in move(NewAmbient:string)
      move::= out startMovement(input Name, output CommunicationList[])→
MOVE1;
      MOVE1::= out exit(Name,Parent)→ MOVE2;
      MOVE2::= out enter(Name, NewAmbient)→MOVE3;
      MOVE3::= out finishMovement(input Name, input CommunicationList[]);
      MOVE4:: in changeLocation(Name, NewLocation);
 Preconditions
 //This precondition is concerned with the interaction with the ambient
      in  changeLocation(Name, NewLocation)
                                 if {self.Name==Name};


 Played_Roles
  CapParent for ICapability ::=startMovement!(Name, CommunicationList)→
                                exit!(Name,Parent)→ enter!(Name,
NewAmbient)→
                                finishMovement!(Name,CommunicationList) →
                                changeLocation?(Name, NewLocation);
  CUSTMOVESBIDDER for IMobility::= move?(NewAmbient);


 Protocol
 // The mobility should be first executed by the customer and then it can
moveback to the customer location by its own.
  BIDDERDIST:= begin→ BIDDERDIST1;
  BIDDERDIST1:= CUSTMOVESBIDDER.move?(NewAmbient)→ BIDDERDIST2;
  BIDDERDIST2:= move(custLocation)+end;

End_Distribution Aspect BidderDist
```

```
Distribution Aspect ADist using IGetLocation
 Attributes
  Constant
  location : string NOT NULL;
  physicalLocation: loc NOT NULL;
 Services
  begin(input ParentAmbient: string, input PhysicalLocation: lo)
```

```
     Valuations
      [begin (ParentAmbient, PhysicalLocation)]
                          location := ParentAmbient,
                          physicalLocation:=PhysicalLocation;
    in/out getLocation( output Location:string)
     Valuations
         [in getLocation(output Location)] Location := location;

  end;
Protocol
  DIST:= begin→ DIST1;
  DIST1:= (getLocation?(Location)→ getLocation!(Location))+ end;
End_Distribution Aspect ADist
```

```
Distribution Aspect Dist
 Attributes
  Constant
  location : string NOT NULL;
 Services
  begin(input ParentAmbient: string)
     Valuations
      [begin (ParentAmbient)] location := ParentAmbient;
  end;

 Protocol

  DIST:= begin→ end;


End_Distribution Aspect Dist
```

```
Distribution Aspect RootDist using IGetLocation

Attributes
  Constant
  location : string(NULL);

Services
  begin()
     Valuations
      [begin ()]
                          location := null,
   in/out getLocation( output Location:string)
     Valuations
         [getLocation(output Location)] Location := location;
  end;

Protocol
  DIST:= begin→ DIST1;
  DIST1:=(getLocation?(Location)→ getLocation!(Location))+ end;


End_Distribution Aspect RootDist
```

```
Functional Aspect CustFunct using ICustProc, ICustBidder

 Attributes
  Variables
    iSaleroom:string,
    iSaleNum:string;
    iDateAuction:Date;
    iLotNumber:string;
    interested: boolean;
    maximumBid: double;
    currentQuantity: double;
    currentsituation: string;

 Services
   begin;
       setInterested(input CustomerInterest:boolean)
      Valuations
        [setInterested(input CustomerInterest)]
                                        interested:=CustomerInterested;


       saveItem(SaleRoom, SaleNum,DateOfAuction,Lotdescrip)
       Valuations
         {interested==true}[saveItem(SaleRoom, SaleNum, DateOfAuction,
                                    LotNumber)]
                     iSaleRoom:=saleroom, iSaleNum:=SaleNum,
                     iDateOfAuction:=DateAuction,
                     iLotdescrip:=Lotdescrip;
         {interested==false}[saveItem(SaleRoom, SaleNum, DateOfAuction,
                                      LotNumber)]
                     iSaleRoom:=NULL, iSaleNum:=NULL,
                     iDateOfAuction:=NULL,
                     iLotdescrip:=NULL;

   setMaximumBid(input MaximumBid:double)
       Valuations
         [setMaximimBid(input MaximumBid)] maximumBid:=MaximumBid;
   pay(input Quantity);



 //this service is used to create the attachments of the Bidder and the
Procurement at runtime. So the customer connects the agents at runtime with
the auctions before he sends them. He creates the attachments between the
Procurement and the two auctions and then he creates the attachments between
the bidder and the auction it has been decided to bid in.
  out createAttachment( InstanceName:string, Ports: Port: string,
                      InstanceName:string, Port: string);

  //Customer and bidder
  out biddingInf(input iSaleRoom, input iSaleNum, input iDateofAuction,
               input iLotNumber, input maximumBid);
  out changeMaximumBid(input maximumBid);
  in/out biddingStatus( input Quantity: double, input Situation: string)
     Valuations
         [in biddingStatus( input Quantity: double, input Situation: string)]
         currentQuantity:=Quantity, currentSituation: Situation;

//Customer receives from proc the lots available and it answers if it is
//interested
  Transactions
   in/out NOTIFYPRODINTEREST(input Saleroom:string, input SaleNum:string,
                            input DateOfAuction:string,
                            input Lotdescrip: string,
                            output Interested:boolean);

     notifyProdInterest = setInterested?(CustomerInterested)→ SAVEITEM;
     SAVEITEM = saveItem?(SaleRoom, SaleNum, DateOfAuction,LotNumber);
         Valuations
           [NOTIFYPRODINTEREST(SaleRoom, SaleNum,DateOfAuction,Lotdescrip,
Interested)]
           Interested =interested;

     in NewMaximumBid()
      NewMaximumBid::= setMaximumBid(input Maximumbid)→NewMaximumBid1;
      NewMaximumBid1::= out changeMaximumBid(input maximumBid);

  triggers
       setMaximumBid(input MaximumBid)  when interested==true;
       pay(currentQuantity)  when currentSituation==”LOT SOLD TO YOU”;

  Played_Roles
   CUSTPROC for ICustProc ::= sendProductCust?(output Interested: boolean,
                                              input Saleroom,
                                              input SaleNum,
                                              input DateOfAuction,
                                              input Lotdescrip,
                                              input lotNumber)→
                              sendProductCust!(output Interested: boolean,
input Saleroom,
                                             input SaleNum,input
DateOfAuction,
                                             input Lotdescrip, input
lotNumber);
   CUSTBIDDER for ICustBidder ::= biddingInf!(input iSaleRoom, input
iSaleNum, input
                                            iDateofAuction, input
iLotNumber, input
                                            maximumBid)→
                                   (changeMaximumBid!(input MaximumBid)+
                                     biddingStatus!(input Quantity,input
Situation)+
                                     biddingStatus?(input Quantity,input
Situation));


Protocol
       ……

End_Functional Aspect CustFunct
```

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
```

```
 Attributes
  Variables
    keywords: string NOT NULL;
    limitDate: Date NOT NULL;
    saleroom: string;
    saleNum:string;
    dateOfAuction: Date;
    lotdescrip: string;
    keepSearching: boolean;
    finishedSearching:boolean;

 Services
  begin(input Keywords: string, LimitDate: Date)
      Valuations
       [begin(input Keywords, input LimitDate)]
                                 keywords:= Keywords, limitDate:= LimitDate;
  setKeepSearchingToTrue()
      Valuations
       [setKeepSearchingToTrue()] keepSearching:=true;
  finishedSearchingWithoutResults()
      Valuations
       [finishedSearchingWithoutResults()] finishedSearching:=true;

//Procurement and the Auction
   in/out searchforlot(input Keywords:string, output Saleroom:string,
                       output SaleNum:string, output DateOfAuction:string,
                       output Lotdescrip:string)
      Valuations
       {DateOfAuction<= limitDate}[in searchforlot(input Keywords,
                                                   output SaleRoom,
                                                   output SaleNum,
                                                   output DateOfAuction,
                                                   output Lotdescrip)]
          saleroom := Saleroom, saleNum:= SaleNum,
          dateOfAuction:= DateOfauction, lotdescrip:=Lotdescrip;

//Procurement and the Customer
     in/out notifyProdInterest(input Saleroom:string,input SaleNum:string,
                            input DateOfAuction:Date,
                            input Lotdescrip:string,
                            output Interested: boolean);
     Valuations
       {Interested== false}[in notifyProdInterest(input Saleroom, input
SaleNum,
                                                  input DateOfAuction, input
Lotdescrip,
                                                  output Interested)]
                                          keepSearching:=true;
       {Interested== true}[ in notifyProdInterest(input Saleroom, input
SaleNum,
                                                  input DateOfAuction, input
Lotdescrip,
                                                  output Interested)]
                                          keepSearching:=false;

   end;
  triggers
     notifyProdInterest (Saleroom,SaleNum, DateOfAuction, Lotdescrip,
                          Interested)
                       when {DateOfAuction<=limitDate};
     searchforlot(input Keywords, output Saleroom, output SaleNum,
                  output DateOfAuction, output Lotdescrip)
                       when  keepSearching==true;
     searchforlot(input Keywords, output Saleroom, output SaleNum,
                  output DateOfAuction, output Lotdescrip)
                       when  {dateOfAuction<=saleDate & saleroom==NULL};

Played_Roles
CUSTPROC for ICustProc ::= notifyProdInterest!(Saleroom, SaleNum,
                                             DateOfAuction, Lotdescrip,
                                             Interested)
                            →
                            notifyProdInterest?( Saleroom, SaleNum,
                                             DateOfAuction, Lotdescrip,
                                             Interested);
 PROCURAUCT for IProcurAuction ::= searchforlot!(Keywords, SaleRoom,
                                                 SaleNum, DateOfAuction,
                                                 Lotdescrip)
                                   →
                                   searchforlot?(Keywords, SaleRoom,
                                                 SaleNum, DateOfAuction,
                                                 Lotdescrip);

Protocol

  PROCURFUNCT:= begin(Keywords, LimitDate)→ PROCURFUNCT1;
  PROCURFUNCT1:= setKeepSearchingToTrue() + end()+ PROCURFUNCT2;
  PROCURFUNCT2:= PROCURACUCT_searchforlot!(Keywords, SaleRoom,
                                                 SaleNum, DateOfAuction,
                                                 Lotdescrip)
                 →
                 PROCURACUCT_searchforlot?(Keywords, SaleRoom,
                                                 SaleNum, DateOfAuction,
                                                 Lotdescrip)
                 →
                 (PROCURFUNCT2 + PROCURFUNCT3
                 + finishedSearchingWithoutResult?()) ;
  PROCURFUNCT3:= CUSTPROC_notifyProdInterest!(Saleroom, SaleNum,
                                    DateOfAuction, Lotdescrip,
                                    Interested)
                  →
                 CUSTPROC_notifyProdInterest?(Saleroom, SaleNum,
                                    DateOfAuction, Lotdescrip,
                                    Interested)
                  →
                 (PROCURFUNCT1 + PROCURFUNCT2);

End_Functional Aspect ProcurFunct
```

```
Functional Aspect BidderFunct using ICustBidder, IBidderAuct

Attributes
  Variable
    lotSaleRoom: string;
    lotSaleNum: string;
    lotDateofAuction: string;
    lotNumber: string;
    lotMaximumBid: real;
    currentBiddingAmount: real;
    biddingSituation: string ;

Services

  //Customer and bidder
  in biddingInf(input iSaleRoom: string, input iSaleNum:string, input
                iDateofAuction:Date, input iLotNumber:string, input
MaximumBid: real)
     Valuations
         [in bidding Inf(input iSaleRoom, input iSaleNum, input
iDateofAuction, input
                         iLotNumber, input maximumBid)]
         lotSaleRoom:=iSaleRoom, lotSaleNum:=iSaleNum,
lotDateofAuction:=iDateofAuction,
         lotNumber:= iLotNumber, lotMaximumBid:= maximumBid;

  in changeMaximumBid(input NewMaximumBid:double)
     Valuations
         [in changeMaximumBid(input NewMaximumBid)]
           lotMaximumBid:= NewMaximumBid;
 //the bidder tells the customer the current bid and if it with it, against
it, lot sold
   to it, finished.
  out biddingStatus(input currentBiddingAmount, input biddingSituation);


// bidder and auction

   in startAuction(input biddingAmount1:real)
      Valuations
         [startAuction(input biddingAmount1)]
              currentBiddingAmount:=biddingAmount1, biddingSituation:="lot
against you";
   in changeStatus(input NewBid: real)
      Valuations
         [changeStatus(input NewBid)]
             currentBiddingAmount:=NewBid, biddingSituation:="lot against
you";

   in/out bid(input currentBiddingAmount: real, output Situation:string)
     Valuations
          [out bid(input currentBiddingAmount, output Situation )]
              Situation:= biddingSituation;
   in finishedAuction(input WonBid: real, input Situation: string)
       Valuations
         [finishedAuction(input WonBid, input Situation)
              currentBiddingAmount:=CurrentBid, biddingSituation:=Situation;

  out register(input UserName, input Password, input iSaleRoom);

triggers
   out bid(input currentBiddingAmount, output Situation) when
        biddingSituation=="bid against you" &

        currentBiddingAmount<=lotMaximumBid;

   out biddingStatus(currentBiddingAmount, biddingSituation) when
```

```
                                    biddingSituation==”lost” or
biddingSituation==”LOT SOLD TO YOU”;

Played_Roles

CUSTBIDDER for ICustBidder ::= biddingInf?(input iSaleRoom, input iSaleNum,
input
                                             iDateofAuction, input
iLotNumber, input
                                             maximumBid)→
                                   (changeMaximumBid?(input MaximumBid)+
                                     biddingStatus!(input Quantity,input
Situation));
BIDDERAUCT for IBidderAuct ::= register!(input UserName, input Password,
iSaleRoom)→
                               startAuction?(input biddingAmount1)→
                               bid!(input currentBiddingAmount, output
Situation )→
                               (changeStatus?(input NewBid)
                                + finishedAuction(input WonBid, input
Situation));


End_Functional Aspect BidderFunct
```

```
Functional Aspect AuctFunct using IProcurAuction, IBidderAuct, ICustAuct

 Attributes
  Variables

    saleroom: string NOT NULL;
    saleNum: string NOT NULL;
    dateOfAuction: Date NOT NULL;
    lotdescrip: string NOT NULL;
    biddingAmount: double NOT NULL;
    nextBid: real;


 Services
  begin(input Saleroom: string, input SaleNum: string,
        input DateOfAuction: Date, input Lotdescrip: string,
        input StbiddingAmount: double)
      Valuations
       [begin(input Saleroom, input SaleNum, input DateOfAuction,
              input Lotdescrip, input StbiddingAmount)]
        saleroom:= Saleroom, saleNum:=SaleNum, dateOfAuction:=DateOfAuction,
        lotdescrip:=Lotdescrip, StbiddingAmount:= biddingAmount,


  register(input UserName: string, input Password:string,
           input SaleRoom: string);
```

```
//Procurement and the Auction. The auction returns a lot with which are
compatible with the keywords in the lot description. It checks the keywords
and returns a product or another
in/out searchforlot(input Keywords:string, output Saleroom:string,
                    output SaleNum:string, output DateOfAuction:string,
                    output Lotdescrip:string)
      Valuations
        {Keywords==lotDescrip}[in searchforlot(input keywords,
                                               output SaleRoom,
                                               output SaleNum,
                                               output DateOfAuction,
                                               output Lotdescrip)]
          Saleroom := saleroom1, SaleNum:= saleNum1,
          DateOfAuction:= dateOfAuction1, Lotdescrip:=lotdescrip;

  incrementNextBid()
     Valuations
      {biddingAmount>=100 & biddingAmount<200}
                       [incrementNextBid()]nextBid:= biddingAmount+10;
      {biddingAmount>=200 & biddingAmount<300}
                       [incrementNextBid()]nextBid:= biddingAmount+20;

//Bidder and the Auction.
   in register (input UserName, input Password, iSaleRoom)
   out startAuction(input biddingAmount1, input "bid against you");
   out finishedAuction(WonBid, situation);
   out changeStatus(input nextBid);


 in bid( input bidAmount: real, output Situation: string)
      Valuations
      {bidAmount==nextBid} biddingAmount:= bidAmount, Situation:= "bid with
you";

trigger
   incrementNextBid() when biddingAmount==nextBid;


 Played_Roles

  PROCURAUCT for IProcurAuction ::= searchforlot?(input Keywords,
                                                 output SaleRoom,
                                                 output SaleNum,
                                                 output DateOfAuction,
                                                 output Lotdescrip)
                                    →
                                    searchforlot!(input Keywords,
                                                 output SaleRoom,
                                                 output SaleNum,
                                                 output DateOfAuction,
                                                 output Lotdescrip);
  BIDDERAUCT for IBidderAuct ::= register?(input UserName,
                                          input Password, iSaleRoom)→
                                 startAuction!(input biddingAmount1)→
                                 (bid?(input currentBiddingAmount,
                                       output Situation )+
                                  changeStatus?(input NewBid)+
                                  finishedAuction(input WonBid,
                                                 input Situation));
 Protocol
 //

End_Functional Aspect AuctFunct
```

```
Coordination Aspect AuctionCnctrCoor using IAuct

  Services
    begin;
        in/out bid(input ProductID: string, input Quantity: real);
       in/out lookforProduct(input Description: string,
                             output ProductID: string ,
                             output CurrentBid:  real);
       in/out notifyBuy(Buy : bool);
       in/out registerForPayment( input UserName: string,
                                  input Password: string,
                                  input ProductID: string,
                                  output Register: bool);
    end;

  Played_Role
    AUCTCUST for IAuct =
    (
      (bid?( ProductID,Quantity )
       →
       (bid!( ProductID,Quantity ) )
      +
      (lookforProduct?( Description, ProductID, CurrentBid )
       →
       (lookforProduct!( Description, ProductID, CurrentBid) )
      +
      (notifyBuy?( Buy )
       →
       (notifyBuy!( Buy ) )
      +
      (registerForPayment?( UserName, Password, ProductID, Register)
       →
       (registerForPayment!( UserName, Password, ProductID, Register) )
    );

        CUSTAUCT for IAuct =
    (
      (bid?( ProductID,Quantity )
       →
       (bid!( ProductID,Quantity ) )
      +
      (notifyBuy?( Buy )
       →
       (notifyBuy!( Buy ) )
    );


  Protocol
    AUCTIONCNCTRCOOR = begin.COORD;
    COORD =
    (
      (AUCTCUST.bid?( ProductID,Quantity ) →
       CUSTAUCT.bid!( ProductID,Quantity ) ).COOR
    +
      (AUCTCUST.lookforProduct?( Description, ProductID, CurrentBid ) →
       CUSTAUCT.lookforProduct!( Description, ProductID, CurrentBid)).COOR
    +
      (AUCTCUST.notifyBuy?( Buy ) )→
       CUSTAUCT.notifyBuy!( Buy )).COOR
    +
      (AUCTCUST.registerForPayment?( UserName, Password, ProductID, Register)
→
       CUSTAUCT.registerForPayment!( UserName, Password, ProductID, Register)
).COOR
    +
      (CUSTAUCT.bid?( ProductID,Quantity ) )→
       AUCTCUST.bid!( ProductID,Quantity )  ).COOR
    +
      (CUSTAUCT.notifyBuy?( Buy ) )→
       AUCTCUST.notifyBuy!( Buy ) ).COOR
    +
        end
    );

End_Coordination Aspect AuctionCnctrCoor;
```

```
Coordination Aspect AgentCustCnctrCoor using ICustAgent, IMobility

  Services
    begin;
     in/out purchase(changeProductDescription(input ProductDescr: string);
     in/out changeMaxBidQuantity(input NewMaxBidQuantity: real);
     in/out move(NewAmbient: string);
    end;

  Played_Role
    CUSTAGENT for ICustAgent =
    (
      (changeProductDescription?( ProductDescr)
       →
       changeProductDescription!( ProductDescr) )
      +
      (changeMaxBidQuantity?( NewMaxBidQuantity)
       →
       changeMaxBidQuantity!( NewMaxBidQuantity) )
    );
    AGENTCUST for ICustAgent =
    (
      (changeProductDescription?( ProductDescr)
       →
       changeProductDescription!( ProductDescr) )
      +
      (changeMaxBidQuantity?( NewMaxBidQuantity)
       →
       changeMaxBidQuantity!( NewMaxBidQuantity) )
    );
        AGENTCUSTMOB for IMobility =
    (
     move?( NewAmbient)
       →
      move!( NewAmbient)
    );
    CUSTAGENTMOB for IMobility =
    (
      move?( NewAmbient)
       →
       move!( NewAmbient)
    );

  Protocol
    AGENTCUSTCNCTRCOOR = begin.COORD;
    COORD =
    (
      (CUSTAGENT.changeProductDescription?( ProductDescr ) →
        AGENTCUST.changeProductDescription!( ProductDescr ) ).COOR
    +
      (AGENTCUST.changeProductDescription?( ProductDescr ) →
        CUSTAGENT.changeProductDescription!( ProductDescr ) ).COOR
    +
      (AGENTCUSTMOB.move?( NewAmbient) →
        CUSTAGENTMOB.move!( NewAmbient) ).COOR
    +
        (CUSTAGENTMOB.move?( NewAmbient) →
        AGENTCUSTMOB.move!( NewAmbient) ).COOR
    +
        end
    );

End_Coordination Aspect AgentCustCnctrCoor;
```

B.3 Ambients

```
Ambient_Site type HostSite
Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Ports
     InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
     ECapabilitiesPort: ICapability Played_Role Mobile.Child;
     EServicesPort: ICall Played_Role ACoordination.Client;
     InServicesPort: ICall Played_Role ACoordination.Server ;

 End_Ports
 Weavings
     ADist.getLocation(Location)instead
                                MobilityAspect.getParent(Parent);

End Ambient_Site type HostSite;
```

```
Ambient_Virtual type Root

 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect RootDist;

 Ports
     InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
     ECapabilitiesPort: ICapability Played_Role Mobile.Child;
     EServicesPort: ICall Played_Role ACoordination.Client;
     InServicesPort: ICall Played_Role ACoordination.Server ;
 End_Ports

 Weavings
     Dist.getLocation(Location) instead
                                MobilityAspect.getParent(Parent);

End Ambient_Virtual type Root;
```

B.4 Components

```
Component_type Customer

 Import Distribution Aspect CustDist;
 Import Functional Aspect CustFunct;
 Weavings
  CustDist.move(NewAmbient) after CustFunct.biddingInf(iSaleRoom,
                                                       iSaleNum,
                                                       iDateofAuction,
                                                       iLotNumber,
                                                       maximumBid);
 End_Weavings

 Ports
     MOVEProcPort: IMobility Played_Role CustDist.MOVEProc;
     MOVEBidderPort: IMobility Played_Role CustDist.MOVEBidder;

     CUSTPROCPort: ICustProc Played_Role CustFunct.CUSTPROC;
     CUSTBIDDERPORT: ICustBidder Played_Role CustFunct.CUSTBIDDER
     CUSTAUCTPort: ICustAuct Played_Role CustFunct.CUSTAUCT;
 End_Ports
   new(input ParentAmbient: string)
       {
        CustDist. begin(ParentAmbient);
        CustFunct.begin();
        }
        {
             CustDist.end();
             CustFunct.end();
         }


End Component Customer;
```

```
Component_type Procurement
 Import Distribution Aspect ProcurDist;
 Import Functional Aspect ProcurFunct;
 Ports
     DCapPort: ICapability Played_Role ProcurDist. CapParent;
     DMovingPort: IMobility Played_Role ProcDist.CUSTMOVESPROC;
     CUSTPROCPort: ICustProc Played_Role ProcurFunct. CUSTPROC;
     PROCURAUCTPort: IProcurAuction Played_Role ProcurFunct. PROCURAUCT;
 End_Ports

 Weavings
   //This weaving is for moving to the customer location
   ProcurDist.setCounter2()
        afterif(Interested==true)
                       ProcurFunct.in notifyProdInterest(input Saleroom,
                                                         input SaleNum,
                                                         input DateOfAuction,
                                                         input Lotdescrip,
                                                         output Interested);

   ProcurDist.addcounter() after
                       ProcurFunct.finishedSearchingWithoutResults();

   ProcurFunct. setKeepSearchingToTrue() after
                       ProcurDist.move(nextAuctionSiteLoc);
 End_Weavings

End Component_type Procurement;
```

```
 Component_type Bidder
 Import Distribution Aspect BidderDist;
 Import Functional Aspect BidderFunct;
 Ports
     DCapPort: ICapability Played_Role BidderDist. CapParent;
     DMovingPort: IMobility Played_Role BidderDist.CUSTMOVESBIDDER;
     CustBidderPort: ICustBidder Played_Role BidderFunct.CUSTBIDDER;
     BidderAuctPort: IBidderAuct Played_Role BidderFunct. BIDDERAUCT;
 End_Ports

 Weavings
   //This weaving is for moving to the customer location after the auction
has finished.

   BidderDist.move(custLocation) after BidderFunct.finishedAuction(input
WonBid, input
   Situation);
 End_Weavings

End Component_type Bidder;
```

```
Component_type Auction
 Import Distribution Aspect Dist;
 Import Functional Aspect AuctFunct;
 Ports
     ProcurAuctPort: IProcurAuction Played_Role AuctFunct.PROCURAUCT;
     CustAuctPort: ICustAuct Played_Role AuctFunct.CUSTAUCT;
     BidderAuctPort: IBidderAuct Played_Role AuctFunct.BIDDERAUCT;
 End_Ports


End Component_type Auction;
```

B.5 Connectors

```
Connector_type CustomerCnctr
 Import Distribution Aspect Dist;
 Import Coordination Aspect CustCnctrCoor;
 Ports
    CustPortProcur: ICustProc;
    CustPortProcurMo: IMobility;
    CustPortBidder: ICustBidder
    CustPortBidderMo: IMobility;´
    ProcurPortCust: ICustProc;
    ProcurPortCustMo: IMobility;
    BidderPortCust: ICustBidder
    BidderPortCustMo: IMobility;
 End_Ports

End Connector_type CustomerCnctr;
```

```
Connector_type AuctionCnctr
 Import Distribution Aspect Dist;
 Import Coordination Aspect CustCnctrCoor;
 Ports
    AuctPortCust: ICustAuct;
    CustPortAuct: ICustAuct;
    CnctAuctPortBidder: IBidderAuct;
    BidderPortAuct: IBidderAuct;
    ProcurPortAuct: IProcurAuction;
    AuctPortProcur: IProcurAuction;
 End_Ports

End Connector_type AuctionCnctr;
```

B.6 Configuration

```
Architectural_Model_Configuration AuctionConf =

New MobileAuction
{
       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)

       Auction1 = new AuctionHouse (“London, King street”, 1876”,
                                  “1 Jun”, “Spanish painting”, “800”,
                                  “AuctionSite”);
         Customer1 = new Customer(“ClientSite”);
         Procurement1 = new Procurement (“ClientSite”);
         Bidder1 = new Bidder(“painting”, “3 Jul”, “ClientSite”);
         AgentCustCnct1 = new AgentCustCnct(“ClientSite”);
         AuctionCnct1 = new AuctionCnct(“AuctionSite”);
         AttchAuct1Cnct = new AttchAuctCnct (AuctionCnct1,
                                         CnctAuctPortBidder,
                                         AuctionHouse1,
                                         BidderAuctPort);
         AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                                  AuctionCnct1, CustPortAuct);
       ATT1 = new CustCnctrAtt(CUST, AgentsPort,
                              CustAgentPort, ACNCTR);
       ATT2 = new CustCnctrAtt(CUST, MovePort,
                              AccountOp_port, ACNCTR);
       ATT3 = new CustAuctCnctrAtt(CUST, AuctionPort,
                              CustAgentMobPort, ACNCTR);
       ATT4 = new CnctrAgntsAtt(ACCNCTR, AgentCustPort,
                              EDistServicesPort, AAP);
       ATT5 = new CnctrAgntsAtt(ACCNCTR, AgentCustMobPort,
                              EMobilityPort, AAP);
       ATT6 = new AgntsHostAtt(AAP, DCapabilitiesPort,
                              ICapabilitiesPort, CLIENTSITE);
       ATT7 = new PurAAPAtt(PUR, MoveAgentPort,
                              InMovilityPort, AAP);
       ATT8 = new PurAucAtt(PUR, PURAUCPort,
                              AuctionPortCUST, AUCNCTR);
       ATT9 = new PurCnctrAtt(PUR, COLLPURPort,
                              PurchaserPort, ACNCTR);
       ATT10 = new ColCnctrAtt(COL, CollPurPort,
                                CollectorPort, ACNCTR);
       ATT11 = new ColAuctAtt(COL, COLLAUCTPort,
                                AuctionPortCUST, AUCNCTR);
       ATT12 = new ColCustAtt(COL, CustAgentPort,
                                CustAgentPort, ACCNCTR);
       ATT13 = new AuctCnctrAtt(AUCT, AuctionPort,
                                AuctionPortAUCT, ACNCTR);

};
```

INDEX