Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

DISTRIBUTION IN AN ASPECT-ORIENTED COMPONENT BASED SOFTWARE ARCHITECTURE THROUGH PRISMA

"DISTRIBUCIÓN EN UNA ARQUITECTURA SOFTWARE ORIENTADA A ASPECTOS Y COMPONENTES PRISMA"

Programa de Doctorado: **Programación Declarativa e Ingeniería de la Programación**

Nour Ali

Dirigido por: Dr. Isidro Ramos Salavert

Dr. Jose Ángel Carsí Cubel

Valencia, Septiembre 2004

ABSTRACT

This research work provides a conceptual model for specifying distributed systems. The conceptual model describes how distributed software architectures can be described through PRISMA. PRISMA, is an architectural model which combines the Aspect-Oriented Software Development (AOSD) and the Component-Based Software Development (CBSD). Therefore, PRISMA has been extended incorporating primitives that allow the mobility, replication and distributed communication of its architectural elements. In addition, the PRISMA Architecture Description Language (ADL) incorporates the constructs for the distribution model. A graphical notation for the distribution primitives in PRISMA ADL is defined.

In addition, this research describes some distribution patterns. Theses patterns describe situations in which the software architecture needs to reconfigure its location topology at run-time either by moving or replicating its architectural elements to cope with fault tolerance problems, new system requirements or other changes in the performance of the software architecture. The patterns are presented through a template to enable their understanding and reusability. The PRISMA ADL has been used to describe them.

ACKNOWLEDGMENTS

This work is dedicated for all those who believe in peace and harmony.

To my parents, for their love, care and encouragement.
To my brother and sisters, Mohamed, Majd and Dana, and to all my Family

To Isidro, for accepting me as a student,
To Pepe, for his comments and advisments
I will gratitude you eternally

To Jenny, Pato and Javi,
For the time they once spent discussing this research,
For interchanging their knowledge
In areas as software architectures, evolution, modelling, distribution and mobility.

I thank the ISSI group for accepting me as another member
To Hilario for sending mails which are so funny,
To Isidro, Mari Carmen, Pepe, Pato and Javi,
To my Lab partners Carlos, Manolo, Artur, Maria Eugenia, And my wonderful Elena and Jenny

To Isa, Gonzalo, Manolo, Artur, Carlos and Elena,
All of you form part of my "Vella" Family,
For the wonderful times I spent with you
Javi, Jenny and Nelly

To my home and land, to my holy children
I ask God to make your future bright
And to live peacefully.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

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

Moreover, considering distribution at analysis and design phases of software development generates high quality distributed applications. This increment of quality is due to taking into account distribution from the beginning of software development so applications are prepared to support its non-functional requirements. However, if distribution is not considered from the beginning, applications should be changed when they arrive to an implementation phase because they are not prepared to support distribution and the traceability between phases of the life cycle is usually lost.

Actually, many CASE tools are able to generate applications following the automatic prototyping paradigm which was proposed by Balzer [8]. They are called model compilers and are able to automatically generate applications from conceptual models. During the last decade, the research efforts were dedicated to formalize the models to automatically generate applications mainly using the object oriented paradigm such as OASIS [51], Oblog[50] or Troll [23].

As a consequence of the poor capability of the object-oriented software development to describe complex structures of distributed information systems, the Component-Based Software Development (CBSD) and the Aspect-Oriented Software Development (AOSD) have emerged. The CBSD [68] promises to control the complexity of system construction by

coupling entities that provide specific services. The AOSD [7] allows separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. Aspects can be reused and manipulated independently of the rest of properties of the system. However, currently no model compilers exist that combine the CBSD and AOSD to generate distributed applications.

In Section 1.1 of this chapter, the objectives of the research work are introduced; then a brief presentation of the PRISMA architectural model. Finally, the structure of the document is presented.

1.1 Research Objectives

The objective of this research work is to provide conceptual models with the PRISMA specification language and the graphical notation for the description of distributed software systems using the CBSD and AOSD.

The necessary distribution primitives are included in PRISMA [54] architectural models in order to be able to describe the architectures of distributed systems. In this way, PRISMA becomes a modelling specification language of distributed systems through specifying the distribution properties at the analysis and design phases of software development.

In the future, the PRISMA architectural model is pretended to become a framework that permits the automatic generation of distributed information systems. For the possibility of this framework the OASIS conceptual model languages have been extended to preserve its evolution and code generation capacity.

1.2 PRISMA Architectural Model

PRISMA [54] is a model that integrates the component based software development [68] (CBSD) and the aspect oriented software development [7] (AOSD) to define architectural models. In addition, PRISMA has reflexive properties through a meta-level that allows the architectural elements to evolve and to reconfigure dynamically at execution time. PRISMA defines its architectures by using an architecture description language (ADL) that describes the architectural models at two levels: the type definition level and the configuration level.

1.2.1 AOSD in PRISMA

An information system is characterized through a set of concerns. These concerns which are common in a system (crosscutting concerns) are separated in the PRISMA model in reusable entities called aspects. In addition, these entities do not only increase the reusability but also improve the maintenance of the architectural models due to the fact that the modification of a concern only affects on its proper characteristics which are not tangled but centralized.

PRISMA uses the AOSD to define the internal behaviour and structure of its architectural elements (white box). Each characteristic of the architectural element is separated in an aspect. The aspects that define an architectural element depend on the information system. Each architectural element of an information system can be seen as a prism in its AOSD point of view (the internal view of the architecture element). Each face of the prism is an aspect that defines a specific characteristic. For example, the AOSD view of the architectural element in Figure 1 could belong to a web distributed information system which takes into account context awareness, quality, navigational and security features.

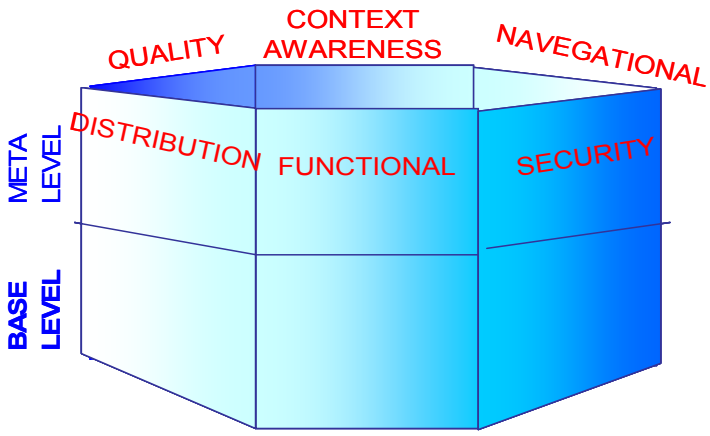
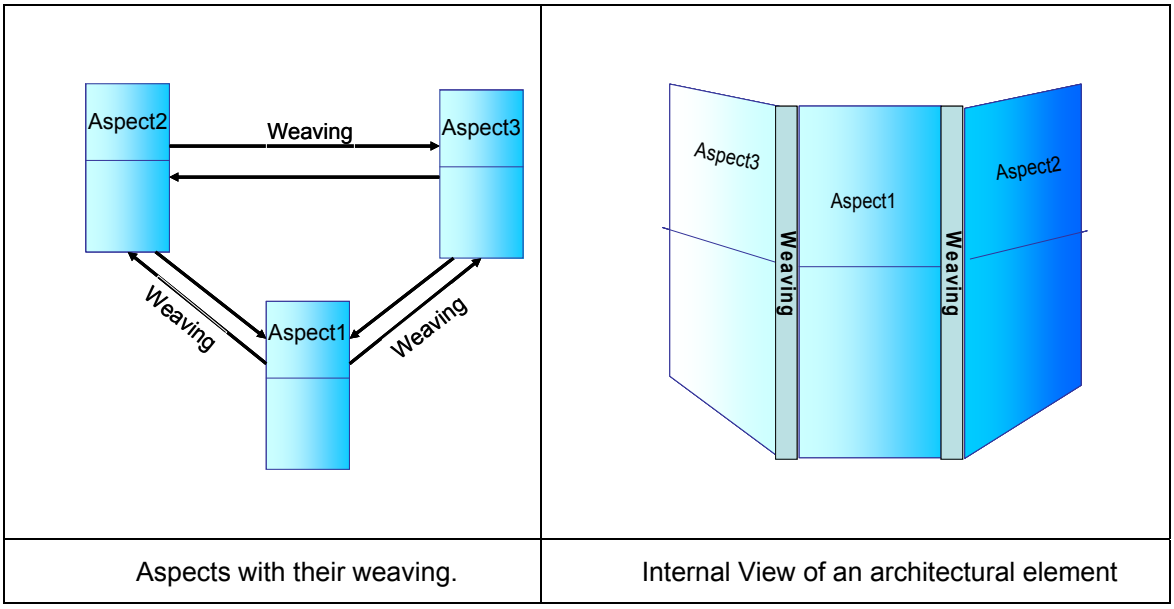


Figure 1 The architectural element seen as a prism in the AOSD point of view

The following aspects and others can be used to describe architectural elements varying in different application domains:

- **Functional Aspect**: The functional aspect captures the semantics of the information system by defining its attributes and behaviour.
- **Coordination Aspect**: The coordination aspect defines the business rules and the synchronisations between architectural elements during their communication.
- **Context-Awareness**: The context-awareness aspect [29] provides information about the context and supports the analysis for retrieving structural properties of the provided information.
- **Quality Aspect**: The quality aspect specifies quality non-functional requirements. It defines the quality attributes that a system needs to evaluate and the quality level that an architectural element needs to satisfy.
- **Navigational Aspect**: The navigational aspect specifies the hyperlinks of the components and the multimedia information they contain.

As architectural elements are formed by a set of aspects, these aspects should be synchronized in order to define the overall behaviour of an architectural element. The synchronizations among the aspects are the aspect weavings [52]. The weavings among the different aspects of an architectural element is defined externally to the aspects in order to provide the total reusability of the aspects in different architectural elements. The weaving is done in such a way to perform the gluing among the aspects and adapt each aspect to the requirements of the architectural elements performance (see Figure 2). Therefore, the weavings of the aspects are defined in the architectural elements specification.



Aspects with their weaving.	Internal View of an architectural element

Figure 2 The weaving is performed externally to the aspects

The synchronizations of the aspects is done by defining the order of execution of the services of each cooperating aspect. Therefore the weaving process is defined through methods. The weaving methods are temporal operations that describe the temporal order of the weaving process. For simplicity, we have the *after*, *before* and *around* methods that are typical to the AspectJ [31] methods which their semantics are as follows:

-after: aspect1.service is executed after aspect2.service

-before: aspect1.service is executed before aspect2.service

-around: aspect1.service is executed except of aspect2.service.

1.2.2 CBSD in PRISMA

The CBSD separates the parts of the system into reusable entities called architectural elements that provide specific services. The PRISMA architectural model has the following architectural elements following the CBSD point of view (black box) (see Figure 3):

- Components: A component is an architectural element that represents part of the functionality of the information system and does not act as a coordinator among other architectural elements. Each component is formed by an identification function, a set of aspects (functional, context-awareness, etc), the internal synchronization of the aspects and one or more ports which's type is a determined interface, which publishes a set of services that are offered and received to and from the environment.
- Connectors: A connector is an architectural element that acts as a coordinator among different architectural elements. A connector is formed by an identification function, a set of aspects (coordination, context-awareness), the weaving of the aspects and one or more roles which's type is a determined interface. The roles represent the interaction points of the different architectural elements that a connector coordinates.
- Systems: A system is a complex architectural element that contains attached components, connectors and other systems. The system can be seen as a component that has its own ports that offer and receive services. The connection relation that establishes a communication channel between the instances of the architectural elements is called an *attachment*. An attachment connects a component port with a connector role. The specification of the connection among the elements that a system includes and the system itself is done through the *bindings*. The bindings are the connections between the ports of the system and the ports or roles of the encapsulated elements.

Figure 3 The external view of an architectural element as a black box with the ports or roles for components or connectors, respectively

An architectural model is a set of instances of components, systems and connectors interconnected between them. For this reason, the instantiation of the architectural elements of a model and the established connections between them are necessary in order to obtain an executable architectural model. A PRISMA architectural model is defined by reusing the different architectural elements that have been defined and stored previously in the PRISMA library. The architectural model is defined by using the attachments connection relationship

which is the same as the attachments relationship used to define the architectural element system.

1.2.3 PRISMA's Meta-Level

PRISMA models have reflexive capabilities through a meta-level. The meta-level permits schema definitions to be converted to data and enable their modification achieving the capacity to evolve such schemas. PRISMA meta-model [55] is defined through meta-classes for each element of a PRISMA architectural model which can evolve and reconfigure. In Figure 4, the meta-model of a PRISMA architectural element is shown. As it can be seen each PRISMA concept and its relationship with the other concepts is defined using the meta-classes such as the meta-class component or connector.

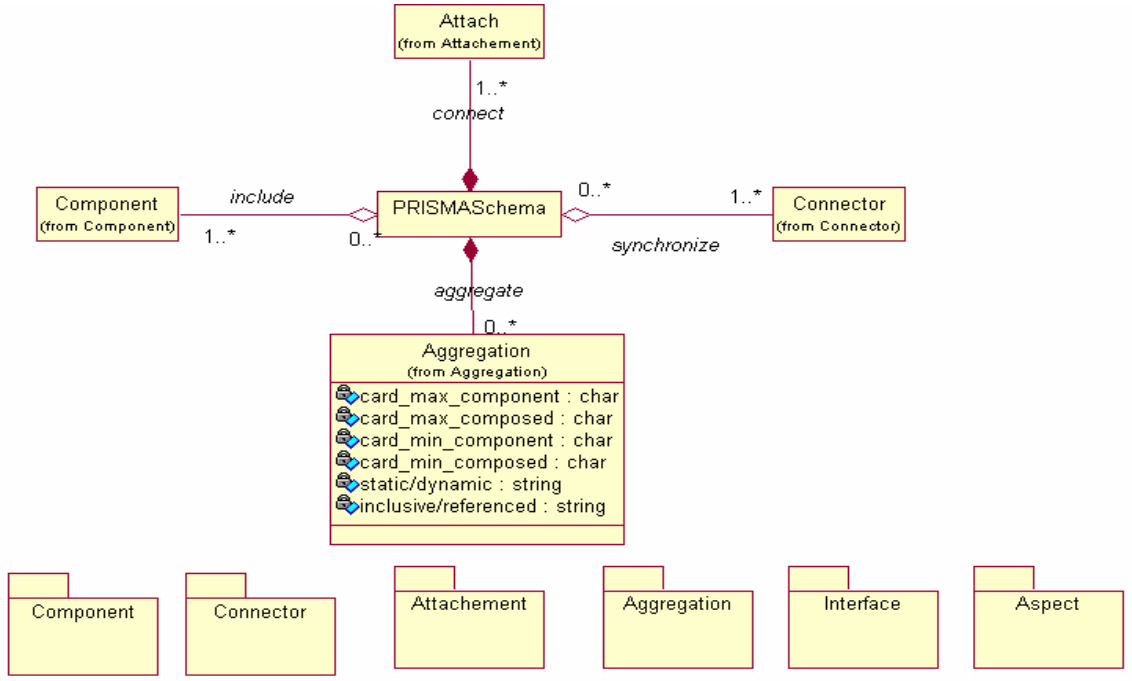


Figure 4 The UML package of the PRISMA architectural model in the meta-level.

PRISMA uses the same approximation used for OASIS schemas in the works of Carsí [13]. Nevertheless, a difference between the OASIS approximation and PRISMA´s is that the meta-level can be generated from the base-level through a reification process. The reification process consists in the partial automatic generation of the needed meta-level to evolve a specific architectural model. The reification process generates the code of the needed meta-classes and their meta-objects in order to evolve the sections of the architecture that are considered volatile. The PRISMA compiler generates the meta-level by only using the sections of the meta-model which allow it to evolve the volatile part of the architecture.

For example, Figure 5 shows the reification process of an aspect. Each aspect of a type has a base level and may have a meta-level. The reification is done from the base level. For

this reason, it is impossible for a type to specify its meta-level without specifying its base level. An aspect defines a meta-level if it reifies some property of the base level.

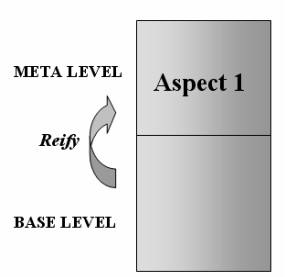


Figure 5 Aspect Reification Process

1.2.4 PRISMA ADL

The PRISMA ADL extends the OASIS [33] (Open Active Software Information Systems) language. This has been necessary in order to allow the description of software architecture concepts such as components, connectors, architectural models, etc. On the other side, the definition of aspects in the language has the OASIS template for defining classes.

The interfaces, aspects, architectural elements (components and connectors) are specified in the PRISMA ADL at the type level as first order citizens of the language. Each first order citizen type is stored in a PRISMA library to enable their reusability, not only in the same architectural model but also in different ones. For example, the same functional aspect can be reused in different components that have the same functionality and can be reused in different components of different architectural elements.

Last but not least, an architectural model (see Figure 4) is defined by using the PRISMA ADL at the configuration level. At the configuration level the architectural model is defined by importing the types it needs from the PRISMA library defined in the type level and instantiating them. Next, the topology of the architecture is specified by attaching components and connectors through an entity called the attachments.

1.3 Structure of the Document

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

Chapter 2 presents the related work to ours. The state of art of the component based software development, the aspect oriented software development, architecture description

languages for distributed systems, aspect oriented software development for distributed systems are pointed out.

Chapter 3 introduces the distribution model of PRISMA. The PRISMA metamodel incorporates to it the necessary concepts to enable it to be distributed, mobile and replicable. The concepts are incorporated following the integration of the AOSD and CBSD of PRISMA. Such that distribution and replication aspects are introduced and the attachments and binding links are extended to become location aware. Then, the architecture description language (ADL) at the type definition level and configuration level for the concepts introduced are fully explained. Next, a brief introduction of the PRISMA infrastructure is introduced. Finally, a brief explanation is given to show how the different decisions to reach the final model where taken.

Chapter 4 describes some distribution patterns. Theses patterns describe situations in which the software architecture needs to reconfigure its location topology at run-time either by moving or replicating its architectural elements to cope with fault tolerance problems, new system requirements or other changes in the performance of the software architecture. The patterns are presented through a template to enable their understanding and reusability. The PRISMA ADL has been used to describe them.

Chapter 5 defines a graphical notation for the distribution primitives in PRISMA ADL defined in this work. This is done by extending OMG's Unified Modelling Language (UML) through incorporating the distribution primitives to the PRISMA profile.

Chapter 6 sums up the main contributions of the work and suggest some future works.

CHAPTER 2. AN OVERVIEW OF THE RELATED WORKS

2.1 Introduction

This chapter presents an overview of the topics related to the research presented in this work. In section 2.2, the different distributed systems technologies and proposals are introduced. Section 2.3, gives an overview of the AOSD and how it has been used in other works for distribution, mobility and replication. Section 2.4, discusses software architectures and ADLs that have been proposed for distribution system. Finally, some graphical notations proposed for distributed and mobile systems are presented.

2.2 Distributed Systems

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

- Wolfang Emmerich

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.

In the following a brief explanation of the most recent proposals using middlewares are presented:

- **OMG's CORBA** [17] is an object based middleware which offers an interface definition language (IDL) and an object request broker (ORB). The IDL specifies the interfaces among the CORBA objects. The IDL is used to abstract over the fact that objects can be implemented in any suitable programming language. In addition, the IDL is responsible to ensure that data is correctly interchanged among the different programming languages. The ORB is responsible for transparently directing method invocations to the appropriate target object, and a set of *services* (e.g. naming, time, transactions, replication etc.)
- **Microsoft® .NET Remoting** [43][59] provides a framework that allows objects to interact with one another across application domains. The framework provides a number of services, including activation and lifetime support, as well as communication channels responsible for transporting messages to and from remote applications. The framework can be extended to achieve what is required.
- **Jini** network technology [37] is an open architecture that enables developers to create network-centric hardware or software services that are highly adaptive to change. The Jini architecture specifies a way for clients and services to discover each other and to collaborate across the network. When a service joins a Jini network, it advertises itself by publishing an object that implements the service API. A client finds services by looking for an object that supports the API. When the client gets the service's published object, it will download any code it needs in order to communicate with the service, thereby learning how to "talk" to the particular service implementation via the API.
- **Lime** [45][34] is a Java-based middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility, or both. Lime is specifically targeted at the complexities of ad-hoc mobile environments. The goal of Lime is to provide the simple Linda model of coordination in mobile environments via tuple spaces.

Other proposals try to make the development of the functionality totally independent from the configuration of the distribution. One of these proposals is GNATDIST[47]. This tool allows communication and distribution of applications' partitions developed with GNAT-Glade, which is a compiler of the Ada95 language. In this model two similar languages exist: one for the specification of the functionality and another one for the specification of the distribution.

The CLIP group of the Madrid University of Technology [18] proposed an extension to the Ciao Prolog language for specifying the distribution of its modules. In this system, all modules are specified as if they were local and the distribution is configured with a new specification in the same language (with the extension proposed). Next, the compiler knows which modules should be distributed and these are compiled in a particular manner. It is important to take into

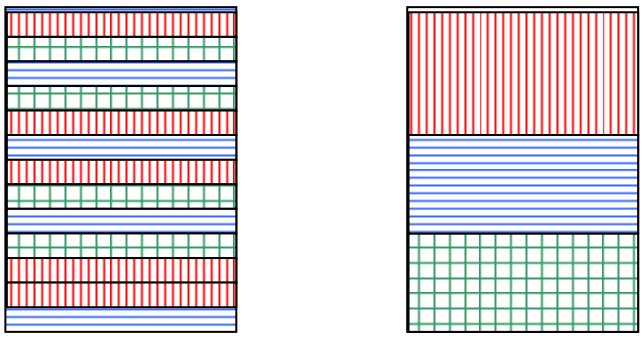
account that the programmer separates the functionality from the distribution and only uses a single language.

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

2.3 Aspect-Oriented Software Development (AOSD).

Object-Oriented Software Development (OOSD) [20] has aided software development by introducing real domain concepts such as encapsulation and inheritance providing a higher level of reusability than procedural programming. However, OOSD fails to modularize many concerns of today's complex systems. AOSD was first introduced in the work of Kiczales [32] as a new programming technique. The work shows how aspect-oriented programming contributes in many improvements such as the higher level of modularity, higher level of reusability and maintainability and evolution through the possibility of only manipulating a certain concern independently of the other concerns of an information system.

In [31] Kiczales presents the extension of Java for the aspect oriented programming (AOP) called AspectJ formally defining the concepts of this technique. Kiczales defines an *aspect*, the basic entity of AOP, as a well-modularized crosscutting concern. Figure 6, is an illustration typically used to show how aspect-oriented programming uses the separation of concerns. In (a) the concerns are tangled in the code while the crosscutting or common concerns of the code are separated in (b) showing the elegance of the code achieved using AOP separating the crosscutting code in an *aspect*. Another fundamental concept introduced by Kiczales is the *joinpoint*. The *joinpoints* are well-defined points in the execution of the programme which make possible the coordination between the aspect code and the non aspect code. The coordination of the aspect and non-aspect code is called weaving.



(a) (b)

Figure 6 The tangling code in the traditional programmes is shown in (a) and the separated code in aspect-oriented programming in (b).

An aspect in AspectJ is a module that implements crosscutting, contains pointcuts, advice and ordinary Java declarations. A pointcut is a set of joinpoints. Advice is a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut. AspectJ supports three main types of an advice: before, after and around.

The aspect oriented models are either static or dynamic models. 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 to one entity. This model has the drawback that at execution time the aspects cannot be independently manipulated failing to gain a high level of evolution, maintenance and reusability. An example to this model is AspectJ.

Many proposals are emerging following the dynamic model where the separation of the aspects and non-aspects entities is conserved at all moments even at execution time. The dynamic model allows the manipulation of the concrete aspects at execution time allowing a high level of evolution, reusability and maintenance. An example is the work in [57] which provides dynamic weaving based on the Java Virtual Machine Debugger Interface (JVMDI). In addition, the JAsCo language [67] integrates AOSD and CBSD at the implementation level by extending the JavaBeans and introducing connectors to perform dynamic aspect weaving. Other proposals that use dynamic weaving (dynamic model) are the works in [56][28].

The success encountered in AOP made the aspect community raise the aspect concepts to earlier phases of the life cycle as in the design [2] [64] , analysis and requirements[10][25][61] of software development.

In this research work, two types of identified aspects are going to be used: the distribution and replication aspect. Therefore in the following we present related works to the distribution and replication aspects.

2.3.1 Distribution Aspects

The distribution aspect in most proposals has been defined to encapsulate the properties necessary to enable a remote communication.

The first work done on separating the distribution concerns from the functionality using aspect oriented programming was in Lope's dissertation[35] . The work proposed the design and implementation of a programming language framework – D – for a representative class of distributed systems. In D, some important distribution issues were identified and made programmable externally to the implementation of the application's functionality. D provided for syntactic separation of distribution concerns, such as creation and coordination of threads, and communication between execution spaces.

The work of Soares in [65], proposes a pattern for the distribution concern using AspectJ named PaDA(see Figure 7). This pattern provides a remote communication between two components (client and server) of a system. Soares proposes three aspects: a client-side aspect to call remotely to a server component, a server-side aspect to enable the reception of remote calls and an exception handling aspect. The client-side aspect is weaved with the source component, the server-side aspect is weaved with the target component and the exception handling aspect is weaved with both components. The PaDA pattern achieves a high level of modularity that makes the system source code API-independent, a high level of maintenance in which the communication API can be changed without affecting on the system functionality and facilitates the testing of the functional requirements without of the distribution because the concerns are separated.

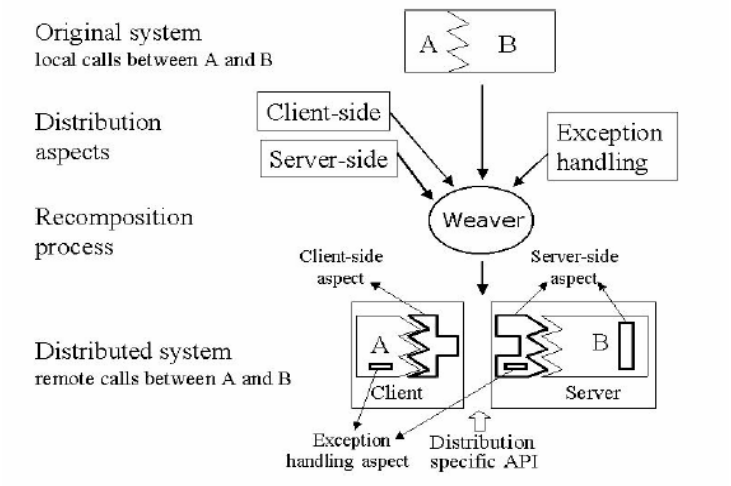


Figure 7 PaDA's Structure

Nevertheless in [62], the[66] experience of implementing a distributed web-based information system with AspectJ was discussed identifying drawbacks of AspectJ. The most essential drawback was that the definition of a pointcut identifes (by using methods signatures, class names, etc.) specific points of a given system, the aspects become specific for that system, or for systems adopting the same naming conventions, decreasing reuse possibilities. The works suggest that either aspect parameterization or code generation tools when developing with Aspects should be supported.

The thesis dissertation of Herrero, works on the aspect-oriented paradigm on the conceptual level to automatically generate code [28]. The distribution aspect in his proposal contains the reference table and the communication platform. The reference table maintains the references to all the invocated objects. The communication platform is responsible of the necessary mechanisms for receiving and sending massages from a communication platform to another.

2.3.2 Replication Aspects

The replication aspect, sometimes called fault tolerance aspect, specifies the necessity of the system to continue working even when one or more of the components fail. This aspect defines the properties necessary to support failures. Generally, the main mechanism to solve failures is the replication. The replication consists basically of the capacity of a component to duplicate.

In the proposal of Herrero[28], the replication aspect specifies either two techniques for replicating: the Active Replication and 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.

2.4 Software Architectures

Although software architectures have been used since decades in software development, there is no consensus on a concrete definition for it. Recently, many researchers are still discussing its concepts and definitions in events such as SFM 2003 [21] and SI-SE 2004 [16]. Many definitions appear in the Software Engineering Institute's Web site [15]. A typical definition which uses David Garlan is:

A software architecture for a system is the structure or structures of the system, which comprise elements, their externally-visible behavior, and the relationships among them.

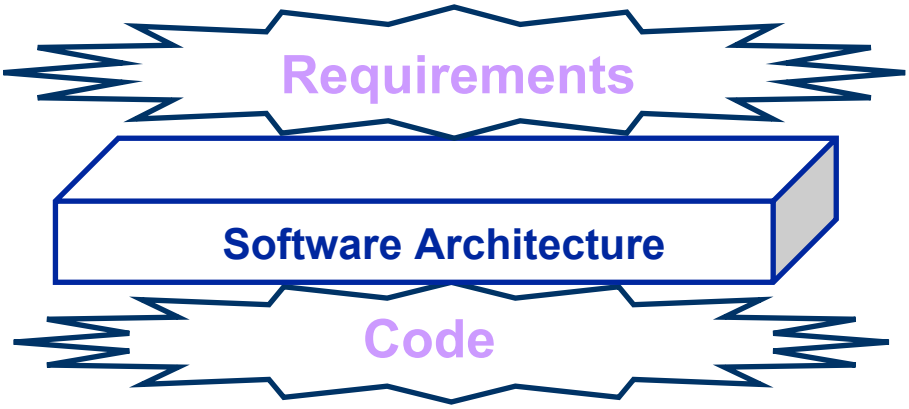


Figure 8 An illustrarion used by David Garlan to show the role of software architectues

Software architecture is a technique used in software development of large complex systems. Software architectures represent the description of both the system structure and system behaviour. The structural view describes how the system is made up of interconnected units called components. The behavioural view describes the interaction of the systems components to achieve the overall functionality of the system. Figure 8 An illustrarion used by David Garlan to show the role of software architectues shows that software architecture is an intermediate phase between requirements and code in software development. It is a phase which gives design analysis and guidelines to predict the final product.

2.4.1 Architecture Description Languages Supporting Distributed Systems

Architecture Description Languages (ADLs) evolved from Module Interconnection Languages (MILs) first defined in 1975 in the works of DeRemer and Kron [19]. Examples of MILs are configuration languages as DURRA [9]which basically separate the computation from the structure of the system. MILs had certain drawbacks in dealing with architectural issues and in posing restrictions to software developers such as restricting that each module should describe the other modules it interacts.

ADLs provide a way to define software architectures using a formal notation. Many ADLs have been proposed each for a particular domain. A great study on a comparison of a set of existing ADLs is the work of Medvidovic [42]. In this section, a discussion of some ADLs is going to be presented with a comparison with the goals of our work in achieving distribution communication, mobility and replication of architectural elements.

Wright [6] is a formal ADL which abstractly describes the architecture using components and connectors. Wright provides the description of architectural configurations and styles. It

describes the behaviour of the components using a CSP-like notation. However, a great pitfall of the language is that it is static and cannot describe dynamic distributes systems with mobility and replication.

Rapide [37] allows architectural modelling, simulation, analysis and code generation capabilities. However, this ADL does not have the necessary constructs to describe distributed, mobility and replication of software architectures.

Despite, software architectures arose to simplify the construction of dynamic distributed systems, few ADLs have been exploring the area of distributed systems. The first research that provided significance results in the area of distributed systems was carried out by Kramer and Magee at the Imperial College in London in 1995 in the ADL named Darwin [38]. Darwin uses π-calculus to define the semantics of distributed message-passing. Darwin builds architectures through defining composite components which consist of the binding of instantiated components which are given locations at instantiation time. Darwin has also been used in the CORBA environment to specify the overall architecture of component-based applications [39]. However, we cannot find new advances to Darwin in constructing software architectures with mobile and replicable components. Moreover, Darwin only supports constrained dynamic manipulation of the architecture, i.e. the changes must be planned.

The work in [49] introduces features that an ADL should consider in order to specify dynamic architectures such as composition, reusability and configuration. In this work a formal configuration language is presented describing a method for a reconfiguration model at run-time. On the other hand, the reconfiguration model is not formal. In addition, this work neglects a distribution model for specifying distributed message-passing among components and connectors.

In the works of Mascolo and Ciancarini [14][40], MobiS a specification language based on a tuple-space based model which specifies coordination by multiset rewriting is introduced. The works show that MobiS can also be used to specify architectures containing mobile components. However, these works do not specify the mobility concerns from the rest of the functionalities of the software architectures.

The ADL C2Sadel has adapted a style to support distribution and mobility. The style [41] provides software connectors that are able to move components. They exploit the modeling and implementation infrastructure for an architectural style that supports distributed and heterogeneous applications. However, this approach has the drawback that there is no separation between coordination and distribution. In this way, the components are the only architectural elements that are mobile while the connectors are static.

Recent works done in supporting distribution and mobility are the works in [36] which describe the semantics in externalising a distribution dimension. This distribution dimension is very similar to a connector but instead of containing the business logic, it controls the rules for mobility and locations achieving a separation between computation, coordination and

distribution. Our approach is quite similar, but we use the aspect oriented approach which encapsulates distribution issues of components in entities called aspects. We use the aspect-oriented approach because it supports the independence of the aspects from the functionalities thereby, achieving reusability, adaptability and evolution. In addition, it is a considerably mature and tested approach at the implementation level.

In [24], one of the most recent works, an ADL that addresses issues of mobility is proposed. The ADL is still not formally defined but discusses an own-defined graphical notation with no semantics. It is only a proposal and is still not mature enough, however it is another emphasis for the necessity for an ADL that supports the constructs and primitives to define dynamic distributed systems.

2.5 Graphical Notation for Distributed and Mobile Systems

Modelling techniques simplify the construction of applications. The idea behind modelling is to be able to automatically generate applications from visual diagrams that are easier to create and understand. They also facilitate for software engineers the analysis of their designs visually. Problems can be detected and solved visually without working with code. As distributed systems are complex and large applications, they essentially need a graphical notation to represent and verify that the applications meet their desired requirements.

The Unified Modelling Language (UML) [69] is the most accepted graphical notation used in industry. Therefore, most of the proposals that use a graphical notation for distributed and mobile systems extend the general purpose language (UML) to represent their metamodels. In this section we discuss some of the proposals that use different UML diagrams for the graphical notation of their distributed models.

The works by Kaveh and Emmerich [30] exploit the limited services that middlewares offer and discuss the problems that are encountered by using them, such as deadlocks and safety problems. They also define and formalize UML stereotypes that support the designs of the distributed applications. Initially, they chose the UML class to represent the system at a type level and the interaction diagrams to model the system at an instance level. The use of interaction diagrams limited them in obtaining full advantage of model checking techniques. Finally, they used the class diagrams, state diagrams [27] and object diagrams. The state diagrams are used to maintain the ability to model the dynamic behaviour and also hold all possible interleaving of object interactions. The UML object diagrams where chosen in order to verify different run-time configurations without any modifications to the state or class diagrams.

In the paper [11] an extension to UML class and activity diagrams to model mobile systems is presented. Locations can be nested and mobile too. They introduce stereotypes to model mobile objects, locations, and activities like moving or cloning. They introduce two notational variants of activity diagrams for modelling mobility. One variant is location centered

and focuses on the topology of locations. The other one focuses on the actor responsible for an activity.

In [46], an Architecture Description Language (ADL) devoted to the design of mobile agent systems to be implemented in a MASIF compliant platform. The ADL is defined as a UML profile called the MASIF-DESIGN profile. It enables the designer to describe the platform he/she uses, to locate the agents in the platform and to define the elements required from the platform for the achievement of the distribution transparencies. The stereotypes used are the UML subsystem, node, package and component.

2.6 Summary and Conclusions

In this chapter, an overview of the related works has been presented. After revising the state of art, we encounter a necessity for treating distributed systems at a conceptual level using AOSD and ADLs. Although software architectures have emerged for building distributed systems, the actual ADLs neglect constructors for describing them. Aspect-oriented programming has achieved great benefits to distributed systems by separating the distribution concerns from the rest of functionalities, however there is no consensus on an aspect definition at a conceptual level. In addition, distributed systems suffer from having an adequate graphical representation.

This research work, proposes a conceptual model for describing software architectures of distributed systems combining the AOSD and CBSD having an associated specification language and graphical notation.

CHAPTER 3. THE DISTRIBUTION MODEL IN PRISMA

3.1 Introduction

The PRISMA model introduced in section 1.2, does not enable the specification and modelling of software architectures of distributed systems. Thus, the PRISMA model should be extended to incorporate to its framework the appropriate characteristics and properties that are essential for distributed systems. As a result not only the framework has to be extended, but also the PRISMA model and its Architecture Description Language (ADL).

This incorporation should be done by preserving the concepts and objectives of the PRISMA architectural model combining the aspect-oriented and component-based software development. Thus, the construction of the software architectures of distributed systems should be constructed by defining PRISMA's interfaces, aspects, architectural elements and meta-level. In addition, PRISMA's objectives of achieving reusability, maintainability, evolution and dynamic reconfiguration should also be reached for the distributed software architectures definition.

In this chapter, the primitives to define distributed software architectures in PRISMA are presented. These primitives are incorporated to the PRISMA meta-model in order to specify architectural elements of distributed and mobile systems whose replication is allowed. In addition, these primitives have been incorporated to the PRISMA ADL at both the type definition and configuration level. Also, the proposal of a PRISMA infrastructure is introduced in order to provide distribution services to the model. Finally, an analysis is shown to demonstrate how the different decisions where taken to reach to the actual distributed model and what attributes where considered in order to achieve to an optimized model.

3.2 Incorporation of a Distribution Model to PRISMA

The incorporation of distribution to the PRISMA architectural model has been done following the combination of the aspect-oriented approach and the component based approach presented in [53] . A distribution aspect is defined in order to encapsulate the distribution properties, which is weaved with other aspects to form distributed architectural elements. In addition, as component based distributed systems may malfunction for a variety of reasons, including network failures and software errors a replication aspect is included to the set of aspects to encapsulate the replication properties which are also weaved with the other aspects to form a fault tolerance architectural element. At the same time, distribution-aware information is added also at an architectural level. At the architectural level the distribution model provides a location hierarchy where the connection relations: attachments among components and connectors and binding links among architectural elements and systems form a distributed Domain Name Server (DNS) of the architecture. As a result both the aspect-oriented view and the architectural view of the distribution dimension are complementary for our distribution model in PRISMA.

3.2.1 AOSD view of the Distribution Model of PRISMA

An aspect is a crosscutting concern i.e. a concern that is present in many components of the information system. Distribution is a clear crosscutting concern in a distributed system. Each distributed component has its properties that enable it to be distributed. Replication properties also crosscut different architectural elements of a software architecture. For this reason, another important aspect for large and complex distributed systems is replication which for instance can be used for solving fault tolerance problems. Therefore, in PRISMA a distribution and a replication aspect are defined in order to support the distribution and replication properties of an architectural element.

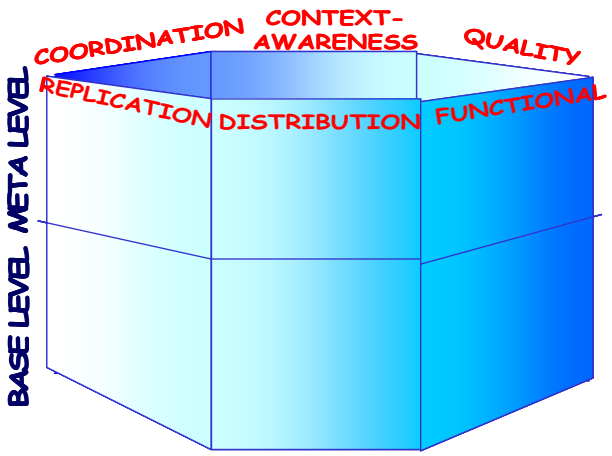


Figure 9 A PRISMA architectural element with a distribution and replication aspect.

3.2.1.1 The Distribution Aspect

A PRISMA distribution aspect should be added to the set of aspects types of the model in order to enable the definition of software architectures of distributed systems (see Figure 9). The distribution aspect specifies the features and strategies that manage the dynamic location of the instances of the architectural elements of a software architecture. Therefore, if the system is local a distribution aspect is not added to the set of aspects of the architectural element.

The distribution aspect deals with all the properties related to distribution and changes in location. To define the distribution aspect the general concepts of aspects fully explained in the work of Perez [53] are used. The distribution characteristics are defined through the aspects attributes and services. The change of state of the architectural element is specified through the valuations of the aspects. The restrictions associated to these changes are defined using constraints and preconditions. The consequences following certain conditions are specified by triggers.

The distribution aspect has to be added to the set of aspects of the meta-model. This is done by defining a distribution meta-class which inherits from the meta-class Aspect of the meta-model. The distribution aspect includes both the general concepts of the meta-class aspect (see Figure 10).

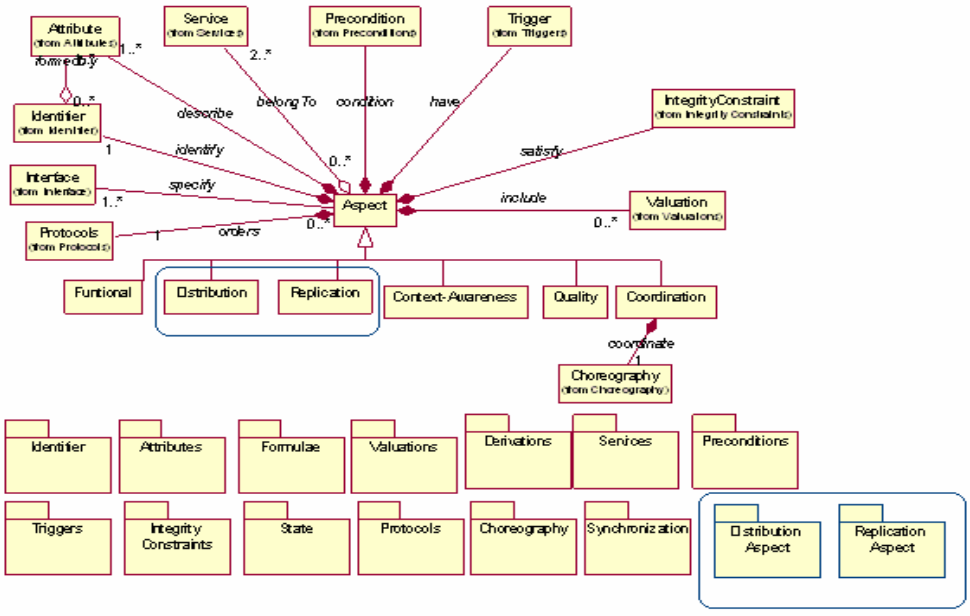


Figure 10 The Distribution and Replication aspect added to the Aspect Package of the PRISMA meta-model

Each architectural element with a distribution aspect must have a location. The location is a predefined PRISMA attribute of the distribution aspect that has as an abstract data type (domain) called *loc*. This data type models the sites in which architectural elements of an architecture domain are located. Therefore, the values of a location attribute should be one of the values of the *loc* data type designed in the architecture domain. In this way, PRISMA can remain independent of any notation of locations (nodes) and can be used for designing different kinds of location mechanisms. For example, in the geographical space, each location represents the coordinates of the earth using a GPS (Geographical Positioning System) where as in logical space the locations are IP addresses of a LAN network.

The distribution aspect can specify the ability of an architectural element to be mobile or the ability of an architectural element to affect on the locality of an external architectural element through the PRISMA service *move*. Therefore, if a distribution aspect does not have the PRISMA service *move* it means that it is not mobile nor it can affect on the locality of other architectural elements of the architecture. The PRISMA service *move* can be **in**, **out** or both. If the *move* is an **in**, the instance of the architectural element type is mobile. If the move is an **out** then the architectural element instance which it is formed by the aspect can move an external architectural element of the software architecture. In addition, the arguments of the service *move* can be **input** or **output**. When the argument is an **input** the argument is returning a result to the distribution aspect and when the argument is an **output** the argument is passing a parameter.

In Figure 11, the package of the distribution aspect is shown. The distribution aspect is directly related with the meta-class *Attribute*, the meta-class *Service* and the meta-class *Component* and *Connector.* A distribution aspect must have one to many *Attributes* (1..n-o..n). This is due to the fact that a distribution aspect must have at least an attribute named *location* with *loc* as a data type. This is specified as a constraint. Moreover, the distribution aspect must have two Services (o..n- 2..n). This is because the initializing *begin* service and the finalization *end* service must exist. In addition, a proper distribution service *move* can optionally exist depending on the requirements. In order to specify that the services *begin*, *end* and *move* are services of the distribution aspect a constrain is defined. The distribution aspect can optionally be part of a component or connector depending if we are working on a distributed system or not therefore the cardinality of the aggregation between the meta-class *distribution* and the meta-class *component* and the meta-class *connector* is (o..n-o..1). In addition, a component and connector can only have a distribution aspect.

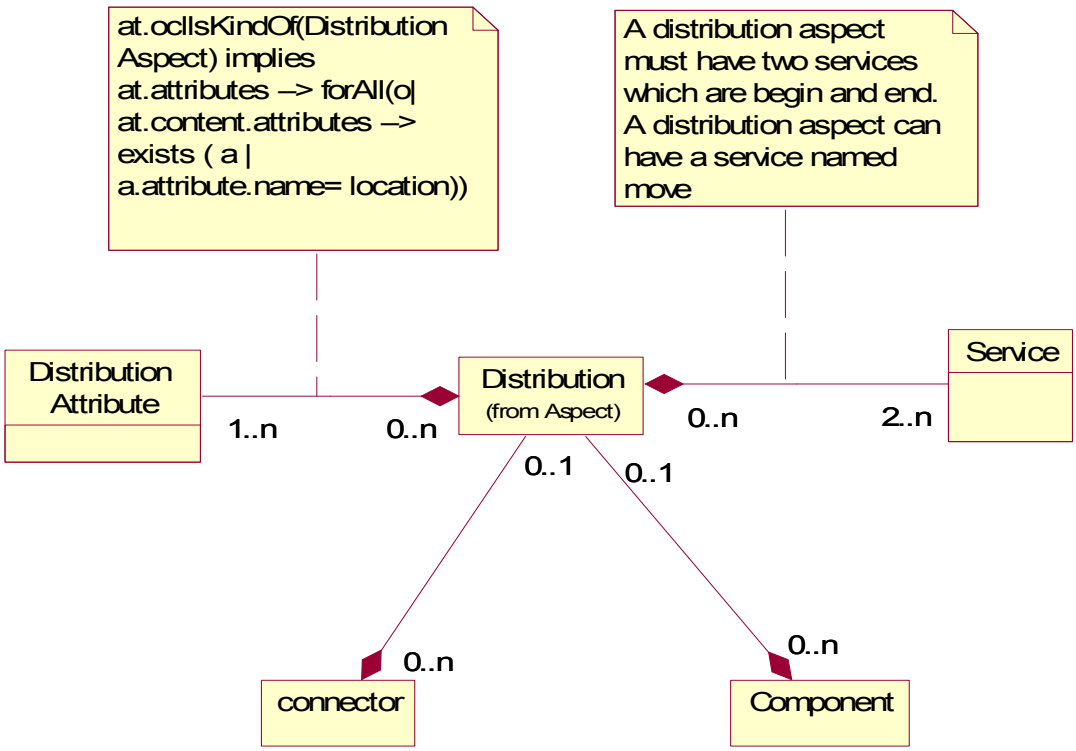


Figure 11 Distribution Aspect package of the PRISMA meta-model

The distribution aspect can also be formed by patterns that also use some predefined services, however this is fully explained in Chapter 4.

3.2.1.2 The Replication Aspect

Many models consider replication as the sequence of a copy and move operation without considering it as a primitive. However, in our proposal we consider to take replication as a primitive and separate the replication properties from those of distribution. In this way, we achieve a higher level of modularization and maintenance such that we can change the replication properties without affecting on those of distribution and the distribution requirements can be tested without those of replication. For this reason, the replication can be independent of the distribution and mobility. Thus, architectural elements can be replicated independently of being in a distributed system or not.

The replication aspect is also included in the PRISMA meta-model by inheriting from the meta-class aspect (see Figure 10). The replication aspect as the distribution aspect uses the general concepts of an aspect to specify all the properties related to the replication of an architectural element. The replication characteristics are defined through the aspects attributes and services. The change of state of the replication aspect is specified through the valuations.

The restrictions associated to these changes are defined using constraints and preconditions. The consequences fulfilling certain conditions are produced by triggers. As the distribution aspect, the replication aspect also includes the protocols as part of its definition in order to organize the necessary services to define the process which replicates an instance of an architectural element.

The replication aspect must have the PRISMA service *replicate*. The *replicate* service has a replica's location as an argument therefore its domain should be of type *loc*. If the service *replicate* is an ***in*** then the aspect specifies that its architectural element can be replicated. If the service *replicate* is an ***out*** then the aspect specifies that the architectural element of the aspect can cause the replication of an external architectural element of the architecture.

In Figure 12, the package of the replication aspect is shown. The replication aspect is related with three meta-classes: *Service*, *Component* and *Connector*. The aggregation between the meta-class *Service* and *Replication* has the cardinality (o..n-3..n). This is to indicate that a replication aspect must have at least three services: the *begin* service, the *end* service and the *replicate* service. To constraint the existence of these three services a constraint is defined. The aggregation relation between the meta-class Replication and the meta-classes *Component* and *Connector* has the cardinality (0..n-0..1). This is due to the fact that the replication aspect is an optional aspect for the architectural elements. Only a replication aspect can exist for each component or connector.

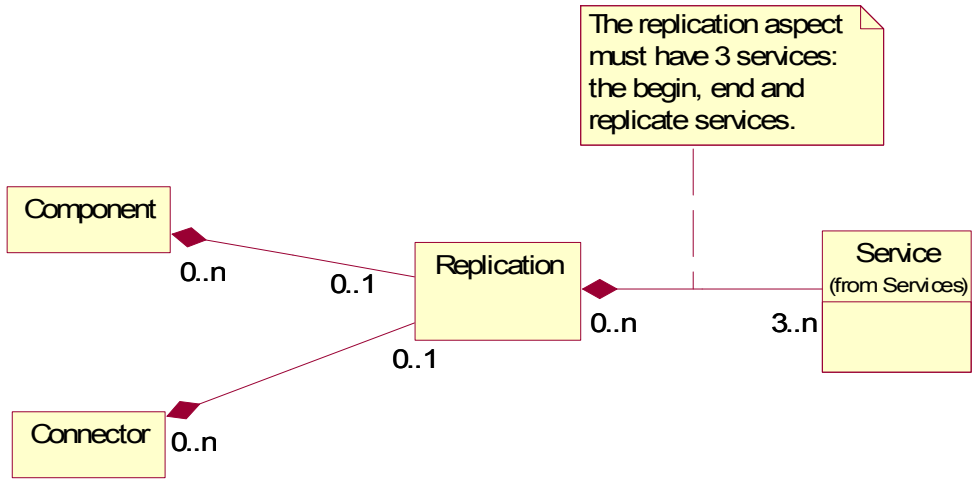


Figure 12 Replication Aspect package of the PRISMA meta-model

The replication aspect as the distribution aspect can be formed of patterns which also need some additional predefined services. This is explained in detail in Chapter 4.

3.2.2 The architectural View of Distribution

An architectural model in PRISMA is built by attaching the instances of architectural elements through the attachments which can be simple architectural elements (components and connectors) or complex architectural elements (systems). Also, the system's bindings connect the system with the elements (components or connectors) it encapsulates.

As the attachments and binding links define the connections among the instances of the architectural elements via the interfaces (ports and roles), they can also define the connections between the locations of the connected instances. As a consequence, the attachments and binding links can be seen as the software artefacts in which instances of architectural elements are registered once instantiated and given a location value in order to publish their ports or roles, that is, in order to offer and provide their services remotely to other instances of the architectural model. Therefore, the attachments and binding links have to be location aware of the architectural elements they connect.

In Figure 13, an example of a distributed architectural model is shown. There are 4 nodes (locations): component_1 and system_1 are deployed on Node 1, connector_1 and component_2 are deployed on Node 2, connector_Arch and component_3 are deployed on Node 3 and connector_2, component_4 and system_2 are deployed on Node 4. The connections between connectors' roles and components or systems ports are the attachments. While the connections between a system ports and the ports of components are the binding links.

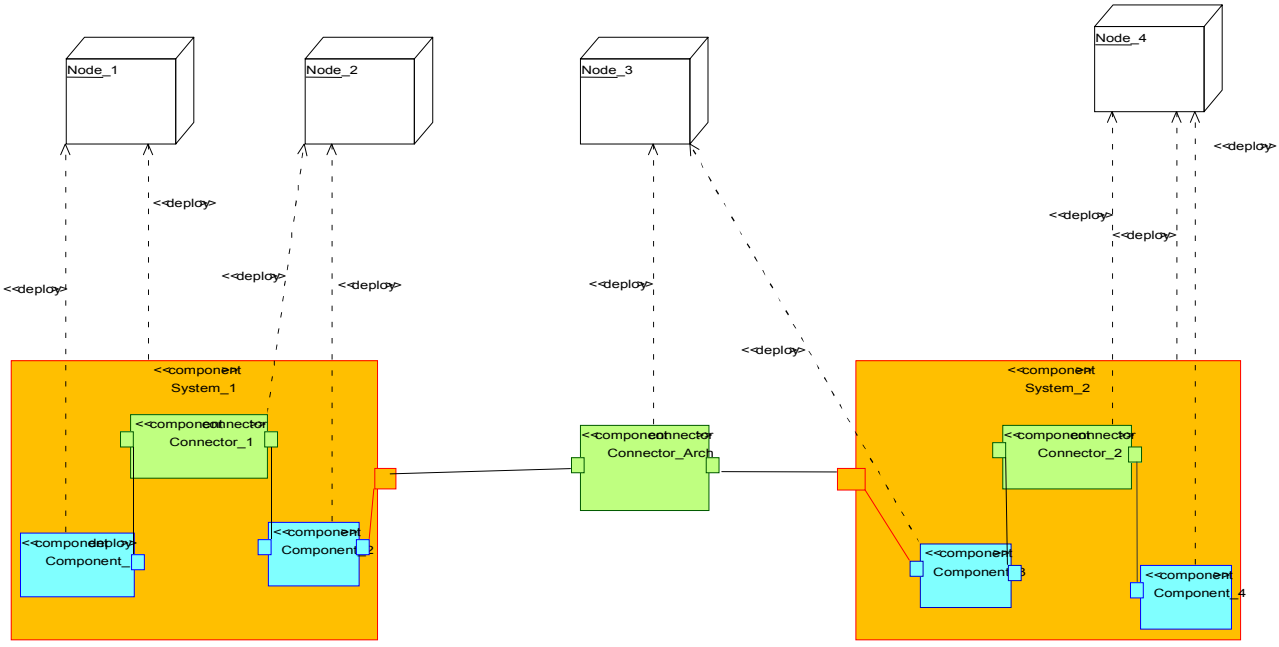


Figure 13 A distributed architectural model in PRISMA connected with attachments and binding links

As a result of the incorporation of the distribution model into PRISMA , the PRISMA metamodel has been extended. This extension can be seen in Figure 14 and 12, the location attribute of the distribution package has been included in the attachments package (see Figure 11). So in the meta-model the location of the two connected architectural elements is added to the attachments. Moreover, the locations of the system and the element it encapsulates are also appended to the meta-model in the bindings package (see Figure 15).

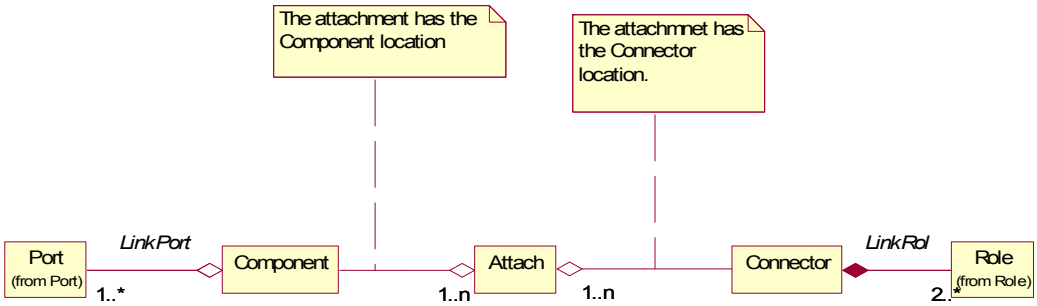


Figure 14 The attachments package of the PRISMA meta-model

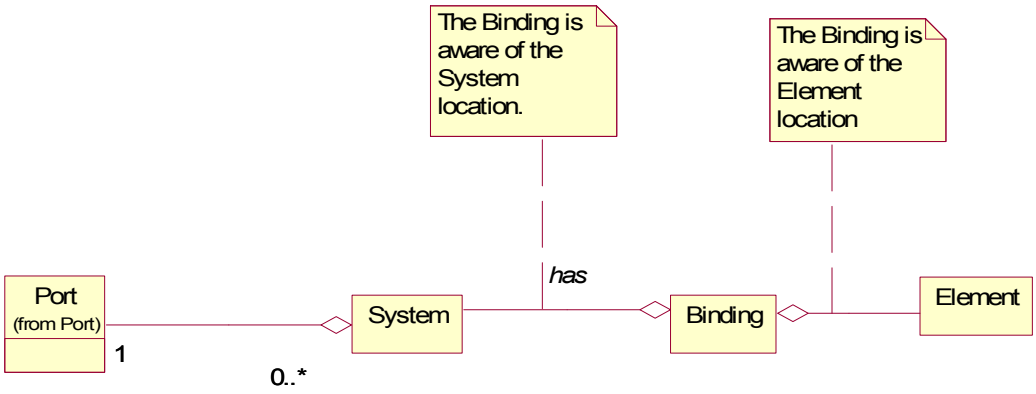


Figure 15 The bindings package the PRISMA meta-model

The attachments and the systems bindings form a distributed Domain Name Service (DNS) when performing distributed communication in the PRISMA model. This optimizes the performance and scalability of distributed systems specified in PRISMA due to the fact that problems resulted from having a centralized DNS are reduced by balancing the load between the distributed DNSs. At the same time, this provides a description mechanism to establish location hierarchies which is crucial for mobile and ubiquitous systems.

The location hierarchies in PRISMA are easily described due to the fact that architectural models are hierarchical. Such that bindings represent hierarchical connections (connect an architectural element from a lower level of aggregation with the system which is a composite component) and attachments are the non-hierarchical connections. As a consequence, the

locations of PRISMA software architectures can be represented in a tree-like structure. The root of the tree is the architectural model and its leafs represent the locations of the direct elements (systems, components or connectors) it contains. If the element is a system then its direct leafs are the locations of the architectural elements that it encapsulates (see Figure 16).

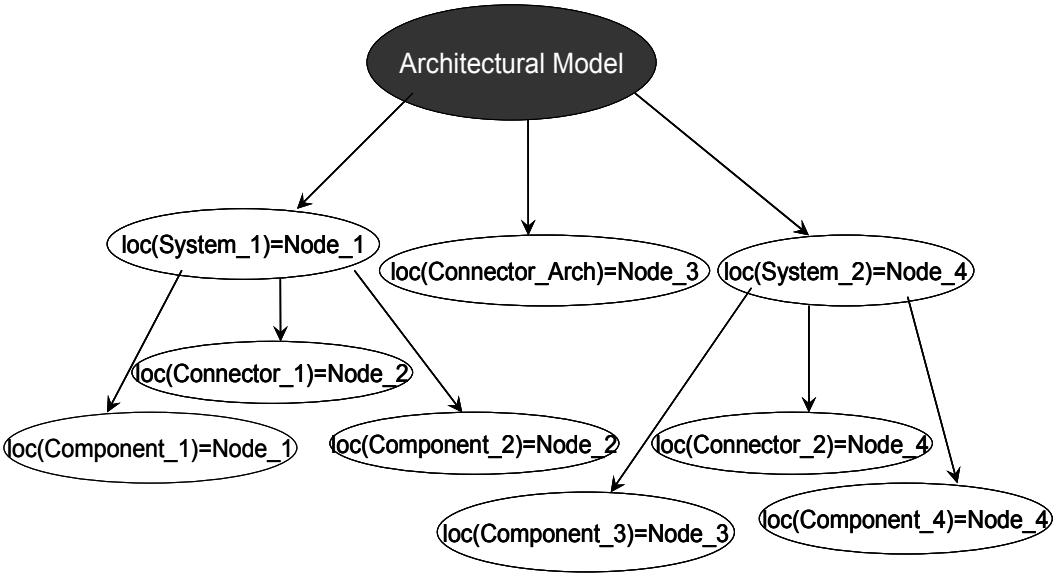


Figure 16 Representing the locations of the architectural model example of Figure 13 hierarchically.

3.3 The Architecture Description Language

The architecture description language (ADL) in PRISMA is based on OASIS 3.0 [33]. OASIS is a formal language for defining conceptual models of object-oriented information systems which permits the validation and automatic generation of applications. The general template of a PRISMA aspect is practically identical to a class in OASIS with some slight differences. On the other hand further extensions of the language were necessary for aspect weaving, architectural elements, attachments, binding links and others. They are presented in detail in [53].

The separation of the PRISMA ADL to define architectural elements at a type level using type definition language and at an instance level using the configuration language has many advantages. The type definition language is able to define architectural elements at a type level combining the AOSD and CBSD, storing the first order citizens in a types library in order to enable their reusability. Nevertheless, the configuration language imports the types and instantiates them defining the topology of the software architecture through the attachments and binding links.

In the following we are going to explain in detail the distribution and replication aspect templates with their proper properties using the type definition language. Next, the distribution

part of the instances and attachments of the architectural view is showed using the configuration language.

3.3.1 The Type Definition Language

The type definition language specifies the interfaces, aspects and architectural elements at a type level.

3.3.1.1 Interfaces

The interfaces are necessary in order to publish services of the different architectural elements. All the services a specific interface publishes have to belong to the same aspect type i.e. the services of a specific interface have to belong to either a distribution aspect or to a replication aspect but not both. The syntax of the type definition language for specifying an interface is the following:

```
Interface name
   Services
     service1;
      ...
     servicen;
End_Interface name;
```

For instance, a specification for publishing the service move of a distribution aspect is the following:

```
Interface IMblty
    Services
       move ( input New_location:loc);
End_Interface IMblty;
```

An interface for the replication service is the following:

```
Interface IReplicate
  Services
  replicate ( input New_location:loc);
End_Interface IReplicate;
```

3.3.1.2 Aspects

In this section the template of a PRISMA aspect is presented. This template is the following:

```
aspect_type Aspect name using interface1, ... interfacen;
      Attributes
      [<attribute_name> (<attribute_type>)]+

      Services
      begin <service_name> (<arg_service>)
      [as <service_name> (<arg_service)];
      end <service_name> (<arg_service>)
      [as <service_name> (<arg_service)];
      in/out <service_name> (<arg_service>)
      [as in/out <service_name> (<arg_service)];
              .......
      End_Services

      Valuations
             <observability_attribute_formulas>
             ......
      End_valuations

      Constraints
        static      <static_restrictions>
             ....
      End_Constraints

      Preconditions
        <formula_precondition_event>
        .......
      End_Preconditions

      triggers
        <trigger_event_formula>
                ......
      End_triggers

      Operations [transactions]
        <transaction_formula>
        ......
      End_Operations

      Protocols
```

```
        <protocol_formula>
    End_Protocols

End_ aspect_type Aspect name;
```

In the heading of the aspect, the type of properties an aspect defines is specified: distribution or replication. Moreover, the interfaces which the aspect gives them semantics are indicated. For instance, it is optional for the distribution aspect to use any interfaces depending whether the distribution aspect needs to publish some of its services or not. For example if the distribution aspect does not specify a mobility behaviour that has to do with the environment then the distribution aspect does not have to use any interfaces as the following:

```
Distribution Aspect Distributed;
```

On the other hand, when a distribution aspect allows the environment to affect on its behaviour it uses the interfaces. For example, when a distribution aspect specifies that the environment can control its mobility, that is to say that the environment can decide to move an architectural element, the distribution aspect should use an interface as follows:

```
Distribution Aspect Mobile using IMblty;
```

The case is similar in the replication aspect. If the replication of the architectural element is not affected by the environment and the replication of architectural elements of the environment are not influenced by the architectural element that specifies the aspect then interfaces are not used in the heading. Whereas the interface should be used in the heading if the contrary is the case. For example, the following is the heading of a replication aspect that its replication is only affected by the state of the architectural element.

```
Replication Aspect replicable;
```

The distribution and replication aspects attributes section can use predefined attributes as well as other attributes necessary to specify the properties of the aspect. In the case of the replication aspect there are no obligatory attributes that should be part of the attributes section, whereas it can use some predefined and other attributes depending on the information systems specification requirements. In the case, of the distribution aspect the predefined attribute location must be part of its attributes section. This attribute has the data type *loc* and uses the reserved words *NOT NULL* to indicate that a value has to be given to the attribute *location* at the time of instantiating the architectural element of the distribution aspect. In the following, an example of the attributes section of a distribution aspect with the predefined and

obligatory *location attribute*, some other optional predefined attributes and attributes that the analyst adds to the specification depending on the requirements of the system is shown:

```
Attributes
 location : loc NOT NULL;
 requestRate : nat(0);
 arrivalRate : nat(0);
 newLocation : loc;
```

The **services** section as the rest of aspects of PRISMA, contain the published services of the interfaces which the aspect uses and the non published ones that are proper and internal to the aspect. The distribution and replication aspects have four types of services: the initialization, finalization, modification and query services. The *begin* and *end* are reserved words for the initialization and finalization services respectively. The begin service of an aspect refers to the initialization of the aspect when an architectural element type is instantiated. The begin service gives the values to the necessary attributes at instantiation like the *NOT NULL* attributes. For example, the attribute location of the distribution aspect is given a value through the begin service because it is *NOT NULL*. The *end* service finishes the execution of the aspect.

The semantic definition of the aspects set of services is differentiated between the client and server behaviour. For instance, the *move (newLocation: loc)* of the distribution aspect can perform different actions depending on a client or a server behaviour. When a component is using the service as a client, the execution of the service will produce a service invocation of another component in which the argument *newLocation* is going to produce a change in the attribute location on the other component. While, whenever the service is executed in the component due to a server behaviour, the argument *newLocation* will change the location of the component. As a result, these behaviours are differentiated syntactically by preceding the service with the reserved words ***in*** and ***out*** to distinguish between the client or server behaviour of the service, respectively. For this reason, in order to indicate that a component can affect on the location of the environment the service ***out*** *move(newLocation: loc)* is used. While, when the component indicates that it can be moved either due to an external petition or due to a change of its state the ***in*** *move(newLocation: loc)* service is used. The same case is applied in the replication aspect with the *replicate(newLocation: loc)* service. When the aspect indicates that it can be replicated the service is syntactically represented by ***in*** *replicate(newLocation:loc)* and when the component of a replication aspect can cause a replication of another architectural element of the architecture the service ***out*** *replicate(newLocation:loc)* is used. When, neither an **in** nor an **out** proceed a service it is considered to be by default an **in**.

An argument of a service can be of type ***input*** or ***output***. If the argument is an ***input*** then the aspect is receiving a value from the environment. However, when the argument is an ***output*** the aspect is passing (sending) the value of a parameter to another architectural element. When an ***input/output*** argument is not specified, the argument is considered to be an input if the service is an **in**. While when the service is an out and the argument is not specified, the argument is considered to be an ***input*** by default. The service section of a distribution aspect can look as follows:

```
Services
 in move(NewLocation : loc);
 out move(NewLocation : loc);
 calculateArrivalRate( input ArrivalRate : nat)
```

The **valuations** specify how the execution of a service changes the values of the attributes and consequently changes the state of the aspect and the architectural element. As a result the valuations related with services with *out* arguments are optional to be specified depending on the requirements, while it is obligatory to specify the valuations of the services with *in* arguments. The valuations are formulas of dynamic logic [26] of type $\varphi \rightarrow [a]\phi$ and are interpreted as: "if in a determined state of the aspect, φ is satisfied and the action a happens, in the posterior immediate state ϕ is satisfied".

In the following, the valuations of the *move(**in** NewLocation: **loc**)* of the distribution aspect is specified in order to change the value of the location attribute to the newLocation when the service executed.

```
Valuations
  [move( input NewLocation : loc)] location = NewLocation;
End_Valuations
```

Constraints are formulas based on the state of the aspect and that must be satisfied each time a service is executed in the whole life of the instances of the architectural element types that incorporate the aspect. In the case a constraint is not satisfied, the next state of the aspect is the last one in which the constraints where satisfied. Constraints can be classified into static and dynamic, depending if they refer to only one state or relate to different status. To specify dynamic constraints temporal operators are used such as *always*, *sometimes*, *since* etc. Constraints are used for example in the distribution aspect to specify the restrictions of the locations a component can move. In the following distribution aspect the location of the component is constrained to always be inside this interval:

```
Constraints
 always { location>0 & location< x};
```

Preconditions establish the conditions that have to be satisfied so that a service can be executed. So it is not enough that a service is invoked in a server but also that certain conditions have to be satisfied. In dynamic logic, the preconditions have the formula $\neg\phi$[a] false, where ϕ is a well-formed formula (wff) which is interpreted as a condition of the occurrence of the indicated action. Its semantic is “if it is not achieved, the occurrence of the action does not change to the next state of the instance”. For example, an architectural element instance should not perform a replication if the number of services executed do not exceed a certain threshold. This is indicated by the following precondition:

```
Preconditions
 replicate(AnotherLocation)( if { serviceCount >= x});
```

A **trigger** permits a service defined in the aspect of an architectural element to be obligatory executed whenever the aspect that defines the trigger reaches to a state that satisfies the condition of the trigger. The triggers can cause an obligatory execution of a service of an external architectural element when the arguments of the service have an out argument. When the architectural element of the aspect in which the obligation is established is the client of such service, the execution of the action occurs as an internal service request.

In the context of dynamic logic, the trigger has the form $\phi[\neg a]$ false, where ϕ is a well-formed formula (wff) interpreted as an occurrence condition of the indicated action. The significance is “if ϕ is satisfied and the indicated action is not executed, then the instance type does not reach a valid state”. Therefore, it is essential that the action occurs for the instance to reach the next state. An example of a trigger in the distribution aspect is when the request rate of the architectural element exceeds a certain threshold the element obligatory has to move a new location. The trigger is as follows:

```
triggers
    move(NewLocation) when {requestRate>=requestRateLimit};
```

An **operation** is a non elemental and non atomic service. An operation is a process where the sequence of established actions should occur. An atomic operation is called a transaction which has to be specified with the reserved word transaction. An example of a transaction in the replication aspect is when the set of services have to be accomplished in

order to replicate an architectural element to the location of a client which most frequently invokes a service. Thus the location of the client should be found then the replication is performed. The transaction is as the following:

```
operations
replication(Service, CL, ClientL)transaction:
replication ≡
out findClientLocationMostInvoked(Service).REPLICATE1;
REPLICATE1 ≡
in responsefindClientLocationMostInvoked(CL).REPLICATE2;
REPLICATE2 ≡   replicate(ClientL);
```

A **protocol** defines a sequence of actions which's occurrence is allowed (can occur). For instance, a protocols section of a distribution aspect first has to be initialized at the instantiation of the architectural element then the architectural element can be move or it can be destroyed. This protocol is specified as follows:

```
Protocols
        DISTRIB0 ≡ begin.EXTMBILE;
        EXTMBILE ≡  move + end;
```

3.3.1.3 Architectural elements

Previously, we have introduced the full template of the distribution and replication aspect. In the following, how the aspects can be weaved to form an architectural element is going to be explained. As noticed, in the specifications of the aspects no dependencies between the aspects and the architectural elements where considered. This provides the reusability of an aspect in different architectural elements. So after defining an aspect and storing it in the type's repository, the architectural elements can reuse these stored aspects if they are compatible with its behaviour requirements. Then, the weaving which is defined in the architectural element's specification indicates the synchronizations and dependencies among the aspects adapting these synchronizations to the behaviour of the architectural element.

Next, the specification of a bank system account component is defined showing the full specification of the replication and distribution aspect. The distribution aspect specifies a distributed and mobile behaviour. The aspect also offers the ability to check if the any locations are compatible with the location borders of the aspect by using the service *checkLocation(input Location:loc, output:bool)*. Before any move, the new location is checked to be in the location interval and if it is the element is moved. This is specified in the valuation of the ***move****(Newlocation:loc)*. The constraint always checks that the location of the element is

inside the interval of locations. The protocols assign the order of the execution of the services. At the beginning of the instantiation of the architectural element the values of the location, maximum location value and minimum location values are assigned. Then a sequence of location checking occurs, or a sequence of moves of the element occurs or the distribution aspect ends in the case the instance of the architectural element is destroyed.

```
Distribution Aspect MobileAccount using IMobility

        Attributes
          location: loc NOT NULL;
          locMax: loc NOT NULL;
          locMin: loc NOT NULL;

        Services
          begin(Location:loc, LocMAx:loc, LocMin:loc)
            Valutions
                 [begin(Location,LocMax, LocMin)]
                   {location:=Location & locMax:=LocMax &
          locMin:=LocMin;}

          checkLocation( input Location:loc, output
          checkC:bool)
             Valutions
               [checkLocation(input Location:loc,output
               checkC: bool)]
                  {Location>locMin & Location<locMax}

               [checkLocation(Location,checkC)]checkC:=true;
                  {Location<locMin & Location>locMax}

               [checkLocation(Location,checkC)]checkC:=false;

          move(Newlocation:loc)
             Valuations
                 {NewLocation>locMin & NewLocation<locMax}
                   [move(input NewLocation)] location =
                     NewLocation;

      Constraints
         always{ location >locMin & location< locMax};
```

```
        Protocols
            mobility :
            mobility  ≡ begin.MOBILITY2;
            MOBILITY2≡ end+checkLocation + move.MOBILITY2;

    End_Distribution Aspect MobileAccount;
```

The replication aspect specifies the behaviour of replicating. In the following replication aspect the service `replicate(NewLocation)` is executed by an architectural element which is not the one of the replication aspect. This is indicated by using the interface IReplicate in the heading of the aspect. The `replicate(NewLocation)` does not have a valuation associated to it because it is a service of the meta-level which does not change the state of the architectural element when executed.

```
Replication Aspect ReplicateAccount using IReplicate

      Services
       in replicate(NewLocation:loc);

      Protocols
           replication:
           replication  ≡  begin.REPLICATION1;
           REPLICATION1 ≡ end + replicate.REPLICATION1;

  End_Replication Aspect ReplicateAccount;
```

The template of an architectural element is divided into four fundamental parts: the heading, the ports or roles, the aspects and the weavings. The heading starts with either the reserved word *Component_type* or *Connector_type* depending if the architectural element is a component or a connector respectively. The ports or the roles are specified by giving for each port or role an interface type. Then the aspects which form the architectural element are imported. Finally, the weaving which synchronizes the aspects are specified indicating if a synchronization is an **after**, **before** or **around**.

```
      Component_type/Connector_type  name
            Port/Roles
                  P_i : interface_i;
                  … …
            End_Port/End_Roles;
```

```
[Aspect_type Aspect Import name]
      Weaving

                  <formula_weaving>;

      End_Weaving;

End_Component_type/Connector_type  name;
```

The account component is defined by defining the ports of the component and giving each one of them a type of interface, by importing the replication and distribution aspect previously specified and a functional aspect and the weavings of the aspects. The first weaving of the account component specifies that a component moves when a customer changes its home address. The second weaving specifies that the destination location is checked using the distribution aspect before a replication occurs. So the replication execution depends if the destination location is valid or not.

```
Component_type Account
  Port
    replicationPort: IReplicatei;
    distributionPort: IMblty;
    functionalPort: IAccount ;
  End_Port;

  Import Functional Aspect FAccount;
  Import Distribution Aspect MobileAccount;
  Import Replication Aspect ReplicateAccount;

  Weaving
     move(NewLocation) after changeAddress(Address);
     checkLocation(Location,Checkc)
           beforeif(Checkc=true)
                 replicate(Newlocation);
   End_Weaving

End_Component_type Account;
```

3.3.2 The Configuration Language

Distributed systems using the PRISMA ADL to describe their software architectures, obtain many advantages due to the separation of its language into the description of the Type Level and Configuration Level. The type definition enables us to describe some distributed and replication behaviours independently of the contexts they will be applied gaining a high reusability of these behaviours in different contexts.

On the other hand, distributed systems highly depend on the configuration of the architectures and on the execution of the distributed instances due to their dynamic environments. The dynamic change of the configuration of the architecture at run time, reacting to the changes in the environment, is called reconfiguration. Distributed systems need to reconfigure due to failures such as failures in the networks, software errors and node failures. The configuration language defines the architectural model by attaching instances of architectural elements. It permits to specify an initial state of the software architecture and then execute services of the meta-level dynamically to enable reconfiguration.

At the configuration level, the architectural element instances of the software architecture are assigned a physical location and registered in the architectural model to offer their services remotely. The architectural model is aware of the locations of its connected instances through the attachments.

The template of the definition of an architectural model is as follows:

```
Architectural Model <name of model>

  Import Types
   Components
    <name of type>, … … ;
   Connectors
    <name of type>, … … ;
   Systems
    <name of type>, … … ;
 End_Import Types;


  Instances
  <name_instance>: <name_type>;
                  ……
End_Instances;


Attachments
 Associated_connector <connectorName>;
                        <formula_connection>;
                        ……
 End_associated_connector <connectorName>;
```

```
End_Attachments;

End Architectural Model <model name>
```

In Figure 17, an architectural model of a distributed account and ATM of a bank system is represented. Account1 and ATM1 are instances connected through the instance Connector1. Account1 and Connector1 are residing on Node1 while ATM1 resides on a different node Node2. In the following this example is going to be specified using the Configuration Language.

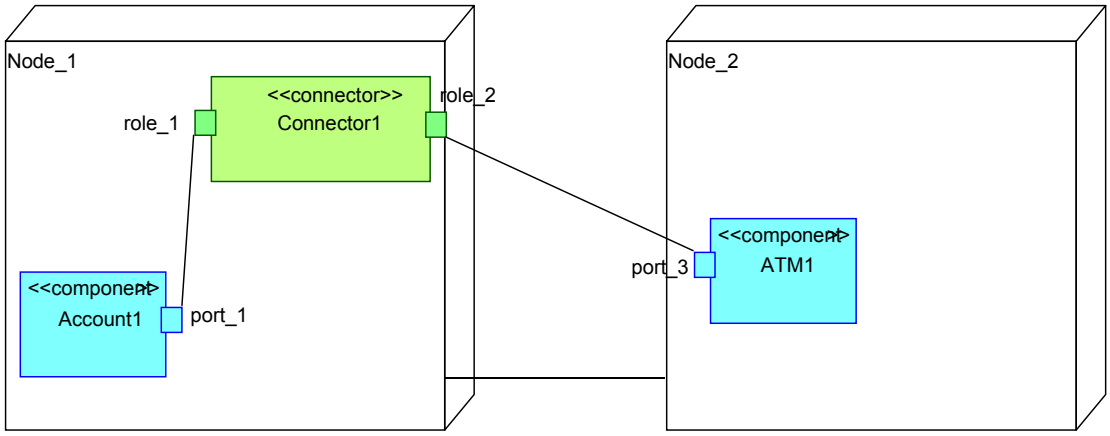


Figure 17 A distributed architectural model of a simple bank system of an account and ATM

The specification of an architectural model is marked with a heading and an ending. The heading gives a name to the architectural model. For instance, marking the architectural model with the name distriuted_bank is as follows:

```
Architectural Model  <distributed_bank>
   ......
End_Architectural Model <distributed_bank>;
```

The architectural elements types that participate in the architectural model are imported from the PRISMA library. These types where previously defined with the type definition language and stored for their usability. For instance, the Account component type is the one

defined in section 3.3.1 with the distribution and replication aspects. The importation of the architectural model's types in Figure 17 which are Account, Connector and ATM is as the following:

```
Import Types
   Components
        Account, ATM;
   Connectors
       Connector;
End_Import Types;
```

Once the necessary types are imported, the types are instantiated. The instantiation section is delimited with the reserved word Instances. At instantiation the necessary values have to be given to the aspect's attributes which are NOT NULL values through the begin service. The following shows the Instances section for instantiating the Account, Connector and ATM to Account1, connector1 and ATM1. The instances give values to the attributes location, locMax and locMin previously defined in the distribution aspect. Therefore, determining on Figure 17 the instance Account1 and Connector1 are located in Node1 and ATM1 is in Node2. Then, the three instances have the LocMin equal to Node1 and LocMax to Node2 indicating that the valid locations should be either Node1 or Node2.

The creation of the instances is made through the service **NewInstanceOf,** where the name of the instance, the type of the instance and the set of required attributes for creating the instance is passed. The required attributes are those who have the **NOT NULL** property in their aspect type definition. The syntax of the service is the following:

```
NewInstanceOf(Name: String, Type: PRISMA_Architectural_Element,
                        ListOfRequiredAttributes: List);
```

The list of required attributes vary depending on the type that the instance belongs to, since each type imports a set of different aspects and each aspect has a quantity and a type of different required attributes.

```
Instances
  NewInstanceOf (Account1: String, Account: Component, Location:
  Node1, LocMin: Node1, LocMAx: Node2);
  NewInstanceOf (Connector1: String, Connector: Connector
  Location: Node1, LocMin: Node1, LocMAx: Node2);
```

```
NewInstanceOf (ATM1: String, ATM: Component, Location: Node1,
LocMin: Node1, LocMAx: Node2);
End_Instances;
```

After defining the instances of the architectural model, the connections between them are established building the topology of the architecture. These connections are indicated by the reserved word attachments. The attachment formula specifies which port of a component instance is attached with which role of a connector instance and the location of the component and connector instance.

```
Attachments
 Associated_connector Connector1;
  Account1.(port1, Node1) ←→ Connector1.(role1, Node1);
  ATM1.(port2, Node2) ←→ Connector1.(role2, Node1);
 End_associated_connector CnctSubasta;

End_Attachments;
```

The specification of a system is quite similar with the sections of an architectural model, with the addition of a bindings link section. Instead of having a heading and ending indicated with the reserved word **Architectural Model**, the reserved word is replaced with the reserved word **System_type**. Moreover, the system has an additional section for the binding links which is marked with the reserved word Bindings.

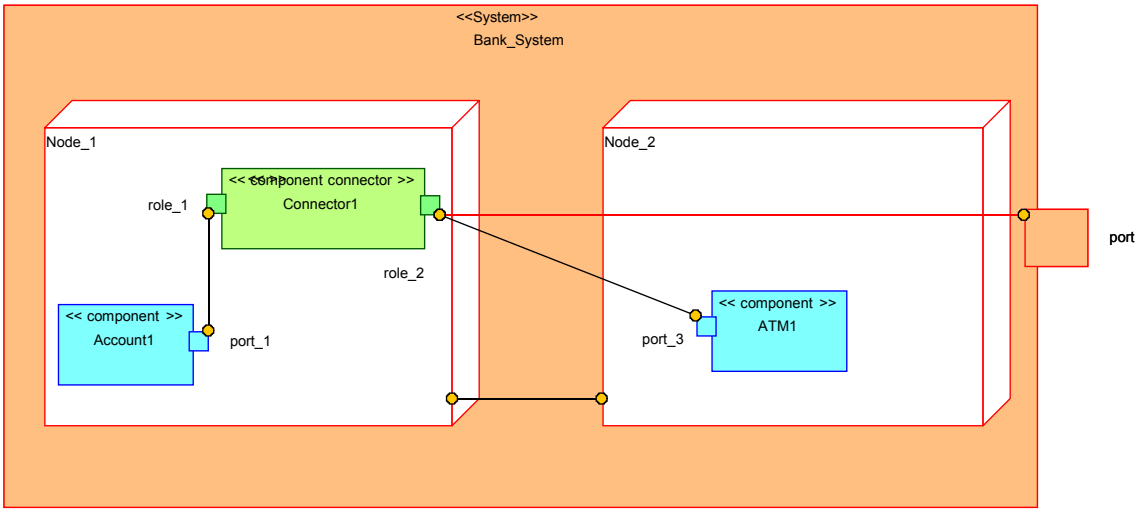


Figure 18 A system architectural element of a distributed bank system

In Figure 18, the same bank system example represented in an architectural model in Figure 17, is represented as a system architectural element that has its own port. The connection between role_2 of Connector1 and port of the Bank_System is a binding link. The binding links also have to be location aware of the connected architectural element instances. The following specifies the system bank system:

```
System_type Bank_System

   Import Types
    Components
       Account, ATM;
    Connectors
      Connector_Arch;
   End_Import Types;


   Instances
      Account1 (Location: Node1, LocMin: Node1, LocMAx: Node2):
       Account;
      Connector1 (Location: Node1, LocMin: Node1, LocMAx: Node2):
       Connector;
      ATM1 (Location: Node1, LocMin: Node1, LocMAx: Node2): ATM;
  End_Instances;


  Attachments
   Associated_connector Connector1;
    Account1.(port1, Node1) <<>> Connector1.(role1, Node1);
    ATM1.(port2, Node2) <<>> Connector1.(role2, Node1);
   End_associated_connector CnctSubasta;
  End_Attachments;


  Bindings
  Connector1.(role2,Node1) --- port;
  End_Bindings;


End_System_type Bank_System;
```

3.4 Mobility, Replication and Reconfiguration

The reconfiguration of a software architecture is the change in the structure of the architecture configuration. Mobility and Replication cause the reconfiguration of the software architecture.

The semantics of a ***move(loc)*** service in PRISMA is the following:

- Creates an instance of the architectural element in the new location conserving its state.
- Modifies all the attachments connected to the architectural element instance, updating them to the new location. For example, lets consider the following attachment between c1 and con: c1(port1,node1)←→con(role1, node2). If c1 moves to node3, then the attachment becomes: c1(port1,node3)←→con(role1,node2). If the architectural element is a system then its binding also have to be updated.
- The instance in the previous location is deleted.

The semantics of a **replicate*(loc)*** service in PRISMA is the following:

- Creates an instance of the architectural element (a replica) in the new location conserving its state.
- Creates exactly the same attachments and bindings (in case of a system) with the architectural element instances connected with the original replica.

3.5 PRISMA's Infrastructure for Distribution

PRISMA provides guidelines for building large, complex and distributed systems. For these guidelines to be useful they must to be accompanied by support for their implementation. Thus an infrastructure must be implemented in order to support the different functionalities of PRISMA in the different technologies. Some functionalities can be either directly used through the offered services of the platforms or by implementing additional PRISMA functionality using their platforms.

The PRISMA meta-level contains the services that provide the creation of the base level, evolution and dynamic reconfiguration of the software architecture at execution time. These meta-level services and other PRISMA services that can be used at execution time are being implemented in the PRISMA infrastructure. This provides the PRISMA functionalities in different programming languages, currently using C# (see Figure 19).

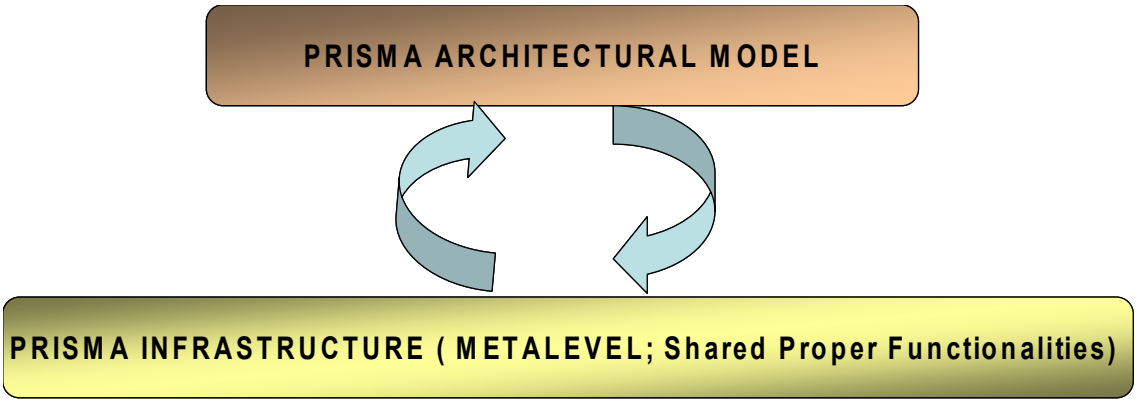


Figure 19 PRISMA Infrastructure

The services of this infrastructure are of two types. The first type of services are those services that a cooperation is needed between the architectural elements of the software architecture and the infrastructure, that is, the architectural elements send some requests to the infrastructure and the infrastructure sends them results. While, the second kind of services are the ones that can be claimed as alarms in which the infrastructure sends orders to the architectural elements without their own petition.

This infrastructure can be seen as an abstract layer between the PRISMA architectural model specification and the different platforms of technology. In other words, it is an abstract middleware that hides some complex functionality from the final application.

3.6 A Distribution Analysis For PRISMA

This section demonstrates two approximations that were taken into account in order to reach to the final distribution architectural model presented in the previous sections. Each possible approximation has a different view of distribution where each one has its own benefits and drawbacks. The concluding distribution model is the combination of both views of distribution achieving to a distribution model with the benefits of both approximations.

To explain the approximations, a simple bank system with different branches is going to be used to show the benefits and drawbacks of each approximation. The example is the following (see Figure 20):

An administrator of a bank system is a mobile user. She/he determines where to move her/himself i.e. she/he controls from where to access to the bank system. Whereas, the administrator can also move a bank account to her/his location. Therefore, the mobility of the account can be controlled by the environment. As well, when the account detects that its principal bank office has changed, the account moves itself to the location of this bank office (i.e. the component moves due to a change in its state).

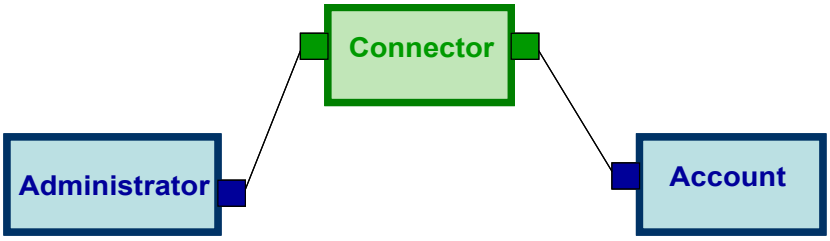


Figure 20 Distributed Administrator and Account connected in a Bank System architectural model

The two approximations that were taken into account are:

- Architectural elements are unaware of their location and thus a different entity in the architectural model should be responsible for distribution matters

- Architectural elements are location aware and have can control decisions on their distribution.

In the PRISMA model the entities that can be taken into consideration for being responsible of the distribution characteristics are the proper architectural elements and the attachments. The architectural elements can have distribution properties in order to be aware of their proper locations and to be in control of the location changes. While the attachment can be the entity responsible of the distribution characteristics of the architectural elements it attaches. The attachment is chosen to have this responsibility due to the fact that the attachment is the only entity in the PRISMA model which does not loose its reusability and is aware of both components and connectors.

3.6.1 Case1: Components and Connectors are Unaware of their Distribution.

This approach proposes that the architectural elements are unaware of their location and are not in control of their mobility. The entities responsible of the distribution properties are the attachments (see Table 1). The attachments should be aware of the locations of the architectural elements they connect, have the mobility services and restrict the locations of the architectural elements they connect.

Table 1 Instances of components and connectors unaware of their location and Attachments are the entities that are location aware of the instances they attach

Location / ArchElement	Component_Instance	Connector_Instance
Component	----	----
Connector	----	----
Attachment	X	X

For this approximation, each attachment is specified by identifying which port of an instance of a component is attached with which role of an instance of a connector. Then for each attachment, the distribution properties of its participants (component and connector instances) are specified. For example, for the attachment between the administrator and connector, the distribution properties of the administrator and the distribution properties of the connector are specified. The specification of the attachments for the bank system example would look as the following:

```
Attachments

Administrator.Port<>Connector.Role2;
Administrator.location=X;
Administrator.Constraints
  always{location< x}
Administrator.Services
  move(newLoc:loc);
    Valuations
     location:=newLoc;
Connector.location=Y;
Connector.Constraints
  always{location>0 & location< x}
Connector.Services
  move(newLoc:loc)
    Valuations
     location:=newLoc;

Connector.Role1<>Account.Port
Connector.location=Y;
Connector.Constraints
   always{location>0 & location< x}
 Connector.Services
   move(newLoc:LOC);
     Valuations
       location:=newLoc;
```

```
Account.location=Z;
......
```

After we have seen how the specification of an attachments example would look like in this approximation, the following remarks could be concluded concerning reusability, mobility, expressiveness and distributed communication:

- **Reusability**: The distribution properties as noticed are tangled in the specification of the attachments. A mechanism that provides reusability should be applied. PRISMA uses the aspect-oriented approach in which we could use distribution aspects to encapsulate the properties, however, an aspect should be weaved with a functionality which attachments neglect. This is why in the previous specification, for each of the attachments of the connector the same distribution properties are repeated.
- **Mobility**: The mobility of each architectural element has to go through the attachments of that architectural element. For example, when the Administrator decides to move an account to its location the move service of the Account should be invoked in all its associated attachments. In order to do so the Account cannot be attached to other components by a different connector of the Account. Another limitation is that there cannot be any constraint that the environment of a component cannot affect on its mobility. This is due to the fact that for the attachments all architectural elements are the environment. Therefore, they cannot differentiate between the petitions of mobility that come from the environment and the petition for mobility that occur due to the change of state of an architectural element.
- **Expressiveness**: The expressiveness of mobility is very complicated. This is due to the fact that all requested mobility services will pass through the attachments. In this case, there should be a mechanism to differentiate between the distribution services requested to the attachments and the services requested to the architectural elements that pass through the attachments.
- **Distributed Communication**: The communication is very appropriate. This is because the attachments are aware of the locations of each of the components and connectors they attach. In this way, when the attachments receive the service of an interface (port or role), they know to whom to send it and to where. This approximation could be very similar to a name service in which the names of the interfaces and there references are stored.

Going through the previous points we can conclude that this approach has a good contribution in distribution communication. However, the reusability which is one of the goals that PRISMA achieves is lost for the distribution properties. In addition, the mobility of the architectural element and its expressiveness is somehow complicated. Therefore, we had to solve them in some manner.

3.6.2 Case 2: Components and Connectors are Aware of their Distribution by adding a Distribution Aspect.

In this section, we are going to analyze the situation where the architectural elements are aware of their location and have the authority to control it (see Table 2). In PRISMA the properties of architectural elements are encapsulated in aspects which are then weaved with each other. Therefore, in this approach the distribution properties of each architectural element are described in a distribution aspect that is added to the set of other aspect that composes it.

Table 2 Instances of components and connectors are the only entities of the architecture aware of their locations through their distribution aspect.

Location / ArchElement	Component_Instance	Connector_Instance
Component	X	----
Connector	----	X
Attachment	----	----

This approach is used in the definition of the distribution aspects in [4] and contains the full example specified in this approach. The distribution properties are specified including a location attribute at the type definition level. Then the distribution aspect is weaved with the other aspects of the architectural element in the Component definition by specifying the synchronizations of the services between the aspects. Then in the configuration level when the architectural elements are instantiated the location of each instance is given a value. A simple specification of the account architectural element aware of its location by weaving a distribution aspect to its set of aspects is the following:

```
Distribution Aspect Mbile using Mblty

 Attributes
```

```
        location: LOC NOT NULL;
        X: LOC NOT NULL;

    Constraints
        always{ location>0 & location< x} ;

    Services
        move(out Location)

  End_Distribution Aspect Mbile;

 Component_type Account

    Ports
       Moves_Acc: Mblty;
    End_Ports;

    Functional Aspect
       Import BankFunctionality: Functional Aspect;
      Weaving
         Import  Distribution  Aspect : strangerAffectsMyMobiility
         move(LOC) after Change_PBankOffice;
      End_Weaving;

 End_Component Account;
```

Analyzing this approach the following can be noticed:

- **Reusability**: The distribution properties are encapsulated in a distribution aspect which can be specified independently of the architectural elements they will be added to. Thus, different architectural elements with same distribution behaviour can use the same distribution aspect.
- **Mobility**: The mobility of each architectural element can be controlled by the architectural element. This allows to restrict e.g. that the Administrator is in control of its own locations and that the environment cannot affect its locality. In addition, in this way an Account can specify that both the environment (the Administrator can move it) and its mobility can depend on its state at the same time.
- **Expressiveness**: The expressiveness of mobility is expressed as an additional service of the architectural element. The architectural element can specify that it offers its service to the environment by offering it in a port.

- **Distributed Communication**: As the architectural elements are unaware of the other architectural elements of the architecture, the locations of the destinations of the messages are unknown. This lacks the availability to model a distributed communication.

3.6.3 The Actual Distribution Model in PRISMA

Each approximation previously presented for distribution in PRISMA has its benefits and drawbacks. Table 3 resumes the different properties of each approximation. The second distribution approximation of adding a distribution aspect to the architectural elements is quite a tidy approach because it follows the PRISMA characteristics and uses the aspect-oriented approach to define a distribution model. On the other side, it is weak in specifying a distributed communication in the PRISMA model. While, the first approximation of externalizing the distribution characteristics in the attachments gives additional functionality to PRISMA, in specifying the distribution communication and representing a Domain Name Server which exists nowadays in the technologies, such as CORBA [17].

Table 3 The contribution of each distribution approximation on the PRISMA model.

Properties / Approach	Reusability	Mobility	Expressiveness	Distributed Communication
1	X	X	X	√
2	√	√	√	X

Table 4 Components and Connectors are aware and control their locations. In addition the Attachments are location aware of the instances they attach.

Location / ArchElement	Component_Instance	Connector_Instance
Component	X	----
Connector	----	X
Attachment	X	X

As a conclusion, a combination of the two previous approaches can be made in order to gain the advantages of both approximations as it can be seen in Table 4. We can preserve the idea of adding a distribution aspect with the distribution properties that have control over the mobility and as well add location information of the instances of the architectural elements to the attachments. In this way we can have the benefits of both approaches. We achieve to an adequate distribution model represented at the type definition language with the distribution aspects as and the following specification at the configuration level:

```
Architectural Model

 Import Types
  Components
   Administrator,Account;
  Connectors
   Connector;
 End Import Types

 Instances
  Administrator: Admin(Location=X);
  Connector: Con( Location=Y);
  Account: Acc( Location=Z);
 End_Instances

 Attachments
  Admin(location, Port)<>Con(location, Role1);
  Con(location, Role2)<>Acc(location, Port);

End_Architectural Model
```

The previous specification shows that when the architectural elements are instantiated, a value to their locations is given. Then the location of each instance is registered in the attachments. When an instance changes its location, a notification has to be made to its attachment in order to change the location values. This service is provided by the PRISMA meta-level.

3.7 Summary and Conclusions

This chapter introduces the distribution model proposed for PRISMA. The distribution concepts and primitives have been included in the PRISMA metamodel. The PRISMA

language supports the primitives for defining distributed software architectures with mobile and replicable architectural elements.

Different cases for a distribution model for PRISMA have been considered. Reusability, mobility, PRISMA language expressiveness and distributed communication have been taken into account for reaching to a suitable model.

In the future, Ambient calculus [12] is going to be used to formalize the PRISMA distribution model. Ambient calculus is a calculus that describes the movement of processes and devices.

CHAPTER 4. CONCEPTUAL DISTRIBUTION PATTERNS FOR PRISMA

4.1 Introduction

Patterns are used in software development to identify common problems, abstracting the problem and solution. In this way, a problem-solution catalogue can be obtained where each problem has a name with possible solutions. Thus, patterns break down the problem to decision points making the advantages and disadvantages of the design alternatives (possible solutions) available at each decision point. The solutions of the patterns can be reused adapting them in similar problem situations reducing the time, costs and efforts of starting to solve the problems from scratch. However, experience and skills are fundamental keys required to tailor patterns to your specific requirements.

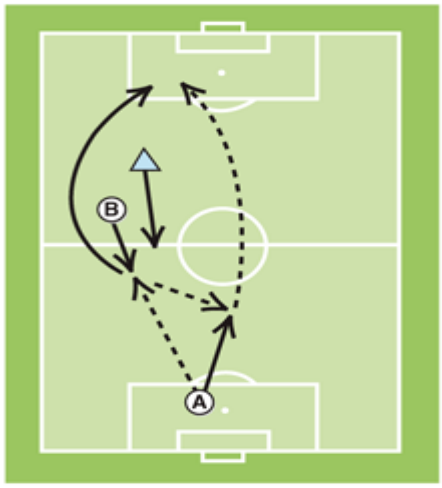


Figure 21 Patterns in soccer

While patterns have been applied recently to software development, people have been using patterns in their daily life since centuries. Patterns have been used to perform complex tasks such as sports (see Figure 21), music, science, architecture [3] and industry. Applying patterns to software development promises the same benefits to software as it does to industrial technology: predictability, risk mitigation, and increased productivity.

In software development patterns emerged in the object-oriented design through the popularity of the book “Design Patterns: Elements of Reusable Object-Oriented Software” [22]. The book presents a set of object-oriented design patterns which help the construction and reuse of software components. The patterns are described through a specification format denominated as the GoF format in order to facilitate their comprehension. Since then, patterns have been used in all areas of software such as patterns for user interface [44].

The huge and complex nature of distributed systems can be influenced by many problems which can produce their breakdown. Thus, we consider patterns as an essential technique in developing distributed software systems. The problem identification and break down and the solution reusability and adaptability which patterns enable, facilitate the development process of the distributed systems.

4.2 Mobility and Replication Reconfiguration Patterns in PRISMA

For developing distributed systems in our PRISMA architectural model, a set of conceptual mobility and replication patterns are described through the PRISMA ADL. The patterns are classified as conceptual following Riehle proposal [63] that a conceptual pattern is a pattern which is described in terms and concepts of the problem space and oriented to specification, independent of their design.

The patterns are stored in a patterns repository to enable the reusability of their specification (see Figure 22). The patterns provide guidelines to the software developer in building the distributed software architecture of the distributed software system reducing its time, cost and effort and increasing its productivity.

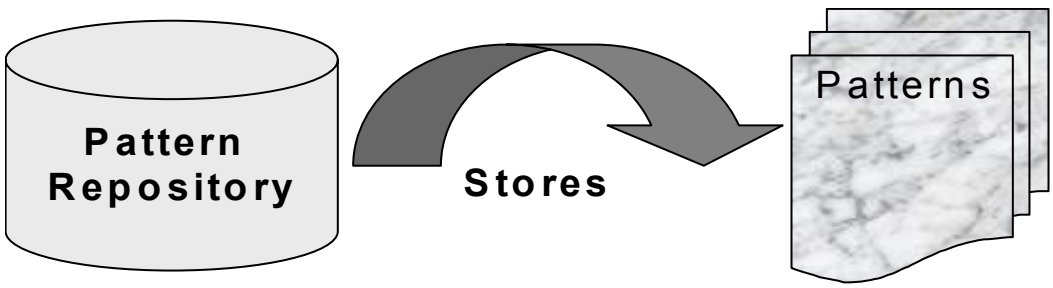


Figure 22 Storage of Pattern Specification in a Repository

The PRISMA patterns describe situations in which the software architecture has to be reconfigured either through mobility or replication of the architectural element instances at

execution-time. These situations occur due to run-time, context and changes in the requirements that can affect on the performance of the architectural instances of the architecture. Therefore, the instances are either relocated or replicated to super pass possible problems. These patterns were first introduced in [5].

Our proposal of using patterns is done through the aspects of the architectural elements. Patterns can easily make up aspects in aspect-oriented software architecture as shown in Figure 23. Each pattern defines the case in which an element can be replicated or moved. The architect can choose these patterns into different replication or distribution aspects. In Figure 23, the small balls number *1* and *3* represent different replication and distribution aspects made of different patterns from a library (repository), and the small ball number *2* represents a functional aspect. The election of the different patterns depends on the behaviour of the architecture that is needed. Analysts have to import them to form either a replication or distribution aspect, and then, aspects are weaved (composed) with other aspects to create an architectural element. As a result, an architectural element is formed by a set of aspects and an aspect is defined by one or more patterns.

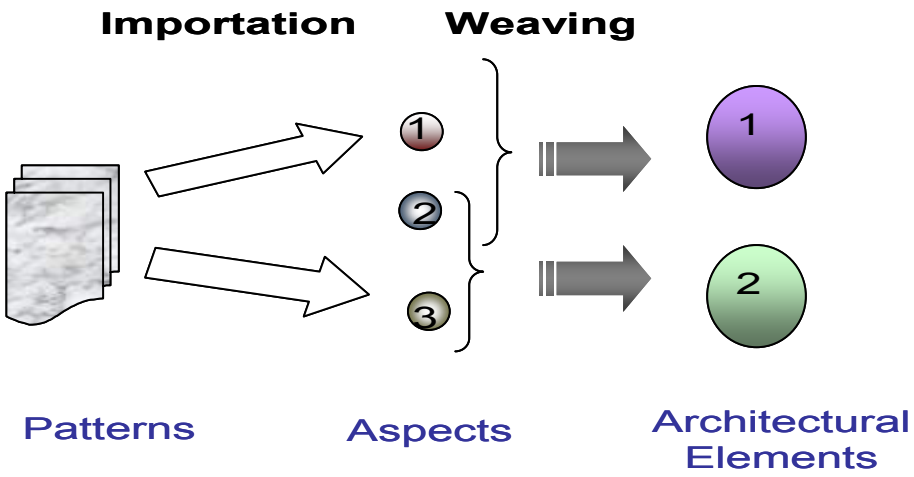


Figure 23 Use of patterns for the development process in PRISMA.

To specify many essential services necessary to specify the patterns, some language primitives are introduced in the PRISMA ADL. These pattern services are services implemented through the PRISMA middleware introduced in section 3.5. These are the additional services implemented above the different platform technologies to provide unique PRISMA functionalities. The services of the infrastructure used in the patterns specification are detailed in Table 5.

Table 5 The PRISMA infrastructure services for the specification of the distribution patterns

Infrastructure Services	Description
move(newlocation:loc)	The architectural element asks the infrastructure to move it to the indicated location specified in the parameter.
replicate(newlocation:loc)	The architectural element asks the infrastructure to replicate it to the indicated location specified in the parameter.
SearchClientLocationMostInvoked(output Services, input clientL:loc)	The architectural element asks the infrastructure to search for the location of the client which most invoked the services specified in the parameter output. The infrastructure returns the location to the architectural element.
SearchServerLocationMostReq(output Services, input serverL:loc)	The architectural element asks the infrastructure to search for the location of the server which most served the services specified in the parameter output. The infrastructure returns the location to the architectural element.
VolumeExceeds()	This service is an alarm that is invoked by the infrastructure to the architectural element informing it that the volume of data interchanged between two architectural elements is exceeded.
LatencyExceeds()	This service is an alarm that is invoked by the infrastructure to the architectural element informing it that the latency of the response time of a service has exceeded.

Infrastructure Services	Description
entersNewLocation(input NewLocation)	This service is invoked by the architectural element to the infrastructure to respond to an alarm. Depending on the alarm the infrastructure returns to the architectural element the adequate new location.

In this chapter, different mobility and replication patterns are introduced through a pattern template. The following sections are part of the template used to explain the distribution patterns:

- **Pattern Number:** The identifier of the pattern in the catalogue.
- **Pattern Name:** A name is given to describe the problem solved.
- **Problem:** The situation to which a component needs to react to by moving.
- **Solution.**
  - **Description:** A brief description of the solution.
  - **Participants:** The different entities of the PRISMA model that participate to specify the patterns
  - **Services:** The services that should be provided in order to be able to solve the problem considering that the distribution issues (distribution aspect) are independent of the functionality of a component.
  - **Graphical Representation:** The representation of the pattern is shown using the UML sequence diagrams to show the interaction among the participants of a pattern.
  - **Representation in an ADL:** The representation of the pattern in an ADL. It is important to notice that in the representation of the ADL the words marked in italics are to distinguish the services of the PRISMA infrastructure from others.

- **Special Cases:** Special cases of the pattern.
- **Consequences:** The impact obtained by applying the pattern on the software architecture.
- **Related Patterns:** Patterns are related to this pattern and the differences between them.
- **Example:** An example is used to explain to show how the pattern is used.

4.3 Mobility Reconfiguration Patterns

This section identifies problems that can be caused due to the in-proper location of the architectural element instances of a software architecture at run-time or situations where the relocation could improve the performance of the architecture. The patterns prevent the occurrence of the problems by detecting an increment of a threshold and then moving the architectural element instances to an adequate location depending on the situation. The patterns discussed in this section are summarized in Table 6.

Table 6 The Mobility Reconfiguration Patterns explained in this section

MP. Number	Pattern Name
MP.01	Excess of the arrival rate.
MP.02	Excess of the request rate.
MP.03	Excess in the volume of data interchanged.
MP.04	Excess of latency.
MP.05	Change in system requirements.

The following patterns are specified using the format explained in Section 4.2.

4.3.1 MP.01- Pattern: Excess of the arrival rate.
Problem.

The arrival rate is the total amount of services that arrive to a component. The higher the rate of the services that arrive to the architectural element, the higher the probability an element acts as a server to different client architectural elements. The element cannot have an optimized communication with all its clients if they are all distributed.

Solution.

-Description:

When the arrival rate exceeds a limit the architectural element moves to the location of the client that most frequently requests its services.

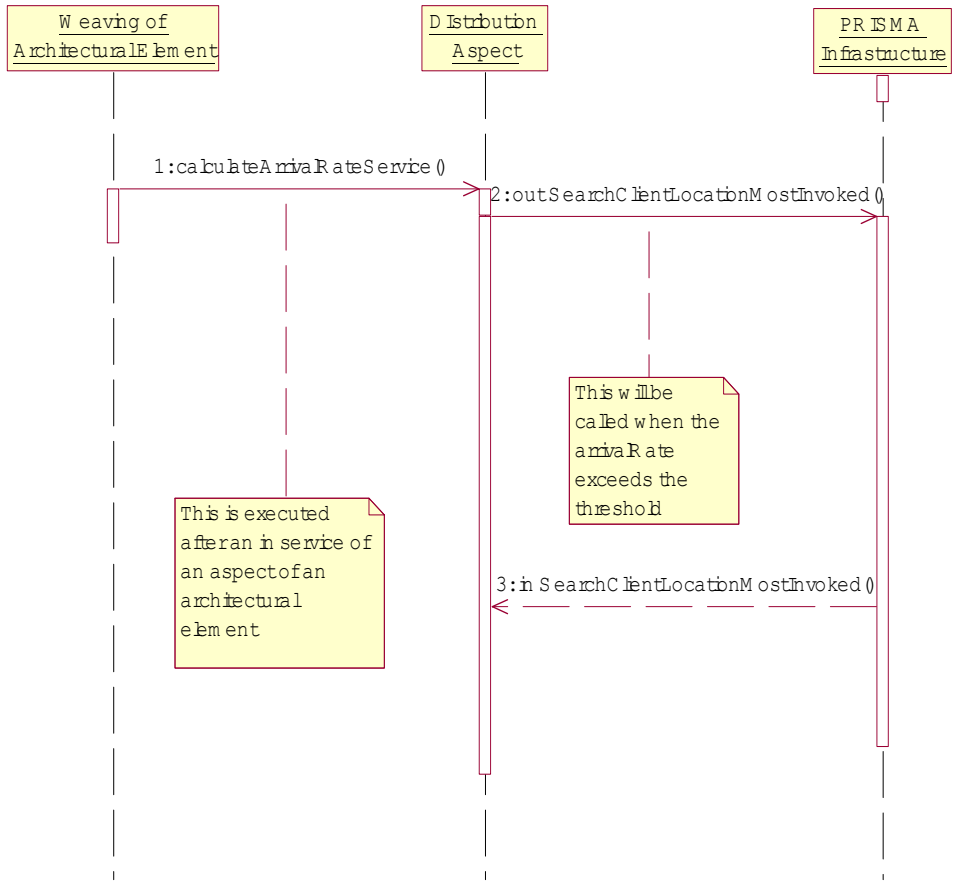
- Participants:

- The aspects of the architectural element: Each time the interested **in** services of an aspect is executed the aspects are weaved with the distribution aspect
- The distribution aspect of the architectural element: The distribution aspect calculates the **in** services invoked and notifies the PRISMA infrastructure that the threshold has exceeded. It receives the proper location (this is to check the location if the location of the architectural element is restricted) and moves to the location.
- The PRISMA infrastructure: The PRISMA infrastructure searches for the most adequate location of the new location.

- Services:

- To calculate the arrival rate: A service which calculates the arrival rate of the services to the component should exist.
- When the arrival rate exceeds a certain limit, the component needs to know which client has requested its services most frequently.
- To move the server component to the location of the client component that most frequently requests its services.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect MP1

 Attributes
  arrivalRate : nat(0);
  arrivalRateLimit : nat NOT NULL;
  clientL : loc;

 Services
  move(ClientL:loc)
  calculateArrivalRateService()
   Valuations
    [calculateArrivalRateService()
    arrivalRate:=arrivalRate+1;
  out SearchClientLocationMostInvoked(Services);
  in SearchClientLocationMostInvoked(CL:loc)
   Valuations
    [SearchClientLocationMostInvoked(CL)] clientL:=CL;
```

```
     Triggers
      MOVE when
               {arrivalRate>=arrivalRateLimit};

     operations
      MOVE( Services, CL, newlocation) transaction:
      MOVE0 ≡ out
SearchClientLocationMostInvoked(Services).MOVE1;
      MOVE1 ≡ in SearchServerLocationMostInvoked(CL).MOVE2;
      MOVE2 ≡ move(newlocation);

    End_Distribution Aspect MP1

    Architectural_type Architectural_Element Name
     Port
               ….
     End_Port;

     Import MP1: Distribution Aspect
     Import AspectName: Aspect_Type

     Weaving
         calculateArrivalRate() after in Services();

     End_Weaving;

    End_Architectural_type Name;
```

Special Cases.

- How many times a service is invoked in the whole life of the server component instead of how many times a service is invoked per unit of time.
- Another case is when we are interested in more than a requested service from only one client.
- To calculate the arrival rate of a concrete interface or port.
- To calculate the arrival rate of the services of a concrete aspect of the architectural element.

Consequences.

- **Maintainability:** When a client and a server are communicating locally, the maintainability of the communication is easily accomplished. This is due to the fact that when the components are close the breakdowns that may affect the communication channel between the client and the server are reduced.
- **Reusability:** The solution can be easily reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.
- **Flexibility:** This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

RP1.

Example.

A bank system consists of bank offices, accounts and ATMs. Each one of them is represented in a record of a distributed database. The ATM acts as a server component to the Bank Administrator. The administrator invokes the *howmanytransactions* service which indicates the number of transactions of an ATM. the analyst needs to move the ATM record to its client location (the Bank Administrator) when the Bank Administrator invokes a certain a number of times the *howmanytransactions.* These requirements are applied in order to facilitate the communication of the ATM with its client. The specification of the ATM with its distribution aspect is the following:

```
Distribution Aspect MP1

  Attributes
   location: loc NOT NULL;
   arrivalRate : nat(0);
   arrivalRateLimit : nat NOT NULL;
   clientL : loc;

  Services
   begin(Location:loc, ArrivalRateLimit:nat)
     Valuations
       [begin(Location,ArrivalRateLimit)] location:=Location
     &                 arrivalRateLimit:=ArrivalRateLimit;
   move(Newlocation:loc)
     Valuations
      [move(ClientL)] location:=ClientL;
      calculateArrivalRate();
     Valuations
           [calculateArrivalRate()]arrivalRate:=arrivalRate+1;
```

```
   out SearchClientLocationMostInvoked(Services);
   in SearchClientLocationMostInvoked(CL:loc)
     Valuations
     [SearchClientLocationMostInvoked(CL)] clientL:=CL;

 Triggers
    MOVE when
               {arrivalRate>=arrivalRateLimit};

 operations
      MOVE( Services, CL, newlocation) transaction:
      MOVE0 = out
SearchClientLocationMostInvoked(Services).MOVE1;
      MOVE1 = in SearchServerLocationMostInvoked(CL).MOVE2;
      MOVE2 = move(newlocation);

 Protocols
   MP1 ≡begin. MP11;
   MP11 ≡  MOVE + END;

End_Distribution Aspect MP1;

Component_type ATM
    Port
               ….
    End_Port;

    Import MP1: Distribution Aspect
    Import ATMFunc: Functional Aspect ;
        Weaving

           calculateArrivalRate() after
                                          in
howmanytransactions();

        End_Weaving;

End_Component_type ATM;
```

4.3.2 MP.02- Pattern: Excess of the request rate.

Problem.

The request rate is the total rate of services that a client architectural element requests. The request rate indicates how much essential is the dependence of this architectural element to the functionalities of other elements. The client cannot have an optimized communication between its servers if they are far apart.

Solution.

-Description:

When the request rate exceeds a limit the client element moves to the location of the server to which most frequently requests.

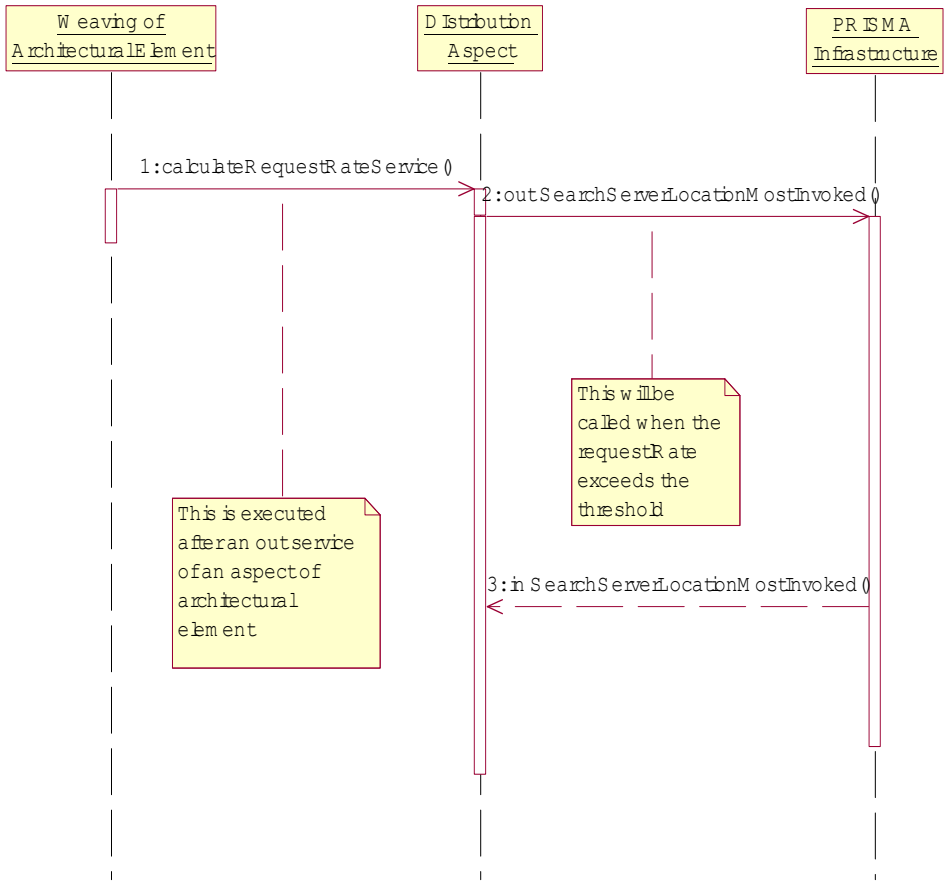
- Participants:

- The aspects of the architectural element: Each time the interested **out** services of an aspect are requested the aspects are weaved with the distribution aspect
- The distribution aspect of the architectural element: The distribution aspect calculates the **out** services requested and notifies the PRISMA infrastructure that the threshold has exceeded. It receives the proper location (this is to check the location if the location of the architectural element is restricted) and moves to the location.
- The PRISMA infrastructure: The PRISMA infrastructure searches for the most adequate location for the new location.

- Services:

- To calculate the total request rates of the services.
- When the request rate exceeds a certain limit, the architectural element needs to be aware of which server it requests most frequently.
- To move the client component to the location of the server element it most frequently requests.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect MP2

    Attributes
      requestRate : nat(0);
      requestRateLimit: nat NOT NULL;
      serverL : loc;

    Services
      calculateRequestRate()
        Valuations
          [calculateRequestRate()] requestRate:= requestRate
    +1;
      in SearchServerLocationMostReq(SL:loc)
        Valuations
            [SearchServerLocationMostReq(SL)] serverL:=SL;
```

```
    Triggers
     MOVE(Services, SL, Newlocation) when
               {requestRate >=requestRateLimit};

    Operations
     MOVE(Services, SL, Newlocation) transaction:
     MOVE0 = out
    SearchServerLocationMostReq(Services).MOVE1;
     MOVE1 = in SearchClientLocationMostReq(SL).MOVE2;
     MOVE2 = move(Newlocation);
End_Distribution Aspect MP2

Architectural_type Architectural_Element Name
      Port
                 ….
      End_Port;

      Import MP2: Distribution Aspect
      Import AspectName: Aspect_Type

       Weaving
           calculateRequestRate() after out Services();

       End_Weaving;

End_Architectural_type Architectuarl_Element Name;
```

Special Cases.

- For example, a special case could be how many times a client architectural element requests the set of service in its whole life instead of how many times a client architectural element requests services per unit of time.
- Another case is when we are interested in more than a requested service from the same client.

Consequences.

- **Maintainability**: This pattern optimizes the communication between the client and a server. However, as a client can have many servers the client constantly moves every certain unit of time to a server that it highly requests.
- **Reusability**: The solution can easily be reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

No related patterns.

Example.

A bank system consists of bank offices, accounts and ATMs. Each one of them is represented in a record of a distributed database. The ATM acts as a client component to the Account. The ATM requires the *balance* and the *withdraw* services. These requirements are applied in order to facilitate the communication of the ATM with its client. The specification of the ATM with its distribution aspect is the following:

```
Distribution Aspect MP2

  Attributes
    location: loc NOT NULL;
    requestRate : nat(0);
    requestRateLimit : nat NOT NULL;
    serverL : loc;

  Services
    begin(Location:loc, RequestRateLimit:nat)
      Valuations
      [begin(Location,RequestRateLimit)]
      location:=Location &
      requestRateLimit:=RequestRateLimit;
```

```
     move(Newlocation:loc)
      Valuations
      [move(ServerL)] location:=ServerL;
     calculateRequestRate()
      Valuations
      [calculateRequestRate()] requestRate:= requestRate +1;
     in SearchServerLocationMostReq(SL)
      Valuations
      [SearchServerLocationMostReq(SL)] serverL:=SL;

Triggers
     MOVE(Services, SL, newlocation) when
                {requestRate >=requestRateLimit};

Operations
     MOVE(Services, SL, newlocation) transaction:
     MOVE0 ≡ out SearchServerLocationMostReq(Services).MOVE1;
     MOVE1 ≡ in SearchServerLocationMostReq(SL).MOVE2;
     MOVE2 ≡ move(newlocation);

 Protocols
        MP2 ≡ begin. MP21;
        MP21 ≡  MOVE + END;

 End_Distribution Aspect MP2;


 Component_type ATM
     Port
               ….
     End_Port;

     Import MP2: Distribution Aspect
     Import ATMFunc: Functional Aspect ;

     Weaving
       calculateArrivalRate() after
                                        out balance();
        calculateArrivalRate() after
                                        out withdrawal();

      End_Weaving;
```

```
End_Component_type ATM;
```

4.3.3 MP.03- Pattern: Excess in the volume of data interchanged.

Problem.

The volume of data interchanged in a network can generate saturation in the network, especially, in cases where the target of the data can not process such amount of information. Many times the bandwidth of the communication channel can't be increased and it is necessary to cancel the communication in order to avoid saturation.

Solution.

-Description:

In order to guarantee that the destiny architectural element receives the data, the source component moves to the location of the destination.

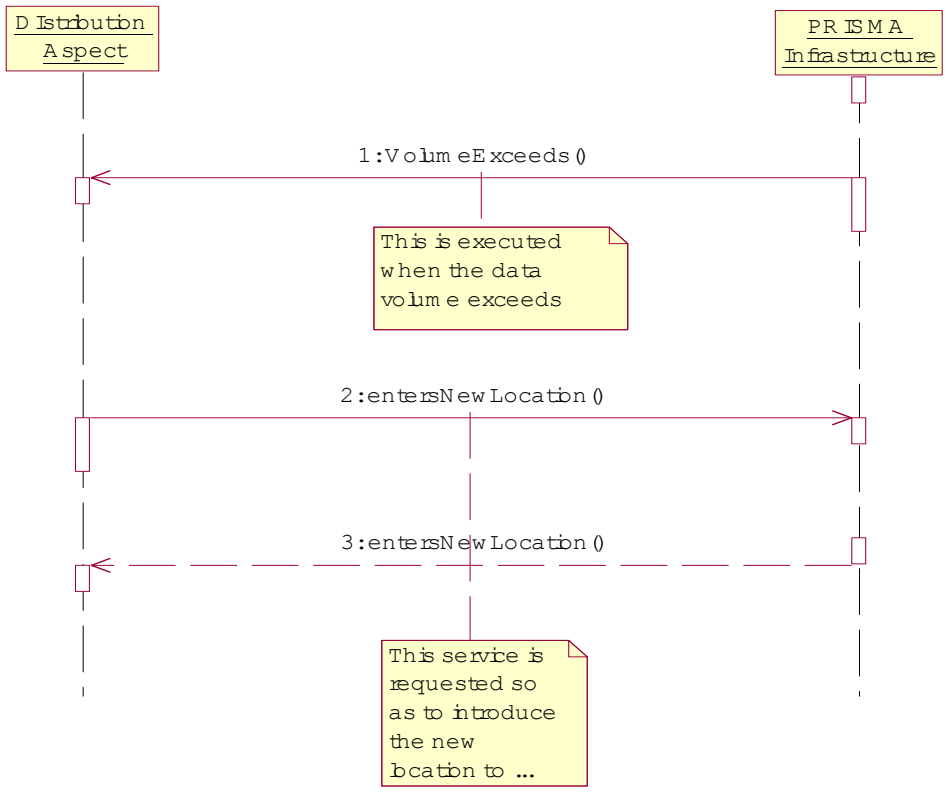
- Participants:

- The distribution aspect of the architectural element: The distribution aspect receives the proper location (this is to check the location if the location of the architectural element is restricted) and moves to the location.
- The PRISMA infrastructure: The PRISMA infrastructure notifies the distribution aspect of an excess of the data volume and searches for the most adequate location for the new location.

- Services:

- To detect when the volume of data reaches a certain limit.
- To move the component to the location of the receiver of the data.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect MP3

  Attributes
    exceedVolume: bool(false);
    newlocation: loc;

  Services
    in VolumeExceeds()
      Valuations
        [VolumeExceeds()] exceedVolume:=true;
    in entersNewLocation(NewLocation)
```

```
        Valuations
          [entersNewLocation(NewLocation)]
newLocation:=Newlocation;

    Triggers
     MOVE() when
                {exceedVolume=true; };

    Operations
     MOVE(Services, SL, newlocation) transaction:
     MOVE0 ≡ out entersNewLocation().MOVE1;
     MOVE1 ≡ in entersNewLocation(NewLocation).MOVE2;
     MOVE2 ≡ move(newlocation);

    End_Distribution Aspect MP3
```

Special Cases.

- The volume of data interchanged with a concrete component

Consequences.

- **Maintainability**: This pattern ensures that the data interchanged between two components reaches its destination.
- **Reusability**: The solution can easily be reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

RP2.

Example.

In a bank system there are elements that calculate statistics from the daily operations of each ATM every night. In this situation, it would be a bad idea to send

all the operations from all the ATMs via the net at the same time interval, because it could probably cause a collapse. The element could consider the amount of data to be sent and move itself to the ATMs in order to get all the data locally, without affecting the current time response of the whole system.

```
Distribution Aspect MP3

Attributes
    location: loc NOT NULL;
    exceedVolume: bool(false);
    newlocation: loc;

Services
  begin(Location:loc)
    Valuations
     [begin(Location)]
      location:=Location;
  in move(NewLocation)
    Valuations
      [move(NewLocation)] location:=Newlocation;
  in VolumeExceeds()
    Valuations
      [VolumeExceeds()] exceedVolume:=true;
  in entersNewLocation(NewLocation)
    Valuations
     [entersNewLocation(NewLocation)]
newLocation:=Newlocation;

Triggers
  MOVE() when
              {exceedVolume=true };

Operations
  MOVE(Services, SL, newlocation) transaction:
  MOVE0 ≡ out entersNewLocation().MOVE1;
  MOVE1 ≡ in entersNewLocation(NewLocation).MOVE2;
  MOVE2 ≡ move(newlocation);

Protocols
    MP3 ≡ begin. MP31;
    MP31 ≡  MOVE + END;

End_Distribution Aspect MP3
```

```
Component_type ATM
   Port
            ….
   End_Port;

   Import MP3: Distribution Aspect
      …

   End_Component_type ATM;
```

4.3.4 MP.04- Pattern: Excess of latency.

Problem.

Latency is the time it takes for a service to reach the server plus the time it takes the server to respond (response time). The latency of a requested service exceeds a specific time means that problems are occurring in the network. If a certain element detects that a latency of its message has exceeded, then the element has to try to accommodate his situation as easily as possible so as to continue his work.

Solution.

-Description:

The architectural element has to try to accommodate itself as easily as possible in order to continue working. To accomplish this it moves itself to the location of its server.

- Participants:

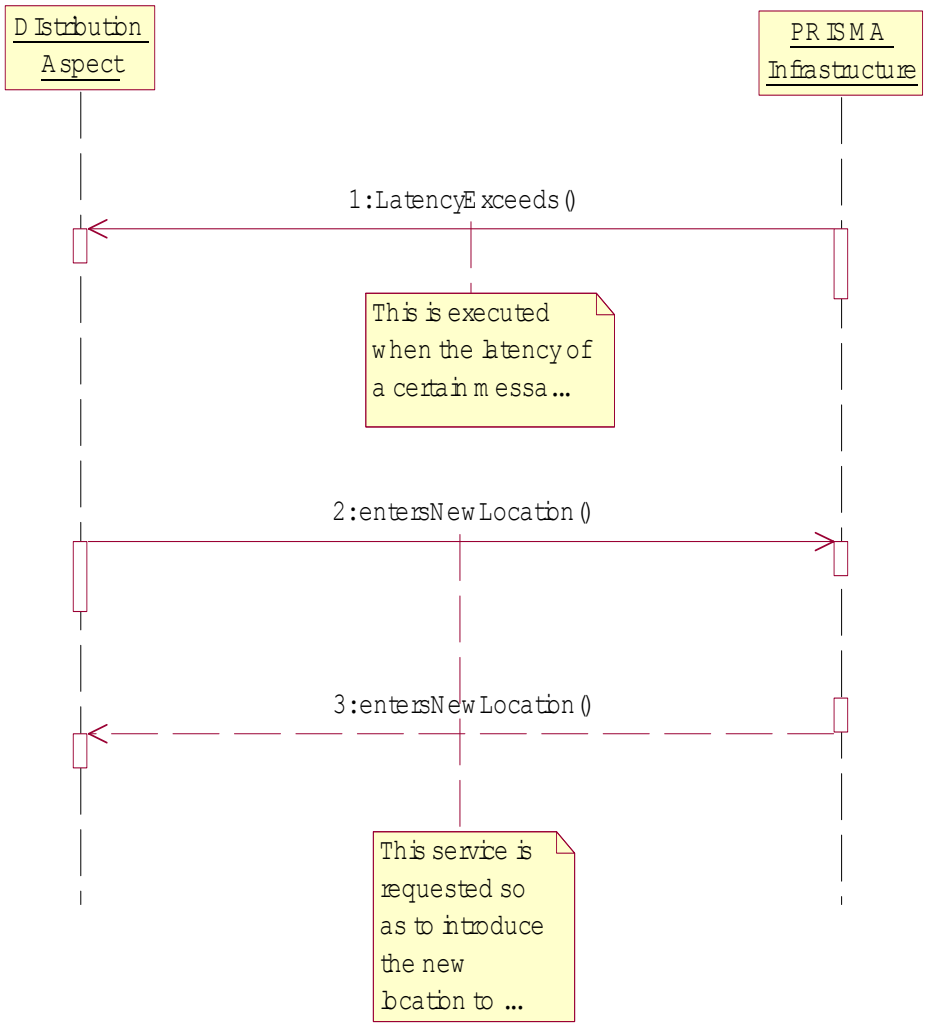
- The distribution aspect of the architectural element: The distribution aspect receives the proper location (this is to check the location if the location of the architectural element is restricted) and moves to the location.
- The PRISMA infrastructure: The PRISMA infrastructure notifies the

distribution aspect of an excess of the latency and searches for the most adequate location for the new location.

- Services:

- To calculate the latency.
- To move the component to the location of the server.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect MP4

   Attributes
    exceedLatency: bool(false);
    newlocation: loc;
```

```
    Services
     in LatencyExceeds()
       Valuations
          [LatencyExceeds()] exceedLatency:=true;
     in entersNewLocation(NewLocation)
       Valuations
          [entersNewLocation(NewLocation)]
newLocation:=Newlocation;

    Triggers
     MOVE() when
                {exceedLatency=true};

    Operations
     MOVE( newlocation) transaction:
     MOVE0 ≡ out entersNewLocation().MOVE1;
     MOVE1 ≡ in entersNewLocation(NewLocation).MOVE2;
     MOVE2 ≡ move(newlocation);

  End_Distribution Aspect MP4
```

Special Cases.

- For example, a special case could be the average latency of the requested services of a client component.

Consequences.

- **Maintainability**: This pattern optimizes the communication between the client and a server even though there are problems in the network.
- **Reusability**: The solution can be easily reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.

- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

RP3.

Example.

In a bank system there are elements that calculate statistics from the daily operations of each ATM every night. In this situation, it would be a bad idea to send all the operations from all the ATMs via the net at the same time interval, because it could probably cause a collapse. The element could consider the amount of data to be sent and move itself to the ATMs in order to get all the data locally, without affecting the current time response of the whole system.

```
Distribution Aspect MP4

Attributes
 exceedLatency: bool(false);
 newlocation: loc;

Services
 begin(Location:loc)
   Valuation
    [begin(Location)]
     location:=Location;
 in LatencyExceeds()
   Valuations
     [LatencyExceeds()] exceedLatency:=true;
 in entersNewLocation(NewLocation)
   Valuations
     [entersNewLocation(NewLocation)]
newLocation:=Newlocation;

Triggers
 MOVE() when
            {exceedLatency=true};
```

```
    Operations
      MOVE( newlocation) transaction:
      MOVE0 ≡ out entersNewLocation().MOVE1;
      MOVE1 ≡ in entersNewLocation(NewLocation).MOVE2;
      MOVE2 ≡ move(newlocation);

    Protocols
      MP4 ≡begin. MP41;
      MP41 ≡  MOVE + END;

End_Distribution Aspect MP4

Component_type ATM
  Port
            ….
  End_Port;

  Import MP4: Distribution Aspect
     …

End_Component_type ATM;
```

4.3.5 MP.05- Pattern: Change in system requirements.

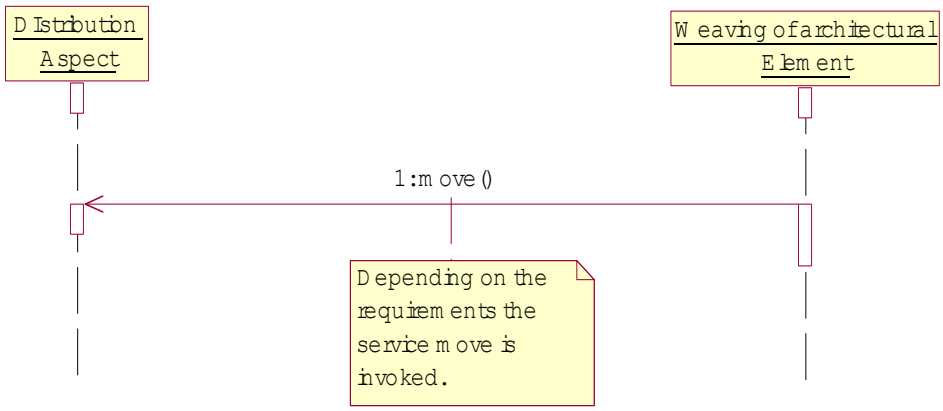
Problem.
System requirements are volatile and are continuously changing. Occasionally, these new system requirements alter the location of the architectural element. Therefore, the architectural element needs to adapt to the new requirements by moving itself to a new location.
Solution.
-Description:
The architectural element is relocated to the new location depending on the new requirement.
- Participants:

- The aspects of the architectural element: When an interesting change occurs a weaving is performed with the distribution aspect.
- The distribution aspect of the architectural element: The distribution aspect moves the architectural element.

- Services:

- To assign a new location depending on the requirements.
- To move the component to the new location.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect MP5

    Services
      in move(NewLocation)
        Valuations
          [move(NewLocation)] newLocation:=NewLocation;

End_Distribution Aspect MP5

Architectural_type Architectural_Element Name
    Port
              ….
    End_Port;

    Import MP5: Distribution Aspect
    Import AspectName: Aspect_Type
```

```
    Weaving
        move() after Service();

    End_Weaving;
End_Architectural_type Architectuarl_Element Name;
```

Special Cases.

- The new location is determined by consulting a database.

Consequences.

- **Maintainability**: This pattern ensures the independence of the evolution of the distribution issues between from the other functionalities of the component.
- **Reusability**: The solution can be easily reused by any entity responsible of the distribution properties of an architectural model independently from its functionality. Moreover, an architectural model which does not make this separation can reuse the pattern however, does not achieve to an independence of the functionality from the distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the components functionalities.

Related Patterns.

RP4.

Example.

This pattern can be applied to the example of the bank system in the case a customer changes its address. Therefore, the main bank office that attends the customers' requests and services will have to change to a bank office nearer to his/her new address. In this situation the customer's account will move to the new bank office.

```
Distribution Aspect MP5

  Attributes
     location:loc NOT NULL;

  Services
    begin(Location:loc)
      Valuations
       [begin(Location)]
        location:=Location;
    move(newlocation:loc)
      Valuations
       [move(NewLocation)] location:=NewLocation;

  Protocols
  MP5 ≡ begin. MP51;
  MP51 ≡  move + END;

End_Distribution Aspect MP5

Component_type Account
   Port
            ….
   End_Port;

   Import MP5: Distribution Aspect
   Import FAccount: Functional Aspect
   Weaving
         move() after changeAddress();
   End_Weaving;

End_Component_type Account;
```

4.4 Replication Reconfiguration Patterns

This section presents the patterns where replication is used to prevent problems caused due to the in-proper location of the architectural element instances of a software architecture at run-time. The patterns prevent the occurrence of the problems by detecting an increment of a threshold and then making a distributed replication the architectural element instances to an adequate location depending on the situation. The patterns discussed in this section are summarized in Table 7.

The difference between the patterns in this section and the patterns in the previous section is that in many cases the architectural elements cannot be moved or a local copy is necessary. In many cases, replication is preferable to mobility for balancing load either of services or data. In PRISMA, we make it flexible so that the analyst determines the most adequate situation. However, the analyst needs to keep in mind that the same mobility and replication patterns are incompatible with each other, i.e. if a mobility and replication pattern have the same name only one of them should be applied but not both.

Table 7 Replication Reconfiguration Patterns explained in this section

RP. Number	Pattern Name
RP.01.	Unbalanced load.
RP.02.	Excess of Volume of Data.
RP.03.	Latency of services.
RP.04.	System Requirements and Configuration Adaptation.

The replication patterns are presented in the following:

4.4.1 RP.01- Pattern: Unbalanced load.

Problem.

An architectural element can reach to a state of unbalanced load due to the high request rate of a certain service. Components that reach to an unbalance load are mainly architectural elements that are essential for the functionality of the system so if an architectural element reaches to this state not only this element enters this fatal state but also the whole system.

Solution.

-Description:

When the arrival rate exceeds a limit the architectural element replicates to the location of the client that most frequently requests its services.

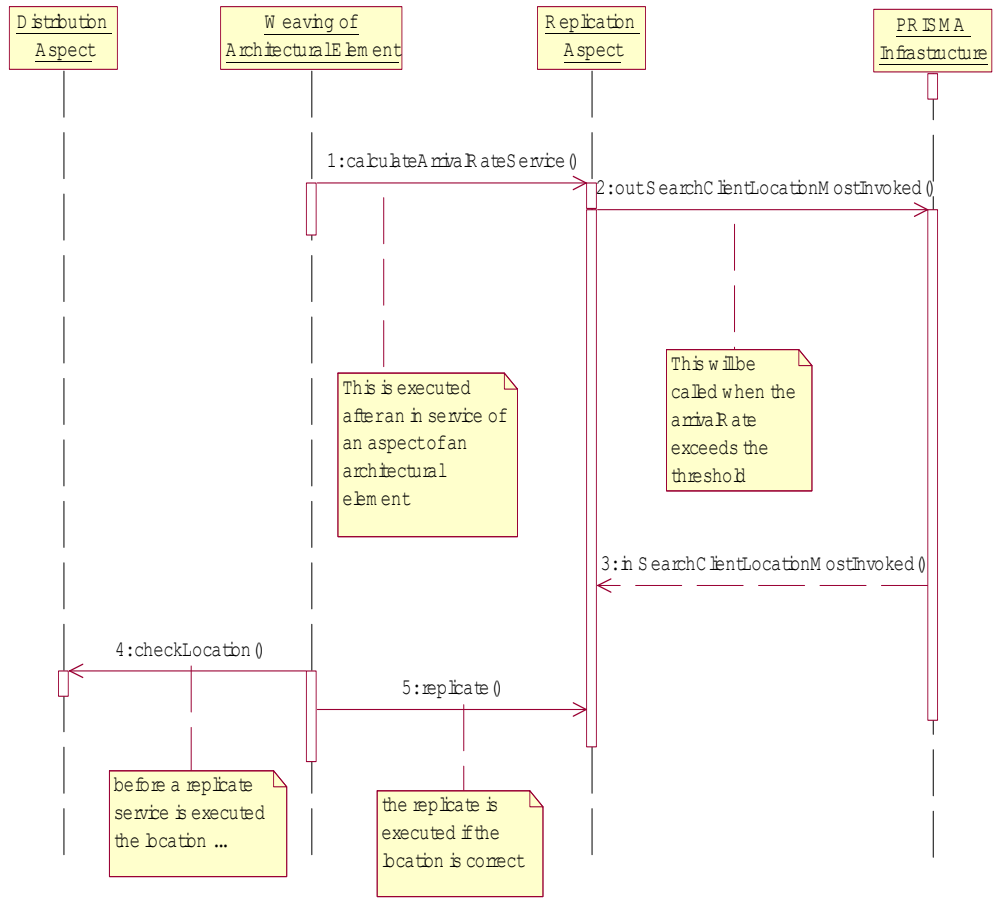
- Participants:

- The aspects of the architectural element: Each time the interested in services of an aspect is executed the aspects are weaved with the replication aspect.
- The replication aspect of the architectural element: The replication aspect calculates the in services invoked and notifies the PRISMA infrastructure that the threshold has exceeded. It receives the proper location (this is to check the location if the location of the architectural element is restricted) and replicates to the location.
- The distribution aspect of the architectural element: It receives the proper location (this is to check the location if the location of the architectural element is restricted)
- The PRISMA infrastructure: The PRISMA infrastructure searches for the most adequate location of the new location.

- Services:

- To calculate the arrival rate of services.
- To detect when the arrival rate exceeds a certain limit.
- To replicate the server component to the locations of the clients of the services.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect D

 Attributes
   locMin: loc NOT NULL;
   locMax: loc NOT NULL;

Services
   checkLocation( input Location:loc, output checkC:bool)
     Valutions
       [checkLocation(input Location:loc,output
checkC:bool)]
            {Location>locMin & Location<locMax}

[checkLocation(Location,checkC)]checkC:=true;
            {Location<locMin & Location>locMax}
```

```
[checkLocation(Location,checkC)]checkC:=false;

    End_Distribution Aspect D

    Replication Aspect RP1
      Attributes
       arrivalRate : nat(0);
       arrivalRateLimit : nat NOT NULL;
       clientL : loc;

      Services
       replicate(ClientL:loc)
       calculateArrivalRateService()
         Valuations
          [calculateArrivalRateService()]
                        arrivalRate:=arrivalRate+1;
       out SearchClientLocationMostInvoked(Services);
       in SearchClientLocationMostInvoked(CL:loc)
         Valuations
          [SearchClientLocationMostInvoked(CL)] clientL:=CL;

      Triggers
       REPLICATE when
                   {arrivalRate>=arrivalRateLimit};
      Operations
       REPLICATE( Services, CL, ClientL) transaction:
       REPLICATE0 ≡ out

SearchClientLocationMostInvoked(Services).REPLICATE1;
       REPLICATE1 ≡ in

SearchServerLocationMostInvoked(CL).REPLICATE2;
       REPLICATE2 ≡ replicate(ClientL);

    End_Replication Aspect RP1

    Architectural_type Architectural_Element Name
      Port
                 ….
      End_Port;

      Import RP1: Replication Aspect
      Import D: Distribution Aspect
```

```
        Import AspectName: Aspect_Type

         Weaving
              calculateArrivalRate() after in Services();
              checkLocation(Location,Checkc)
                       beforeif(Checkc=true)

replicate(Newlocation);
             End_Weaving;

      End_Architectural_type Architectuarl_Element Name;
```

Special Cases.

- To control the input services of a specific port of an architectural element
- To control the set of input services of the architectural element.
- Instead of controlling the load of a specific port, is to control the load of the entire imports of an architectural element and detect which one of the set is causing the highest load to the architectural element.

Consequences.

- **Maintainability**: When a client and a server are communicating locally, the maintainability of the communication is easily accomplished. This is due to the fact that when the components are close the breakdowns that may affect the communication channel between the client and the server are reduced.
- **Reusability**: The solution can be easily reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

MP1.

Example.

A bank system consists of bank offices, accounts and ATMs. Each one of them is represented in a record of a distributed database. The ATM acts as a server component to the Bank Administrator. The administrator invokes the *howmanytransactions* service which indicates the number of transactions of an ATM. the analyst needs toreplicate the ATM record to its client location (the Bank Administrator) when the Bank Administrator invokes a certain a number of times the *howmanytransactions.* These requirements are applied in order to facilitate the communication of the ATM with its client and to balance the load. The specification of the ATM with its replication aspect and distribution aspect is the following:

```
Distribution Aspect D

  Attributes
    location: loc NOT NULL;
    locMin: loc NOT NULL;
    locMax: loc NOT NULL;

  Services
     begin(Location:loc, LocMAx:loc, LocMin:loc)
       Valutions
        [begin(Location,LocMax, LocMin)]
          location:=Location & locMax:=LocMax
&locMin:=LocMin;

     checkLocation( input Location:loc, output
checkC:bool)
       Valuations
        [checkLocation(input Location:loc,output
checkC:bool)]
            {Location>locMin & Location<locMax}

        [checkLocation(Location,checkC)]checkC:=true;
            {Location<locMin & Location>locMax}

        [checkLocation(Location,checkC)]checkC:=false;

  Protocols
    DISTRIBUTE :
    DISTRIBUTE  ≡ begin.DISTRIBUTE1;
    DISTRIBUTE1≡ end+checkLocation;

End_Distribution Aspect D
```

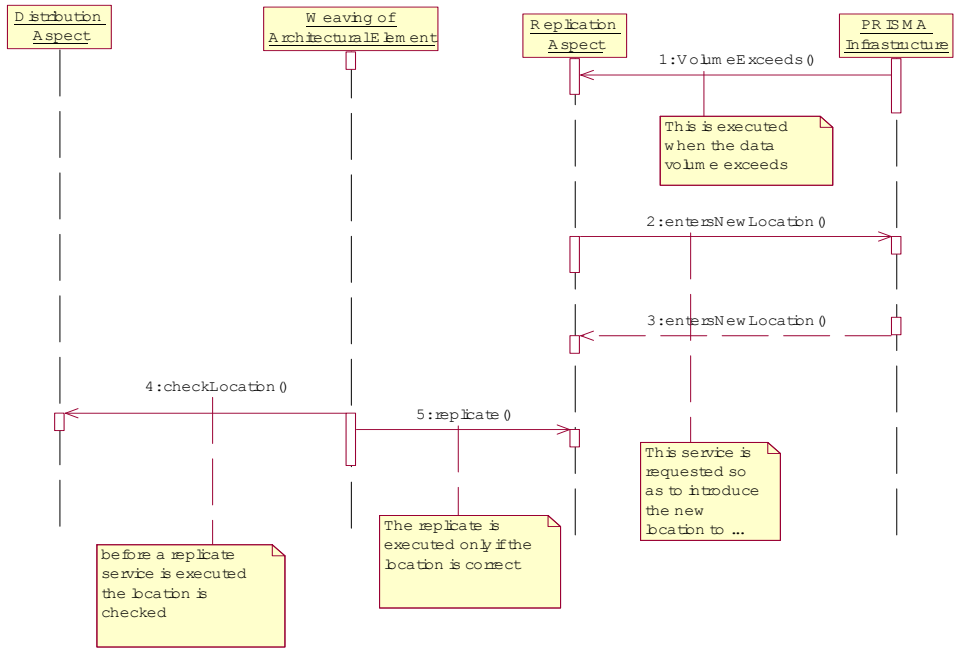
```
Replication Aspect RP1

Attributes
  arrivalRate : nat(0);
  arrivalRateLimit : nat NOT NULL;
  clientL : loc;

Services
  begin(ArrivalRateLimit:nat)
    Valuations
     [begin(ArrivalRateLimit)]
      arrivalRateLimit:=ArrivalRateLimit;
  replicate(ClientL:loc)
  calculateArrivalRateService()
    Valuations
     [calculateArrivalRateService()]
                  arrivalRate:=arrivalRate+1;
  out SearchClientLocationMostInvoked(Services);
  in SearchClientLocationMostInvoked(CL:loc)
    Valuations
     [SearchClientLocationMostInvoked(CL)] clientL:=CL;

Triggers
  REPLICATE when
            {arrivalRate>=arrivalRateLimit};

Operations
  REPLICATE( Services, CL, ClientL) transaction:
  REPLICATE0 ≡ out
SearchClientLocationMostInvoked(Services).REPLICATE1;
  REPLICATE1 ≡ in
SearchServerLocationMostInvoked(CL).REPLICATE2;
  REPLICATE2 ≡ replicate(ClientL);

End_Replication Aspect RP1


Component_type ATM
Port
           ….
```

```
End_Port;

Import RP1: Replication Aspect;
Import D: Distribution Aspect;
Import ATMFunc: Functional Aspect ;

Weaving

  calculateArrivalRate() after
                                  in howmanytransactions();
  checkLocation(Location,Checkc)
              beforeif(Checkc=true)
                                     replicate(Newlocation);


 End_Weaving;

End_Component_type ATM;
```

4.4.2 RP.02- Pattern: Excess of the volume of data.

Problem.
The volume of data interchanged in a network can generate saturation in the network, especially, in cases where the target of the data can not process such amount of information. Many times the bandwidth of the communication channel can't be increased and it is necessary to cancel the communication in order to avoid saturation.
Solution.
-Description:
In order to guarantee that the destiny architectural element receives the data, the source component replicates to the location of the destination balancing the volume of data and facilitating the communication.
- Participants:

• The replication aspect of the architectural element: The replication aspect receives the proper location and replicates after checking the correctness of the location. • The distribution aspect of the architectural element: It checks the correctness of the location if the location of the architectural element is restricted. • The PRISMA infrastructure: The PRISMA infrastructure notifies the replication aspect of an excess of the data volume and searches for the most adequate location for the new location.
- Services:
• To detect when the volume of data reaches a certain limit. • To replicate the component to the location of the receiver of the data.
- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect D

  Attributes
     locMin: loc NOT NULL;
     locMax: loc NOT NULL;

  Services
     checkLocation( input Location:loc, output checkC:bool)
       Valuations
         [checkLocation(input Location:loc,output
checkC:bool)]
              {Location>locMin & Location<locMax}

         [checkLocation(Location,checkC)]checkC:=true;
               {Location<locMin & Location>locMax}

         [checkLocation(Location,checkC)]checkC:=false;

End_Distribution Aspect D


Replication Aspect RP2

  Attributes
```

```
        exceedVolume: bool(false);
        newlocation: loc;

    Services
       replicate(Newlocation:loc);
       in VolumeExceeds()
         Valuations
           [VolumeExceeds()] exceedVolume:=true;
       in entersNewLocation(NewLocation)
         Valuations

[entersNewLocation(NewLocation)]newLocation:=Newlocation;

    Triggers
       REPLICATE() when
                  {exceedVolume=true};

    Operations
        REPLICATE (Newlocation,newLocation) transaction:
        REPLICATE0 ≡ out entersNewLocation().REPLICATE1;
        REPLICATE1 ≡ in entersNewLocation(NewLocation).
REPLICATE2;
        REPLICATE2 ≡ replicate(newlocation);

    End_Replication Aspect RP2
```

Special Cases.

- The volume of data interchanged with a concrete component

Consequences.

- **Maintainability**: This pattern ensures that the data interchanged between two components reaches its destination.
- **Reusability**: The solution can easily be reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

MP3.

Example.

In a bank system there are elements that calculate statistics from the daily operations of each ATM every night. In this situation, it would be a bad idea to send all the operations from all the ATMs via the net at the same time interval, because it could probably cause a collapse. The element could consider the amount of data to be sent and replicate itself to the ATMs in order to get all the data locally, without affecting the current time response of the whole system.

```
Distribution Aspect D

  Attributes
      location: loc NOT NULL;
      locMin: loc NOT NULL;
      locMax: loc NOT NULL;

  Services
      begin(Location:loc, LocMAx:loc, LocMin:loc)
          Valuations
           [begin(Location,LocMax, LocMin)]
               location:=Location & locMax:=LocMax &
locMin:=LocMin;
      checkLocation( input Location:loc, output
checkC:bool)
          Valuations
           [checkLocation(input Location:loc,output
checkC:bool)]
```

```
                {Location>locMin & Location<locMax}

              [checkLocation(Location,checkC)]checkC:=true;
                {Location<locMin & Location>locMax}

              [checkLocation(Location,checkC)]checkC:=false;


       Protocols
           D :
           D  ≡ begin.D1;
           D1≡ end+checkLocation;

   End_Distribution Aspect D;


Replicaiontion Aspect RP2

     Attributes
      exceedVolume: bool(false);
      newlocation: loc;

     Services
      in replicate(NewLocation)
      in VolumeExceeds()
        Valuations
          [VolumeExceeds()] exceedVolume:=true;
      in entersNewLocation(NewLocation)
        Valuations
          [entersNewLocation(NewLocation)]
  newLocation:=Newlocation;

      Triggers
      REPLICATE() when
                {exceedVolume=true };

      Operations
        REPLICATE(NewLocation, newlocation) transaction:
        REPLICATE0 ≡ out entersNewLocation().REPLICATE1;
        REPLICATE1 ≡ in entersNewLocation(NewLocation).
  REPLICATE2;
        REPLICATE2 ≡ replicate(newlocation);

       Protocols
```

```
    RP2 ≡ begin.RP21;

    RP21 ≡  replicate + END;


End_Replication Aspect RP2


Component_type ATM
    Port
                ….
    End_Port;

    Import D: Distribution Aspect
    Import RP2: Replication Aspect

    Weaving
        checkLocation(Location,Checkc)
                    beforeif(Checkc=true)
                                        replicate(Newlocation);


    End_Weaving

End_Component_type ATM;
```

4.4.3 RP.03- Pattern: Excess of latency.

Problem.

Latency is the time it takes for a service to reach the server plus the time it takes the server to respond (response time). The latency of a requested service exceeds a specific time means that problems are occurring in the network. If a certain element detects that a latency of its message has exceeded, then the element has to try to accommodate his situation as easily as possible so as to continue his work.

Solution.

-Description:

The architectural element has to try to accommodate itself as easily as possible in order to continue working. To accomplish this it replicates itself to the location of its server.

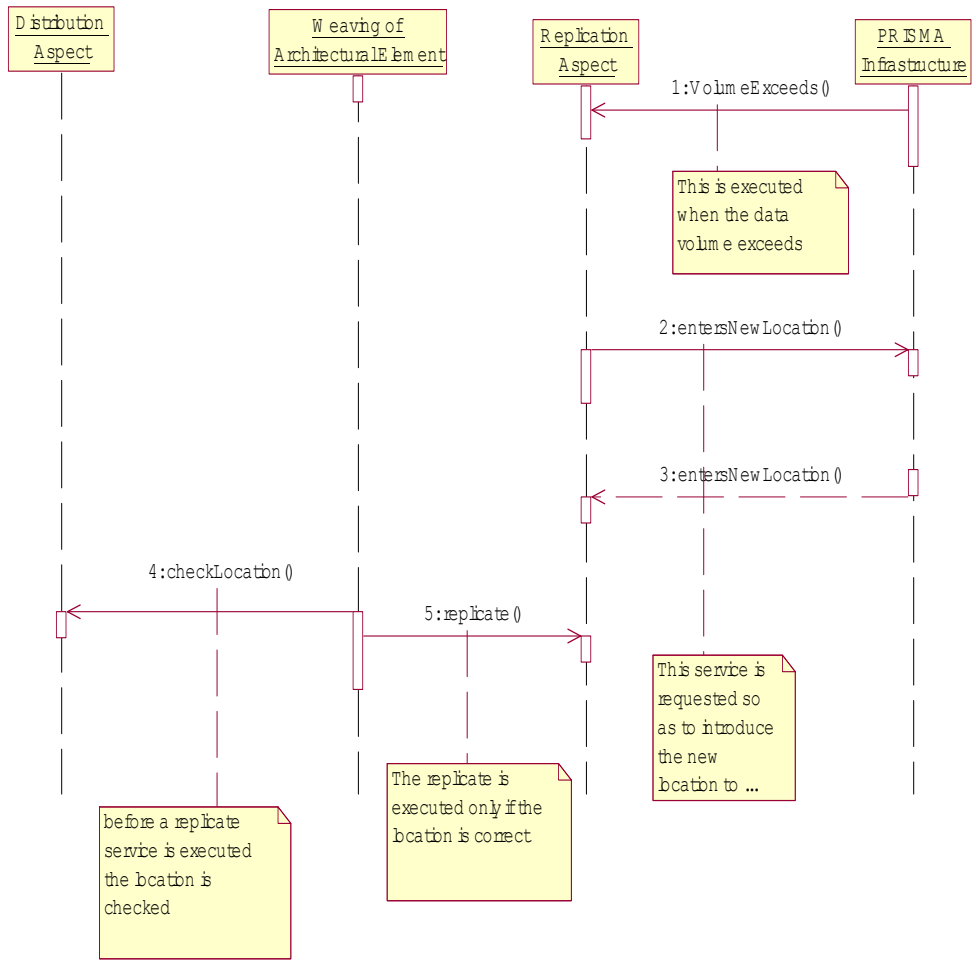
- Participants:

- The replication aspect of the architectural element: The replication aspect receives the proper location and replicates after checking the correctness of the location.
- The distribution aspect of the architectural element: It checks the correctness of the location if the location of the architectural element is restricted.
- The PRISMA infrastructure: The PRISMA infrastructure notifies the replication aspect of an excess of the latency and searches for the most adequate location for the new location.

- Services:

- To calculate the latency.
- To replicate the component to the location of the server.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect D

  Attributes
    locMin: loc NOT NULL;
    locMax: loc NOT NULL;

  Services
   checkLocation( input Location:loc, output checkC:bool)
     Valution
       [checkLocation(input Location:loc,output checkC:bool)]
            {Location>locMin & Location<locMax}

[checkLocation(Location,checkC)]checkC:=true;
            {Location<locMin & Location>locMax}

[checkLocation(Location,checkC)]checkC:=false;
```

```
End_Distribution Aspect D;


Repliation Aspect RP3

  Attributes
       exceedLatency: bool(false);
       newlocation: loc;

  Services
       in LatencyExceeds()
         Valuations
           [LatencyExceeds()] exceedLatency:=true;
       in entersNewLocation(NewLocation)
         Valuations
           [entersNewLocation(NewLocation)]
       newLocation:=Newlocation;

  Triggers
      REPLICATE(NewLocation, newLocation) when
                 {exceedLatency=true;};

  Operations
      REPLICATE(NewLocation, newLocation) transaction:
      REPLICATE0 ≡ out entersNewLocation().REPLICATE1;
      REPLICATE1 ≡ in entersNewLocation(NewLocation).
REPLICATE2;
      REPLICATE2 ≡ replicate(newlocation);

  End_Replication Aspect RP3
```

Special Cases.

- For example, a special case could be the average latency of the requested services of a client component.

Consequences.

- **Maintainability**: This pattern optimizes the communication between the client and a server even though there are problems in the network.
- **Reusability**: The solution can be easily reused by any entity responsible for the distribution properties of an architectural model independently from its functionality. However, an architectural model which does not make this separation can still reuse the pattern; but, it does not achieve to the independence of the functionality from distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the component functionalities.

Related Patterns.

MP4.

Example.

In a bank system there are elements that calculate statistics from the daily operations of each ATM every night. In this situation, it would be a bad idea to send all the operations from all the ATMs via the net at the same time interval, because it could probably cause a collapse. The element could consider the amount of data to be sent and replicate itself to the ATMs in order to get all the data locally, without affecting the current time response of the whole system.

```
Distribution Aspect D

      Attributes
       location: loc NOT NULL;
       locMin: loc NOT NULL;
       locMax: loc NOT NULL;

    Services
     begin(Location:loc, LocMAx:loc, LocMin:loc)
               Valution
                [begin(Location,LocMax, LocMin)]
                   location:=Location & locMax:=LocMax &
locMin:=LocMin;
     checkLocation( input Location:loc, output checkC:bool)
       Valuation
        [checkLocation(input Location:loc,output checkC:bool)]
             {Location>locMin & Location<locMax}
```

```
[checkLocation(Location,checkC)]checkC:=true;
             {Location<locMin & Location>locMax}

[checkLocation(Location,checkC)]checkC:=false;

    Protocols
    D :
    D  ≡ begin.D1;
    D1≡ end+checkLocation;

  End_Distribution Aspect D;


   Repliation Aspect RP3

     Attributes
      exceedLatency: bool(false);
      newlocation: loc;

    Services
      in LatencyExceeds()
        Valuations
          [LatencyExceeds()] exceedLatency:=true;
      in entersNewLocation(NewLocation)
        Valuations

 [entersNewLocation(NewLocation)]newLocation:=Newlocation;

    Triggers
      REPLICATE(NewLocation, newLocation) when
                 {exceedLatency=true;};

   Operations
      REPLICATE(NewLocation, newLocation) transaction:
      REPLICATE0 ≡ out entersNewLocation().REPLICATE1;
      REPLICATE1 ≡ in entersNewLocation(NewLocation).
REPLICATE2;
      REPLICATE2 ≡ replicate(newlocation);

   Protocols
       REPLICATEP ≡begin. REPLICATEP1;
       REPLICATEP1 ≡  replicate + END;
```

```
End_Replication Aspect RP3

Component_type ATM
    Port
                ….
    End_Port;

    Import D: Distribution Aspect
    Import RP3: Replication Aspect

    Weaving
     checkLocation(Location,Checkc)
                                beforeif(Checkc=true)
                                      replicate(Newlocation);

   End_Weaving

 End_Component_type ATM;
```

4.4.4 RP.04- Pattern: System Requirements and Configuration Adaptation.

Problem.
System requirements are volatile and are continuously changing. Occasionally, these new system requirements alter the configuration of the software architecture. Therefore, the architectural element needs to adapt to the new requirements by replicating itself to a new location.
Solution.
-Description:
The architectural element is replicated to the new location depending on the new requirement.
- Participants:
• The aspects of the architectural element: When an interesting change

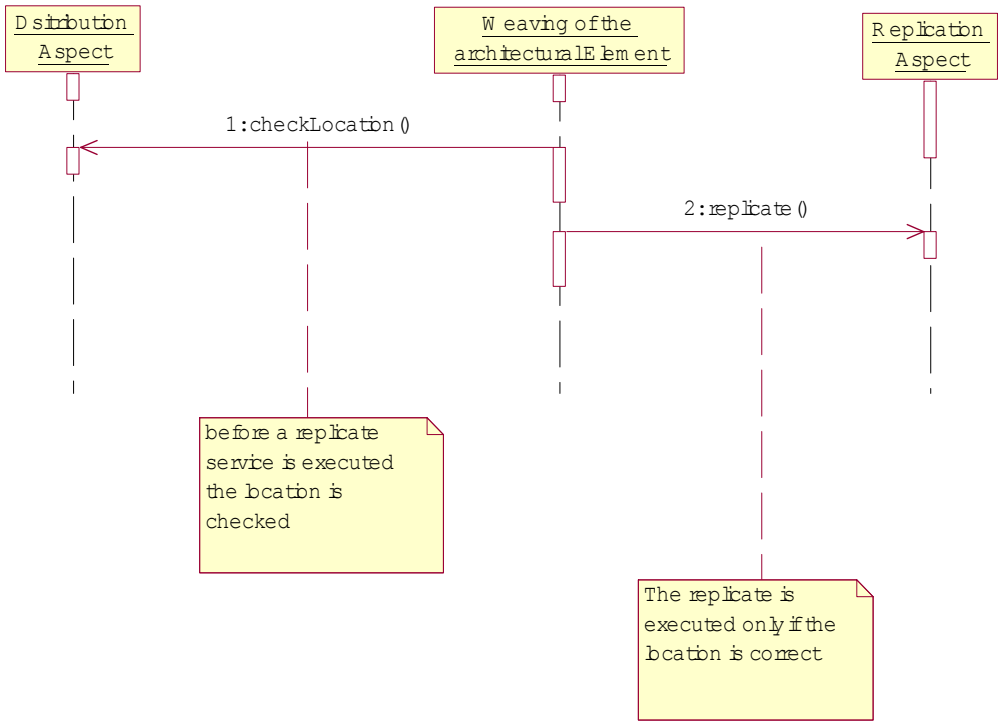
occurs a weaving is performed with the replication aspect

- The distribution aspect of the architectural element: The distribution aspect checks the new location.
- The replication aspect of the architectural element: The replication aspect replicates the architectural element.

- Services:

- To assign a new location depending on the requirements.
- To replicate the component to the new location.

- Graphical Representation:



-Representation in ADL:

```
Distribution Aspect D

  Attributes
    locMin: loc NOT NULL;
```

```
      locMax: loc NOT NULL;

   Services
    checkLocation( input Location:loc, output checkC:bool)
     Valuations
      [checkLocation(input Location:loc,output checkC:bool)]
           {Location>locMin & Location<locMax}

[checkLocation(Location,checkC)]checkC:=true;
           {Location<locMin & Location>locMax}

[checkLocation(Location,checkC)]checkC:=false;

 End_Distribution Aspect D;


 Replication Aspect RP4

   Services
    in replicate(NewLocation:loc)

 End_Replication Aspect RP4

 Architectural_type Architectural_Element Name
   Port
              ….
   End_Port;

   Import RP4: Replication Aspect;
   Import D: Distribution Aspect;
   Import AspectName: Aspect_Type;

      Weaving
        replicate() after Service();
        checkLocation(Location,Checkc)
                    beforeif(Checkc=true)
                             replicate(Newlocation);

       End_Weaving;

 End_Architectural_type Architectuarl_Element Name;
```

Special Cases.

- The new location is determined by consulting a database.

Consequences.

- **Maintainability**: This pattern ensures the independence of the evolution of the distribution issues from the other functionalities of the component.
- **Reusability**: The solution can be easily reused by any entity responsible of the distribution properties of an architectural model independently from its functionality. Moreover, an architectural model which does not make this separation can reuse the pattern however, does not achieve to an independence of the functionality from the distribution.
- **Flexibility**: This pattern is easily adapted to other distributed behaviours and to the components functionalities.

Related Patterns.

MP5.

Example.

This pattern can be applied to the example of the bank system in the case a customer changes its address. Therefore, the main bank office that attends the customers' requests and services will have to change to a bank office nearer to his/her new address. In this situation the customer's account will replicate to the new bank office.

```
Distribution Aspect D

  Attributes
    location: loc NOT NULL;
    locMin: loc NOT NULL;
    locMax: loc NOT NULL;

  Services
   begin(Location:loc, LocMAx:loc, LocMin:loc)
             Valuations
              [begin(Location,LocMax, LocMin)]
                 location:=Location & locMax:=LocMax &

locMin:=LocMin;
     checkLocation( input Location:loc, output checkC:bool)
       Valuations
```

```
        [checkLocation(input Location:loc,output checkC:bool)]
             {Location>locMin & Location<locMax}

[checkLocation(Location,checkC)]checkC:=true;
             {Location<locMin & Location>locMax}

[checkLocation(Location,checkC)]checkC:=false;

    Protocols
        DISTRIBUTE :
        DISTRIBUTE  ≡  begin.DISTRIBUTE1;
        DISTRIBUTE1≡ end+checkLocation;

  End_Distribution Aspect D;


  Replication Aspect RP4

    Services
     replicate(newlocation:loc)

    Protocols
     REPLICATEP ≡ begin. REPLICATEP1;
     REPLICATEP1 ≡  replicate + END;

  End_Replication Aspect RP4

  Component_type Account
    Port
              ….
    End_Port;

    Import D: Distribution Aspect
    Import RP4: Replication Aspect
    Import FAccount: Functional Aspect

     Weaving
         replicate() after changeAddress();
         checkLocation(Location,Checkc)
                        beforeif(Checkc=true)
                                  replicate(Newlocation);
      End_Weaving;
```

```
End_Component_type Account;
```

4.5 Example: An Architectural Element specifying more than a Distribution Patterns

In this section we present in detail the explanation of the example used several times to describe the patterns. At the same, the example will use two patterns from the above to show how more than a pattern can be applied for an architectural element and use the proposal of Figure 23.

A bank system consists of bank offices, accounts and ATMs. Each one of them is represented in a record of a distributed database. The ATM acts as a server component to the Bank Administrator. The administrator invokes the *howmanytransactions* service which indicates the number of transactions of an ATM. In addition, the ATM acts as a client component to the Account. The ATM requires the *balance* and the *withdraw* services. The required services are in an interface called Operations, specified as follows in PRISMA:

```
Interface IOperations
 Services
 balance();
 withdraw();
End Interface IOperations
```

In this example, the analyst wants the ATM record to move to its server location (the Account) when it requests a certain limit of times the services *balance* and *withdraw*. Also, the analyst needs to move the ATM record to its client location (the Bank Administrator) when the Bank Administrator invokes a certain number of times the *howmanytransactions.* These requirements are applied in order to facilitate the communication of the ATM with its client and server.

In the example **MP.01** and **MP.02** need to be applied to the ATM distribution aspect. In **MP.01 Excess of the arrival rate** is specified. Going back to the **Solution** part of the template of **MP.01**, the first step in specifying the pattern is to calculate the arrival rate. In Figure 5, the *calculateArrivalRate()* service sums 1 to the attribute *arrivalRate* each time the interested service is invoked. In the second and third step, to detect when the arrival rate exceeds a certain limit and to move the server component to the location of the client of the services of the specific service are indicated with the **trigger**. The **trigger** calls a transaction *MOVE* that has to be executed as one unit. This transaction consists in finding the client which most frequently invoked the service and moving to this location. The transaction invokes the service ***out*** *SearchClientLocationMostInvoked(Service)* of the infrastructure which takes the name of the Service and returns the client location that most frequently invokes the service. The client location is returned with the service ***in*** *SearchClientLocationMostInvoked(CL)*. The value of the argument is set as the value of the client location by the *valuation*.

In **MP.02 Excess of the request rate** is specified. Going back to the **Solution** part of the template of **MP.02**, the first step in specifying the pattern is to calculate the request rate of the interested services. The *calculateRequestRate()* service sums 1 to the attribute *RequestRate* each time the interested services are invoked. In the second step, to move the component when the request rate exceeds the limit is indicated with a **trigger** similar to the explanation of the **MP.01.** and using also services of the infrastructure to know the location of the server of the interface.

The distribution aspect uses the two patterns **MP.01** and **MP.02**.

```
Distribution Aspect MP1_MP2

Attributes
location: loc NOT NULL;
arrivalRate : nat(0);
arrivalRateLimit : nat NOT NULL;
clientL : loc;
requestRate : nat(0);
requestRateLimit : nat NOT NULL;
serverL : loc;


Services
    begin(Location:loc, ArrivalRateLimit:nat)
```

```
    Valuation
     [begin(Location,ArrivalRateLimit)]
      location:=Location &
                  arrivalRateLimit:=ArrivalRateLimit;
move(newlocation:loc)
  Valuations
   [move(newlocation)] location:=newlocation;
calculateArrivalRate();
  Valuations
   arrivalRate:=arrivalRate+1;
out SearchClientLocationMostInvoked(Services);
in SearchClientLocationMostInvoked(CL)
  Valuations
   [SearchClientLocationMostInvoked(CL)] clientL:=CL;
calculateRequestRate()
  Valuations
   [calculateRequestRate()] requestRate:= requestRate +1;
 in SearchServerLocationMostReq(SL)
  Valuations
      [SearchServerLocationMostReq(SL)] serverL:=SL;

 Triggers
  MOVEC( Services, CL, newlocation)  when
           {arrivalRate>=arrivalRateLimit};
  MOVES(Services, SL, newlocation) when
           {requestRate >=requestRateLimit};

 operations
  MOVES (Services, SL, newlocation) transaction:
  MOVE0 = out SearchServerLocationMostReq(Services).MOVE1;
  MOVE1 = in SearchClientLocationMostReq(SL).MOVE2;
  MOVE2 = move(serverL);

  MOVEC( Services, CL, newlocation) transaction:
  MOVE0 = out SearchClientLocationMostInvoked(Services).MOVE1;
  MOVE1 = in SearchServerLocationMostInvoked(CL).MOVE2;
  MOVE2 = move(clientL);


Protocols

  MBLTYP ≡begin. MBLTYP1;
```

```
        MBLTYP1 ≡  MOVE + END;
    End_Distribution Aspect MP1;
```

The component ATM is specified in the following. The component imports the distribution aspect which uses the patterns **MP.01.** and **MP.02.** Then the distribution aspect has to be synchronized with the functionality of the component by the *weaving*. The *calculateArrivalRateService()* (which is a service from the distribution aspect) is activated ***after*** the service **in** *howmanytransactions*() is activated(which are services of the functionality of the component). In this way, the distribution aspect is connected with the functionality of the server-side of the component.

The *calculateRequestRate()* (which is a service from the distribution aspect) is activated ***after*** the service **Out** balance and **Out** *withdraw()* are activated(which are services of the functionality of the component and form the interface Ioperations). In this way, the distribution aspect is connected with the functionality of the client-side of the component.

```
Component_type ATM
     Port
                ….
     End_Port;

     Import MP1_MP2: Distribution Aspect
     Import ATMFunc: Functional Aspect;
     Weaving

            calculateArrivalRate() after
                                        in howmanytransactions();
            calculateRequestRate() after Out withdraw();
            calculateRequestRate() after Out balance();


     End_Weaving;

End_Component_type ATM;
```

From the previous specification of the bank system example, note how the patterns can be highly reused through the aspects. Any component can simply import the distribution aspect that imports different patterns and adapt it to the functionality of the component. Thus, we achieve a high level of pattern reusability and a high level of reusability of the distribution aspect in different components.

4.6 Summary and Conclusions

On one hand, distributed systems are actually a necessity and not an additional accessory. However, many problems are encountered in distributed systems which may cause the collapse of the system if no solution is given. This chapter identifies many common problems in distributed systems giving a form of solving them through the mobility and replication of the architectural elements that compose the systems.

The PRISMA distribution model takes into account these problems and solutions by organizing them into patterns in which the solutions to the problems can be easily reused. Therefore, the patterns have been specified using the PRISMA ADL. Also, the PRISMA infrastructure has to provide extra services to implement the patterns.

In the future, once a tool is developed for the PRISMA framework, the patterns will be included in the tool. In the case the analyst needs to apply one the identified patterns, the analyst only has to choose the pattern name needed as an option provided by the tool. The tool will have the patterns stored in a repository and automatically use the specification of the pattern solution. The analyst can visually see the specification and can manually adapt the pattern to necessities of the situation.

CHAPTER 5. GRAPHICAL NOTATION FOR DISTRIBUTION IN PRISMA

5.1 Introduction

"Modelling is the future" were the words of Bill Gates on the 29th of March, 2004 when the interviewer of eweek [48] asked him about modelling. In this interview, Bill Gates gave his vision about the future and the improvements gained by modelling such as customizing software visually with less written code. The tendency of modelling is not only Microsoft's vision for the improvement of software development but also many other companies such as IBM.

Microsoft is promising a great success for their graphical design tools that will appear as part of Visual Studio 2005 [60]. They provide a UML-like class diagram that is capable of reflecting all the language features of Visual Basic .NET and C# in the Microsoft .NET environment. They offer the ability, once you create your diagrams and once the code is generated, to navigate from the diagrams to code and vice-versa to ensure that the system is functioning accurately.

As Visual studio 2005 is not in the market yet, in this work OMG's (Object Management Group) UML [69] is going to be used to represent the graphical notation of our ADL. This is chosen due to the fact that UML up-to-date is the most extended modelling language and many visual design environments incorporate UML such as IBM Rational Rose [62], Poseidon[58] or Microsoft's Visio.

In this chapter, we introduce UML and the mechanisms of extending it. Then, we explain the elements of the basic PRISMA UML profile defined in [53] that are necessary for defining the extension and incorporating the concepts related with distribution. Finally, the UML profile of PRISMA for distribution is presented.

5.2 Bases to define a UML Profile

The standard omg modelling language UML 2.0 [69] is a general purpose language that can be specialized for different domains using extensibility mechanisms. Profiling is the standard, built in mechanism in UML. The intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaption is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile.

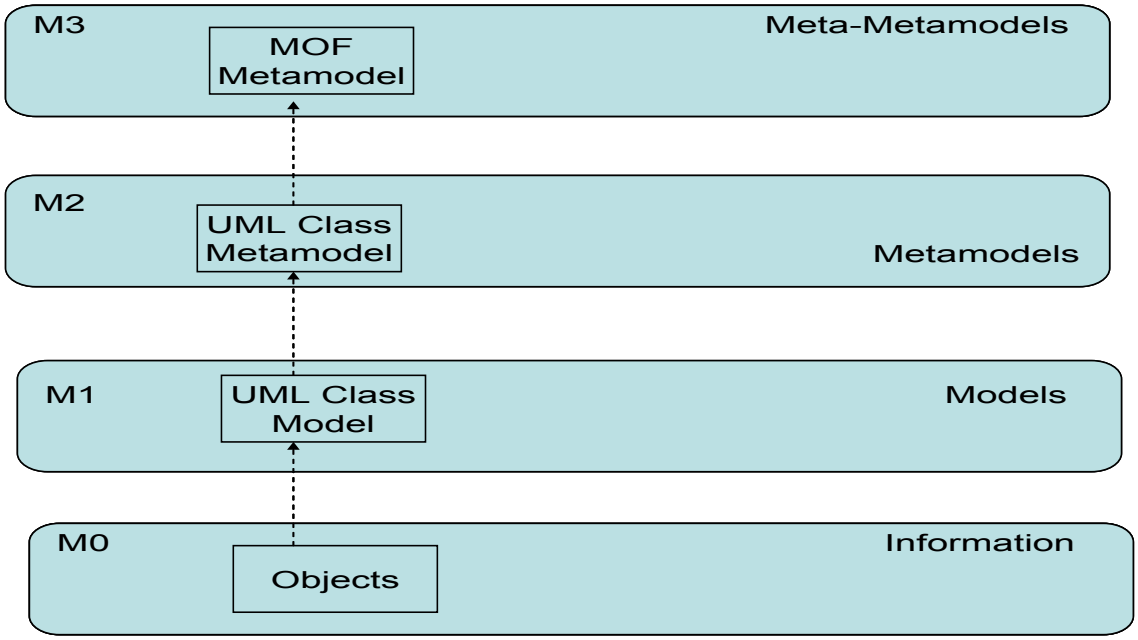


Figure 24 4-Level Metamodeling Framework

The UML metamodel is capable of having such extensions due to the fact that is based on a 4-level metamodelling framework (see Figure 24) instantiated from MOF: the meta-metamodel M3 is where the MOF metamodel is defined, the metamodels M2 where the UML metamodel is defined as an instance of the MOF metamodel, the models M1 and the instances.

UML has two forms of extension: the Heavyweight UML extension and Lightweight UML extension. The Heavyweight UML extension extends the UML metamodel directly through MOF mechanisms for example by defining new subclasses in the metamodel. The Lightweight UML extension is to extend UML through a profile. In our work the lightweight extension is chosen in order to incorporate the profile to a CASE tool.

A profile consists of stereotypes, meta-attributes (tagged values in UML 1.5) and constraints. A stereotype is used to define specialized model elements on a core UML model

element (or other stereotypes) by defining supplemental semantics. The meta-attributes are individual modifiers with user-defined semantics which gives additional information that is required to use the model. The meta-attributes are typed with a standard data type. The constraints are restrictions attached to the stereotypes to make the stereotype different from the UML model element.

5.3 Background of the PRISMA Profile

In this section, the basic stereotype <<aspect>> of the PRISMA Profile which is necessary to this work is going to be presented. This is essential due to the fact that the concepts of the PRISMA distribution model are going to be incorporated to the profile by extending the <<aspect>> defined in the works of Perez [53]. Therefore it is essential to have an understanding of the basic PRISMA profile concepts.

The stereotype <<aspect>> is as follows:

- Aspect

The base class of the <<aspect>> stereotype which is associated to the aspect concept of the PRISMA model is the UML metaclass *class* (see figura 35). This is due to the fact that the aspect as the metaclass *class* is described by a template with attributes, operations, methods and semantics. In addition, they have in commun that a *class* can use a set of interfaces for specifying a set of operations.

Stereotype	Base Class	Parent	Meta-attributes	Description
<<Aspect>>	Class	N/A	None	An aspect specifies the structure and behaviour o fan architectural element from a determined concern.

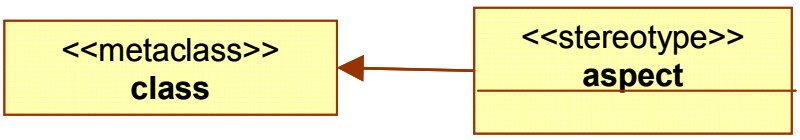


Figure 25 The stereotype <<aspect>> extends class

5.4 The UML Profile for the distribution Model

This section includes the concepts of the distribution aspect and replication aspect to the PRISMA profile. In the following, the inclusion of both aspects is explained in detail.

5.4.1 Distribution Aspect

To define the necessary primitives of the distribution aspect a stereotype called <<distribution aspect>> has been defined.

The base class of the stereotype <<distribution aspect>> (see Figure 26) which is associated to the concept of the distribution aspect of PRISMA is the metaclass UML *class*. In addition the parent of the stereotype <<distribution aspect>> is the stereotype aspect <<aspect>> of the PRISMA model. This is due to the fact that a PRISMA distribution aspect consists of the same parts of a PRISMA aspect with *attributes*, *services*, *valuations*, *constraints*, *preconditions*, *transactions* and *protocols*. Some constraints are specified for the distribution aspect. The distribution aspect must have a PRISMA attribute called *location.* In addition, the distribution aspect has a meta-attribute called mobility. If the meta-attribute mobility has a value not equal to "0" then the distribution aspect must have a PRISMA service called *move*. The explanation is described in Table 8.

Table 8 The <<distribution aspect>> stereotype

Stereotype	Base Class	Parent	Meta-attribute	Description	Constraints
<<Distribution Aspect>>	Class	<<aspect>>	mobility	The <<Distribution Aspect>> is an aspect that is used to model PRISMA architectural elements that are distributed.	1) A distribution aspect must have a location PRISMA attribute. OCL Context at.oclIsKindOf(Distribution Aspect) implies at.attributes ➔ forAll(o |at.contect.attributes➔ exists (a |a.attribute.name = location)) 2)A service move shall

Stereotype	Base Class	Parent	Meta-attribute	Description	Constraints
					not exist if mobility=0. OCL Context at.oclIsKindOf(Distribution Aspect) & at.oclIsTypeOf (mobility) & mobility=0 implies at.operation ➔ forAll (o | at.contect.operation ➔ does not exists (op | o.operation.name =move))

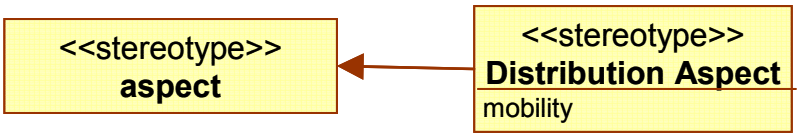


Figure 26 The <<Distribution Aspect>> stereotype extends <<aspect>>

The mobility meta-attribute of the stereotype <<distribution aspect>> is defined to indicate when a distribution aspect has a mobile behaviour. The value of the meta-attribute can have the values 0,1,2 and 3. When the value of mobility is 0, a service move in the distribution aspect does not exist. When the value of mobility is 1 then the distribution aspect has an *in move*. When the value is 2, an *out move* is specified and when the value is 3 both *in move* and *out move* exist in the template of the distribution aspect. This is indicated in detail in Table 9.

Table 9 The mobility meta attribute

Meta-Attribute	Description	Constraints
mobility	The mobility meta-attribute is used when the architectural element is either mobile=”1” or affects on the mobility of the others= “2” or both =”3”.	1) If the mobility meta-attribute is set to 1 then the move operation will be an “in”. OCL Context at.oclIsKindOf(Distribution Aspect) & at.oclIsTypeOf (mobility) & mobility=1 implies at.operation ➔ forAll (o | at.contect.operation ➔ exists (op | o.operation.name =in move)) 2) If the mobility meta-attribute is set to 2 then the move operation will be an “out”. OCL Context at.oclIsKindOf(Distribution Aspect) & at.oclIsTypeOf (mobility) & mobility=2 implies at.operation ➔ forAll (o | at.contect.operation ➔ exists (op | o.operation.name =out move)) 3) If the mobility meta-attribute is set to 3 then the move operation will be an “in” and an “out”. OCL Context at.oclIsKindOf(Distribution Aspect) & at.oclIsTypeOf (mobility) & mobility=3 implies at.operation ➔ forAll (o | at.contect.operation ➔ exists (op | o.operation.name =in move) & (op|o.operation.name = out move))

5.4.2 Replication Aspect

To define the necessary primitives of the replication aspect three stereotypes have been defined: <<replication aspect>>.

The base class of the stereotype <<replication aspect>> (see Figure 26) which is associated to the concept of the replication aspect of PRISMA is the metaclass UML *class*. In addition the parent of the stereotype <<replication aspect>> is the stereotype aspect <<aspect>> of the PRISMA model. This is due to the fact that a PRISMA replication aspect consists of the same parts of a PRISMA aspect with *attributes*, *services*, *valuations*, *constraints*, *preconditions*, *transactions* and *protocols*. Some constraints are specified for the replication aspect. The replication aspect must have a PRISMA service called *replicate.* In addition, the replication aspect has a meta-attribute called replicable to specify the replicable behaviour. The explanation is described in Table 10 and in Figure 27 the stereotype <<replication aspect>> is shown graphically.

Table 10 The <<replication aspect>> stereotype represented in a tabular way.

Stereotype	Base Class	Parent	Meta-Attributes	Description	Constraints
<<Replication Aspect>>	Class	<<aspect>>	replicable	The <<Replication Aspect>> is an aspect that is used to model PRISMA architectural elements that can be replicated.	1) A replication aspect should not have a “0” value for the meta-attribute replicable. OCL Context at.oclIsKindOf(Replication Aspect) & at.oclIsTypeOf (replicable) implies ➔ replicable!=0 2) A service replicate should be defined for each value of replicable.

Figure 27 The stereotype <<replication aspect>> extends the stereotype <<aspect>>

The replicable meta-attribute of the stereotype <<replication aspect>> is defined to indicate when a replication aspect has a replicable behaviour. The value of the meta-attribute can have the values 1,2 and 3. When the value of replicable is 1 then the replication aspect has an *in replicate*. When the value is 2, an *out replicate* is specified and when the value is 3 both *in replicate* and *out replicate* exist in the template of the replication aspect. This is indicated in detail in Table 11.

Table 11 the replicable meta-attribute

Meta-Attributes	Description	Constraints
replicable	The replicable meta-attribute is used when the architectural element can either be replicated=”1” or can replicate an external architectural element = “2” or both =”3”.	1) If the replicable meta-attribute value is set to 1 then the replicate operation will be an “in”. OCL Context at.oclIsKindOf(Replication Aspect) & at.oclIsTypeOf (replicable) & replicable=1 implies at.operation ➔ forAll (o | at.contect.operation ➔ exists (op | o.operation.name =in replicate))

Meta-Attributes	Description	Constraints
		2) If the replicable meta-attribute value is set to 2 then the replicate operation will be an “out”. OCL Context at.oclIsKindOf(Replication Aspect) & at.oclIsTypeOf (replicable) & replicable=1 implies at.operation ➔ forAll (o | at.contect.operation ➔ exists (op | o.operation.name =out replicate)) 3) If the replicable meta-attribute value is set to 3 then the replicate operation will be an “in” and an “out”. OCL Context at.oclIsKindOf(Replication Aspect) & at.oclIsTypeOf (replicable) & replicable=3 implies at.operation ➔ forAll (o | at.contect.operation ➔ exists (op | o.operation.name =in replicate) & (op|o.operation.name = out replicate))

5.5 Summary and Conclusions

This chapter has presented a first step for defining a graphical notation for the PRISMA distribution model. Two UML stereotypes have been defined: the <<distribution aspect>> stereotype and the <<replication aspect>>. The stereotypes have their proper meta-attributes that determine some of the services the aspects can provide. To model the aspects the UML class diagram is going to be used.

In the future, we would like to use the UML deployment diagrams to represent the deployment of the instances of the architectural elements in different locations. The graphical notation will be used by the analysts to model the distributed software architectures using a developed CASE tool. In this way, the analysts do not have to learn the PRISMA language instead they can easily use visual diagrams.

CHAPTER 6. CONCLUSIONS AND FURTHER WORK

As a conclusion, this chapter sums up the main contributions of the work, scientific publications and gives some suggestions for further work.

6.1 Summary of the Contributions

The research of this work has been motivated by the observation that although Architecture Description Languages (ADL's) were initially developed in order to describe software architectures of distributed systems, the ADL's do not support constructors for defining distributed, mobile and replicable software architectures. Therefore, the most original contribution of the work is the definition of the primitives to describe software architectures of distributed, mobile and replicable software systems at a conceptual level.

The PRISMA architectural model approach which integrates the aspect oriented software development and component based software development has been the context of this research. This work has included the necessary primitives to enable PRISMA to become a model to describe complex software architectures of dynamic distributed systems. A distribution that describes the dynamic location of the architectural elements has been included with its properties to the PRISMA metamodel. In addition, a replication aspect has also been added to the set of possible aspects of a PRISMA architectural model by also adding its characteristics to the PRISMA metamodel. Also, the attachments and binding links relations that are essential to define systems and architectural models are extended to become location aware and become the communication channels that enable remote calls between the distributed architectural elements.

The primitives necessary to describe distributed, mobile and replicable architectural elements and architectural models have been incorporated to the PRISMA ADL at the two levels of abstraction: at the type definition level and at the configuration level. To use PRISMA

ADL separated into these two levels of abstraction provides benefits in specifying distributed systems. Thus at the type definition level the distribution, mobile and replicable properties can be reused due to the storage of the aspect types in the PRISMA libraries and due to the externalization of the weaving from the distribution and replication aspects. In addition, the configuration level enables to give the specific properties of the distributed systems at execution time depending on the topology of the software architecture.

Next, we have presented some distribution patterns that provide the analyzer with some guidelines to apply to its distributed systems using the PRISMA model. The patterns describe situations in which the mobility and replication of the architectural elements are recommended in order to prevent fault tolerance problems, to adapt to new changes to the requirements and to have an efficient performance at run time. The patterns have been structured in a template to facilitate their appliance and reusability in the different participants of the patterns.

Finally, it has been worked on a UML profile to present all the introduced concepts. This is to permit the modelling of the PRISMA architectural models with the necessary primitives that enable it to describe software architectures of distributed software systems graphically.

As a result of this work, the PRISMA architectutral model becomes a framework to describe distributed systems at an analysis and design level (conceptual level).

6.2 Related Publications

This work is based on a set of research publications. The following international and national publications were obtained:

- Jennifer Pérez, **Nour Ali,** Jose A. Carsí, Isidro Ramos, Elena Navarro. *Designing Software Architectures with an Aspect-Oriented Language,* Journal on Aspect Orientation, ISSN 1548-3851. (To appear)
- **Ali, N.H.,** Perez J., Ramos I. “High Level Specification of Distributed and Mobile Information Systems”, Proceedings of Second International Symposium on Innovation in Information & Communication Technology ISIICT 2004, Amman, Jordan, 21-22 April, 2004.
- **Ali, N.H.,** Silva J., Jaen, J.,Ramos I., Carsi, J.A., and Perez J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proceedings of 15th IASTED, Parallel and Distributed Systems, Acta Press (Marina del Rey, C.A., USA, November 2003), p 820-826.
- **Nour H. Ali,** Jennifer Pérez, José Ángel Carsi, Isidro Ramos. Aspect Reusability in Software Architectures. Poster in the 8th International Conference of Software Reuse(ICSR), Madrid , Spain, Julio, 2004.

- Perez J., **Hussein N.,** Ramos I., Pastor J.A., Sanchez P., Álvarez B. *Desarrollo de un Sistema de Teleoperación utilizando el enfoque PRISMA.* ISBN.: 84-688-3836-5, Actas VIII Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Alicante, noviembre 2003, PÁGINAS: 411-420.(short paper)
- **Ali, N.**, Carsi, J.A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Malaga. (Accepted as short paper)
- Silva J., **Hussein N.,** Carsi J.A., Ramos I. El aspecto de distribución de PRISMA ISBN.: 84-688-3836-5, Actas VIII Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Alicante, noviembre 2003, paginas 127-136.(short paper)
- Perez J., **Ali N H**., Ramos I., Carsi, J.A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes, AOSD workshop in collaboration with VIII Jornadas de Ingeniería del Software y Bases de Datos, Universidad de Extremadura Departamento de Informática Informe Técnico nº 20/2003, Alicante, November, 2003 p 27-36.
- **Nour H. Ali**, Josep Silva, Javier Jaén, Isidro Ramos, Jose A. Carsí, Jennifer Pérez. Distribution Patterns in Aspect-Oriented Component- Based Software Architectures Actas IV jornadas de trabajo de Distributed Objects, Languages, Methods and Environments, DOLMEN, Alicante, Noviembre 2003. P: 74-80.

6.3 Further Work

In the near future the research is going to be concentrated on the appliance of the proposed model languages and patterns to case studies, specifically in the area of the teleoperation systems.

A fundamental complementary for the specification of the conceptual distribution patterns to the PRISMA architectural model is the use of dependency analysis techniques and tools. These allow the fact to identify the interdependent elements of the architectural model that can be influenced when an architectural elements.

Another important task is to be done, is to identify and implement transformation patterns from the PRISMA model to different distributed platforms such as .Net Remoting and CORBA. These patterns are necessary in order to build the model compiler.

In addition, a case tool is going to be developed to incorporate textual and graphical notation of PRISMA and patterns to enable develop the distributed applications in different platforms and programming languages.

BIBLIOGRAPHY

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

[2] Aldawud, O., Elrad, T., Bader, A. *A UML Profile for Aspect Oriented Modeling,* Workshop on Aspect Oriented Programming, OOPSLA 2001.

[3] Alexander C. Notes on the synthesis of Form. Harvard University Press, 1964.

[4] Ali, N.H., Perez J., Ramos I. "High Level Specification of Distributed and Mobile Information Systems", Proceedings of Second International Symposium on Innovation in Information & Communication Technology ISIICT 2004, Amman, Jordan, 21-22 April, 2004.

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

[6] Allen, R.J. A Formal Approach to software Architecture. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.

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

[8] Balzer, R. Software Tech. in the 1990's: Using a new Paradigm", IEEE, 1983.

[9] Barbacci, M.R., Doubleday, D., Weinstock, C. B. and Lichota, R. W. *DURRA: An Integrated Approach to Software Specification, Modeling and Rapid Prototyping.* Technical Report CMU/SEI-91-TR-21, Software Engineering Institute (SEI), septembre 1991

[10] Brito, I., Moreira, A. Towards a Composition Process for Aspect-Oriented Requirements. Early Aspects 2003: Aspect-Oriented Requirements Engineering

and Architecture Design, workshop of the 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, 17 March 2003.

[11] H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing, "*Extending Activity Diagrams to Model Mobile Systems*" In Proceedings of International Conference NetObjectDay (NODe) '02, LNCS 2591 Springer, Germany, Oct 2002, pp. 278-293.

[12] Cardelli, L. and Gordon, A.D. Types for Mobile Ambients. Proceedings of the 26th ACM Symposium on Principles of Programming Languages, 1999. pp 79-92.

[13] Carsí J.A., "*OASIS como marco conceptual para la evolución del software*", Tesis doctoral, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, ISBN 84-699-3372-8.

[14] Ciancarini, P., Mascolo, C. Software Architecture and Mobility**.** Proc. 3rd Int. Software Architecture Workshop (ISAW-3), November, 1998.

[15] Clements, P., Bachmann, F., Bass L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. Documenting Software Architectures: Views and Beyond. Addison Wesley, 2002.

[16] Components- The Future of Software Engineering? The SI-SE 10th Anniversary Symposium. March 18-19, 2004, Zurich.

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

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

[19] DeRemer, F. and Hans, H.K. Programming-in-the-large versus programming-in-the-small. IEEE Transactions on Software Engineering, SE-2(2), June, 1976, p 80-86.

[20] Eliëns, A. Principles of Object-Oriented Software Development**.** Addison-Wesley (2000), ISBN 0-201-39856-7.

[21] Formal Methods for Software architectures. Third International School on Formal Methods for Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003 Bertinoro, Italy, September 2003, Advanced Lectures. Marco Bernardo and Paola Inverardi(Eds), ISBN 3-540-20083-5, ISSN 0302-9743.

[22] Gamma, E., Helm, R., Johnson, R. and Vlissides J. et al., Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley, 1995. ISBN 0201633612.

[23] Grau A., "*Computer-Aided validation of formal conceptual models*", PhD. Thesis Institute for Software, Information Systems Group, Technical University of Braunschweig, March 2001.

[24] Gruhn, V., Schafer, C. An Architecture Description Language for Mobile Distributed Systems. Software Architecture: First European Workshop, EWSA 2004, St Andrews, UK, Springer-Verlag Heidelberg ISSN: 0302-9743, ISBN: 3-540-22000-3, May 21-22, 2004, p 212-218.

[25] Grundy, J., Aspect-Oriented requirements Engineering for Component-based Software Systems. In 4th IEEE int'1 Symp. on RE, 1999, IEEE CS Press,p 84-91.

[26] Harel D. (1984); "Dynamic Logic"; *in Handbook of Philosphical Logic II,* editors D.M. Gabbay, F. Guenthner; pp.497-694, Reidel.

[27] Harel D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3):231-274, 1987.

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

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

[30] Kaveh, N.,Emmerich, W. "Validating Distributed Object and Component Designs", Third International School on Formal Methods 2003, LNCS, Bertinoro, Italy, September 2003, pp. 63-91.

[31] Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palm, J., Griswold, W.G. "*An Overview of AspectJ"*, Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 2001.

[32] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J. "*Aspect-Oriented Programming*". In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[33] Letelier P., Sánchez P., Ramos I., Pastor O. OASIS 3.0: "*Un enfoque formal para el modelado conceptual orientado a objeto*". Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721- 663-0, 1998.

[34] Lime Team. Lime Web page, http://lime.sourceforge.net/

[35] Lopes, C. D. **"**A Language Framework for Distributed Computing" Ph.D. Dissertation, College of Computer Science, Northeastern University, Boston, 1997.

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

[37] Luckham, D. C. and Vera, J. An Event- Based Architecture Definition Language. IEEE Transactions on Software Engineering, 21(9):717–734, September, 1995.

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

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

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

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

[42] Medvidovic N., Taylor R.N., "*A classification and Comparison Framework for Software Architecture Description Languages",* IEEE Transactions of SW Engineering, Vol. 26, nº 1, January 2000.

[43] Microsoft .Net Remoting : A Technical Overview, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

[44] Molina, P.J. Especificación de Interfaz de Usuario: De los Requisitos a la generación automàtica. Tesis Doctoral. Marzo de 2003.

[45] Murphy, A.L., Picco G.P. and Roman, G.-C. LIME: A Middleware for Physical and Logical Mobility. In F. Golshani, P. Dasgupta, and W. zhao, editors, Proc of the 21st Int. Conf. On distributed Computing Systems (ICDCS-21), May 20001, p 524-533.

[46] F. Muscutariu and M.-P. Gervais (2001), *On the Modeling of Mobile Agent-Based Systems*, In Proceedings of 3rd IEEE/ACM International Workshop on Mobile Agents for Telecommunication Applications (MATA'01), Montreal, Canada, Lecture Notes in Computer Science n°2164, Springer Verlag, pp219-234

[47] Nana, L., Kermarrec Y., and Pautet, L. GNATDIST: a configuration language for distributed Ada 95 applications: *ACM Tri Ada conference*, Philadelphia, USA, 1996, 63-72.

[48] Taft D.K. What is Bill Gates Thinking? Eweek : Enterprise News and Reviews. 30 March 2004. http://www.eweek.com/article2/0,1759,1556075,00.asp

[49] Virginia C. de Paula, G. R. Ribeiro Justo and P. R. F. Cunha: Specifying Dynamic Distributed Software Architectures, XII Brazilian Symposium on Software Engineering, BCS Press, October, 1998.

[50] Oblog – Object Logic. "*Oblog Software*", Oblog Software S.A. Lisboa, Portugal. URL: http://www.info.fundp.ac.be/~phe/2rare/spd.html.

[51] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*, Procc. of 9th International Conference, CaiSE97, Barcelona, 1997.

[52] Perez J., Ali N H., Ramos I., Carsi, J.A. PRISMA: Arquitecturas Software Orientadas a Aspectos y Basadas en Componentes, AOSD workshop in collaboration with VIII Jornadas de Ingeniería del Software y Bases de Datos, Universidad de Extremadura Departamento de Informática Informe Técnico nº 20/2003,Alicante, November,2003 p 27-36.

[53] Pérez, J., Ramos, I. Oasis como Soporte Formal para la Definición de Modelo Hipermedia Dinámicos, Distribuidos y Evolutivos , Informe Técnico DSIC-II/22/03, Universidad Politécnica de Valencia, Octubre 2003

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

[55] Pérez, J., Ramos, I., Carsí Cubel, J.A. Compilador para la Generación Automática del Metanivel de una Especificación mediante la Reificación de Propiedades del Nivel Base , Informe Técnico DSIC-II/23/03 Universidad Politécnica de Valencia, octubre 2003

[56] 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, 2002 , Enschede, The Netherlands, Pages: 134 – 140

[57] Popovici, A., Gross, T. and Alonso, G. Dynamic Weaving for Aspect-Oriented Programming. In proceedings of the 1st international conference on Aspect-oriented software development.(Enschede The Netherlands, April 2002)

[58] Poseidon website: http://www.gentleware.com/

[59] Rammer, I. Advanced .Net Remoting. Apress; 1 edition (April 5, 2002), ISBN: 1590590252.

[60] Randell,B.A., and Lhotka, R. VISUAL STUDIO 2005 Bridge the Gap Between Development and Operations with Whitehorse. MSDN Magazine The Microsoft Journal for Developers. http://msdn.microsoft.com/msdnmag/issues/04/07/whitehorse/default.aspx

[61] Rashid, A., Moreira, A., Araujo, J. Modularisation and composition of Aspectual Requirements. In proceedings of the 2nd international conference on Aspect-oriented software development(Boston Massachusetts, March 2003)

[62] Rational Rose: http://www-306.ibm.com/software/awdtools/developer/rosexde/

[63] Riehle D. and Zullighoven, H. Understanding and Using Patterns in Software Development, 1996.

[64] Suzuki, J., and Yamamoto, Y. Extending UML with Aspects: Aspect Support in the design phase. 3rd Workshop on Aspect-oriented Programming in European Conference on Object Oriented Programming(ECOOP), 1999,299-300.

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

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

[67] Suvee, 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(Boston Massachusetts, March 2003)

[68] Szyperski, C., *Component software: beyond object-oriented programming*, (New York, USA: ACM Press and Addison Wesley, 2002).

[69] Unified Modelling Language UML 2.0: http://www.omg.org/technology/uml/