AMBIENT-PRISMA: Ambients in Aspect-Oriented Software Architectures

Nour Ali Irshaid

Department of Information Systems and Computation
Polytechnic University of Valencia

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science

Supervisor: Prof. Isidro Ramos Salavert

October 2007

ABSTRACT

Nowadays, distributed systems have become part of our daily lives thanks to the success of Internet. Mobility is a paradigm which exploits distributed systems, where mobile components can freely move and interact with other distributed components. However, the development of software systems with distribution and mobility characteristics is a difficult task. Technology-independent models that use software engineering techniques are needed for facilitating the development of these kinds of systems.

Aspect-Oriented Software Development (AOSD), and software architecture are techniques that promise separation of concerns which can improve the development of systems. Distribution and mobility are crosscutting concerns of a software architecture and can be dealt as aspects. Software architectures also need new primitives for describing the topology of a distributed system and mobility behaviours. Ambient Calculus (AC) is a formalism that provides primitives to describe distribution and mobility characteristics in an abstract way. It introduces a concept called ambient which is a bounded place where computation happens.

This thesis presents a framework called Ambient-PRISMA for describing and developing distributed and mobile software systems in an abstract way. Ambient-PRISMA enriches an aspect-oriented software architecture approach called PRISMA with concepts of AC. This enrichment is performed by extending the PRISMA metamodel, Aspect-Oriented Architecture Description Language (AOADL), formalization, CASE Tool and methodology. A case study of an electronic Auction System with mobile agents is used throughout the thesis in order to illustrate the work.

The Ambient-PRISMA metamodel defines how models should be correctly built. The AOADL allows the specification of distributed architectural elements and different mobility behaviours by using aspects. In addition, the topology of a

distributed software architecture can be described. The formalization provides Ambient-PRISMA to describe unambiguous specifications. The CASE Tool consists of a Modelling Tool and a middleware. The Modelling Tool allows the specification of Ambient-PRISMA software architectures in a graphical way. The middleware allows Ambient-PRISMA applications to execute on .NET technology. The methodology provides guidelines for an analyst for modelling and developing applications.

RESUMEN

Hoy en día, los sistemas distribuidos forman parte de nuestra vida diaria gracias al éxito de Internet. La movilidad es un paradigma que explota los sistemas distribuidos, donde los componentes móviles pueden moverse y interactuar libremente con otros componentes distribuidos. Sin embargo, el desarrollo de sistemas software con características de distribución y móvilidad es una tarea dificil. Para facilitar el desarrollo de este tipo de sistemas, es necesario proveer modelos independientes de tecnología que utilicen técnicas de la ingeniería del software.

Técnicas que pueden mejorar el desarrollo de sistemas y que soportan la separación de "concerns" son el Desarrollo Software Orientado a Aspectos (DSOA) y la Arquitectura Software. La distribución y la movilidad son "crosscutting concerns" de una arquitectura software y se pueden considerar como aspectos. Las arquitecturas software también necesitan nuevas primitivas que describen la topología de un sistema distribuido y los comportamientos moviles. El Cálculo de Ambientes (CA) es un formalismo que proporciona primitivas para especificar características de distribución y movilidad de forma abstracta. El CA introduce el concepto de ambiente que es un lugar limitado en donde sucede el cómputo.

Esta tesis presenta Ambient-PRISMA, un marco para describir y desarrollar sistemas software distribuidos y móviles de forma abstracta. Ambient-PRISMA enriquece PRISMA, un enfoque de arquitecturas software orientadas a aspectos, con los conceptos del CA. Este enriquecimiento de PRISMA es realizado ampliando su metamodelo, su Lenguaje de Descripción de Arquitecturas Orientado a Aspectos (LDAOA), la formalización, la herramienta CASE y la metodología. Además, se utiliza un caso de estudio de un sistema de una subasta electrónica con agentes móviles a lo largo de la tesis para ilustrar el trabajo.

El metamodelo de Ambient-PRISMA define cómo sus modelos deben ser construidos correctamente. El LDAOA permite la especificación de elementos

arquitectónicos distribuidos y de diversos comportamientos de movilidad usando aspectos. Además, el LDOA posibilita la descripción de la topología de una arquitectura software distribuida. La formalización permite describir Ambient-PRISMA sin ambigüedad. La herramienta CASE esta compuesta de una herramienta de modelado y un middleware. La herramienta de modelado permite la especificación de arquitecturas software de Ambient-PRISMA de forma grafica. El middleware permite la ejecución de aplicaciones de Ambient-PRISMA sobre la plataforma .NET. La metodología proporciona al analista pautas para poder modelar y desarrollar aplicaciones.

RESUM

Hui en dia, els sistemes distribuïts formen part de la nostra vida diària gràcies a l'èxit d'Internet. La mobilitat és un paradigma que explota els sistemes distribuïts, en el qual els components mòbils poden moure's i interactuar lliurement amb altres components distribuïts. No obstant això, el desenvolupament de programari amb característiques de distribució i mobilitat és una tasca difícil. Per tal de facilitar el desenvolupament d'aquest tipus de sistemes, cal proveir de models independents de la tecnologia que empren tècniques de l'enginyeria del programari.

El Desenrotllament de Programari Orientat a Aspectes (DPOA), i l'Arquitectura de Software són tècniques que poden millorar el desenvolupament de sistemes i que suporten la separació de "concerns". La distribució i la mobilitat són “crosscutting concerns” d'una arquitectura de programari i es poden considerar com a "aspectes". Les arquitectures de programari també necessiten noves primitives per tal de descriure la topologia d'un sistema distribuït i els comportaments de mobilitat. El Càlcul d'Ambients (CA) és un formalisme que proporciona primitives per a especificar característiques de distribució i mobilitat de forma abstracta. Aquest introdueix el concepte d'ambient que és un lloc delimitat on el còmput es duu a terme.

Aquesta tesi presenta Ambient-PRISMA, un marc de treball per a descriure i desenvolupar sistemes de programari distribuïts i mòbils de forma abstracta. Ambient-PRISMA enriqueix amb el conceptes del CA a PRISMA, un enfocament d'arquitectures de programari orientat a aspectes. Aquest enriquiment a PRISMA és realitzat ampliant el seu meta-model, el seu Llenguatge de Descripció d'Arquitectures Orientat a Aspectes (LDAOA), la formalització, l'eina CASE i la metodologia. A més a més, s'utilitza un cas d'estudi que versa sobre un sistema de subhasta electrònica amb agents mòbils al llarg de la tesi per tal d'il·lustrar el treball desenrotllat.

El meta-model d'Ambient-PRISMA defineix com els seus models han de ser construïts correctament. El LDAOA permet l'especificació d'elements arquitectònics distribuïts i de diversos comportaments de mobilitat mitjançant aspectes. A més, el LDAOA possibilita la descripció de la topologia d'una arquitectura de programari distribuïda. La formalització permet a Ambient-PRISMA descriure especificacions sense ambigüitat. L'eina CASE consisteix d'una eina de modelatge i un "middleware". Per una banda, l'eina de modelatge permet l'especificació d'arquitectures de programari d'Ambient-PRISMA de forma gràfica. Per altra banda, el "middleware" permet a les aplicacions Ambient-PRISMA executar-se sobre la tecnologia .NET. La metodologia proporciona als analistes pautes per a poder modelar i desenvolupar aplicacions.

KEYWORDS

KEYWORDS:

Software Architectures, Aspect-Oriented Software Development (AOSD), Distributed Systems, Mobile Systems, Ambient Calculus, Model Driven Engineering (MDE).

PALABRAS CLAVE:

Arquitecturas Software, Desarrollo de Software Orientado a Aspectos (DSOA), Sistemas Distribuidos, Sistemas Moviles, Calculo de Ambientes, Ingeniería Dirigida por Modelos.

PARAULES CLAU:

Arquitectures de Programari, Desenvolupament de Programari Orientat a Aspectes (DPOA), Sistemes Distribuïts, Sistemes Mòbils, Cálcul d'Ambients, Enginyeria Dirigida per Models.

We Will Return

…

Beloved Palestine! How do I live

Away from your plains and mounds?

The feet of mountains that are dyed with blood

Are calling me

And on the horizon appears the dye

The weeping shores are calling me

And my weeping echoes in the ears of time

The escaping streams are calling me

They are becoming foreign in their land

Your orphan cities are calling me

And your villages and domes

My friends ask me, "Will we meet again?"

"Will we return?"

Yes! We will kiss the bedewed soil

And the red desires are on our lips

Tomorrow, we will return

And the generations will hear

The sound of our footsteps

....

Abdelkarim Al-Karmi (Abu Salma) - 1951

TABLE OF CONTENTS

ABSTRACT 3

RESUMEN 5

RESUM 7

KEYWORDS 9

TABLE OF CONTENTS 13

LIST OF FIGURES 21

LIST OF TABLES 25

PART I. INTRODUCTION 27

CHAPTER 1 29

INTRODUCTION 29

1.1 MOTIVATION AND RATIONALE 30
1.2 OBJECTIVES OF THE THESIS 32
1.3 STRUCTURE OF THE THESIS 33

PART II. STATE OF ART 37

CHAPTER 2 39

DISTRIBUTION AND MOBILITY 39

2.1 INTRODUCTION 39
2.2 DISTRIBUTED SYSTEMS 39
2.3 MOBILE SYSTEMS 41
2.4 MODELLING DISTRIBUTION AND MOBILITY 43
2.4.1 Unified Modelling Language (UML) 44
2.4.2 Process Algebra 45
2.4.3 Other Approaches 47
2.5 CONCLUSIONS 48

CHAPTER 3 51

SOFTWARE ARCHITECTURES FOR DISTRIBUTION AND MOBILITY 51

3.1 INTRODUCTION 51
3.2 BASIC CONCEPTS OF SOFTWARE ARCHITECTURES 52
3.2.1 Architecture Description Languages (ADLs) 53
3.2.2 Components 55
3.2.3 Connectors 56
3.2.4 Configurations 57
3.2.5 Views 58

3.3 ARCHITECTURE DESCRIPTION LANGUAGES FOR DISTRIBUTION AND MOBILITY 58
3.3.1 Darwin 59
3.3.2 C2Sadel 62
3.3.3 Community 64
3.3.4 MobiS 66
3.3.5 LAM Model 66
3.3.6 π-ADL 67
3.3.7 Con Moto 68
3.4 COMPARISON 70
3.5 CONCLUSIONS 74

CHAPTER 4 77

ASPECT-ORIENTED SOFTWARE DEVELOPMENT FOR DISTRIBUTED SYSTEMS 77

4.1 INTRODUCTION 77
4.2 ASPECT-ORIENTED SOFTWARE DEVELOPMENT 79
4.2.1 Crosscutting Concerns 79
4.2.2 Aspect 80
4.2.3 Weaving 80
4.3 ASPECT-ORIENTED SOFTWARE DEVELOPMENT FOR DISTRIBUTED SYSTEMS 82
4.3.1 Implementation 85
4.3.1.1 AspectJ 85
- A Pattern for Distribution Aspects (PaDA) 86
- The Mobility Aspect Pattern 88

4.3.1.2 General Object to EJB Conversion Helper (GOTECH) Framework 89
4.3.1.3 DJCutter 89
4.3.1.4 PROgrammable extenSions of sErvices (PROSE) 90
4.3.1.5 AspectIX 91
4.3.2 Analysis and Design 92
4.3.2.1 The D-Framework 93
4.3.2.2 Composition Filters 96
4.3.2.3 UML All pUrpose Transformer (UMLAUT) 97
4.3.2.4 The Disguises Model Approach 98
4.3.2.5 AWED 100
4.4 CONCLUSIONS 100

PART III. PRELIMINARIES 103

CHAPTER 5 105

PRELIMINARIES 105

5.1 CASE STUDY: AUCTION SYSTEM 105
5.1.1 Customer 106
5.1.2 Procurement 107
5.1.3 Bidder 107
5.1.4 LotOnAuction 108
5.1.5 AuctionHouse 108
5.2 AMBIENT CALCULUS 109

5.3 CONCLUSIONS 110

CHAPTER 6 113

PRISMA 113

6.1 INTRODUCTION 113
6.2 PRISMA MODEL OVERVIEW 114
6.2.1 Architectural Model 119
6.2.2 Interfaces 121
6.2.3 Aspects 122
6.2.3.1 Attributes 123
6.2.3.2 Services 125
6.2.3.3 Valuations 126
6.2.3.4 Preconditions 130
6.2.3.5 Constraints 131
6.2.3.6 Transactions 132
6.2.3.7 Played_Roles 133
6.2.3.8 Protocols 134
6.2.4 Simple Architectural Elements: Components and Connectors 135
6.2.5 Connections 140
6.2.5.1 Attachments 140
6.2.5.2 Bindings 142
6.2.6 Systems 144
6.2.7 Configuration of an Architectural Model 148
6.3 PRISMA METHODOLOGY 149
6.4 PRISMA CASE TOOL 152
6.4.1 PRISMA Metamodel in DSL 154
6.4.2 PRISMA Modelling Tool 155
6.4.3 PRISMANET Middleware 156
6.4.4 PRISMA Model Compiler 159
6.5 CONCLUSIONS 159

PART IV. AMBIENT-PRISMA 163

CHAPTER 7 165

AMBIENT-PRISMA: CHARACTERISTICS AND METAMODEL 165

7.1 INTRODUCTION 165
7.2 CHARACTERISTICS OF AMBIENTS IN AMBIENT-PRISMA 173
7.2.1 Interfaces 173
7.2.1.1 ICall Interface 173
7.2.1.2 ICapability Interface 174
7.2.1.3 IGetLocation Interface 175
7.2.1.4 IRoute Interface 176
7.2.2 Aspects 176
7.2.2.1 ACoordination Coordination Aspect 178
7.2.2.2 MobilityAspect Mobility Aspect 179
7.2.2.3 Distribution Aspect 182
7.2.3 Weavings 184
7.2.4 Ports 185

7.2.5 Attachments *186*
7.2.6 Bindings *190*
7.2.7 Kinds of Ambients *192*
7.3 AMBIENT-PRISMA METAMODEL 194
7.3.1 Connector Package *195*
7.3.2 Ambient Package *196*
7.3.3 KindsOfAmbient Package *201*
7.3.4 Attachments Package *203*
7.3.5 Bindings Package *206*
7.3.6 AmbientPRISMAArchitecture Package *206*
7.3.7 Data Types Package *207*
7.4 CONCLUSIONS 208

CHAPTER 8 211

AMBIENT-PRISMA LANGUAGE 211

8.1 INTRODUCTION 211
8.2 THE TYPE DEFINITION LEVEL 212
8.2.1 Architectural Model syntax *212*
8.2.2 Interfaces *213*
8.2.3 Predefined Interfaces *214*
8.2.3.1 ICall Interface 214
8.2.3.2 ICapability Interface 215
8.2.3.3 IGetLocation Interface 216
8.2.3.4 IRoute Interface 217
8.2.4 Aspects *218*
8.2.4.1 Distribution Aspects of Components, Connectors and Systems 219
8.2.4.2 Triggers 222
8.2.5 Predefined Aspects *224*
8.2.5.1 ACoordination Aspect of an ambient 225
8.2.5.2 MobilityAspect aspect of an ambient 226
8.2.5.3 Distribution Aspects of Ambients 231
8.2.6 Components, Connectors and Systems *235*
8.2.7 Ambients *238*
8.2.8 Attachments *240*
8.3 THE CONFIGURATION LEVEL 241
8.4 CONCLUSIONS 244

CHAPTER 9 247

AMBIENT-PRISMA FORMALIZATION 247

9.1 INTRODUCTION 247
9.2 CHANNEL AMBIENT CALCULUS 248
9.3 FORMALIZATION OF AMBIENT-PRISMA CONCEPTS 251
9.3.1 Simple Architectural Elements *252*
9.3.2 Components *252*
9.3.3 Connectors *253*
9.3.4 Ports *254*
9.3.5 DCapPort Port *257*

9.3.6 Aspects *259*
9.3.6.1 Attributes 260
9.3.6.2 Valuations 261
9.3.6.3 Service Execution 261
9.3.6.4 Protocols 262
9.3.7 Ambients *263*
9.3.8 Attachments *265*
9.3.9 Bindings *271*
9.3.10 Architectural Model Configuration *275*
9.4 CONCLUSIONS 276

PART V. TOOL AND METHODOLOGY 277

CHAPTER 10 279

AMBIENT-PRISMA CASE TOOL 279

10.1 INTRODUCTION 279
10.2 THE MODELLING TOOL FOR AMBIENT-PRISMA 280
10.3 AMBIENT-PRISMANET 285
10.3.1 Ambient-PRISMANET Architecture *286*
10.3.2 Middleware's List Management *288*
10.3.2.1 Detecting Failures and Recoveries of Notifiers *292*
10.3.2.2 A Notifier recovering from a failure *293*
10.3.2.3 The failure of a Leader *294*
10.3.3 Distributed Domain Name Server (DNS) *295*
10.3.4 Distribution Management *298*
10.3.5 Distributed Transactions *300*
10.3.6 Execution Model of Transactions *301*
10.3.7 Transactional Context at an architectural element level *304*
10.3.8 Transactional Context at a distributed architectural level *305*
10.3.9 Ambient-PRISMA Execution Model *309*
10.3.9.1 Ambient Construct *309*
10.3.9.2 The Mobility Aspect *310*
10.3.9.3 The Coordination Aspect *312*
10.3.9.4 The three kinds of Ambients *313*
10.3.9.5 Group Ambients *314*
10.3.9.6 Site Ambients *315*
10.3.9.7 Virtual Ambients *316*
10.3.10 Distributed Communication *317*
10.3.10.1 Extending PRISMA Attachments *318*
10.3.10.2 Initial Configuration of Attachments *320*
10.3.10.3 Reconfiguration of Attachments for Mobility *322*
10.3.11 Mobility through Ambients *327*
10.3.11.1 The Moving service *328*
10.3.11.2 The Accept service *329*
10.4 CONCLUSIONS 331

CHAPTER 11 333

AMBIENT-PRISMA METHODOLOGY ___ 333

11.1 DETECTION OF ARCHITECTURAL ELEMENTS AND ASPECTS ___ 333
11.1.1 Identification of Virtual and Site ambients ___ 334
11.1.2 Identification of Components, Connectors and Systems ___ 334
11.1.3 Identification of Group ambients ___ 335
11.1.4 Identification of distribution and replication aspects ___ 335
11.2 SOFTWARE ARCHITECTURE MODELLING ___ 336
11.2.1 Interfaces ___ 337
11.2.2 Aspects ___ 337
11.2.3 Components, Connectors, and Systems ___ 338
11.2.4 Ambients ___ 339
11.2.5 Instantiation and Configuration ___ 339
11.3 CODE GENERATION ___ 340
11.4 CONCLUSIONS ___ 341

PART VI. CONCLUSIONS AND FURTHER WORKS ___ 343

CHAPTER 12 ___ 345

CONCLUSIONS AND FURTHER WORKS ___ 345

12.1 CONCLUSIONS ___ 345
12.2 FURTHER WORKS ___ 352

BIBLIOGRAPHY ___ 355

APPENDIX A ___ 373

AMBIENT-PRISMA AOADL SYNTAX ___ 373

A.1 ARCHITECTURAL MODEL ___ 373
A.2 INTERFACES ___ 373
A.3 ASPECTS ___ 374
A.4 ATTRIBUTES ___ 375
A.5 SERVICES ___ 375
A.6 PRECONDITIONS ___ 376
A.7 TRANSACTIONS ___ 376
A.8 CONSTRAINTS ___ 377
A.9 TRIGGERS ___ 377
A.10 PLAYED_ROLES ___ 378
A.11 PROTOCOL ___ 378
A.12 PORTS ___ 378
A.13 WEAVINGS ___ 379
A.14 VIRTUAL AMBIENTS ___ 379
A.15 SITE AMBIENTS ___ 380
A.16 GROUP AMBIENTS ___ 380
A.17 COMPONENT ___ 381
A.18 CONNECTORS ___ 381
A.19 ATTACHMENTS ___ 382
A.20 CONFIGURATION ___ 382

APPENDIX B 385

AMBIENT-PRISMA SOFTWARE ARCHITECTURE OF THE AUCTION SYSTEM CASE STUDY 385

B.1 INTERFACES 385
B.2 ASPECTS 386
B.2.1 Distribution Aspect of the Customer component 386
B.2.2 Distribution Aspect of the Procurement component 386
B.2.3 Distribution Aspect of the Bidder component 388
B.2.4 Distribution Aspect of the HostSite ambient 389
B.2.5 Distribution Aspect of the Inmobile elements 390
B.2.6 Distribution Aspect of the Root ambient 390
B.2.7 Distribution Aspect of MobileAmb 391
B.2.8 Functional Aspect of the Customer component 391
B.2.9 Functional Aspect of the Procurement component 393
B.2.10 Functional Aspect of the Bidder component 395
B.2.11 Functional Aspect of the AuctionHouse system 397
B.2.12 Functional Aspect of the LotOnAuction component 398
B.3 AMBIENTS 402
B.4 COMPONENTS 403
B.4.1 Customer 403
B.4.2 Procurement 404
B.4.3 Bidder 405
B.4.4 LotOnAuction 405
B.4.5 WrappAspSystem 406
B.5 SYSTEMS 406
B.5.1 AuctionHouse 406
B.6 CONNECTORS 407
B.6.1 AgentCustCnct 408
B.6.2 AuctionHouseCnct 408
B.6.3 CnctLotAuction 408
B.7 ATTACHMENTS 408
B.8 INITIAL CONFIGURATION 409

ACRONYMS 411

INDEX 413

LIST OF FIGURES

Figure 1. Graphical notation of a composite component pipeline in Darwin taken from [Mag95] _60_
Figure 2. Connectors intermediating between distributed components taken from [Dos99] _63_
Figure 3. A mobile component specified in Community taken from [Lop04] _65_
Figure 4. A structural model in Con Moto taken from [Gru04] _68_
Figure 5. (a) The tangling concerns in the traditional applications (b) the separated concerns in AOSD taken from [Lad03]. _80_
Figure 6. Code to handle a distributed client/server object in .NET Remoting _84_
Figure 7. PaDA's Structure taken from [Soa02a] _87_
Figure 8. Fragments of a distributed object in AspectIX taken from [Hau98] _92_
Figure 9. Run-time Architecture for D's remote objects taken from [Lop97] _94_
Figure 10. Filters in the composition filter model taken from [Ber04] _96_
Figure 11. Structure of the Disguises Model taken from [Her03] _99_
Figure 12. Structure of the distribution aspect taken from [Her03] _99_
Figure 13. A UML deployment diagram with the architectural elements of the Auction System architectural elements _106_
Figure 14. The Textual and Visual Syntax of Ambient Calculus constructs taken from [Car98b] _110_
Figure 15. Applying the enter capability to the ambient n taken from [Car01] _110_
Figure 16. Crosscutting Concerns in PRISMA software architectures taken from [Per06b] _115_
Figure 17.Views of a PRISMA architectural element taken from [Per06b] _116_
Figure 18. Communication between the white box and the black box views taken from [Per06b] _117_
Figure 19. Systems taken from [Per06b] _118_
Figure 20. The package ArchitectureSpecification of the PRISMA metamodel taken from [Per06b] _120_
Figure 21. Syntax of the architectural model in the AOADL _121_
Figure 22. The package Interfaces of the PRISMA metamodel taken from [Per06b] _121_
Figure 23. The package SignatureOfService of the PRISMA metamodel [Per06b] _122_
Figure 24. Specification of the interface ICustProc _122_
Figure 25. The metaclass Aspect of the package Aspects of the PRISMA metamodel taken from [Per06b] _123_
Figure 26. Specification of the header of an aspect with interfaces (ProcurFunct) _123_
Figure 27. Specification of variable attributes of the aspect ProcurFunct aspect _124_
Figure 28. Specification of some services of ProcurFunct aspect _126_
Figure 29. Specification of a valuation of in `changeMaximumBid` *of the BidderFunct aspect* _127_
Figure 30. Specification of a valuation of in/out `searchforlot` *of the AuctFunct aspect* _128_
Figure 31. Specification of a valuation of in/out `searchforlot` *of the ProcurFunct aspect* _130_

Figure 32. Specification of a precondition in the BidderFunct aspect 131
Figure 33. Specification of a constraint in the ProcurFunct aspect 132
Figure 34. Specification of a transaction in the CustFunct aspect 133
Figure 35. Specification of played_roles of ProcurFunct aspect 134
Figure 36. Specification of protocols of ProcurFunct aspect 135
Figure 37. The package ArchitecturalElements of the PRISMA metamodel taken from [Per06b] 136
Figure 38. The package KindsOfArchitecturalElements of the PRISMA 138
Figure 39. The package Connectors of the PRISMA 138
Figure 40. Specification of the Customer component type 139
Figure 41. The package Attachments of the PRISMA metamodel taken from [Per06b] 141
Figure 42. Specification of attachments in the agents auction architectural model 141
Figure 43. AttchCustAuct attachment that connects AuctionHouseCnct connector and Customer component 142
Figure 44. A possible configuration of AttchCustAuc attachment that connects AuctionCnct and Customer instances 142
Figure 45. The package Bindings of the PRISMA metamodel taken from [Per06b] 143
Figure 46. The package Systems of the PRISMA metamodel taken from [Per06b] 144
Figure 47. The Systems package of the PRISMA metamodel taken from [Per06b] 145
Figure 48. Specification of the AuctionHouse system 148
Figure 49. Specification of the Auction Configuration 149
Figure 50. The methodology of the PRISMA approach 150
Figure 51. PRISMA CASE PARTS 153
Figure 52. DSL Tools Framework: Domain Model of PRISMA taken from [Per06b] 155
Figure 53. PRISMA Type Modelling Tool taken from [Per06b] 156
Figure 54. PRISMANET middleware architecure taken from [Per06b] 157
Figure 55. Namespaces of the module PRISMA Execution Model taken from [Per06b] 159
Figure 56. A Customer component and a Procurement component connected through a AgentCustCnct connector 166
Figure 57. The graphical notation of an Ambient-PRISMA ambient 168
Figure 58. A possible Configuration of the Auction Mobile Agent Case Study where the Procurement1 component is in the AuctionSite ambient 170
Figure 59. Procurement1 component in the ClientSite ambient 172
Figure 60. Specification of the ICall Interface 174
Figure 61. Partial specification of the ICapability Interface 175
Figure 62. Specification of the IGetLocation Interface 175
Figure 63. Specification of the IRoute Interface 176
Figure 64. Partial specification of the ACoordination Aspect 179
Figure 65. Partial specification of the MobilityAspect aspect 181
Figure 66. Partial specification of an aspect of kind Distribution 183
Figure 67. A mobile ambient called MobileAmb 183
Figure 68. Predefined weaving between the MobilityAspect aspect and a distribution aspect called ADist 184
Figure 69. Partial specification of ambients ports 186
Figure 70. Procurement1 component in the AuctionSite ambient 187
Figure 71. Procurement1 component in the ClientSite ambient 187

Figure 72. Operation of the attachments associated to the InServicesPort ports and the EServicesPorts of ambients ____ *190*
Figure 73. The architectural elements of AuctionHouse1 system located in the AuctionSite ____ *191*
Figure 74. AuctionHouse1 system architectural elements distributed between the AuctionSite ambient and the LondonSaleRoomSite ambient ____ *192*
Figure 75. Kinds of ambients in a possible network ____ *194*
Figure 76. Connector Package of the Ambient-PRISMA metamodel ____ *196*
Figure 77. Ambient Package of the Ambient-PRISMA metamodel ____ *198*
Figure 78. OCL constraints associated to the Ambient metaclass ____ *200*
Figure 79. The KindsOfAmbients package of the Ambient-PRISMA metamodel ____ *202*
Figure 80. The Attachments package of the Ambient-PRISMA metamodel ____ *204*
Figure 81. OCL constraints for the Attachment metaclass ____ *205*
Figure 82. The Binding package in the Ambient-PRISMA metamodel ____ *206*
Figure 83. The AmbientPRISMAArchitecture package of the Ambient-PRISMA metamodel ____ *207*
Figure 84. The package DataTypes of Ambient-PRISMA ____ *208*
Figure 85. Architectural Model template in Ambient-PRISMA ____ *213*
Figure 86. MobileAgentsModel Architectural Model ____ *213*
Figure 87. IMobility Interface ____ *214*
Figure 88. Specification of the ICall Interface ____ *214*
Figure 89. Specification of the ICapability Interface ____ *215*
Figure 90. Specification of the IGetLocation Interface ____ *217*
Figure 91. Specification of the IRoute Interface ____ *217*
Figure 92. Specification template of aspects in Ambient-PRISMA ____ *218*
Figure 93. Speification of the BidderDist distribution aspect ____ *221*
Figure 94. Specification template of triggers ____ *223*
Figure 95. Specification of a trigger in the BidderFunct aspect ____ *223*
Figure 96. Specification of a trigger in the ProcurDist aspect ____ *224*
Figure 97. Specification of the ACoordination Aspect ____ *226*
Figure 98. Specification of the MobilityAspect aspect ____ *229*
Figure 99. A distribution aspect of a Site ambient ____ *233*
Figure 100. A distribution aspect of a Group or a Virtual ambient ____ *233*
Figure 101. A distribution aspect of the Root ambient ____ *234*
Figure 102. A distribution aspect called MobileAmbDist aspect ____ *235*
Figure 103. Specification of the Bidder component ____ *236*
Figure 104. Partial Specification of the AuctionHouse system ____ *238*
Figure 105. Specification Template of Ambients ____ *239*
Figure 106. Specification of HostSite ambient ____ *239*
Figure 107. Some attachments of the `MobileAgentsAuctionModel` *architectural model* ____ *240*
Figure 108. Specification Template of configurations of ambient-PRISMA architectural models ____ *243*
Figure 109. Specification of the MobileAgentsAuctionConf configuration ____ *244*
Figure 110. Specification of the Bidder component ____ *253*
Figure 111. Specification of the AuctionHouseCnct connector ____ *254*
Figure 112. Specification of the IMobility interface ____ *254*

Figure 113. Partial Specification of ICapability Interface 257
Figure 114. Specification of HostSite ambient 265
Figure 115. Customer1 component and AgentCustCnct connector connected in ClientSite ambient 268
Figure 116. An attachment that connects a port of an ambient to a port of a child architectural element 269
Figure 117. Operation of the attachments associated to the InServicesPort port 270
Figure 118. Specification of the MobileAgentsAuctionConf configuration 275
Figure 119. Definition of Ambients in the DomainModel 281
Figure 120. Tool box with ambients 282
Figure 121. Site ambient shape 283
Figure 122. Modelling Tool indicating a violation of a hard constraint for an ambient 284
Figure 123. Modelling of a Group ambient at the Type Definition Level 285
Figure 124. Distributed Run-time Environment of Ambient-PRISMANET 286
Figure 125. Ambient-PRISMANET architecture 288
Figure 126. Leader-Notifier Interaction 290
Figure 127. The Panel of Known Middlewares 291
Figure 128. Sequence diagram when a failure of a middleware occurs 293
Figure 129. Pseudocode Fragment of the initialization of a middleware 294
Figure 130. Decentralized DNS in Ambient-PRISMANET 296
Figure 131. DNSServices class of the DNS layer 297
Figure 132. Executing model for mobility 299
Figure 133. The Transactional Manager and the Transactional Context classes 302
Figure 134. The StateCareTaker and the AspectMemnto classes 305
Figure 135. An inverse operation 307
Figure 136. A log functioning 307
Figure 137. Classes which implement the log 309
Figure 138. A segment of the `Ambient` *class constructor* 309
Figure 139. The MobilityAspect class and the ICapability interface 310
Figure 140. The AmbientCoordinationAspect class and the ICall interface 312
Figure 141. The three kinds of ambients in Ambient-PRISMANET 313
Figure 142. . Specification of the HostSite ambient 315
Figure 143. Specification of the Root ambient 316
Figure 144. Specification of attachments by the user in the AOADL 318
Figure 145. Procurement1 component in the ClientSite ambient 319
Figure 146. The Attachment class in Ambient-PRISMANET 319
Figure 147. Bottom-Up algorithm for searching routes 321
Figure 148. Creation of attachments 322
Figure 149. TraceOperation Structure 323
Figure 150. The move transaction of the ProcurDist aspect in the AOADL 324
Figure 151. Configuration after that Procurement1 has exited ClientSite 326
Figure 152. Configuration after that Procurement1 has requested the finishMovement service to AuctionSite 327
Figure 153. The Procurement1 component in the Root ambient 330
Figure 154. The modelling stage of the Ambient-PRISMA methodology 336

LIST OF TABLES

Table 1. Comparison of ADLs that address distribution and mobility (Part 1) ___________ *72*
Table 2. Comparison of ADLs that address distribution and mobility (Part 2) ___________ *73*
Table 3 Comparison of Aspect-Oriented Models that support Distribution ____________ *101*
Table 4. Comparison between Attachments and Bindings __________________________ *144*
Table 5. Description of the ICapability interface services __________________________ *216*
Table 6. The notations used in this chapter ____________________________________ *249*
Table 7. Representation of the middlewares in the Panel for Known Middleware _______ *291*
Table 8. List of TraceOperation structure in ClientSite ____________________________ *325*
Table 9. List of TraceOperation structure in Root _________________________________ *325*

PART I.INTRODUCTION

CHAPTER 1

INTRODUCTION

"cogito ergo sum" I think, therefore I exist

René Descartes

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

Ambient-PRISMA is supported by a framework that includes a metamodel, a language, a Modelling Tool and a middleware for developing Ambient-PRISMA aspect-oriented software architectures. The methodology that we propose follows the Model Driven Engineering (MDE). This allows the specification of Ambient-PRISMA software architectures using its Aspect-Oriented Architecture Description Language (AOADL), both in a textual and a graphical way. The AOADL is independent of

development platforms and is based on a formal language in order to be precise and preserve non-ambiguity. Code generation is supported by the middleware available in the CASE tool prototype.

1.1 MOTIVATION AND RATIONALE

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

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

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

- Software Architecture [Sha96] is considered to be the bridge between the requirements and implementation phases of the software life cycle. The software architecture of a system describes its structure in terms of computational (components) and coordination (connectors) units of software. Architecture Description Languages (ADLs) specify the functional and coordination properties of these software units in a formal way. However, few ADLs provide the needed constructs for describing distribution or mobility features in an abstract way.
- Component Based Software Development (CBSD) [Szy02] and Aspect-Oriented Software Development (AOSD) [Fil04] are two techniques that promise increased lends of reusability, flexibility, and maintainability through the software development process. CBSD proposes to construct the system by connecting entities that provide and require services. AOSD allows the separation of concerns by modularizing crosscutting concerns in separate entities called aspects. Distribution has been clearly identified as a crosscutting concern; separating it from the other concerns of the software system, will decrease cost and efforts to maintain and reuse software.
- Model Driven Engineering (MDE) [Sch06] is a software development approach based on transformations between models. MDE proposes the development of software by using models that describe a system in a technology-independent way. These models can be transformed to technology dependent models, which describe a system based on a specific technology.
- A formalism that provides mechanisms to describe distribution and mobility properties is Ambient Calculus (AC) [Car98a]. AC introduces the concept of ambient, which represents boundaries where computation occurs. Ambients can model the location hierarchy encountered in distributed systems and model the mobility as the crossing of the locations boundaries.
- PRISMA [Per05b] is an approach that integrates the advantages of Component-Based Software Development (CBSD) [Szy02] and Aspect-Oriented Software Development (AOSD) [Fil04] to specify software

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

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

1.2 OBJECTIVES OF THE THESIS

The main goal of this thesis is to investigate how the combination of the techniques and approaches of CBSD, AOSD, the MDE, and Ambient Calculus support the development of distribution and mobile software systems. The result of this combination is a framework that provides the definition of distributed and mobile software systems.

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

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

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

- To formalize the primitives of Ambient-PRISMA with an appropriate formalism in order to ensure unambiguous behaviours of the distributed and mobile software architectures specified through the AOADL.
- To provide a graphical support for the distribution and mobility characteristics incorporated to the AOADL in order to make friendly designs.
- To develop a middleware and a catalogue of code generation patterns. The middleware must permit the execution of the distributed and mobile software architectures based on the proposed model and the code generation patterns must provide the rules to generate automatically the source code of a specific programming language and distributed platform from the primitives incorporated to the proposed AOADL.
- To propose a methodology that guides an analyst through the development process of aspect-oriented software architectures of distributed and mobile software systems.
- To validate the model, the expressiveness of the language and the tool by developing case studies of distributed and mobile software system.

1.3 STRUCTURE OF THE THESIS

This remainder of this thesis is organized in the following chapters:

- Chapter 2: Distribution and Mobility

  This chapter presents concepts of distributed and mobile software systems. It also reviews different approaches for modelling these kinds of systems.

- Chapter 3: Software Architectures for Distribution and Mobility

This chapter provides an introduction to software architectures and how distribution and mobility have been dealt at an architectural level.

- Chapter 4: Aspect-Oriented Software Development for Distributed Systems
  This chapter provides an introduction to aspect-oriented software development, shows how the development of distributed software systems benefit from AOSD techniques and how different AOSD approaches have addressed distributed software.

- Chapter 5: Preliminaries
  This chapter introduces the Auction System case study that has been chosen to illustrate the work of this thesis. It also gives an overview of Ambient Calculus.

- Chapter 6: PRISMA
  This chapter gives an overview of PRISMA. It presents parts of its metamodel. Also, the software architecture of the Auction System case study is specified in PRISMA using the Aspect-Oriented Architecture Description Language (AOADL). Finally, the PRISMA CASE tool and methodology of PRISMA are presented.

- Chapter 7: Ambient-PRISMA: Characteristics and Metamodel
  This chapter presents Ambient-PRISMA. It explains the characteristics of a primitive called ambient and how it has been introduced in PRISMA for specifying aspect-oriented software architectures of distributed and mobile software systems. It also presents how the PRISMA metamodel has been enriched in order to obtain the Ambient-PRISMA metamodel.

- Chapter 8: Ambient-PRISMA Language

This chapter presents the language constructs that have been defined in order to permit the specification of distributed and mobile aspect-oriented software architectures.

- Chapter 9: Ambient-PRISMA Formalization

  This chapter presents the formalization of Ambient-PRISMA.

- Chapter 10: Ambient-PRISMA Case Tool

  This chapter presents how a Modelling Tool supports the modelling of Ambient-PRISMA specifications in a graphical and friendly way. It also presents the Ambient-PRISMANET middleware which supports the execution of Ambient-PRISMA in the .NET platform.

- Chapter 11: Ambient-PRISMA Methodology

  This chapter presents the Ambient-PRISMA methodology which offers guidelines to the analyst for developing distributed and mobile software systems.

- Chapter 12: Conclusions and Further Works

  This chapter presents the main contributions of the thesis and future works.

- Appendix A: Ambient-PRISMA AOADL Syntax

  This appendix presents the BNF of the Ambient-PRISMA AOADL.

- Appendix B: Ambient-PRISMA Software Architecture of the Auction System Case Study

  This appendix presents the complete specification of the Auction System using the Ambient-PRISMA language.

PART II. STATE OF ART

CHAPTER 2

DISTRIBUTION AND MOBILITY

"Your close neighbour is better than your faraway brother."

Popular Palestinian Quote

2.1 Introduction

This chapter presents the fundamentals for understanding distributed and mobile software systems. An overview of the basic concepts of distribution and mobility are introduced as well as how distributed and mobile software systems have been modelled.

The chapter is structured as follows: Section 2.2 defines what a distributed system is. Section 2.3 presents the different kinds of mobility encountered in distributed systems. Section 2.4 reviews different approaches that have been used for modelling distribution and mobility. Finally section 2.5 presents some conclusions.

2.2 Distributed Systems

Nowadays, software systems have a distributed nature in order to cope with non-functional requirements such as scalability, heterogeneity, evolution, and fault tolerance. Many definitions have been given to distributed systems. In the following, some definitions are presented:

<<A system in which hardware and software components located at networked computers communicate and coordinate their actions only by message passing>>

Coulouris et al. in [Col05]

<<A collection of independent computers that appears to its uses as a single coherent system>>

Tanenbaum et al. in [Tan07]

<<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.>>

Emmerich in [Emm00]

From the previous definitions it can be observed that a distributed system consists of components executing on different hosts. These components need to interact with each other in order to appear as a whole system. Each host has a middleware that allows components on the different hosts to interact. The complexity of distribution should be hidden from final users. This hiding is called transparency and it allows users to develop distributed applications in a similar way as centralized ones. Coulouris et al. [Col05] describe different forms of transparency in a distributed system such as access, location, transactions, concurrency, failure handling and relocation.

Nowadays, distributed software systems are built using distributed object or component middleware. Middleware is an intermediate software layer, or 'platform' that sits below the application and above the network operating system. The role of middleware is to ease the task of programming and managing distributed applications by providing services which hide the complexity and heterogeneity of a

distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages [Ber96].

2.3 Mobile Systems

A mobile system is a computing system with mobile entities. At the late 1980s, the first kind of mobile entities appeared. These were mobile processes in operating systems [Mil99a]. An operating system moved a process context (registers, programme counter, etc) from a processor to another for load balancing, fault tolerance, or concurrency purposes.

Currently, computer systems have two kinds of mobile entities: mobile hardware and mobile software. Mobile hardware is the capability of devices to move. This kind of mobility is also referred to as Mobile Computing [Car98b] or Physical Mobility. Software Mobility is the capability of moving a component from one host to another during runtime in a distributed system. This kind of mobility is also referred to as Code Mobility [Fug98], Mobile Computation [Car98b] or Logical Mobility [Fug98].

The notions presented in this thesis proposal can be used to model both hardware and software mobility because they are abstract enough. However, this thesis focuses on Software Mobility. As a result, the discussion presented in the following concentrates on Software Mobility.

A widely used definition for Code Mobility is the following:

<<Code Mobility is the capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location within a computer network.>>

Carzaniga et al. in [Car97]

In the previous definition Code Mobility is the capability of a distributed application to connect and disconnect code fragments to/from different locations at

run-time. Software Mobility is mainly used to achieve application level adaptation such as service customization, dynamic functionality extension, disconnected operation support or fault tolerance [Car97].

Code Mobility can be useful for the following reasons [Tan07]:

- To minimize communications: If data is processed close to where it resides, then messages that cross the network are minimized. For example, a client application moves to a server which manages a huge database in order to perform database operations and only send results across the network.
- To improve performance: A linear speed-up can be achieved by sending small programmes on different sites than only using one programme. For example, in searching in the Web it is better to send many programmes which search on different sites than only sending one.
- To improve flexibility: Distributed systems can dynamically configure without deciding in advance where each part should be executed.

Migration is classified to be proactive or reactive depending on who decides that an object needs to move [Fug98]. In proactive migration (sometimes called subjective or autonomous migration), an object decides when it needs to move and to where it has to move. On the other hand, in reactive migration (sometimes called objective or passive migration) the migration decision is determined by another executing object.

A detailed overview of existing code mobility paradigms is given by Fuggetta et al. [Fug98]. They describe three code mobility paradigms:

- Remote Evaluation allows the proactive shipping of code to a remote host in order to be executed.
- Code on Demand in which the client owns the resources (e.g., data) needed for the execution of a service, but lacks the functionality needed to perform the service.
- Mobile Agents are autonomous objects that carry their state and code, and proactively move across the network

In the Remote Evaluation and the Code on Demand paradigms code only moves. However, in the Mobile Agent paradigm code and state is moved. For mobile agents, mobility is classified into weak or strong mobility [Fug98]:

- Weak mobility happens in systems where the migrant is a data object which starts execution from the beginning after migration. Weak mobility transfers the code which may be accompanied by some initialization data; however the execution state (e.g. threads, stack pointers) is not involved.
- Strong mobility occurs in mobile objects which their execution is interrupted for the migration and once migrated on the destination, the execution is carried forward executing from the interrupted point.

Weak mobility is simpler to implement than strong mobility. This is due to the fact that technological platforms such as JAVA and .NET do not provide the ability to capture the states of threads. As a result, most approaches support weak mobility in order not to change the lower layers of the technological platforms. Examples of approaches which support weak mobility are Aglets [Lan98], ProActive [Bau00], or MAPNET [Sta04], and approaches which support strong mobility are D'Agents [Kot97] or Ara [Pei97].

Strong mobility is supported by two mechanisms: migration and cloning. The migration mechanism destroys the executing object and transmits it to the destination. The cloning mechanism creates a copy of the executing object at the new destination without destroying the executing object. As in migration, cloning can be proactive and reactive.

2.4 Modelling Distribution and Mobility

Semi-formal and formal notations have been used for modelling distribution and mobility. One can notice from the literature that the most widely used notations for modelling distribution and mobility are the Unified Modelling Language (UML) [UML07] and process algebras.

In the following, a review of the different approaches that have been used for modelling distributed and mobile systems is presented.

2.4.1 Unified Modelling Language (UML)

UML is a general purpose language that provides some graphical notations for modelling systems. It provides deployment diagrams for modelling distributed systems. Deployment diagrams show the hardware for a system, the software that is installed on that hardware, and the middleware used to connect the different machines to one another [Rum04].

These kinds of diagrams can only be used for a static view of the distributed software system. As a result, UML has to be extended by using stereotypes and profiles for mobility support. The different diagrams that UML provides have been extended for mobility support such as the class diagram, the sequence diagram [Kus05] or the activity diagram [Gra04].

The work in [Bau00a] extend class diagrams by using UML stereotypes and tagged values for representing the concepts of location, mobile object, and mobile location. Locations can be nested and mobile. The activity diagram is also extended with stereotypes for representing the move action, and the clone action. Furthermore, a location centred view of the activity diagram which focuses on the topology of locations is provided.

In [Kos03], a stereotyped class called <<mobile>> has been introduced as well as a new type of sequence diagrams. This work is based on using sequence diagrams based on Ambient Calculus [Car98b] and Mobile Maude [Dur00]. However, the sequence diagram used in this work is not standard.

The works in [Kav03] define and formalize UML stereotypes that support distributed applications. Initially, they chose the UML class diagram for representing systems at a type level and the interaction diagrams to model systems at an instance level. The use of interaction diagrams limited them in obtaining full advantage of

model checking techniques. Finally, they opted in using the class diagrams, the state diagrams, and the 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 were chosen in order to verify different run-time configurations without any modifications to the state or class diagrams.

Other works have used all UML diagrams: Use Case, Class, Object, State Chart, Sequence, Collaboration, Deployment, and Activity for modelling mobility such as M-UML [Sal04] which is an extension of UML 1.4 and the work presented in [Kle01].

It can be noticed from above that many works have exploited UML for modelling mobility. This is due to the fact that UML is an industrial standard which provides a graphical notation and can be extended for many features. However, UML lacks a precise semantics for its designs.

2.4.2 Process Algebra

Process algebras provide a precise description for interactive concurrent systems and a concise description for computation. The π-calculus [Mil99], which extends the Calculus of Communicating Systems (CCS) [Mil89] with constructs for exchanging channels over channels, was the first process algebra for describing mobile systems. In π-calculus, mobility is indirectly modelled as a change in the links a process has to interact with other processes. However, it does not provide an explicit notion for location.

Posterior process algebras which are based on π-calculus have emerged in order to deal with specific features of distributed and mobile systems. Some of these implicitly provide a notion of location such as SPI-calculus [Aba99] and Higher-Order π-calculus (HO π-calculus) [San93]. The SPI-calculus extends π-calculus with a set of cryptographic primitives. The HO π-calculus allows a process name as well as channel names to be sent along channels for facilitating code mobility modelling.

Other extensions to π-calculus provide an explicit notion of location. These include π_1-calculus [Ama00], Dπ [Hen98], Nomadic π-calculus [Woj00], Seal Calculus [Vit99], and Ambient Calculus [Car98b].

The π_1-calculus provides localities as well as channels and processes. It takes into account failures of localities, remote communication, and mobility of processes and their channels. The topology of a distributed system is flat (not a hierarchy of locations) and a migration decision is objective. The Dπ calculus is similar to π_1-calculus but it only provides local communication and mobility of processes.

The Nomadic π-calculus focuses on the modelling of mobile agents' interactions. It provides primitives for modelling agents and sites as well as channels and processes. In Nomadic π-calculus, a distributed system topology consists of a two-level hierarchy where agents contain processes, and are executed in sites. Agents are mobile and a migration decision is subjective i.e., depends on the mobile agent. Remote communication is not supported but can be simulated by mobility.

The Seal Calculus focuses on code mobility, resource access control, and security. Its main primitives are processes, resources (channels), and seals (logical or physical locations). The topology of a distributed system is represented in a tree structure. In Seal Calculus, only parent child communication can be performed. Seals are mobile and migration is expressed by sending a seal name on a channel. As a result, a seal can only move to its child or to its parent seal. A migration decision is objective i.e., the decision is made by the parent of a mobile seal.

Ambient Calculus (AC) is very similar to Seal Calculus (see section 5.2). Its main primitives are processes, ambients (logical and physical locations) and mobility capabilities which are called *in*, *out* and *open*. The *in* and *out* capabilities allow an ambient to move in and out of other ambients, respectively and the *open* capability allows an ambient to be dissolved. The topology of a distributed system is represented in a tree structure. Ambients are mobile and they can move in a sibling ambient and can move out their parent ambient by triggering mobility capabilities.

The migration decision is subjective. In Ambient Calculus, communication is only local and remote communication can only be simulated by mobility.

Further extensions to AC have been made such as Safe Ambients [Lev00], Boxed Ambients [Bug01] and Channel Ambient Calculus [Phi05] (ChAC). Safe ambients extends AC with a more secure mobility. This is provided by requiring the exited or entered ambient to agree the release or acceptance of an ambient. The Boxed Ambients does not provide the *open* capability as a primitive and adds channels as a primitive in order to allow communication among parent child ambients, similarly to ones provided by the Seal Calculus. The ChAC [Phi05] a variant of Boxed Ambients which provides channel-based interaction and sibling communication (see section 9.2 for more details).

Other process algebra which are not based in π-calculus can be found such as DJoin-Calculus [Fou96] and Kernel Language for Agents Interaction and Mobility (KLAIM) [Nic98] [Bet02]. The DJoin-Calculus is based on the reflexive abstract machine RCHAM. Localities in DJoin form a tree structure and there is no distinction between remote and local communication. However, only localities can move. KLAIM combines higher-order calculus with the Linda coordination language with primitives for locations. The topology of a distributed system in KLAIM can be hierarchically structured. Remote and local communication is also provided. However, only code migration is possible (locations do not move) and the migration decision is objective.

More information about mobile process calculi can be found in [MOC07] and a comparative analysis of distributed mobile calculi is presented in [Bou02]. In addition, a review of process calculus for locality can be found in [Cas01].

2.4.3 Other Approaches

Although UML and process algebra approaches are the most extended, other approaches exist. In this section, these approaches are reviewed for modelling distribution and mobility.

Mobile Maude [Dur00] extends the rewriting logic language Maude for supporting mobile computation. Its primitive notions are processes and mobile objects. Processes are nested and mobile objects can reside in processes.

Mobile Unity [Rom02] provides localities by distinguishing a variable with every unit of specification. It provides a flat topology of locations. Movement is modelled as location assignment. Communication among mobile units is captured using a minimalist coordination language.

Work has been done in modelling mobility paradigms using Object Z, a modelling language that extends the Z specification language. Network nodes are modelled as objects and connection between nodes through objects. However, Object Z does not provide explicit primitives for modelling distributed and mobile systems.

Work has also been done in the conceptual modelling of mobile object systems using language constructs. There is an extension of the TROLL language called Mobile Troll [Alb03] for specifying mobile systems which can clearly distinguish between their mobile and stationary parts. The behavioural part of Mobile Troll is based on an extension of the Distributed Temporal Logic called Module Distributed Temporal Logic [Fil00]. However, the work does not incorporate the flexibility of evolving the stationary parts into mobile parts.

2.5 Conclusions

In this chapter, an overview of the basic concepts of distributed and mobile software systems has been presented. Currently, most software systems have a distributed nature and mobility is a kind of adaptation that can be encountered in these systems. As a result, many approaches have emerged in order to model distribution and mobility.

Most modelling approaches are based on UML or process algebras. UML provides design oriented features that are supported by a graphical notation. This

characterizes UML to be widely used. Although it is a general purpose language, it provides extension mechanisms that allow it to model distribution and mobility. It also promotes separation of concerns by offering different views of the software system. However, it does not provide a precise semantics for its designs provoking their misunderstandings.

Many process algebras have been proposed for distribution and mobility. Some providing implicit support for locations and others explicitly support locations. It can be noticed, that process algebras have been adapted with the advance in distributed systems. However, process algebras are not widely used. This is mainly because they have not been integrated with software engineering practices. As a result, much more work has to be done in order to fill the gap between formal methods and semi-formal methods.

CHAPTER 3

SOFTWARE ARCHITECTURES FOR DISTRIBUTION AND MOBILITY

"The mother art is architecture. Without an architecture of our own we have no soul of our own civilization"

Frank Lloyd Wright

3.1 Introduction

Software architecture is a discipline that focuses on the design and specification of overall system structure. It originated from the proposal of David Parnas for modularizing software in order to simplify the construction of complex and large systems [Hof00]. In software architecture, a system structure is organized from elements that are composed into more complex ones until the overall system structure is obtained. It also deals with structural issues related to its elements such as their communication protocols, data access, functional assignment, and physical distribution [Sha96].

Software architecture bridges the gap between the requirements phase and the implementation phase of the software lifecycle. It is an abstract representation of the system that hides implementation, algorithms or data structure details [Bas03]. In this way, it is the first step for designing a system which needs to fulfil a collection of

desired properties that can be functional or non-functional such as throughput, consistency, and capacity [Sha96].

It can be noticed from the definition of Code Mobility given by Carzaniga et al. in [Car97] (see definition in section 2.3) that Code Mobility causes a change in the structure of a distributed software system. In this way, software architecture techniques can be useful for developing mobile software systems. As a result, the objective of this chapter is to present how distributed and mobile software systems have been described in software architectures.

This chapter is structured as follows: In section 3.2, the basic concepts of software architectures are introduced. The objective of this section is not to provide the reader with an analysis or a summary of the state of art of the concepts and properties of software architectures. These issues have been already treated in other works such as in [Per97], [Sar97], [Kog95] and [Med00]. The objective is to give the reader an introduction to the basic concepts of software architectures that support his/her understanding in the next sections and chapters of this thesis. Section 3.3, gives an overview of the available Architecture Description Languages (ADLs) found in the literature that have addressed distributed and mobile software systems. Section 3.4, presents a comparison among the ADLs presented in section 3.3. Finally, section 3.5 presents conclusions of the comparison performed in 3.4.

3.2 Basic Concepts of Software Architectures

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

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

Perry and Wolf in [Per92]

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

Shaw and Garlan in [Sha96]

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

ANSI/IEEE Std 1471-2000 [IEE00]

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

Bass, Clements,and Kazman in [Bas03]

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

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

3.2.1 Architecture Description Languages (ADLs)

Most box-and-line diagrams of architectural designs are not precise enough for analyzing completeness, consistency or correctness. As a result ADLs emerged. An ADL is a language that describes software architectures by providing a textual syntax and usually a graphical support. The purpose of ADLs is to aid the understanding and communication of software systems. Many ADLs can be found in the literature, each emerged for different purposes and domains. Examples of

ADLs are Rapide [Luc95] which allows architectural modelling, simulation, analysis and code generation capabilities, Wright [All97] which supports the formal specification and analysis of interactions between components, and SADL [Mor95] which focuses on formal architectural refinement. A comparison of different ADLs is provided in [Med00].

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

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

Medvidovic and Taylor [Med00] explain that an ADL provides the description of the building blocks of software architectures. These building blocks are components

(see section 3.2.2), connectors (see section 3.2.3) and configurations (see section 3.2.4). Also, it is important to provide tool support for an ADL in order to make it more useful.

3.2.2 Components

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

A typical definition of components is the following:

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

Clemens Szyperski [Szy02]

Meyer presents seven criteria for describing a component:

<<A component is a software element that:

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

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

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

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

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

6. Is composable with other components.

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

Bertran Meyer in [Szy00]

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

3.2.3 Connectors

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

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

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

- Connectors can be quite sophisticated elaborating complex definitions.
- Connectors' definition should be localized. Having connectors localized supports the maintenance and an improve design of interactions.
- Connectors are abstract. Connectors may be parameterizable and may define different kinds of interactions that are adapted to the components they coordinate at instantiation time. A connector can be instantiated as many times as necessary.
- Connectors may require distributed support.

- Components should be independent. An interface of a component is defined independently of which components will use it or how they use it.
- Connectors should be independent. Connectors should be able to mediate components that can be dynamically changing.
- Relations among components are not fixed. A component can interact with many components and its interaction with each component is different.
- Systems reuse patterns of composition.

3.2.4 Configurations

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

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

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

Shaw and Garlan in [Sha96]

3.2.5 Views

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

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

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

3.3 Architecture Description Languages for Distribution and Mobility

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

- Clients are not aware of where their servers are located in order to perform communications.
- Resources are accessible even though they are distributed

- Elements can move without others noticing it i.e. they are accessible and are able to communicate with the elements and resources in their new location.

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

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

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

Basically, ADLs that support the description of mobile software architectures should provide mechanisms for describing dynamic systems. Surveys have been performed on different dynamic ADLs such as in [Cue02] and [Bra04]. In addition, they should include other primitives such as a notion of location and how an architectural element can move from a location to another. In this section, an overview of the ADLs that have addressed distributed and mobile software systems are presented.

3.3.1 Darwin

Magee, Dulay and Kramer at the Imperial College in London were the creators of Darwin, one of the first ADLs that has focused on the specification of software

architectures for distributed systems [Mag95]. Darwin has a textual and a graphical notation. It describes configurations by binding components which provide and require services. Also, Darwin allows the construction of composite components by binding other components with their composite components.

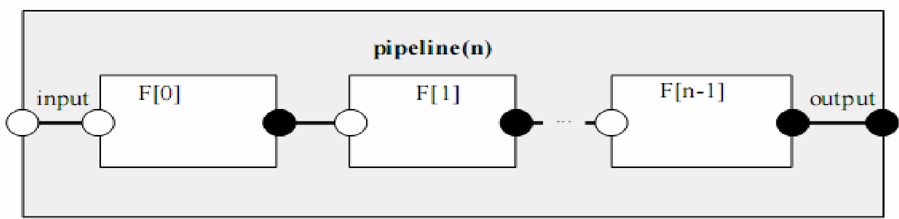


Figure 1. Graphical notation of a composite component pipeline in Darwin taken from [Mag95]

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

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

Darwin supports the configuration of distributed component instances and their remote communication. Component instances are distributed on different machines by using a Darwin expression that assigns an integer to each instance. For example,

to assign that each filter instance of Figure 1 is distributed, the following Darwin expression is used *"F[k]@k+1"*. These integer expressions are then mapped by the runtime system to real machine addresses and the component instances are configured on those machines.

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

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

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

However, in the literature, new advances to Darwin in constructing software architectures with mobile components cannot be found. Since Darwin is based on π-calculus only, mobility can only be simulated by the movement of channels. It lacks primitives to express the movement of components that cross boundaries.

3.3.2 C2Sadel

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

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

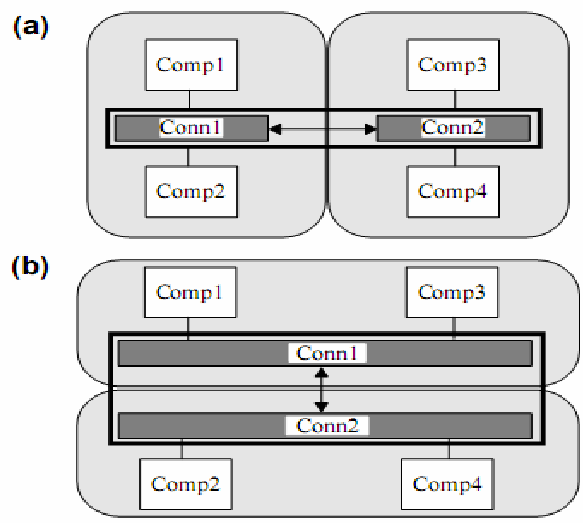


Figure 2. Connectors intermediating between distributed components taken from [Dos99]

C2 provides components to communicate remotely by encapsulating middleware technology access through connectors [Dos99]. The approach consists of implementing a single virtual software connector using a set of segmented connectors. Segments of the virtual connector are linked across the network. Figure 2 shows how connector segments connect distributed components. Shaded ovals represent network boundaries (e.g. hosts). In Figure 2 (a), messages sent to any segment of a virtual connector are broadcast to other segments. In Figure 2 (b), the virtual connector is separated into a top segment and a bottom segment. The connector segments are connected by a middleware. As a conclusion, each host of the distributed system has a connector segment. The connector segments shown in Figure 2 are implemented by the C2 framework.

C2 provides an implementation infrastructure for providing mobility [Med01]. Basically, mobility is provided by evolving the configuration of the distributed software architecture. Evolution is provided by a component called Admin that is connected to each segment connector of the distributed system. To support mobility

of a component the Admin component invokes methods for disconnecting and deleting a mobile component connected to its segment connector. Then, the Admin component connected to another segment (on another host) adds the component and attaches it to its segment.

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

3.3.3 Community

Community [Fia03] is an ADL that is based on category theory [Fia04] and on parallel design languages. It achieves a seperation of concerns in computation, coordination, and configurationas as well as distribution and mobility [Lop02]. The Community Workbench [Oli05] is a tool that provides an environment for defining Community configurations, animating them and probe specific scenarios.

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

In Community, space (a set of locations) is modelled in an abstract way through an abstract data type called loc. Loc models the positions of space depending on the notion that the modelled system needs. In this way, Community can model different

kinds of mobility. A position (location) is assigned to any constitute of a design i.e., channels and actions are location-aware because the distribution is fine-grained.

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

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

Figure 3. A mobile component specified in Community taken from [Lop04]

Figure 3 shows a mobile component specified in Community. It can be observed that the design has the *outloc l* location variable which models that the mobile component controls its own location. Also, it can be observed that each channel and action is assigned with the location *l* using the @ expression. In this case, all constitutes of the component are located in l.

Currently, Community is one of the most expressive ADLs for modelling and analyzing distributed and mobile systems. It also allows an analyst to simulate different mobility behaviours using the Community Workbench. However, Community supports mobility as a change in a value without causing dynamic reconfiguration of the software architecture. Community also does not focus on developing mobile and distributed software systems from its designs.

3.3.4 MobiS

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

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

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

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

3.3.5 LAM Model

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

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

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

3.3.6 π-ADL

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

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

However, the dynamic adaptability that π-ADL provides is limited since changes have to be anticipated. For example, the composite component has to have a

component type defined in its specification in order to allow that component to move to it. Also, π-ADL does not explicitly support location and mobility which are essential for modelling distributed and mobile software systems instead it simulates them by using components and composite components. In this way, the analysis and generation of a software system does not deal with specific properties of distributed systems.

3.3.7 Con Moto

Con Moto is one of the most recent ADLs that have been developed to address mobile and distributed systems [Gru04][Gru05][Sch06] . The architectural model is specified in a behavioural model and a structural model. In Con Moto, the behavioural model uses polyadic π-calculus for expressing non-functional properties and message passing of components in a distributed system. In the structural model, Con Moto provides primitives to describe the structure of a distributed and mobile system. Con Moto also has both textual language which is an XML dialect and a graphical notation (see Figure 4).

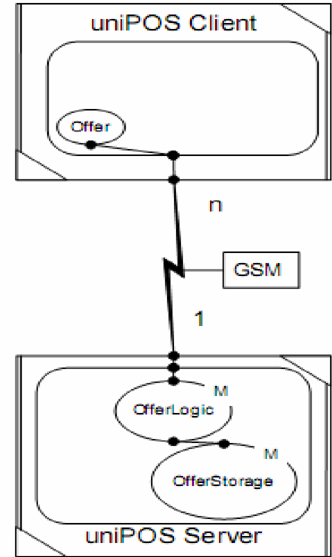


Figure 4. A structural model in Con Moto taken from [Gru04]

Con Moto distinguishes between logical and physical components. Physical components are devices such as PDAs or servers and logical components model software components. The great differences between logical and physical components are that physical components act as execution environments for logical components and physical components have resource constraints. For example in Figure 4, the *uniPOS Server* and the *uniPOS Client* are physical components and the *Offer*, *OfferLogi*c and the *OfferStorage* are logical components. Also, Con Moto distinguishes between physical and logical connectors. Physical connectors connect physical components such as the one between *uniPOS Server* and *uniPOS Client* in Figure 2. Logical connectors allow logical components to communicate. Logical connectors can be embedded in many physical connectors. For example in Figure 2, *Offer* can communicate with *OfferLogi*c.

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

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

3.4 Comparison

There are many features that can be used in order to compare and analyze the ADLs presented in section 3.3. However, the features that are of interest in the scope of this thesis are the ones related with distribution and mobility. The paper of Roman et.al [Rom00] is an appropriate starting point for discovering features that are essential for models that need to support mobility. In this paper, a framework for viewing mobility is described. They propose that space and coordination are the most important dimensions to be considered for dealing mobility, and that models which provide mobility are differentiated on how they assume the unit of mobility, location and context which highly depend on the coordination mechanisms that a model provides.

In the following, the features that have been considered in the comparison are explained:

- **Location:** Locations represent the different positions where a mobile entity can move in space. Locations have to be explicitly dealt as first-class entities and be distinguished from other entities of a model. As a result, space can be modelled and it can be specified where a mobile entity can and cannot move.
- **Mobility Support**: This feature describes how the architectural model supports the movement of a mobile entity. Basically, mobility support determines what implications are caused when an entity is moved such as a reconfiguration of an architecture or a change in a value.
- **Unit of Mobility**: This feature represents the smallest entity of a model that is allowed to move. This feature is important because it represents which entities of a model can be mobile or not.
- **Location-Awareness**: This feature determines whether an entity can be aware of its current location or not. This is important because it allows an entity to take decisions depending on his current location.

- **Migration Decision**: This feature specifies when and what causes an entity to move. This feature is associated with the terms objective and subjective mobility. Objective mobility is the movement of an entity caused by the environment. Subjective mobility is the movement of an entity caused by the same entity.
- **Coordination**: The coordination mechanisms that a model supports determine the context that is seen by each entity. The coordination mechanisms should be specified separately from the functionality of the entity. Since a basic characteristic of ADLs is the separation between coordination and computation, all of them support this characteristic. Therefore, what differentiates the coordination feature in ADLs is whether they support connectors or not.
- **Formalism**: Models have to be formal enough in order to enable a precise description of the semantics of the distribution and mobility properties. The formalism chosen should provide the primitives needed to specify mobility features.
- **Graphical Support**: ADLs provide graphical support in order to be usable. Most ADLs do provide one. However, the objective of this comparison is to discover which ADLs provide a graphical notation for specifically describing the distribution and mobility primitives. A basic graphical notation for distributed systems is the ability to provide a deployment notation, i.e., a graphical notation for specifying where entities are distributed.
- **Middleware**: Middleware implement distribution and mobility primitives that a model provides. In this comparison the objective is to discover to what extend ADLs have been used to develop distributed and mobile systems.
- **Tool Support**: A tool supports a developer during the development process through providing facilities and guides. Some of the facilities can be modelling support, verification, code generation, code execution, and simulation. The objective of this comparison is to discover what facilities do ADL tools provide for distribution and mobility.

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

Table 1 and Table 2) due to the limitation of the page dimensions.

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

	Location	**Mobility Support**	**Unit of Mobility**	**Location-awareness**	**Migration Decision**
Darwin	An integer value that represents a machine	No support	Not defined	No Location-awareness	No Support
C2Sadel	A border connector on each host.	Reconfiguring the architecture	Components mobile, all connectors static	No Location-awareness	Objective
Community	A value of an Abstract Data Type	Change in a value	Fine grained mobility of a component or a connector	Location-aware	Both objective and subjective
MobiS	Spaces that are composite components	Consuming and producing spaces tuples by rules	Spaces	Not explicitly	Both objective and subjective
LAM Model	Components nested in a tree structure	Reconfiguring the architecture	Mobile Agents, components and connectors are static	Not explicitly	Both objective and subjective

	Location	Mobility Support	Unit of Mobility	Location-awareness	Migration Decision
π-ADL	composite components represent locations	Reconfiguring the architecture	sub-components are mobile	Not explicitly	Both objective and subjective mobility
Con-Moto	Physical Components	Reconfiguring the architecture	Components are mobile	No Location-awareness	Not clear

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

	Coordination	Formalism	Graphical Support	Middleware	Tool Support
Darwin	Binding among components, No explicit notion of connector	π-calculus	Support but not for distribution	Implemented on CORBA for remote invocations	Compiler generates C++ code and automatic distributed configuration
C2Sadel	Explicit support for connectors that provide distribution and mobility	Semi-formal, first order logic.	Support but not for distribution	Supports remote invocations and code on demand mobility	Generates application skeletons
Community	Explicit support for connectors that provide distribution coordination	Category theory	Support but not for distribution	No support	Modelling, animation and simulation tool

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

3.5 Conclusions

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

- The notion of location has been introduced as: a new type of connector, a new type of component, a value of a data type, and a composite component.

In the case when it has been introduced as a composite component the notion of location is quite ambiguous since the same concept is used for both representing locations and hierarchical compositions.

- The support of mobility highly depends on how the notion of location is introduced in the ADL. When location is represented by a value, mobility is supported by the change of this value. When location is supported by architectural element (component, connector, composite component) mobility is supported by reconfiguring the software architecture.
- Components are the units of mobility in most ADLs. Community is the only ADL that allows connectors to be mobile.
- Few ADLs allow there elements to be location aware.
- Most ADLs provide objective and subjective mobility.
- Most ADLs provide connectors as first-class entities in order to provide complex coordination mechanisms. However, the interesting approaches are those that provide special coordination mechanisms for distribution such as Community and C2Sadel.
- ADLs have been formalized by using formalisms that do not explicitly provide first-class entities for distribution and mobility.
- Most ADLs provide a graphical support to model components, connectors and configurations. However, an ADL that provides the graphical support for distributing architectural elements is needed. Providing a graphical support for describing the distribution and mobility issues allow the ADL to be more usable. Con-Moto has a friendly notation to distinguish between logical components and the locations where these components are assigned.
- The only ADLs that have been implemented on a specific middleware have been Darwin and C2Sadel. However, Darwin does not provide mobility and the mobility that C2Sadel allows is restricted.
- Each ADL provides tool support with a different focus. For example, Darwin allows the automatic configuration of components to locations. C2Sadel provides skeleton code generation. Community and Con-Moto focus on simulation. π-ADL provides many tools however, there was not any

information available to analyze to what extend does it support code generation of distributed and mobile code.

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

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

CHAPTER 4

ASPECT-ORIENTED SOFTWARE DEVELOPMENT FOR DISTRIBUTED SYSTEMS

"We may take different paths, we may get caught up in different aspects of life... but our love will always be there."

S.A. Sheade

4.1 Introduction

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

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

Edsger Dijkstra in [Dij74]

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

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

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

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

This chapter is structured as follows: Section 4.2 presents basic concepts of AOSD. Section 4.3 explains how AOSD has dealt with distributed systems. Finally, section 4.4 presents some conclusions.

4.2 Aspect-Oriented Software Development

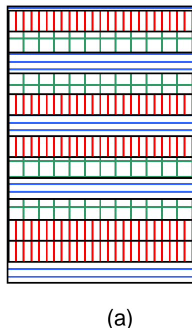
AOSD emerged from AOP. The early contributors to AOP [Kic97] were Cristina Lopes and Gregor Kiczales at the Palo Alto Research Centre (PARC) of the Xerox Corporation. Kiczales was the leader of the team that created AspectJ [Kic01] [Lad03] [Mil04], the first practical implementation of AOP and, currently, the most extended one. In December 2002, the AspectJ project was transferred from Xerox to the open source community at eclipse.org [ASP07].

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

4.2.1 Crosscutting Concerns

A crosscutting concern is behaviour or data that is spread in different modules. Crosscutting concerns can be non-functional requirements such as security, logging, authentication, etc or functional concerns that are spread among different modules. Crosscutting concerns cause the following [Lad03]:

- Tangling of Concerns: It is caused when a module contains multiple concerns at the same time (see Figure 1(a)). As a result, the module is less maintainable, reusable, and comprehensible.
- Scattering of Concerns: It is caused when a single concern is defined in many modules such as performance concerns. Scattering of concerns provoke identical definitions to be repeated in multiple modules.



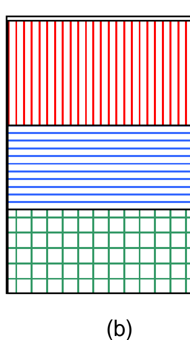


Figure 5. (a) The tangling concerns in the traditional applications (b) the separated concerns in AOSD taken from [Lad03].

4.2.2 Aspect

An aspect is a software entity that provides mechanisms for encapsulating crosscutting concerns. Figure 1(b) shows the elegancy that is achieved through aspects in the modularization and the localization of the crosscutting concerns of Figure 1(a). An aspect normally contains declarations similar to the ones of a class and can contain, depending on the aspect language, the mechanisms that specify the interactions of an aspect with the underlying system. For example, in AspectJ aspects are the following (see section 4.2.3 for definition of pointcuts and advice):

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

Gregor Kizcales et al. [Kiz01]

4.2.3 Weaving

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

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

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

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

Two weaving models are distinguished:

- Static Weaving: Static models are those in which the aspects and non aspect entities are declared as separate entities, however at compilation time the two entities are combined into one. This model has the drawback that at execution time the aspects cannot be manipulated. As a result, aspects fail to gain a high level of evolution, maintenance and reusability. Examples to this model are AspectJ [Kic01] and Composition Filters [Aks94] (see section 4.3.2.2).
- Dynamic Weaving: In this model the separation of the aspects and non-aspects entities is conserved at all moments even at execution time. Aspects

can be woven and unwoven at run-time. Examples to this model are the PROgrammable extenSions of sErvices (PROSE) platform that provides dynamic AOP for Java [Pop02] (see section 4.3.1.4), JAsCo [Suv03], the Disguises Model [San98] (see section 4.3.2.4) and the Dynamic Aspect-Oriented middleware Framework (DAOF) [Pin02].

4.3 Aspect-Oriented Software Development for Distributed Systems

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

The different technological middleware offer constructs that solve a number of problems such as remote access, fault tolerance and security. However, applications that need these kinds of primitives have to introduce them in an inelegant way. For example, Figure 6 shows the code of a C# class that its objects offer and request services to and from other remote objects using .NET Remoting. The code in italics shows the code related to distribution.

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

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

When AOSD techniques are used, all the code related to remote access can be separated into an aspect and then it can be weaved with all the existing classes that need remote access. In this way, the distributed application has a higher quality since maintenance, reusability and understanding of the remote access concern is improved. AOSD is a promising technique for developing distributed systems since it separates concerns that crosscut them. In addition to the remote access concern depicted in Figure 6, [Sub05] detects other crosscutting concerns found in distributed applications such as synchronization, fault tolerance and security.

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

namespace DistributedSearchAgents {
     public class SearchAgent : MarshalByRefObject, INotification {

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

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

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

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

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

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

}//end namespace DistributedSearchAgents
```

Figure 6. Code to handle a distributed client/server object in .NET Remoting

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

4.3.1 Implementation

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

4.3.1.1 AspectJ

AspectJ [Kic97] is a general purpose language that extends the Java object-oriented programming language in order to incorporate aspects. In AspectJ, the base code is programmed using Java objects and in order to define aspects, AspectJ is used. AspectJ provides the aspect construct that contains: pointcuts, advice, and Java constructs. The aspect, pointcuts and advice concepts have been previously introduced in sections 4.2.2 and 4.2.3.

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

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

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

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

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

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

- ***A Pattern for Distribution Aspects (PaDA)***

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

In [Soa02b], Soares reports his experience in restructuring a web-based health complaint system which was initially implemented in plain Java and RMI into AspectJ. Soares removed the RMI specific code from the initial version in pure Java into a set of aspects. He noticed that all the distribution code was tangled both in the server-side and in the client-side. As a consequence of this experience, Soares proposes a pattern called PaDA [Soa02a] that provides a structure for implementing distribution using AOP (see Figure 7). The pattern consists of implementing three distribution aspects: a client-side aspect to call remotely to the server-side, a server-

side aspect to enable the reception of remote calls and an exception handling aspect. The client-side aspect is weaved with the client, the server-side aspect is weaved with the server and the exception handling aspect is weaved with both client and server.

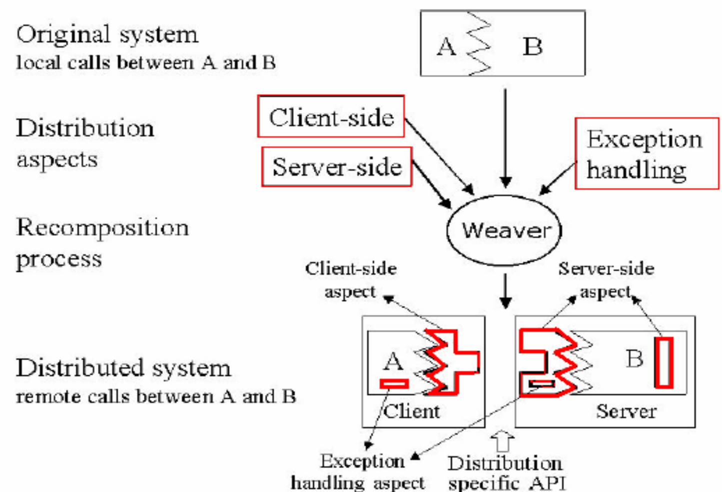


Figure 7. PaDA's Structure taken from [Soa02a]

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

Other techniques can be used in order to separate the distribution code such as factories or adapters' patterns [Gam95]. However, the version of the application where these patterns are used has more code than one in the AspectJ version. Also, the code of the server and the client has to be changed in order to incorporate these

patterns to the application. In this way, the AspectJ version is more maintainable and adaptable.

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

- ***The Mobility Aspect Pattern***

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

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

done in order to improve the traceability of aspects and provide tools or wizards that automate code generation of aspects.

4.3.1.2 General Object to EJB Conversion Helper (GOTECH) Framework

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

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

4.3.1.3 DJCutter

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

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

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

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

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

4.3.1.4 PROgrammable extenSions of sErvices (PROSE)

PROSE is a platform that provides dynamic weaving of aspects [Pop02]. PROSE consists of the Java Virtual Machine (JVM) and a set of libraries. PROSE does not define a new AOP language. It provides aspect constructs that are implemented in pure Java. In order to use PROSE, one has to extend the constructs it provides, e.g. in order to define a specific aspect the base class aspect has to be extended. To

provide dynamic weaving of aspects, PROSE uses the Java Virtual Machine Debugger Interface (JVMDI) to stop the execution of the JVM at join points and then calls the advice behaviour.

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

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

4.3.1.5 AspectIX

AspectIX [Hau98] is a middleware that supports AOP for CORBA distributed objects. The middleware offers the ability to add new aspects dynamically, i.e. it provides dynamic weaving. In AspectIX, an object consists of a set of fragments (see Figure 8). Each fragment has a specific behaviour of an object. For example, a fragment may hold the constraints on the communication channel to another fragment. Aspects are specified in order to determine which fragment represents a distributed object.

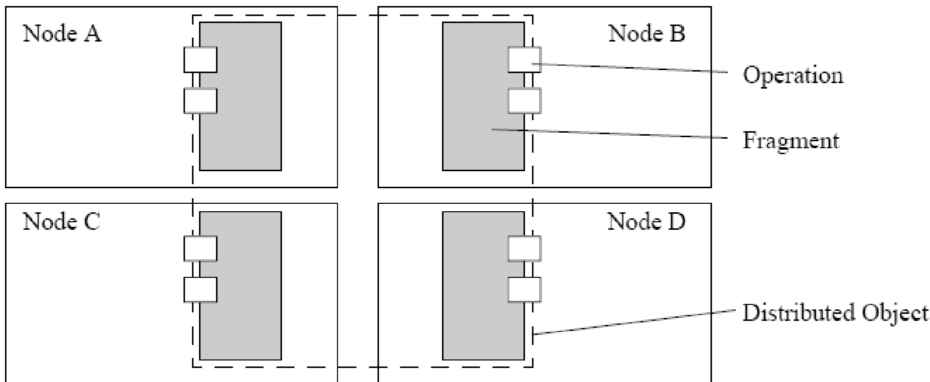


Figure 8. Fragments of a distributed object in AspectIX taken from [Hau98]

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

4.3.2 Analysis and Design

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

4.3.2.1 The D-Framework

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

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

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

DJ is the implementation of D in Java and RMI. A tool called the Aspect Weaver automates the transformations of D constructs (coordinators and portals) into Java classes as well as weaves the aspects to the base code in Java. The Aspect Weaver extends the base code with hooks (meta-data and new code) that transfer the control to the aspects at the beginning and end of methods. Also, a layer called RIDL has

been implemented above RMI in order to support D portals. Figure 9 shows the run-time architecture for a client object (*aObj* in *Space1*) calling a remote object (*aObj* in *Space 2*) that is associated with a D portal. The client object refers to a proxy of the portal (*aObjPP*) that checks if there are illegal remote calls. If there aren't any illegal calls, the portal proxy redirects the call to the portal object (*aObjP*) by using the RMI layer, and as a consequence the portal object redirects the call to the real object.

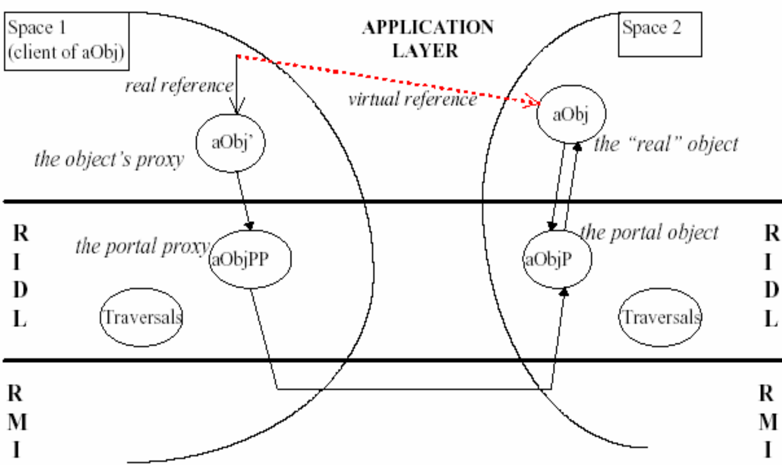


Figure 9. Run-time Architecture for D's remote objects taken from [Lop97]

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

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

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

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

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

As a conclusion, the D approach is the first to provide declarative languages that support aspects. The D languages are domain-specific languages as each of them deals with a specific problem (concern). The study of the crosscutting concerns performed in order to define D and the implementation of DJ were the base for the birth of the actual versions of AspectJ. However, currently, aspect-oriented languages are general purpose languages instead of domain specific. As Cristina Lopes describes in [Lop02a]:

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

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

Cristina Videira Lopes in [Lop02a]

4.3.2.2 Composition Filters

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

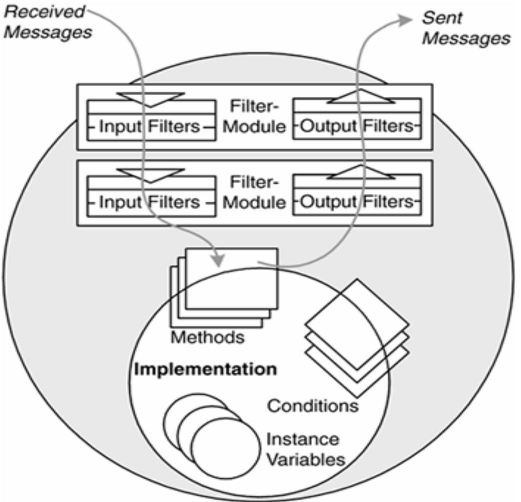


Figure 10. Filters in the composition filter model taken from [Ber04]

Filter modules are composed of a set of filter elements that determine whether a message is either accepted or rejected and what action to be performed in either case (see Figure 10). Filter modules are separated into Input Filters that filter object received messages and Output Filters that filter object sent messages. A filter element consists of:

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

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

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

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

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

4.3.2.3 UML All pUrpose Transformer (UMLAUT)

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

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

4.3.2.4 The Disguises Model Approach

The disguises model [Her03] is a model that separates synchronization, concurrency control, distribution [San00a], and replication from the functional behaviour of a system in aspects. An aspect in this model is a non-functional property and a non-functional property is represented by an aspect. Also, the model provides dynamic weaving of aspects.

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

The disguises model establishes a distinction between objects and aspects by computation reflection techniques [Mae87]. It defines auxiliary objects that establish a connection between the functional object and its aspects. The auxiliary objects are separated into Input and Output objects (see Figure 11). Input Objects intercept intercept input messages to objects and send them to aspects. Output Objects intercept output messages from objects and send them to aspects. Aspects are separated into input and output aspects. Input aspects are activated when a method in the functional object is invoked such as the synchronization and replication

aspects. Output aspects are activated when a functional object invokes a method, such as the distribution aspect.

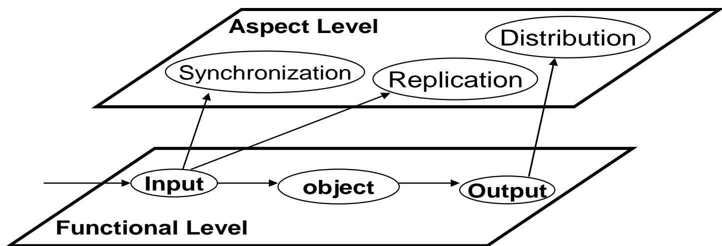


Figure 11. Structure of the Disguises Model taken from [Her03]

The distribution disguise performs the tasks depicted in Figure 12. The aspect publishes the functional object, looks in a reference table, and is in charge of sending remote messages using the platforms. The distribution aspect receives an output message of an object, and then it looks in the reference table in order to search for the receptor object of the message. When it has the remote reference of the receptor, it transforms the message in order to send it through a communication platform to the receptor.

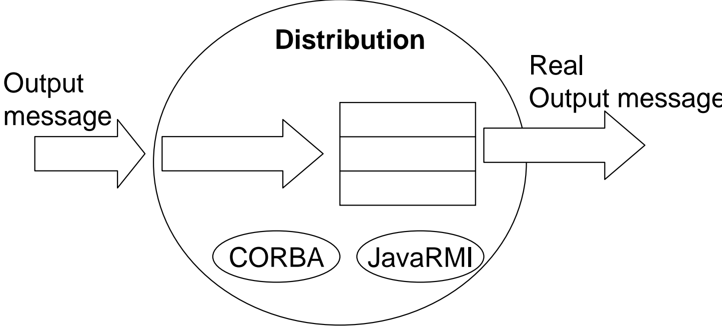


Figure 12. Structure of the distribution aspect taken from [Her03]

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

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

4.3.2.5 AWED

AWED [Ben06] is an aspect-oriented language that includes constructs for supporting the distribution of the aspect-oriented application. The language provides constructs for specifying remote pointcuts that match join points at remote hosts (similarly to DJCutter in section 4.3.1.3) and can determine where remote advices can be executed. Hosts in AWED can be grouped and a pointcut can match on a group of hosts. An aspect in AWED contains a set of fields as well as pointcuts and advices. An aspect construct determines whether it is dynamically deployed on all hosts or only on the local one. Also, AWED provides constructs for specifying how a distributed aspect shares its state with other aspects basing on the type of the aspect, its group or the host. AWED has been implemented by extending JAsCo [Suv03] and by using RMI.

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

4.4 Conclusions

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

It can be concluded that at the implementation level of software development, the AOP languages and platforms have improved the maintenance and complexity of

the code of distributed systems. Also, AOP has been applied for remote communication more than on mobility. This fact is observable because mobility needs a platform that provides dynamicity, including dynamic weaving.

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

Table 3 Comparison of Aspect-Oriented Models that support Distribution

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

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

PART III. PRELIMINARIES

CHAPTER 5

PRELIMINARIES

"All that we are is the result of what we have thought. The mind is everything. What we think we become."

Budha

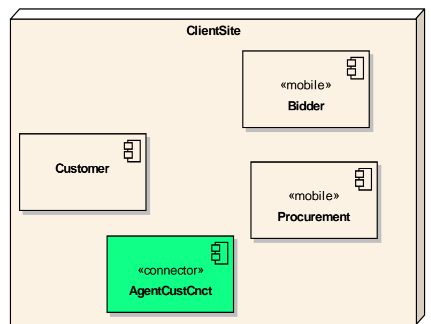
This chapter presents the case study used in order to illustrate the work presented in this thesis. In addition, an overview of Ambient Calculus is given.

5.1 Case Study: Auction System

There is a variety of domains where Ambient-PRISMA can be applied such as distributed bank system, teleoperation systems, mobile phones network, etc. In this thesis, an electronic commerce application is used for illustrating Ambient-PRISMA. The case study consists of an electronic Auction System which offers its products for being bid and sold. These kinds of auctions have been developed in works such as [San00]. The Auction System allows its customers to send mobile agents in order to search for interesting products and bid on their behalf. In this way, the customers can send the agents and disconnect from the network. This reduces network latency and traffic and agents can respond to changes in the auction in a quicker way.

Figure 13 shows a UML deployment diagram that shows the architectural elements that form part of the Auction System software architecture. It shows that the software architecture consists of a *Customer*, a *Bidder*, a *Procurement*, and a *AuctionHouse* component. It also shows that there is a connector called

AgentCustCnct which coordinates the *Customer* component with the *Bidder* and the *Procurement* components and a connector called *AuctionHouseCnct* which coordinates the *AuctionHouse* component with the *Customer*, *Bidder* and *Procurement* components. The *AuctionHouse* component is composed of a component called *LotOnAuction* and a connector called *CnctLotAuction*. The *Customer*, the *Bidder*, and the *Procurement* components are hosted in the *ClientSite* host as well as the *AgentCustCnct* connector. The *AuctionHouse* component and the *AuctionHouseCnct* connector are hosted in the AuctionSite host. The *Bidder* and the *Procurement* components are marked with mobile (<<mobile>>) in order to indicate that they are mobile components. They can move from the *ClientSite* host to the *AuctionSite* host.



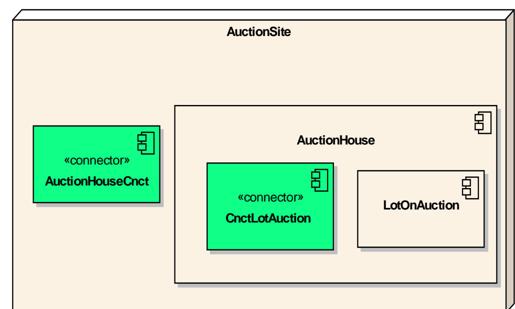


Figure 13. A UML deployment diagram with the architectural elements of the Auction System architectural elements

The functional and distribution behaviour of each component is explained in the following:

5.1.1 Customer

The functionality of the *Customer* component is listed below:

- It asks the *Procurement* component to search for a product with a certain characteristics and that the date of auction is before a specific date.

- It confirms to the *Procurement* component if it is interested in a certain product.
- It asks the *Bidder* component to bid for a specific product at a maximum price limit
- It asks the *Bidder* component about the status of the bidding
- It changes the maximum price limit of a product.

The distribution behaviour of the *Customer* component is listed below:

- The *Customer* component is a stationary component.
- It decides to move the *Procurement* when it needs a product. When the product it needs is found by the Procurement, it moves the *Bidder* component to the *AuctionSite*.
- It can decide to move the *Procurement* and the *Bidder* component back to the *ClientSite* when it believes it is appropriate.

5.1.2 Procurement

The functionality of the *Procurement* component is listed below:

- It asks the AuctionHouse component for searching products that fulfil the descriptions and the limited date given by the *Customer* component.
- It notifies the *Customer* component of a product that fulfils certain characteristics.

The distribution behaviour of the *Procurement* component is listed below:

- It moves to an *AuctionSite* host when the customer orders it.
- It moves back to the *ClientSite* host when the *Customer* confirms that it has found the appropriate product or when it does not find any product that fulfil the characteristics specified by the *Customer* component.

5.1.3 Bidder

The functionality of the *Bidder* component is listed below:

- It keeps biding for a specific product until it wins, the bid reaches a maximum price limit, or when it looses.
- It sends to the Customer component the bidding status
- It receives from the Customer component to change the maximum bid of a product.

The distribution behaviour of the *Bidder* component is listed below:

- It moves to the to an *AuctionSite* host when the customer orders it.
- It moves back to the *ClientSite* host when the auction has finished (won or lost) or when the current biddings exceed the maximum price limit.

5.1.4 LotOnAuction

The functionality of the *LotOnAuction* component is listed below:

- It stores the description of a product on the auction and the auction detail such as the auction date, and the amount which a product has to be sold.
- It returns to the *AuctionHouse* component the description of the product it stores, if the product is pending to be auctioned.
- It determines the quantity of the initial bid of a product when an auction starts and the next bid.
- It starts the auction and ends the auction when the amount of a product is achieved after certain bids.

The distribution behaviour of the *LotOnAuction* component is listed below:

- The *LotOnAuction* component is a stationary component.

5.1.5 AuctionHouse

The functionality of the *AuctionHouse* composite component is listed below:

- It asks for the different *LotOnAuction* components that are available.

- It returns to the Procurement component the products that fulfil to the descriptions it sends.

The distribution behaviour of the *AuctionHouse* composite component is listed below:

- The *AuctionHouse* component is a stationary component.

5.2 Ambient Calculus

Ambient Calculus [Car98a] (AC) is a process algebra that extends π-calculus [Mil99] in order to introduce the concept of ambient. An ambient is a bounded place where computation occurs. Thus, an ambient can be anything with a boundary such as a laptop, a web page, a folder, etc. Each ambient has a set of running computations that can control it. These are responsible for moving an ambient. In addition, an ambient can contain other subambients that have running computations.

Thus, mobility is performed at an ambient level, i.e. ambients are mobile. Also, mobility is performed by crossing boundaries of ambients. AC provides mobility and local communication primitives. These primitives can be expressed in a textual syntax and in a graphical syntax which is called Folder Calculus [Car98b] (see Figure 14). Folder Calculus is a graphical metaphor for AC where ambients are visually represented as folders.

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

Textual Syntax	Visual Syntax	Comments
$(\nu n)P$		New name n in a scope P.
$n[P]$		Folder (ambient) of name n and contents P.
$M.P$		Action M followed by P.
$P \mid Q$	P Q	Two processes in parallel. (Visually: contiguously placed in 2D.)

Textual Syntax	Visual Syntax	Comments
0	○	Inactive process (often omitted).
$!P$		Replication of P.
$\langle M \rangle$		Output M.
$(n).P$		Input n followed by P.
(P)		Grouping

Figure 14. The Textual and Visual Syntax of Ambient Calculus constructs taken from [Car98b]

Some of the primitives that AC provides are called capabilities. Capabilities are actions that can be performed on ambients There are three main types of capabilities: enter, exit and open capabilities. The enter capability orders an ambient to enter another ambient on its same hierarchy level (see Figure 15). The exit capability orders an ambient to exit its parent ambient. The open capability dissolves an ambient leaving the processes that were in it.

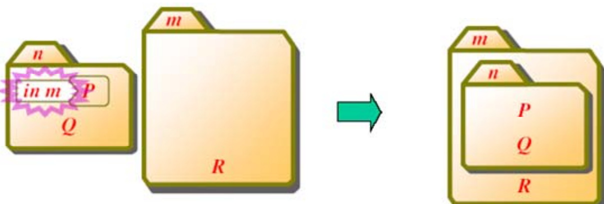


Figure 15. Applying the enter capability to the ambient n taken from [Car01]

5.3 Conclusions

This chapter has introduced the case study that is used throughout the thesis in order to illustrate the work to be presented. The case study of an Auction System which uses mobile agents has been chosen. The Auction System is appropriate for

illustrating this thesis because it is a distributed system where its components can have different mobility behaviors. In addition, an overview of AC is given. The proposal of this thesis is based on AC. As a result, an understanding of AC concepts is important.

CHAPTER 6

PRISMA

"The past is not simply the past, but a prism through which the subject filters his own changing self-image."

Doris Kearns Goodwin

6.1 Introduction

PRISMA is an approach that integrates AOSD and software architecture approaches in order to specify software architectures of complex systems. PRISMA is based on its models, metamodel [Per05b], Aspect-Oriented Architecture Description Language (AOADL) [Per06a], a methodology [Per08], and a CASE tool called PRISMA CASE. The metamodel defines the concepts and constraints needed for defining PRISMA architectural models. The AOADL provides the primitives needed to specify PRISMA software architectures. The PRISMA CASE tool allows the development of aspect-oriented software architectures following the PRISMA approach using graphical modeling tools, verification mechanisms, model compilers to automatically generate code and tools for executing the generated code.

Since this thesis presents an extension to PRISMA, the objective of this chapter is to provide the reader with a presentation to the PRISMA approach in order to permit the comprehension of the coming chapters of this thesis. The reader is referred to [Per06b] for a more detail description of PRISMA. In addition, this chapter

illustrates the primitives that the AOADL provides by specifying the aspect-oriented software architecture of the Auction system case study previously presented without distribution and mobility characteristics.

This chapter is organized as follows: Section 0 presents an overview of the PRISMA model. The primitives that PRISMA provides are illustrated by the metamodel and the Aspect-Oriented Architecture Description Language (AOADL). Section 6.3 explains the methodology that PRISMA proposes for modelling aspect-oriented software architectures. Section 6.4 presents how the PRISMA CASE tool has been developed and how it supports the development of PRISMA software architectures. Finally, section 3.5 concludes the chapter.

6.2 PRISMA Model Overview

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

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

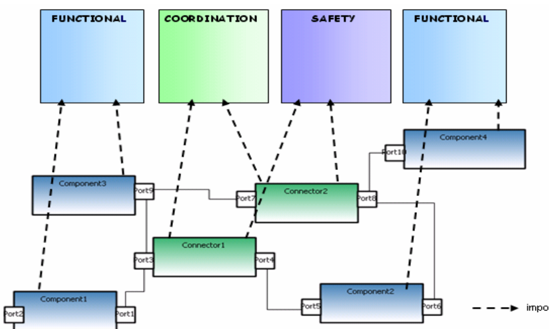


Figure 16. Crosscutting Concerns in PRISMA software architectures taken from [Per06b]

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

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

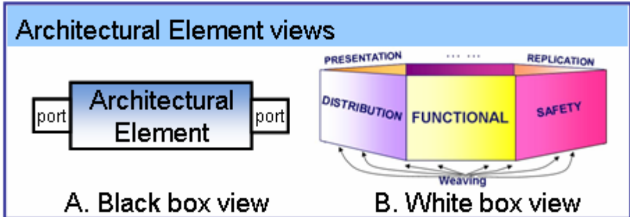


Figure 17.Views of a PRISMA architectural element taken from [Per06b]

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

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

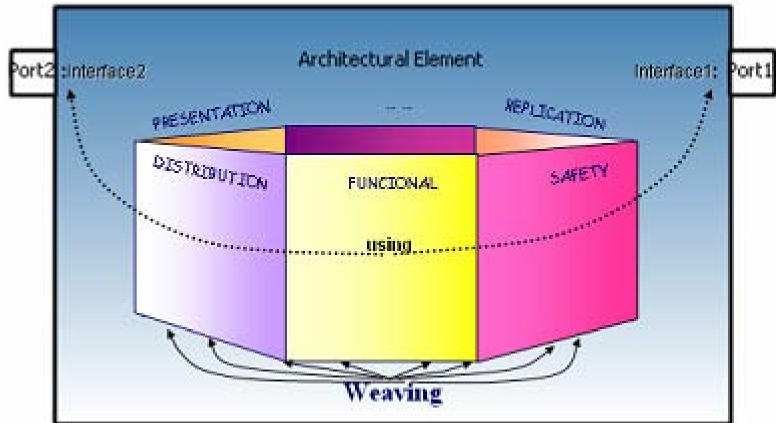


Figure 18. Communication between the white box and the black box views taken from [Per06b]

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

In software architectures, components are connected with connectors. As a result, attachments are the channels that enable the communication between components and connectors. Each attachment is defined by attaching a component port with a connector port. In Figure 16, the lines between component ports and connector ports are attachments.

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

of the complex component (the system) and the ports of the architectural elements that a system contains (see Figure 19).

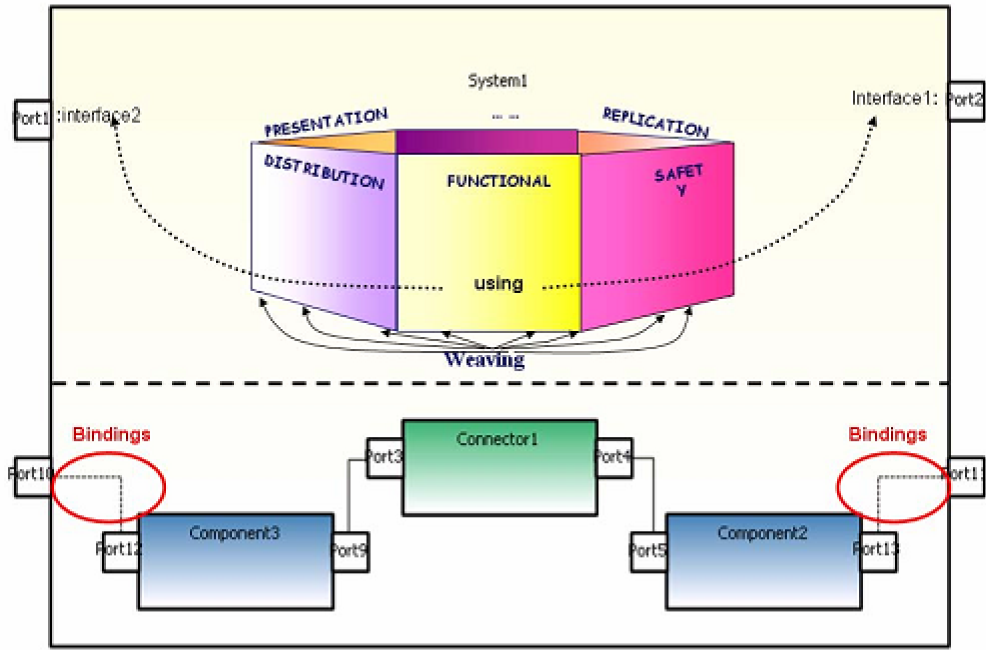


Figure 19. Systems taken from [Per06b]

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

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

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

6.2.1 Architectural Model

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

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

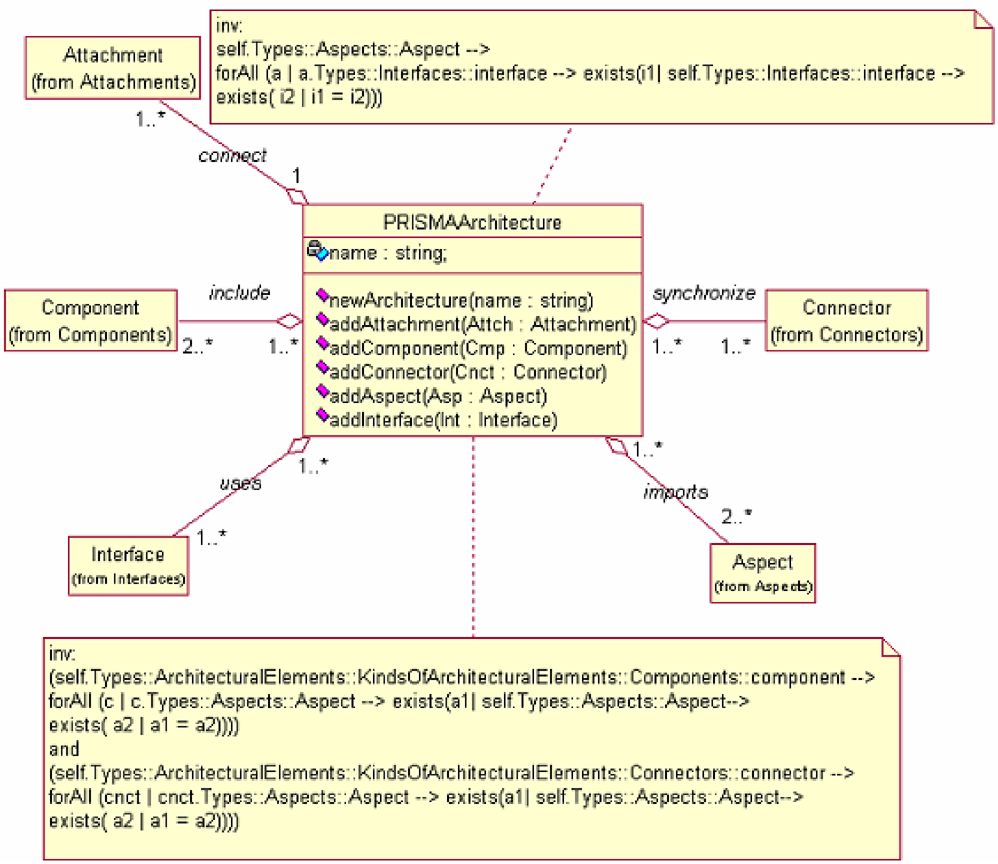


Figure 20. The package *ArchitectureSpecification* of the PRISMA metamodel taken from [Per06b]

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

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

Figure 21. Syntax of the architectural model in the AOADL

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

6.2.2 Interfaces

An interface publishes a set of services. It describes the signature of the services that can be invoked or requested through that interface (see Figure 1). The signature of a service specifies its name and parameters (see Figure 23). Parameters are declared by specifying their *kind* (input/output), name and data type.

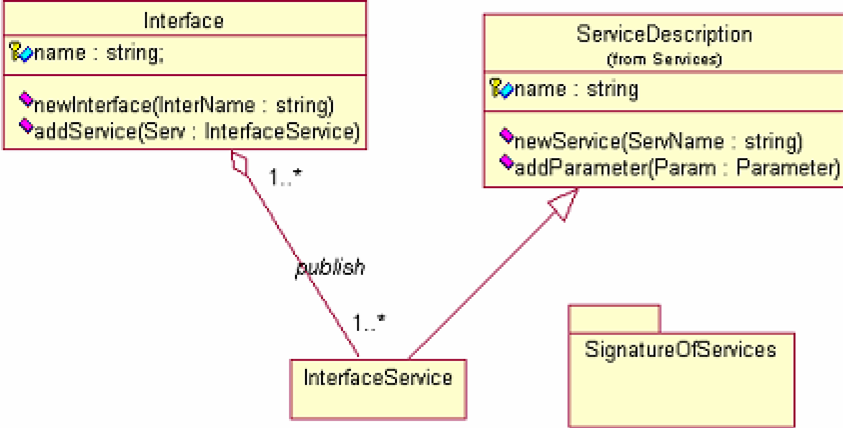


Figure 22. The package *Interfaces* of the PRISMA metamodel taken from [Per06b]

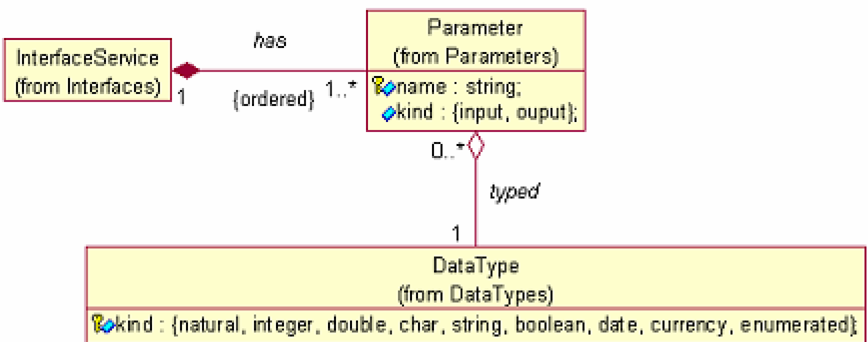


Figure 23. The package *SignatureOfService* of the PRISMA metamodel [Per06b]

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

```
Interface ICustProc
 in/out notifyProdInterest(input LotId, input Saleroom:string,
                           input SaleNum:string, input DateOfAuction:date,
                           input Lotdescrip:string,
                           output Interested: boolean);

End_Interface ICustProc
```

Figure 24. Specification of the interface *ICustProc*

6.2.3 Aspects

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

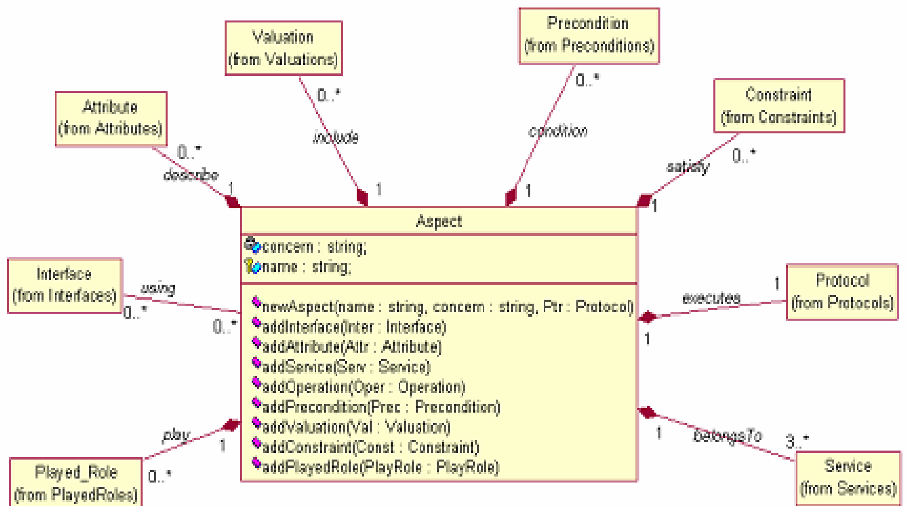


Figure 25. The metaclass Aspect of the package Aspects of the PRISMA metamodel taken from [Per06b]

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

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

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

6.2.3.1 Attributes

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

- *Constant:* The stored values cannot change
- V*ariable:* The stored values can be changed

- *Derived:* The value is calculated on demand applying its derivation rule.

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

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

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

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

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

6.2.3.2 Services

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

Each service has a name and can have a set of parameters (as described in section 6.2.2) whose type can be *input* or *output*. A service can be *in*, *out* and *in/out*. An *in* service has a server behaviour and receives results. An *out* service has a client behaviour and sends results. An *in/out* service has both a client and a server behaviour.

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

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

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

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

6.2.3.3 Valuations

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

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

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

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

    An example of this case is the *changeMaximumBid* service of the *BidderFunct* aspect (see Figure 29). *changeMaximumBid* service is executed by *BidderFunct* aspect when the Customer requests to change the maximum bid of a product. *changeMaximumBid* is of

kind *in* and the valuation assigns the *NewMaximumBid* input parameter to *lotMaximumBid* attribute.

.

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

End_Aspect BidderFunct;
```

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

- ***out***: The service is requested by the aspect.

  Valuation out*:* The postcondition must be satisfied after the service request.

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

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

- The service is provided and executed by the aspect, and the aspect sends the results of the service execution
  - ***Valuation in****:* The postcondition must be satisfied after the service execution. The output parameters of the service cannot be used as a right term of the valuation in any case, neither in the condition nor in the postcondition, due to the fact that the service has not

been executed and its output parameters do not have value. In fact, output parameters are usually used as a left term of the postcondition in order to satisfy the condition that requires output parameters to have a value after the service execution. must be satisfied after the service execution.

Figure 30 shows the specification of the *searchforlot* service of the *AuctHseFunct* aspect. The *searchforlot* service is provided and executed by the *AuctHseFunct* aspect, and the aspect sends the results of the service execution. In this case, a condition checks whether the *Keywords* input parameter matches with the value stored in the *lotDescrip* attribute of the *AuctFunct* aspect and whether the *Saleroom* input parameter is not equal to NULL. If the condition is satisfied, the postcondition of the valuation assigns the values of the output parameters with values stored in attributes of the *AuctHseFunct* aspect.

```
Functional Aspect AuctHseFunct using IProcurAuction, IAuctHseLot
Attributes
  Variables
    lotId: string NOT NULL;
    saleroom: string NOT NULL;
    saleNum: string NOT NULL;
    dateOfAuction: date NOT NULL;
    lotdescrip: string NOT NULL;
Services
 … ….
  in/out searchforLot(input Keywords:string, output LotId: string,
                      output Saleroom:string, output SaleNum:string,
                      output DateOfAuction:string,
                      output Lotdescrip:string)
      Valuations
        {(Keywords==lotDescrip) and (Saleroom!=NULL)}
                       [in searchforlot(Keywords, LotId
                                         SaleRoom, SaleNum,
                                         DateOfAuction, Lotdescrip)]
          LotId:= lotId, Saleroom := saleroom, SaleNum:= saleNum,
          DateOfAuction:= dateOfAuction, Lotdescrip:=lotdescrip;
     … …
   end();
End_Functional Aspect AuctHseFunct;
```

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

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

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

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

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
 Services
 … ….
  in/out searchforLot(input Keywords:string, output LotId: string,
                      output Saleroom:string, output SaleNum:string,
                      output DateOfAuction:date, output Lotdescrip: string)

    Valuations
     {DateOfAuction<= limitDate}[in searchforlot(Keywords, LotId, SaleRoom,
                                                 SaleNum, DateOfAuction,
                                                 Lotdescrip)]
          lotId:= LotId, saleroom := Saleroom, saleNum:= SaleNum,
          dateOfAuction:= DateOfauction, lotdescrip:=Lotdescrip;
    … …
   end();

End_Aspect ProcurFunct;
```

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

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

6.2.3.4 Preconditions

Preconditions establish the conditions that have to be satisfied in order to execute a service of an aspect. In the AOADL, a precondition is specified by indicating the service and the condition for its execution separated by the reserved word *if*. A precondition can be specified in the *ProcurFunct* aspect (see Figure 32). The precondition indicates that the *searchforlot* service cannot be executed unless the *DateOfAuction* parameter is before the *limitDate* attribute. This indicates that the procurement does not search for a lot that is auctioned after a specific date.

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc
… …
  Preconditions
   in searchforlot(Keywords, LotId, SaleRoom, SaleNum,
                  DateOfAuction,Lotdescrip)
                                   if (DateOfAuction<= limitDate);
… …
End_Functional Aspect ProcurFunct;
```

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

6.2.3.5 Constraints

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

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

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

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

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

6.2.3.6 Transactions

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

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

```
Functional Aspect CustFunct using ICustProc, ICustBidder

   Services      … …

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



    saveItem(input LotId:string, input SaleRoom:string,
            input SaleNum:string, input DateOfAuction:date,
            input Lotdescrip:string)
       Valuations
         {interested==true}[saveItem(LotId, SaleRoom, SaleNum,
                                DateOfAuction, LotNumber)]
                    iLotId:=LotId,iSaleRoom:=Saleroom, iSaleNum:=SaleNum,
                    iDateOfAuction:=DateAuction, iLotdescrip:=Lotdescrip;


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

     notifyProdInterest = setInterested?(CustomerInterested)→ SAVEITEM;
     SAVEITEM = saveItem?(SaleRoom, SaleNum, DateOfAuction,LotNumber);
         Valuations
           [in NOTIFYPRODINTEREST(LotId, SaleRoom, SaleNum, DateOfAuction,
                                 Lotdescrip, Interested)]
             Interested =interested;
….
End_Aspect CustFunct;
```

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

6.2.3.7 Played_Roles

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

In Figure 35, the played_roles of the *ProcurFunct* aspect are specified. *ProcurFunct* aspect has two played_roles: *CUSTPROC* and *PROCURAUCT*.

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

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc


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

End_Aspect ProcurFunct;
```

Figure 35. Specification of played_roles of *ProcurFunct aspect*

6.2.3.8 Protocols

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

Figure 36 shows the specification of the protocols of the *ProcurFunct* aspect. The protocols glue the begin service, the end service, the private services of the *ProcurFunct* aspect: *setKeepSearchingToTrue*, and *finishedSearchingWithoutResult*,

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

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc

Protocol

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

End_Functional Aspect ProcurFunct
```

Figure 36. Specification of protocols of *ProcurFunct aspect*

6.2.4 Simple Architectural Elements: Components and Connectors

An architectural element is specified by its set of ports, the aspects that form it, and the aspect weavings. An architectural element is represented in the PRISMA

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

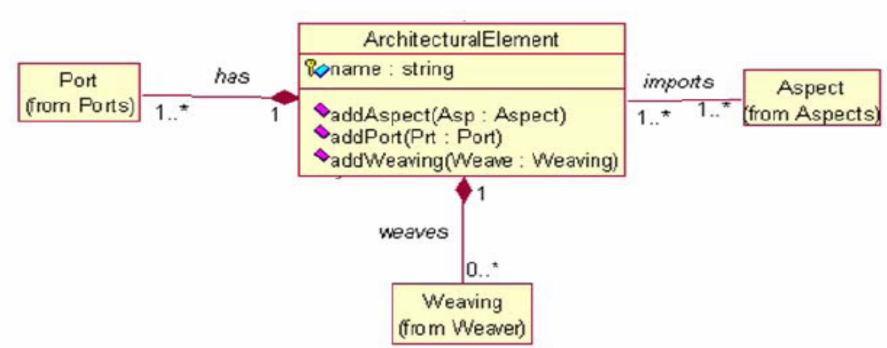


Figure 37. The package *ArchitecturalElements* of the PRISMA metamodel taken from [Per06b]

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

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

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

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

As it can be noticed from section 6.2.3 an aspect definition does not include any references to other aspects. This independence of the aspect specification from other aspects and weavings allows aspects to be reusable. In addition, the fact that the specification of weavings is inside architectural elements provides the flexibility of specifying different behaviours of an architectural element by importing the same aspects and defining different weavings. Therefore, architectural elements definitions import aspects and define the needed weavings.

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

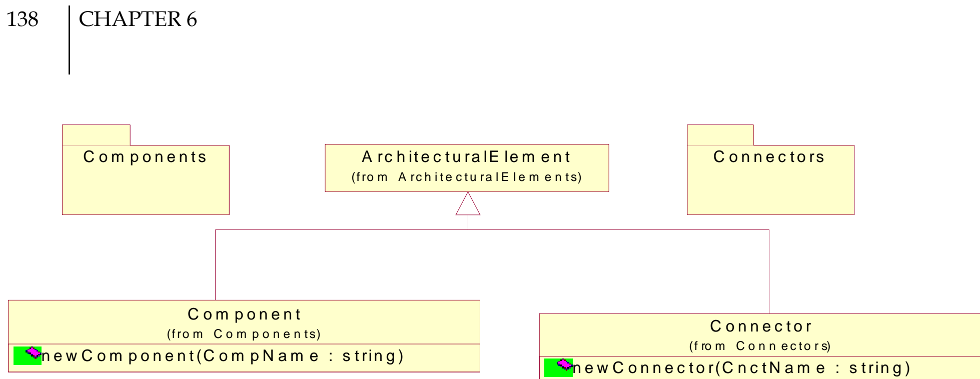


Figure 38. The package *KindsOfArchitecturalElements* of the PRISMA metamodel taken from [Per06b]

The details of the Connectors package of Figure 38 are shown in Figure 39. The *Connector* metaclass has an associated constraint that specifies that a connector must import an aspect whose concern is coordination. Moreover, the *Connector* metaclass has another constraint associated to it which specifies that a connector must have at least two attachments (see section 6.2.5.1) associated to it, and each attachment must connect the connector to two different components.

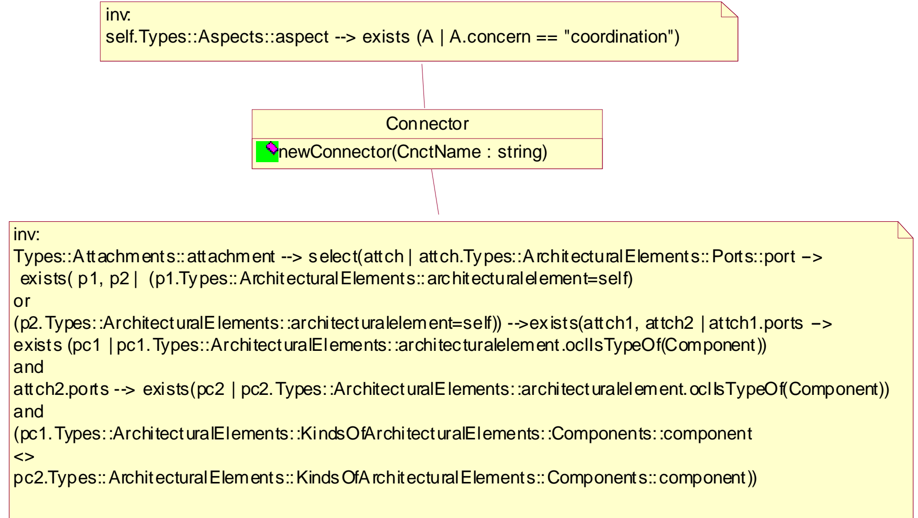


Figure 39. The package *Connectors* of the PRISMA metamodel taken from [Per06b]

An example of an architectural element specification is the *Customer* component specification (see Figure 40). A component specification starts with the reserved word *Component* and ends with the reserved word *End_Component*. The specification consists of a name (*Customer*), the importation of needed aspects (*CustDist*, *CustFunct*), the weavings among the aspects (a weaving between the *move* service of the *CustDist* aspect and the *biddingInf* service of the *CustFunct* aspect), a set of ports: *MOVEProcPort*, *MOVEBidderPort*, *CUSTPROCPort*, and *CUSTBIDDERPORT* and the two sections that allow the creation and destruction of the architectural element.

```
Component Customer

 Import Distribution Aspect CustDist;
 Import Functional Aspect CustFunct;
 Weavings
  CustDist.move(NewAmbient) after
                         CustFunct.biddingInf(LotId, SaleRoom, SaleNum,
                                              DateofAuction,MaximumBid);

 End_Weavings

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

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


End Component Customer;
```

Figure 40. Specification of the *Customer* component type

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

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

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

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

6.2.5 Connections

Connections are communication channels that connect ports of architectural elements. In PRISMA, there are two kinds of connections: Attachments and Bindings. In the following, these are explained.

6.2.5.1 Attachments

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

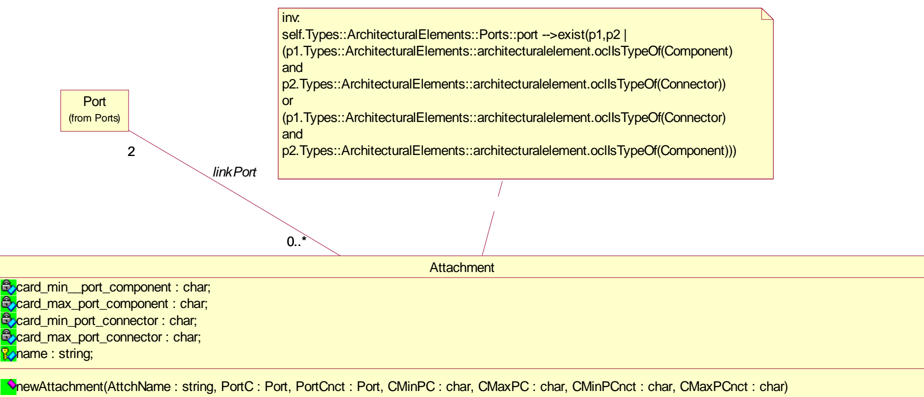


Figure 41. The package *Attachments* of the PRISMA metamodel taken from [Per06b]

Figure 42 specifies attachments of the *AuctionAgents* architectural model. A type of attachment called *AttchAuctCnct* is specified to connect the *CnctAuctPortBidder* port of *AuctionCnct* connector and the *BidderAuctPort* port of the *AuctionHouse* component (or system as shown in section 6.2.6). In addition, the minimum and maximum cardinalities of the attachment are specified. The *AuctionCnct.CnctAuctPortBidder(1,1)* means that only one instance of the *AttchAuctCnct* can be connected to the *CnctAuctPortBidder* port of a *AuctionHouseCnct* instance. The *AuctionHouse.BidderAuctPort(1,1)* means that only one instance of the *AttchAuctCnct* can be connected to the *BidderAuctPort* port of an *AuctionHouse* instance.

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

Figure 42. Specification of attachments in the agents auction architectural model

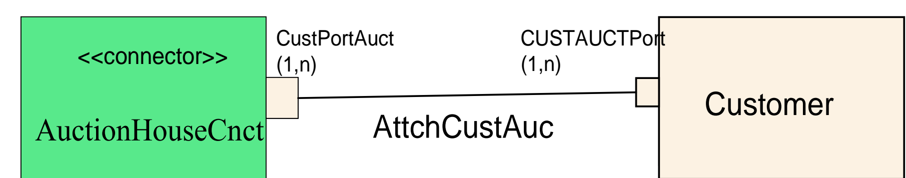


Figure 43. ***AttchCustAuct*** **attachment that connects** ***AuctionHouseCnct*** **connector and** ***Customer*** **component**

Another attachment specified in Figure 42 is the *AttchCustAuc* attachment type that connects the *CUSTAUCTPort* port of the *Customer* to the *CustPortAuct* port of the *AuctionHouseCnct*. Figure 43 shows the graphical notation of the specification of the *AttchCustAuc* attachment. It is also specified that at least one to many *(1,n)* *AttchCustAuc* attachment instances can be connected to the *CUSTAUCTPort* port and that at least one to many *(1,n)* *AttchCustAuc* attachment instances can be connected to the *CustPortAuct* port. Figure 44 shows a possible configuration that can be instantiated from the *AttchCustAuc* attachment type. The *AuctionCnct* instances called *eBayCnct* and *ChristiesCnct* have two *AttchCustAuc* instances connected to their *CustPortAuct* port. The *Customer* instances called *Client1* and *Client2* have two AttchCustAuc instances connected to their CUSTAUCTPort port.

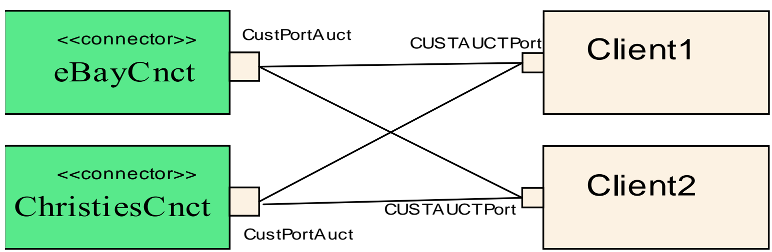


Figure 44. A possible configuration of ***AttchCustAuc*** **attachment that connects** ***AuctionCnct*** **and** ***Customer*** **instances**

6.2.5.2 Bindings

A binding is a connection that defines the composition between a complex component and one of its architectural elements, specifically, between a port of a

complex component and a port of one of its architectural elements. In PRISMA, complex components are called systems (see section 6.2.6).

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

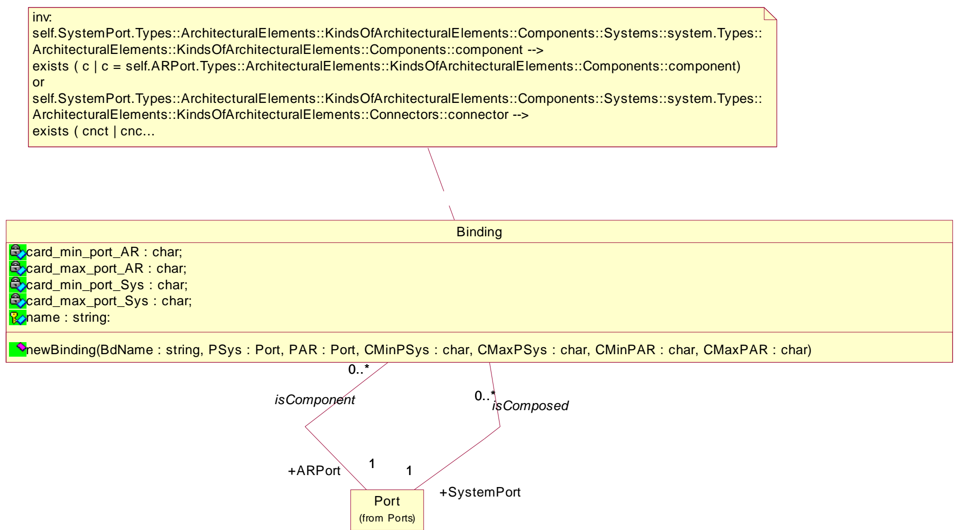
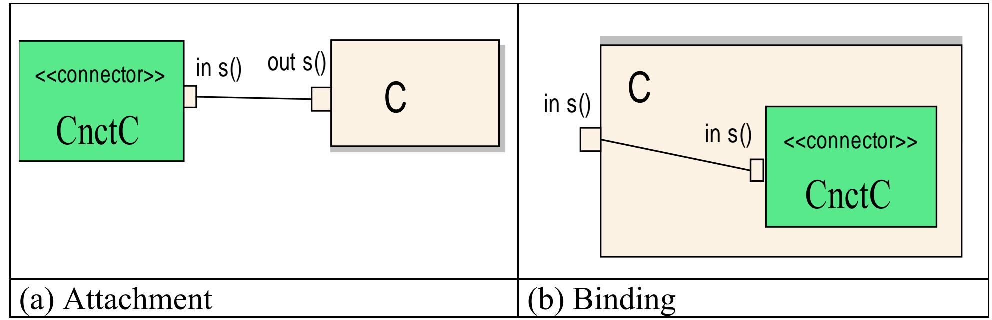


Figure 45. The package Bindings of the PRISMA metamodel taken from [Per06b]

Table 2 shows the difference between an attachment and a binding in terms of their functionality in sending and receiving services among the architectural element ports they connect. When an attachment connects a component *C* that requests a service *s* of kind *out* using its port, a connector *CnctC* receives an *in* invocation of the service *s* through its port (see Table 2(a)). When a binding connects a system *C* that

receives a service *s* invocation of kind *in* using its port, the connector *CnctC* receives also an *in* invocation of the service through its port.

Table 4. Comparison between Attachments and Bindings



(a) Attachment	(b) Binding

6.2.6 Systems

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

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

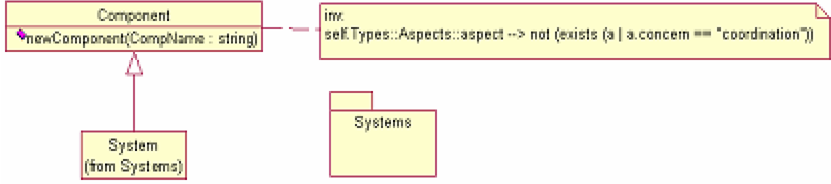


Figure 46. The package Systems of the PRISMA metamodel taken from [Per06b]

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

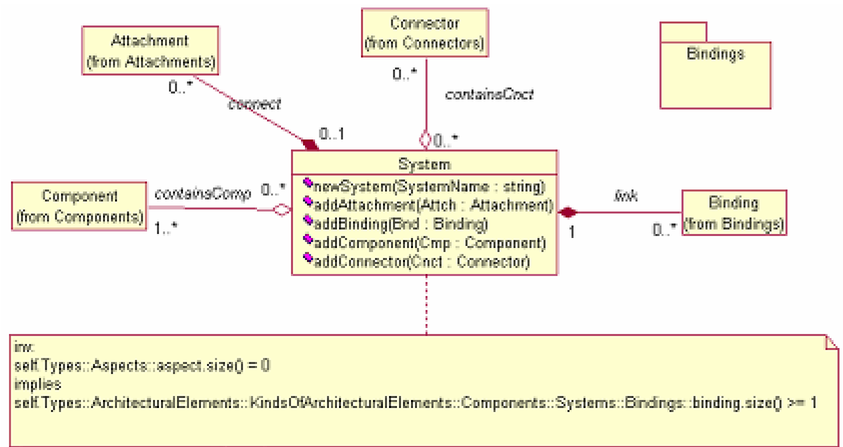


Figure 47. The Systems package of the PRISMA metamodel taken from [Per06b]

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

An example is the *AuctionHouse* system, which imports a functional aspect that allows the system to select the lots that can be candidate of the interest of a customer (see Figure 48.). It has two ports (*ProcurAuctPort* and *SysBidderPort*) to communicate with architectural elements that are not part of the system (see section *Ports,* Figure 48.). The *AuctionHouse* is composed of a *WrappAspSystem* component,

a *LotOnAuction* component and a *CnctLotAuction* connector that synchronizes them (see section *Import ArchitecturalElements,* System_type AuctionHouse

```
 Import Distribution Aspect Dist;
 Import Functional Aspect AuctHseFunct;

 Ports
     ProcurAuctPort: IProcurAuction Played_Role AuctHseFunct.PROCURAUCT;
     SysBidderPort: IBidderAuct Played_Role lotFunct.BIDDERAUCT;
 End_Ports

Import Architectural Elements WrappAspSystem, LotOnAuction, CnctLotAuction;


Attachments
 AttchAWrapCnct:
           CnctLotAuction.CoorWPort(1,1) ←→ WrappAspSystem.LotAuctPort(1,1);
 AttchCnctLot:
           CnctLotAuction.CoorLPort (1,n) ←→ LotOnAuction.LotAuctPort (1,1);
End_Attachements;

Bindings
  BndWrap: ProcurAuctPort(1,1)<--> WrappAspSystem.ProcurAuctPort(1,1);
  BndL: SysBidderPort(1,n)<--> LotOnAuction (1,1)
End_Bindings;


 new(input num_LotOnAuction: integer,input num_CnctLotAuction: integer,
      input num_AttchAWrapCnct: integer, input num_AttchCnctLot: integer,
      input num_ BndWrap: integer, input num_ BndL: integer)
  {
    new WrappAspSys();
    new LotOnAuction(input LotId: string, input Saleroom: string,
                     input SaleNum: string, input DateOfAuction: date,
                     input Lotdescrip: string,
                     input StbiddingAmount: double, input EndDate: date);

     new CnctLotAuction();
     new AttchAWrapCnct(input ArgCnctName: string, input ArgCnctPort: string,
                      input ArgCompName: string, input ArgCompPort: string);
     new AttchCnctLot (input ArgCnctName: string, input ArgCnctPort: string,
                      input ArgCompName: string, input ArgCompPort: string);
     new BndWrap (input ArgSysPort: integer, input ArgAEName: integer,
                  input ArgAEPort: integer);
    new BndL  (input ArgSysPort: integer, input ArgAEName: integer,
                  input ArgAEPort: integer);
```

```
 }
End System_type AuctionHouse;
```

Figure 48.). The *WrappAspSystem* component is a wrapper of the system's aspects that permit the aspects communicate with the architectural elements that it is composed of. The system has two attachments called *AttchAWrapCnct* and *AttchCnctLot*. The *AttchAWrapCnct* attachment allows the WrappAspSystem component to communicate with the *CnctLotAuction* connector. The *AttchCnctLot* attachment allows the *LotOnAuction* component to communicate with the *CnctLotAuction* connector. Also, two bindings called *BndWrap* and *BndL* have been defined in order to publish the services of the *ProcurAuctPort* port of the *WrappAspSystem* component and the services of the *LotAuctPort* port, respectively, to the exterior of the AuctionHouse (see the *Bindings* section of Figure 48).

```
System_type AuctionHouse
 Import Distribution Aspect Dist;
 Import Functional Aspect AuctHseFunct;

 Ports
     ProcurAuctPort: IProcurAuction Played_Role AuctHseFunct.PROCURAUCT;
     SysBidderPort: IBidderAuct Played_Role lotFunct.BIDDERAUCT;
 End_Ports

Import Architectural Elements WrappAspSystem, LotOnAuction, CnctLotAuction;


Attachments
 AttchAWrapCnct:
          CnctLotAuction.CoorWPort(1,1) ←→ WrappAspSystem.LotAuctPort(1,1);
 AttchCnctLot:
          CnctLotAuction.CoorLPort (1,n) ←→ LotOnAuction.LotAuctPort (1,1);
End_Attachements;

Bindings
  BndWrap: ProcurAuctPort(1,1)<--> WrappAspSystem.ProcurAuctPort(1,1);
  BndL: SysBidderPort(1,n)<--> LotOnAuction (1,1)
End_Bindings;


 new(input num_LotOnAuction: integer,input num_CnctLotAuction: integer,
      input num_AttchAWrapCnct: integer, input num_AttchCnctLot: integer,
      input num_ BndWrap: integer, input num_ BndL: integer)
  {
```

```
    new WrappAspSys();

    new LotOnAuction(input LotId: string, input Saleroom: string,
                     input SaleNum: string, input DateOfAuction: date,
                     input Lotdescrip: string,
                     input StbiddingAmount: double, input EndDate: date);

     new CnctLotAuction();

     new AttchAWrapCnct(input ArgCnctName: string, input ArgCnctPort: string,
                       input ArgCompName: string, input ArgCompPort: string);

     new AttchCnctLot (input ArgCnctName: string, input ArgCnctPort: string,
                       input ArgCompName: string, input ArgCompPort: string);

     new BndWrap (input ArgSysPort: integer, input ArgAEName: integer,
                   input ArgAEPort: integer);

    new BndL  (input ArgSysPort: integer, input ArgAEName: integer,
                   input ArgAEPort: integer);
 }
End System_type AuctionHouse;
```

Figure 48. Specification of the AuctionHouse system

After these definitions, the *new* service is specified (see the *new* section of Figure 48). The list of parameters of the *new* service consists of the following parameters: the parameters that provide the cardinalities of each architectural element, binding, and attachment of the system. These cardinalities allow the *AuctionHouse* to know how many instances of each type must be created at configuration time. Finally, the *destroy* service of the *AuctionHouse* is specified by invoking the *destroy* services of the architectural elements that it imports.

6.2.7 Configuration of an Architectural Model

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

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

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

Figure 49. Specification of the Auction Configuration

6.3 PRISMA Methodology

PRISMA proposes a methodology for modeling software architectures using the primitives it provides. The methodology is divided into three steps: detection of architectural elements, software architecture modelling, and code generation. The software architecture modelling consists of modelling: Interfaces, Aspects, Architectural Elements, Systems and Configurations (see Figure 50). The methodology is supported by modelling a software architecture using the PRISMA

AOADL which defines the architectural elements at two levels of abstraction: at the type definition level and at the configuration level. The type definition level defines architectural types which are stored in a repository in order to be reused by other types or specific architectures. The configuration level designs the topology of a specific architectural model by creating and interconnecting instances of the defined architectural elements in the type definition level.

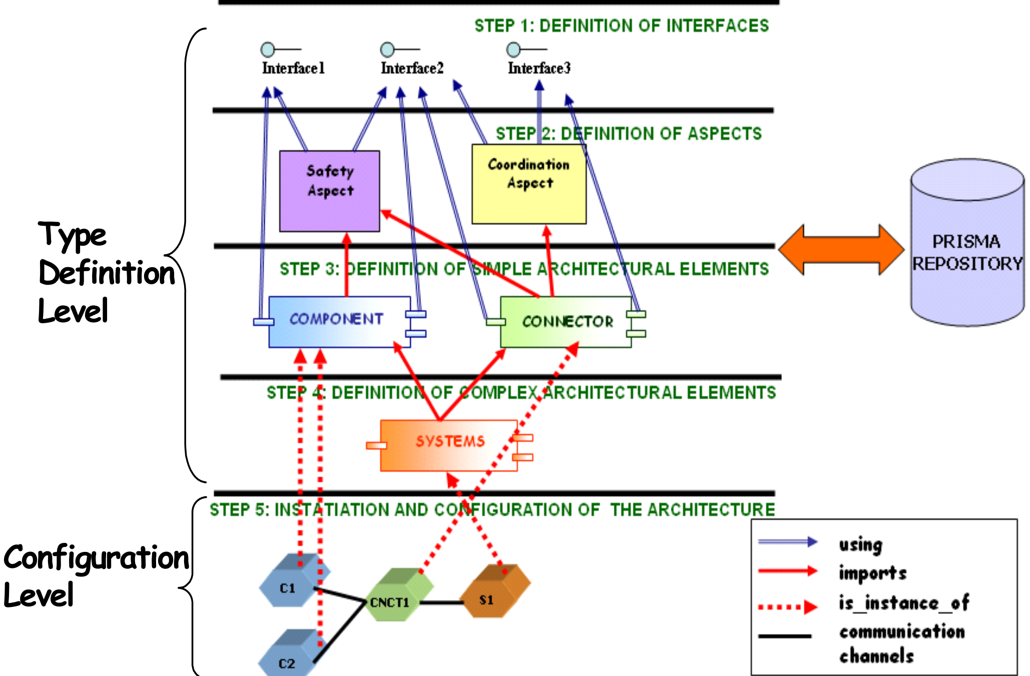


Figure 50. The methodology of the PRISMA approach

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

- STEP 1: Interfaces are the first to be specified due to the fact that it is not necessary to previously define other elements of the model. Interfaces are stored in a PRISMA repository for reuse.

- STEP 2: Aspects can use interfaces to define their behaviour. The same aspect can be used by many aspects. As a result, aspects are defined in the second step. Aspects are reusable entities that define a specific behaviour of a crosscutting concern. The number of aspects for the same concern is decided by the analyst, taking into account the software system and criteria such as reusability and/or understanding. An aspect for a concern or several aspects for the same concern can be defined. Aspects are reusable entities and are stored in the repository. As a result, aspects can be reused in different software architectures.
- STEP 3: The aspects that are defined in step 2 are imported by architectural elements. An architectural element imports the aspects that define the concerns that it requires. An aspect can be imported by many architectural elements (see steps 2 and 3 of Figure 50). Architectural elements types which need to be stored in a repository implies that the storage of their aspects.
- STEP 4: To completely specify a system, the architectural elements that the system is composed of should be previously defined. In addition, the communication channels that permit the communication among them are defined. Systems are defined as patterns or architectural styles that can be reused in any software architecture whenever they are needed. For this reason, they are stored in a repository.
- STEP 5: Architectural elements types that have been stored in a repository are instantiated. As a consequence, architectural element instances have the properties and behaviours of the aspects that their architectural elements types import. The attachments and bindings relationships are instantiated in order to connect the architectural element instances. In this way, a specific software architecture is designed in PRISMA.

It is important to keep in mind that the enumeration of these steps is not a restrictive order. The enumeration simply indicates the dependencies between the

different concepts that arise when the architectural model is being modelled. These dependencies are the following:

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

6.4 PRISMA Case Tool

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

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

PRISMA uses Domain-Specific Languages Tools (DSL Tools) [DSL07] as an integrated framework for developing the PRISMA CASE tool. Figure 51 shows the architecture of the PRISMA CASE and how the parts that compose it allow us to automatically generate executable C# code from architectural models that have been modelled using the modelling tool. This generation is possible thanks to the code generation templates (model compiler), which isolate the specification from the

source code preserving their independence. Until now, the tool generates PRISMA aspect-oriented C# code that is executable in .NET framework thanks to PRISMANET middleware , which gives support to aspect execution over .NET technology.

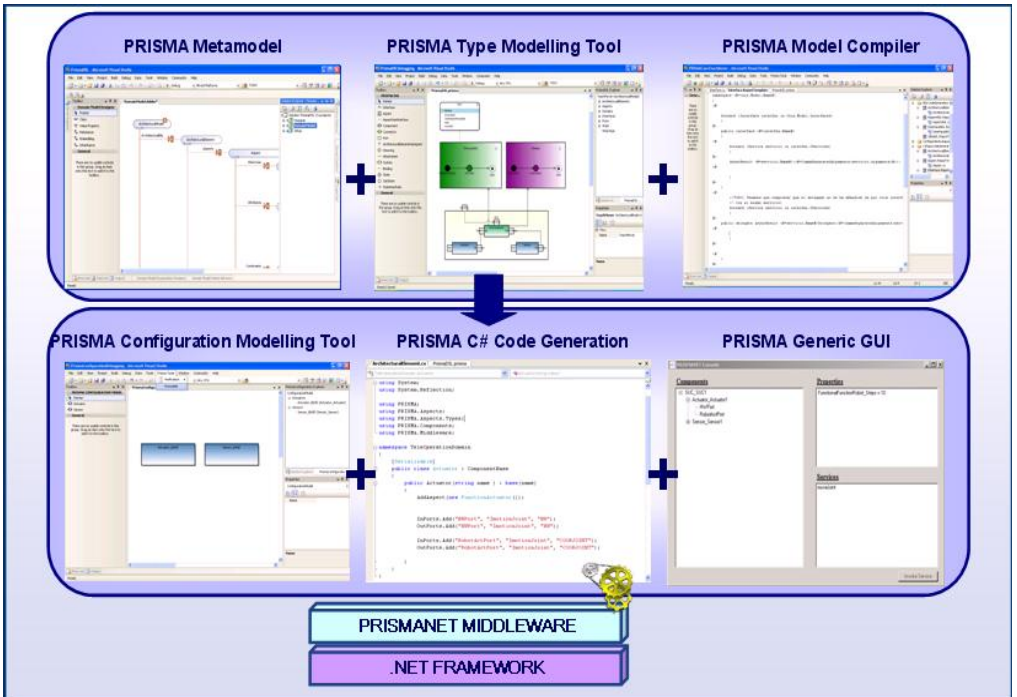


Figure 51. PRISMA CASE PARTS

The *PRISMA Type Modelling Tool* defines all the reusable types (interfaces, aspects, simple architectural elements and systems) and the architectural model of the software system are modelled in a graphical way by dragging and dropping the PRISMA modelling primitives. The *PRISMA Configuration Modelling Tool* models the configuration of the initial architecture of a specific system by instantiating the types and the architectural model that has been defined in the *PRISMA Type Modelling Tool*. Finally, the *PRISMA Model Compile* generates the code of the types and its instances. The code of the types is generated by executing the *PRISMA Model Compiler* from the *PRISMA Type Modelling Tool*, and the code of the instances is

generated by executing the *PRISMA Model Compiler* from the *PRISMA Configuration Modelling Tool*. Next, the execution of the generated code joint the PRISMANET middleware can be launched from the *PRISMA Configuration Modelling Tool* .Once the aspect-oriented software architecture is executed the user can interact with it using the *PRISMA Generic GUI,* which allows the user to execute services, query the value of attributes and validate the correct behaviour of each of the architectural elements that compose the architecture.

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

6.4.1 PRISMA Metamodel in DSL

DSL Tools provides the Domain Model Tool. The Domain Model tool provides a class diagram toolbox for allowing the user to define a domain model. The user represents concepts of its metamodel by using classes and relationships. In this way, each metaclass and relationship of the PRISMA metamodel has been introduced in the Domain Model (see Figure 52). Also, the OCL constraints of the metamodel (some have been presented in section 0) are implemented. These constraints have been implemented in order to check whether or not certain rules are satisfied during the modelling process.

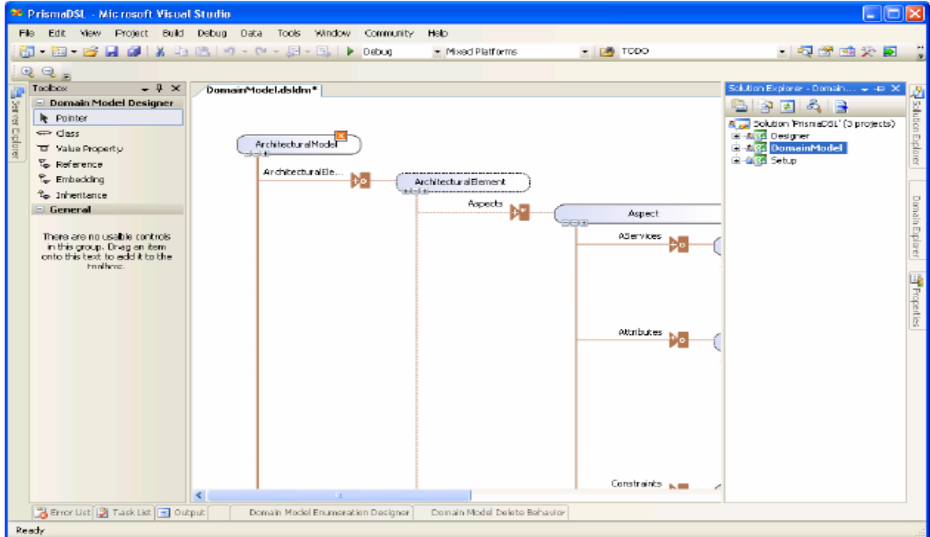


Figure 52. DSL Tools Framework: Domain Model of PRISMA taken from [Per06b]

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

6.4.2 PRISMA Modelling Tool

The PRISMA Modelling tool is supported by the *PRISMA Type Modelling Tool* and the *PRISMA Configuration Modelling Tool*. The *PRISMA Type Modelling Tool* is generated from the projects defined in section 6.4.1. The modelling tool is composed of a toolbox, a drawing sheet, a *model explorer,* a *window of properties* and a PRISMA menu (see Figure 53).

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

constraints are only checked when the model is saved or when requested by the user.

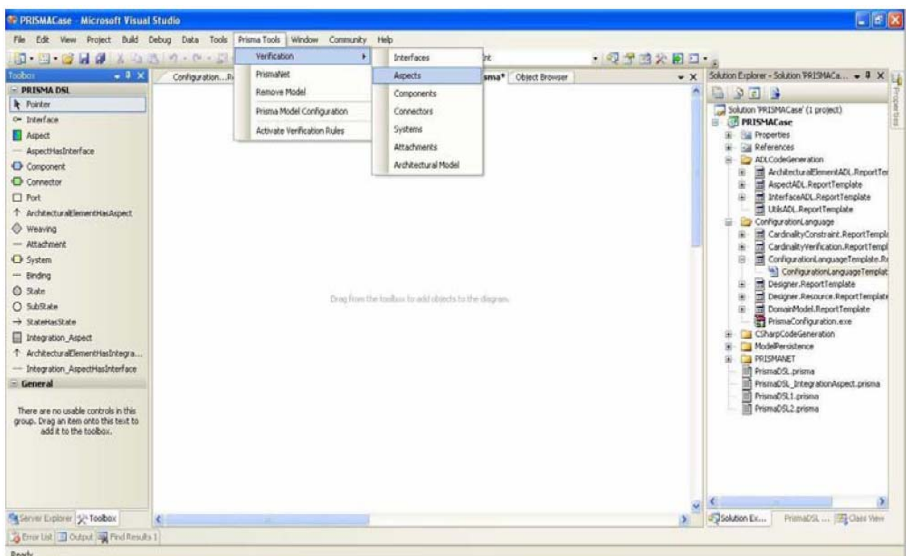


Figure 53. PRISMA Type Modelling Tool taken from [Per06b]

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

6.4.3 PRISMANET Middleware

The .NET platform does not directly support all the PRISMA primitives. As a result, a middleware called PRISMANET [Per05a] has been implemented in C# in order to provide PRISMA constructs in the .NET platform. PRISMANET is the

platform-dependent model of PRISMA in .NET, which permits PRISMA software architectures to be developed and executed on the .NET platform.

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

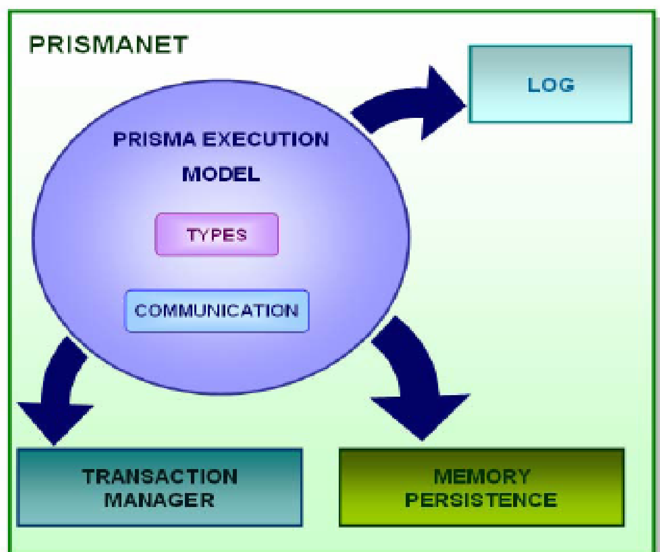


Figure 54. PRISMANET middleware architecure taken from [Per06b]

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

- **PRISMA Execution Model:** This module implements the basic functionality of the PRISMA types. This implementation is divided into two modules that contain the classes that implement this functionality: the Types and

Communications modules. The Types module implements aspects and architectural elements, and the Communications module implements attachments and bindings. As a result, the implementation of the PRISMA application is achieved by extending these classes. The Types and Communication modules have been grouped in namespaces as shown in Figure 55.

- **Memory Persistence**: This module provides services to manage and maintain the instances of architectural elements that are stored in the main memory during their execution. The middleware manages the instances that are locally executed. Some of these services are the loading of architectural elements instances, the creation of execution threads, and the management of architectural element lists.
- **Transaction Manager**: This module provides services to suitably execute transactions.
- **Log**: This module logs every operation that it is performed by the middleware in order to register the execution history of software architectures.

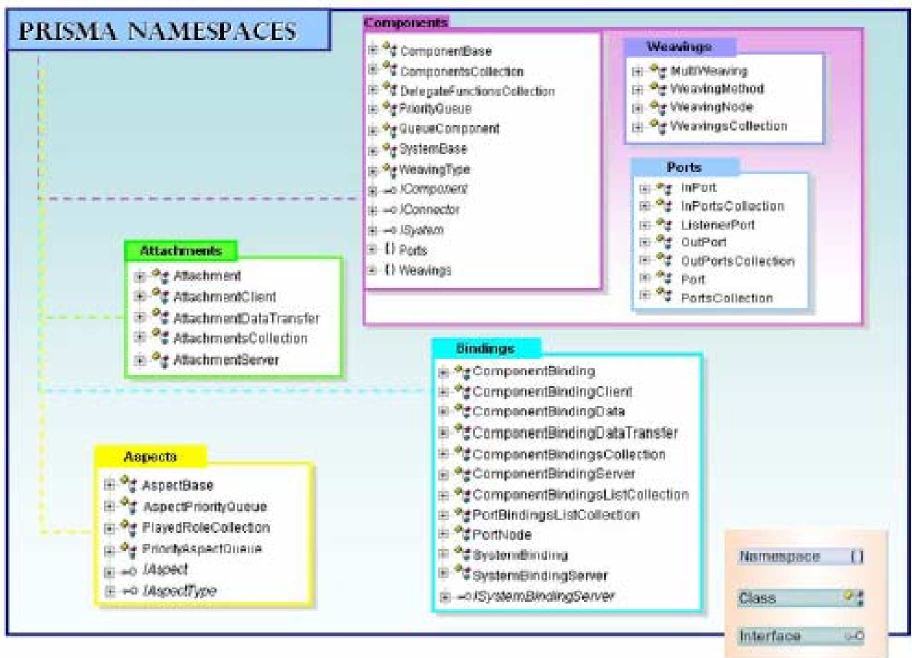


Figure 55. Namespaces of the module *PRISMA Execution Model* taken from [Per06b]

6.4.4 PRISMA Model Compiler

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

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

6.5 Conclusions

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

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

In this case, the metamodel has been introduced in DSL Tools in order to develop the PRISMA framework.

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

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

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

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

- Jennifer Perez, **Nour Ali**, Jose A. Carsi, I. Ramos, Barbara Alvarez, Pedro Sanchez, Juan A. Pastor, "Integrating Aspects in Software Architectures: PRISMA Applied to Robotic Tele-operated Systems", Information and Software Technology, 2007. (accepted)
- Cristobal Costa, **Nour Ali**, Jennifer Perez, Jose Angel Carsi, Isidro Ramos, "Dynamic Reconfiguration of Software Architectures through Aspects", First European Conference on Software Architecture (ECSA 2007)**.** LNCS, vol. 4758**,** pp. 279-283, Madrid, September, 2007 (poster).
- Jennifer Pérez**, Nour Ali**, Jose A. Carsí, Isidro Ramos, "Designing Software Architectures with an Aspect-Oriented Software Architectures", The 9th

International Symposium on Component-Based Software Engineering (CBSE), Lecture Notes Computer Science, Springer Verlang, LNCS 4063, pp. 123-138, ISBN: 0302-9743, Västeras, Sweden, June-July, 2006.

- Jennifer Pérez, **Nour Ali**, Jose A. Carsí, Isidro Ramos, "Dynamic Evolution in Aspect- Oriented Architectural Models", Second European Workshop on Software Architecture, Springer LNCS 3527, pp.59-16, ISSN: 0302-9743, ISBN: 3-540-26275-X , Pisa, Italy, June 2005.
- Mª Eugenia, **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí**, "**DIAGMED: An Architectural model for a Medical Diagnosis", IV workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 1-7, Archena, Murcia, November, 2005. (In Spanish)
- Rafael Cabedo, Jennifer Pérez, **Nour Ali**, Isidro Ramos, Jose A. Carsí, Aspect-Oriented C# Implementation of a Tele-Operated Robotic System, III Workshop on Aspect-Oriented Software Development (DSOA), X Conference on Software Engineering and Databases (JISBD), pp. 53-59, ISBN: 84-7723-670-4, Granada, September, 2005. (In Spanish)
- Cristóbal Costa, Jennifer Pérez, **Nour Ali**, Jose Angel Carsí, Isidro Ramos, "PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures"**,** X Conference on Software Engineering and Databases (JISBD), pp. 27-34, ISBN: 84-9732-434-X, Granada, September, 2005. (In Spanish)
- Jennifer Pérez, **Nour Ali** , Jose A. Carsí, Isidro Ramos, "PRISMA Architecture of the Robot 4U4 Case Study", Technical Report DSIC-II/13/04, pp. 72, Polytechnic University of Valencia, 2004. (In Spanish)
- **Nour H. Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsi, "Aspect Reusability in Software Architectures", The 8th International conference of Software Reuse (ICSR), July, 2004.(poster)
- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, "Development of a Tele-Operation System using the PRISMA Approach", VIII Conference on Software Engineering and Databases

(JISBD), pp. 411-420, ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)

- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Jose A. Carsí, "PRISMA: Aspect-Oriented and Component-Based Software Architectures", Workshop on Aspect-Oriented Software Development (DSOA), Conference on Software Engineering and Databases (JISBD), Technical Report TR-20/2003 of the Polytechnic School of the University of Extremadura, pp. 27-36, Alicante, November, 2003. (In Spanish)

PART IV. AMBIENT-PRISMA

CHAPTER 7

AMBIENT-PRISMA: CHARACTERISTICS AND METAMODEL

"It is part of morality not to be at home in one's home"

Edward W. Said

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

This chapter motivates and discusses details of this combination. It is structured as follows: Section 7.1 explains how the ambient concept has been introduced in PRISMA in order to define Ambient-PRISMA. Section 7.2, presents the characteristics that distinguish an ambient architectural element from other architectural elements. Section 7.3 enriches the PRISMA metamodel with the needed concepts for defining the Ambient-PRISMA metamodel. Finally, Section 7.4 presents conclusions of this chapter.

7.1 Introduction

Ambient-PRISMA is the result of integrating AC concepts in PRISMA for describing distributed and mobile systems. This integration is achieved by enriching PRISMA with an ambient construct.

In AC [Car98a], an ambient is defined as a bounded place where computation happens. The boundary separates what is in the ambient from what is not. In PRISMA, software architectures are described by means of architectural elements. A simple architectural element is either a component or a connector. Components are the entities that are responsible for capturing computations. Connectors are the entities that coordinate computations. Neither of them captures the notion of boundary or location that we could possibly use to capture what ambients are meant to provide.

For example, in the Mobile Agent Auction case study, the *Customer1* and the *Procurement1* are components and the *AgentCustCnct* connector coordinates them (see Figure 56). There is no primitive that represents where these components and connectors are located nor a primitive responsible for the way these move. The objective is to allow PRISMA components to be distributed across several locations and that PRISMA connectors coordinate components regardless of where they reside and they are not responsible for moving them around.

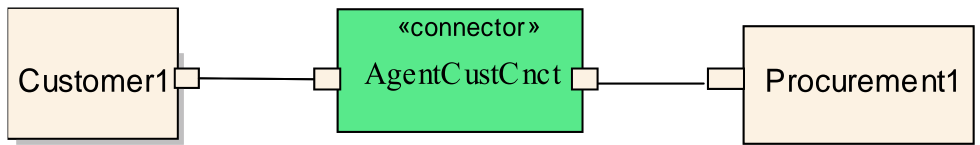


Figure 56. A Customer component and a Procurement component connected through a AgentCustCnct connector

In the following, the first-class entities of PRISMA (see CHAPTER 6) have to be reviewed in order to support the ambient concept as originally proposed by AC.

A PRISMA component has the following characteristics:

- It defines the logic of a software system.
- It cannot have a coordination aspect.
- A component can be composed by other components (Systems) in order to define its functionality.

An ambient, cannot be a component because it does not define the logic of the software system. An ambient has a concrete functionality which is the definition of a boundary that separates what is in this boundary from the outside. As a result, an ambient cannot define its functionality by being composed of other ambients. Therefore, an ambient is not a component. An ambient coordinates what is in its boundary from what is out and needs a coordination aspect to perform this coordination.

A PRISMA system has the following characteristics:

- It defines complex computations of a software system that needs other components to perform them.
- A PRISMA system cannot have a coordination aspect.
- Components or connectors that belong to a system can be directly connected to other connectors that do not form part of the system they belong to.

An ambient is an architectural element that needs to locate other architectural elements in its boundary, however the functionality of an ambient does not depend on the functionality of the elements it locates. Also, elements that are located in an ambient cannot be directly connected to elements of other ambients. Elements that need to be connected to elements of other ambients have to be managed by an ambient in order to control security issues.

A PRISMA connector has the following characteristics:

- It defines coordination mechanisms for the services among components (or systems).
- It needs a coordination aspect.

Ambients also coordinate elements that are contained in their boundary with elements that are outside them. As a result, an ambient needs coordination mechanisms in order to control what can enter or exit its boundary. However, in

PRISMA connectors cannot locate and manage other architectural elements and their connections.

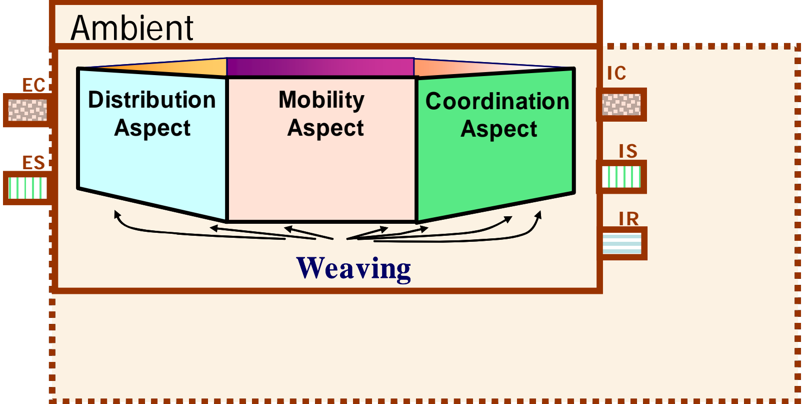


Figure 57. The graphical notation of an Ambient-PRISMA ambient

Therefore, ambients are introduced in PRISMA as a specialized kind of connector that is able to coordinate a boundary which models the notion of location and provides mobility support to other architectural elements. In this way, Ambient-PRISMA preserves the PRISMA connector which coordinates the logic of components and introduces ambients which coordinate components and connectors that are in different ambients for communication purposes. Also, ambients coordinate components and connectors with other ambients for mobility purposes.

Figure 57 shows the graphical representation of an ambient. Ambient-PRISMA ambients have a boundary, an AOSD view, and a CBSD view. A boundary of an ambient represents a place where PRISMA components, connectors, systems and their connections (attachments and bindings) are located. The boundary indicates what is in an ambient from what is out of it. As a result, mobility of an element is performed by crossing a boundary of an ambient and entering a boundary of another ambient. The boundary of an ambient is represented with a dotted line in order to indicate that the functionality of an ambient does not depend on the functionality of its internal architectural elements.

The AOSD view describes an ambient as an architectural element that is composed of different aspects that are weaved together. An ambient has at least three aspects: a coordination aspect, a mobility aspect and a distribution aspect. The mobility aspect of an ambient provides services for allowing architectural elements (including ambients) to move in and out of them. The coordination aspect receives calls from external architectural elements (distributed calls) and redirects them to corresponding internal architectural elements of the ambient. It also receives calls from its internal architectural elements and redirects them to the exterior. The distribution aspect of an ambient stores the name of its parent ambient in order to be aware of its location in an ambient hierarchy and allows internal architectural elements to consult routes of other architectural elements.

The Ambient CBSD view describes it as a black box where it communicates with other architectural elements by using ports through which invocations of services are sent and received. In order for the services of an ambient aspects to be offered, an ambient has at least five ports: two for the mobility aspect called InCapabilitiesPort, and ECapabilitiesPort, two for the coordination aspect called EServicesPort, and InServicesPort, and one for the distribution aspect called InRoutePort. As shown in Figure 57, the InCapabilitiesPort, and ECapabilitiesPort ports are designated by dots. The ECapabilitiesPort port, marked with *EC*, allows the ambient to offer and require the mobility aspect services outside the ambient. The InCapabilitiesPort port, marked with *IC*, allows an ambient to offer the mobility services to architectural elements inside the ambient. The EServicesPort, and InServicesPort ports are designated by stripes. These ports allow architectural elements that are inside the ambient to offer and request services to and from architectural elements outside the ambient. Therefore, the InServicesPort port, marked with *IS* in Figure 57, is for internal architectural elements and the EServicesPort port marked with *ES* is for architectural elements outside the ambient. The InRoutePort port, marked with *IR*, allows internal architectural elements to consult routes of any architectural elements.

In AC, an ambient can be nested within other ambients [Car98a]. Ambients in Ambient-PRISMA can be located inside the boundary of other ambients This allows Ambient-PRISMA software architectures to describe topologies of locations where hierarchies of distributed and mobile systems can be modelled. An ambient hierarchy forms a tree structure which has a root ambient. In Ambient-PRISMA, this root ambient must exist in each software architecture and it is called *Root*.

For example Figure 58 shows a hierarchy of ambients where the *Root* ambient locates two ambients called *ClientSite* and *AuctionSite*. The *ClientSite* ambient locates the *Customer1* component and the *AgentCustCnct* connector. The AuctionSite ambient locates the *Procurement1* component, the *AuctionHouseCnct1*, and the *AuctionHouse1* system.

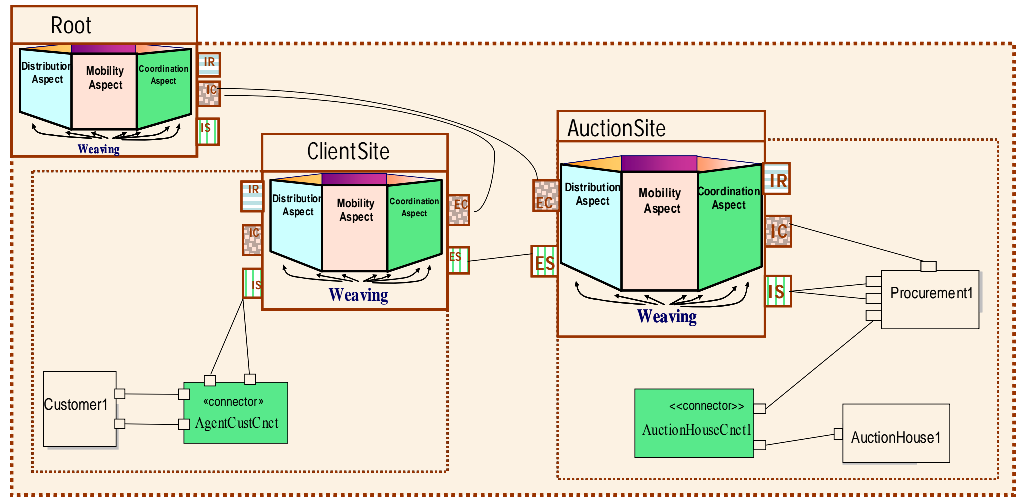


Figure 58. A possible Configuration of the Auction Mobile Agent Case Study where the Procurement1 component is in the AuctionSite ambient

An ambient is a connector because it coordinates architectural elements located in its boundary with the exterior. In this way, an ambient can control what can pass in or out of it. For example in Figure 58, when the *AgentCustCnct* connector sends a service request to the *Procurement1* component, the request has to be synchronized by the *ClientSite* ambient in order to allow it to be sent to the *AuctionSite* ambient,

and the *AuctionSite* ambient sends the service request to the *Procurement1* component. Also, if the *Procurement1* component needs to move from the *AuctionSite* ambient to the *Root* ambient, then the *AuctionSite* ambient has to coordinate the *Procurment1* component with the *Root* ambient in order to allow the *Root* ambient to accept the *Procurement1* component. In this way, the *Procurement1* component crosses the boundary of the *AuctionSite* ambient into the *Root* ambient.

Attachments connect ECapabilitiesPort port of ambients to the InCapabilities of their parent ambient. Attachments can also connect an EServicesPort port of an ambient to an EServicesPort port of its sibling ambient. For example, the ECapabilitiesPort port of the *AuctionSite* ambient is connected to the InCapabilities of the *Root* ambient (its parent ambient) through an attachment. In addition, the EServicesPort port of the *AuctionSite* ambient is connected to the EServicesPort port of the *ClientSite* ambient (its sibling).

Attachments also connect ports of architectural elements located in an ambient to the InCapabilities port or the InRoutePort port of their parent ambient. For example in Figure 58, the *Procurement1* component has a port connected to the InCapabilitiesPort port of the *AuctionSite* ambient because the *Procurment1* component is a mobile architectural element.

Elements of different ambients do not have direct connections (attachments or bindings) between each other. An attachment or a binding only connects elements of the same ambient. For example in Figure 58, an attachment between a port of the *ClientSite* ambient and a port of the *AuctionSite* ambient is possible because the *ClientSite* and the *AuctionSite* are siblings (they share the same parent ambient). An attachment is also possible between the *Customer1* component and the *AgentCustCnct* connector because both of them are in the *ClientSite* ambient. However, the *Procurement1* component does not have an attachment with the *AgentCustCnct* connector. The connection between the *Procurement1* component and the *AgentCustCnct* connector is performed by three attachments: an attachment

between *Procuremet1* and *AuctionSite*, an attachment between *AuctionSite* and *ClientSite*, and an attachment between *ClientSite* and *AgentCustCnct*.

When an architectural element moves from an ambient to another, the attachments or bindings associated to an ambient are reconfigured in order to provide the mobile architectural element with the services it needs. For example when the *Procurement1* component in Figure 58 moves from the *AuctionSite* ambient to the *ClientSite* ambient, the attachments associated to the *Procurement1* are reconfigured (see Figure 59): the attachments that connect the *Procurement1* to the InCapabilities and the InServicesPort port of the *AuctionSite* ambient are removed, the attachment that connected the *Procurement1* component to the *AuctionHouseCnct1* connector is also removed, attachments between the *Procurement1* and the InServicesPort and the InCapabilities ports of the *ClientSite* are created, an attachment that connects the *AuctionHouseCnct1* connector and the InServicesPort of the *AuctionSite* ambient is created, and two attachments that connect the *Procurement1* to the *AgentCustCnct* is created.

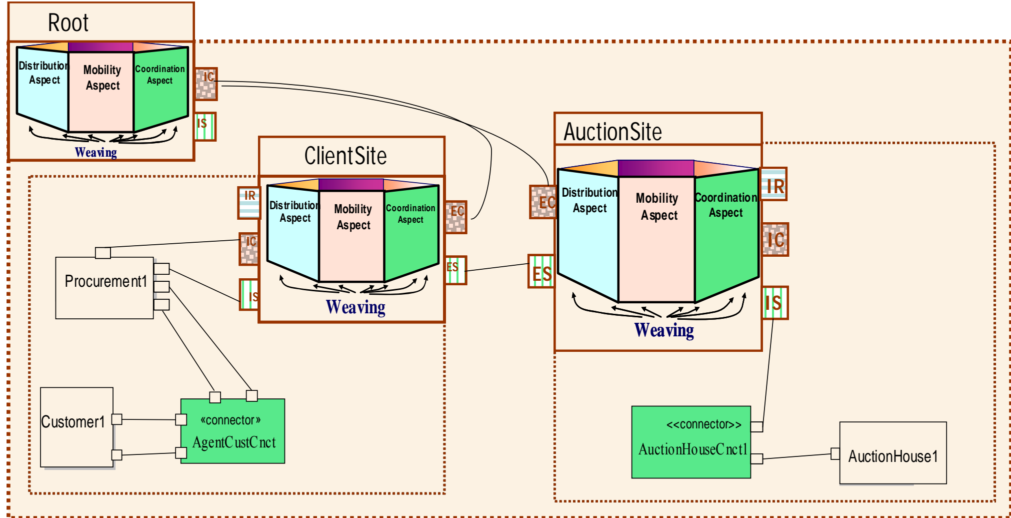


Figure 59. Procurement1 component in the ClientSite ambient

7.2 Characteristics of Ambients in Ambient-PRISMA

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

7.2.1 Interfaces

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

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

7.2.1.1 ICall Interface

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

The *ICall* interface publishes a service called *call* (see Figure 60). Each invocation of a *call* service stores in its parameters the signature of a service requested or provided by one of the architectural elements located in an ambient. The call service

has three parameters: *Name* which stores the name of the original service, *ParamsList* which is a list that stores the parameters of the original service and *ParamType* which is a list that stores whether the parameters of the original service are input or output.

```
Interface ICall

   call(input Name: Service, input ParamsList[]: Parameter,
        input ParamType[]: ParameterKind);

End_Interface ICall
```

Figure 60. Specification of the ICall Interface

The *call* service allows an ambient to process a service provided or required by its architectural element to or from external architectural elements. For example, the *Procurment1* component of Figure 58 requires the *notifyProdInterest(**input** LotId, **input** Saleroom:string, **input** SaleNum:string, **input** DateOfAuction:date, **input** Lotdescrip:string, **output** Interested: boolean);* service from the *Customer1* component through the *AgentCustCnctr* connector. As a result, the *AuctionSite* ambient receives the *notifyProdInterest* service transformed into a call invocation where the Name parameter stores the value *notifyProdInterest*, the *ParamsList* stores [*Saleroom, SaleNum, DateOfAuction, Lotdescrip, Interested*], and the *ParamType* stores [true, true, true, true, false].

7.2.1.2 ICapability Interface

An ambient offers a set of services for allowing architectural elements located in it to be able to move. The interface that publishes the set of services for allowing architectural elements located in it to move is called *ICapability* (see Figure 61). Some of these services are *exit*, *enter*, and *finishMovement*. For example, the *exit* service has an input parameter called *Name*. The *Name* parameter indicates the name of the architectural element that needs to exit. The *enter* service has two input parameters: *Name* and *NewAmbient*. The *Name* parameter indicates the name of the

architectural element that needs to enter another ambient. The *NewAmbient* parameter indicates the name of the ambient that an architectural element needs to enter to. The *finishMovement* service has two input parameters: *Name* and *CommuniList[]*. The Name parameter indicates the name of the architectural element that has finished moving. The *CommuniList* parameter indicates the list of connections (attachments and bindings) that the architectural element has associated.

```
Interface ICapability

   ….
   exit (input Name: ArchitecturalElement);
   enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
   finishMovement(input Name: ArchitecturalElement,
                  input CommuniList[]:string);

    …
End_Interface ICapability
```

Figure 61. Partial specification of the ICapability Interface

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

7.2.1.3 IGetLocation Interface

An ambient publishes a service called *getLocation* through an interface called *IGetLocation*. This service allows an ambient to inform the name of its parent ambient (see Figure 62). The *getLocation* service has one parameter called *Location*. The *Location* parameter is an output parameter that returns the name of the parent ambient.

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

Figure 62. Specification of the IGetLocation Interface

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

7.2.1.4 IRoute Interface

An ambient publishes a service called *getRoute* through an interface called *IRoute* (see Figure 63). This service allows an ambient to provide its internal architectural elements for consulting a route of any architectural element of a software architecture. The *getRoute* service has two parameters called *Name* and *Route[]*. The *Name* parameter is an input parameter that indicates the name of an architectural element. The *Route* is an output parameter which returns a list with the route of an architectural element.

```
Interface IRoute
 getRoute(input Name: ArchitecturalElement, output Route[]: Ambient);
End_Interface IRoute
```

Figure 63. Specification of the IRoute Interface

For example, the *Procurement1* component of Figure 58 can invoke the *getRoute* service to the *AuctionSite* ambient with the value "Procurement1" stored in the *Name* parameter. As a result, *AuctionSite* returns the list [Root, AuctionSite] to the Procurement1 component.

7.2.2 Aspects

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

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

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

- **Distribution Aspect**: This aspect allows an ambient to be aware of its parent, except for the Root ambient which does not have one. It also specifies whether the ambient is mobile or not, and if it is, the distribution aspect can invoke primitives offered by the *MobilityAspect* aspect of parent ambients in order to define mobility behaviours.

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

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

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

7.2.2.1 ACoordination Coordination Aspect

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

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

Since all services of the internal and external elements of an ambient must pass through its *ACoordination* aspect, an ambient must control all the interfaces of its internal elements. Therefore, the *ACoordination* aspect uses the *ICall* interface which is specified in Figure 60 in order not to change its interface each time an architectural element exits or enters an ambient. The *ACoordination* aspect specifies the behaviour of the call service (see Figure 64). It specifies that the *call* service is an in/out i.e. it is required and provided. The *ACoordination* coordination aspect is a generic aspect that is reused by all ambients.

```
Coordination Aspect ACoordination using ICall

Services
   …
   in/out call(input Name: Service, input ParamsList[]: Parameter,
               input ParamType[]: ParameterKind);

   …
```

Figure 64. Partial specification of the ACoordination Aspect

7.2.2.2 MobilityAspect Mobility Aspect

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

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

- exit: architectural elements that need to exit an ambient must invoke the exit service from their parent ambient (AC exit capability).
- enter: architectural elements that need to enter an ambient must invoke the enter service from their parent ambient (AC enter capability).
- accept: ambients that need to release an architectural element must invoke the accept service from the ambient that receives the architectural element.
- copy: architectural elements that need to be replicated must invoke the copy service from their parent ambient. (The specification of the replication).

- startMovement: architectural elements that need to make a sequence of exits or enters must invoke the startMovement service. This service allows an ambient to prepare an architectural element to move.
- finishMovement: architectural elements that finish making a sequence of exits or enters must invoke the finishMovement service. This service provides an ambient to reconfigure attachments once a mobile architectural element reaches destination.
- changeLocation: an ambient invokes the changeLocation service of an architectural element in order to change the value of their location when an architectural element is moved.

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

The *MobilityAspect* aspect uses the *ICapability* interface (see Figure 61) in order to specify the behaviour of the services which it publishes (see Figure 65). Most of the services that the *MobilityAspect* aspect provides are transactional services of reconfiguration, i.e., services that change the configuration of the software architecture.

For example, the *MobiltyAspect* aspect specifies that the *exit* service is a transactional service (see Figure 65). It specifies that the exit transaction is of kind ***in*** i.e., it is provided. The transaction checks whether the architectural element that needs to exit is a child of the ambient that imports the aspect or not. This is checked through the *isChild* service. If the requested architectural element is a child of the ambient, then the *ischild* service returns true through the *isChildOK* output parameter. If *ischild* returns true, then the transaction proceeds to execute *getParent*

service. The *getParent* service returns the name of the parent ambient of the *MobilityAspect* ambient through the *Parent* output parameter. This is needed in order to indicate the destination of the exiting architectural element. Then, the moving process is performed through the *moving* service. The *moving* service has two parameters: *Requested* and *Parent*. The *Requested* parameter indicates the name of the exiting (or moving) architectural element and the *Parent* parameter indicates the destination of the moving architectural element.

```
Mobility Aspect MobilityAspect using ICapability

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

End_Mobility Aspect MobilityAspect
```

Figure 65. Partial specification of the *MobilityAspect* aspect

The *moving* service is a transactional service that consists of removing an element from the boundaries of an ambient, removing attachments between the element and the ambient, removing attachments between the element and the elements of the same ambient and creating attachments between elements that were connected to the element and the ambient.

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

7.2.2.3 Distribution Aspect

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

Each distribution aspect must save the name of the parent ambient of the ambient that imports it. Also, it provides services for allowing other aspects of the ambient or internal architectural elements of the ambient to consult locations and routes. Therefore, a distribution aspect uses the *IGetLocation* interface specified in Figure 62 and the *IGetRoute* interface specified in Figure 63.

Figure 66 shows the specification of a possible distribution aspect of an ambient called *ADist*. The *ADist* aspect uses both the *IGetLocation* and the *IGetRoute* interfaces. The *ADist* stores the name of the parent ambient of the ambient that imports it through the *location* attribute. The location attribute is marked in bold because it is a keyword of Ambient-PRISMA. The *ADist* aspect specifies that the *getLocation* service and the *getRoute* service are of kind in/out i.e., first the aspect receives the request (*in*), then the *Valuation* assigns the value of the output parameter and finally the service sends the value of the output parameter (*out*) to the requester.

```
Distribution Aspect ADist using IGetLocation, IGetRoute
 Attributes
  Variable
   location : Ambient NOT NULL;
  …

 Services
```

```
  …
  in/out getLocation( output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;

  in/out getRoute(input Name: ArchitecturalElement, output Route[]: Ambient)
Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);

  ….
End_Distribution Aspect ADist
```

Figure 66. Partial specification of an aspect of kind Distribution

Although an ambient must have a distribution aspect, and must contain at least the specification shown in Figure 66, a distribution aspect is not a generic aspect that is predefined. In this way, a distribution aspect is defined differently depending on the systems requirements. For example, the distribution aspect of the *ClientSite* and the *AuctionSite* save the name of their parent ambient. Also, these two ambients are not mobile ambients. Therefore, both ambients can reuse the same distribution aspect. However, an ambient that needs to move, such as *MobileAmb* ambient shown in Figure 67, cannot reuse the same distribution aspect as *ClientSite*. As a result, a distribution aspect which the *MobileAmb* ambient imports has to specify mobility strategies.

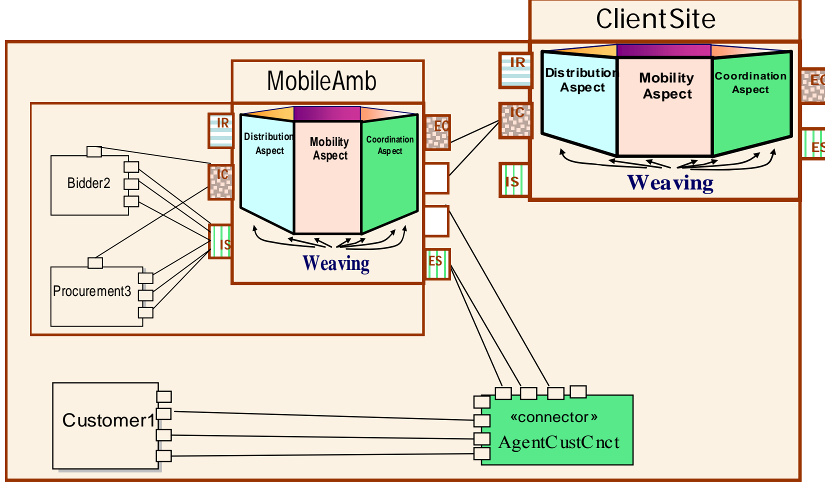


Figure 67. A mobile ambient called MobileAmb

7.2.3 Weavings

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

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

The predefined weaving of each ambient specifies that the *getParent* service of the *MobilityAspect* aspect triggers the *getLocation* service of a distribution aspect (see Figure 68). The *getLocation* service is executed instead of the *getParent* service.

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

Figure 68. Predefined weaving between the MobilityAspect aspect and a distribution aspect called ADist

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

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

7.2.4 Ports

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

As a consequence, each ambient must have at least five ports: two for the *MobilityAspect* aspect and two for the *ACoordination* aspect, and one for a distribution aspect. The two ports for providing/requesting services of the *MobilityAspect* are *InCapabilitiesPort* port and *ECapabilitiesPort* port. The *InCapabilitiesPort* port offers the mobility services of the parent ambient to its architectural elements (Marked with IC in Figure 58 and Figure 67). The *ECapabilitiesPort* port allows an ambient to request the accept service from its parent ambient and receive accept requests from its parent ambient or a sibling ambient (Marked with EC in Figure 58 and Figure 67).

The two ports for providing/requesting services of the *ACoordination* aspect are *InServicesPort* port and *EServicesPort* port. These ports allow architectural elements of an ambient to be in contact with the exterior boundary of an ambient. The port of an ambient that is marked with IS in Figure 58 and Figure 67, is the *InServicesPort* port and it is connected to internal architectural elements. The port of an ambient that is marked with ES in Figure 58 and Figure 67, is the EServicesPort port and it is connected to exterior architectural elements.

The predefined port for a distribution aspect of an ambient is called InRoutePort port, marked with *IR* in Figure 58 and Figure 67. The InRoutePort offer the *getRoute* service to internal architectural elements of an ambient.

Figure 69 shows the specification of the ports section in an ambient specification. It can be observed that the *InCapabilitiesPort* port and the *ECapabilitiesPort* are of type *ICapability* interface i.e., they publish the services of the *ICapability* interface. The *EServicesPort* port and the *InServicesPort* port are of type *ICall* interface, and the *InRoutePort* port is of type *IRoute* interface.

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

Figure 69. Partial specification of ambients ports

7.2.5 Attachments

In PRISMA, an attachment is a communication channel that connects a port of a component to a port of a connector. In Ambient-PRISMA, attachments are extended in order to permit ambients ports to be connected to a port of a component, connector, system or another ambient.

An architectural element in an ambient cannot be directly connected through an attachment to another architectural element in another ambient. An attachment that connects two architectural elements in different ambients is broken down into many attachments. For example in Figure 70, the *Procurement1* component and the *AgentCustCnctr* connector have an attachment that connects them in order to communicate. This attachment is broken down into an attachment that connects the *InServicesPort* port of the *AuctionSite* ambient and a port of the *Procurement1*, an attachment that connects the *EServicesPort* port of the *AuctionSite* ambient and the *EServicesPort* port of the *ClientSite* ambient, and an attachment that connects the *AgentCustCnctr* connector to the *InServicesPort* port of the *ClientSite* ambient. When the *Procurement1* component moves from the *AuctionSite* ambient to the *ClientSite* ambient (see Figure 71), an attachment between the *Procurment1* component and the *AgentCustCnct* connector directly connects them.

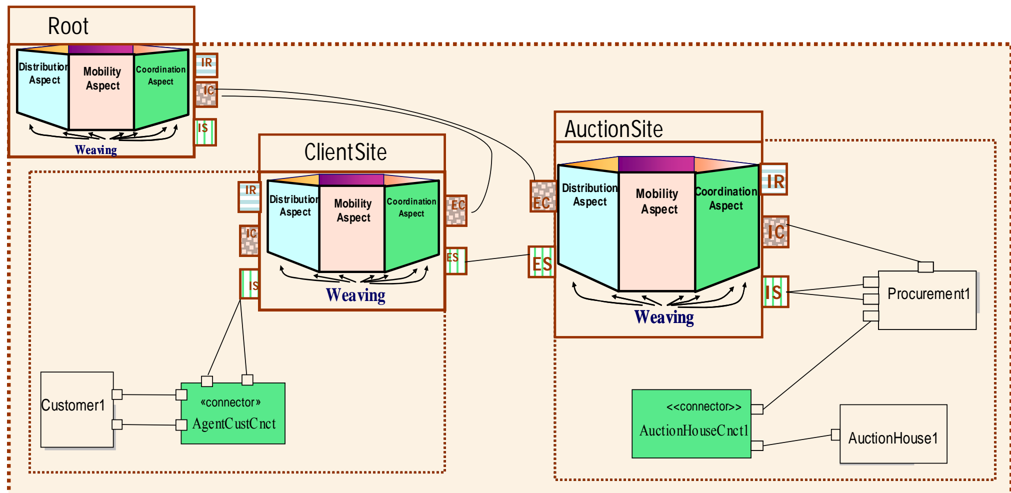


Figure 70. Procurement1 component in the AuctionSite ambient

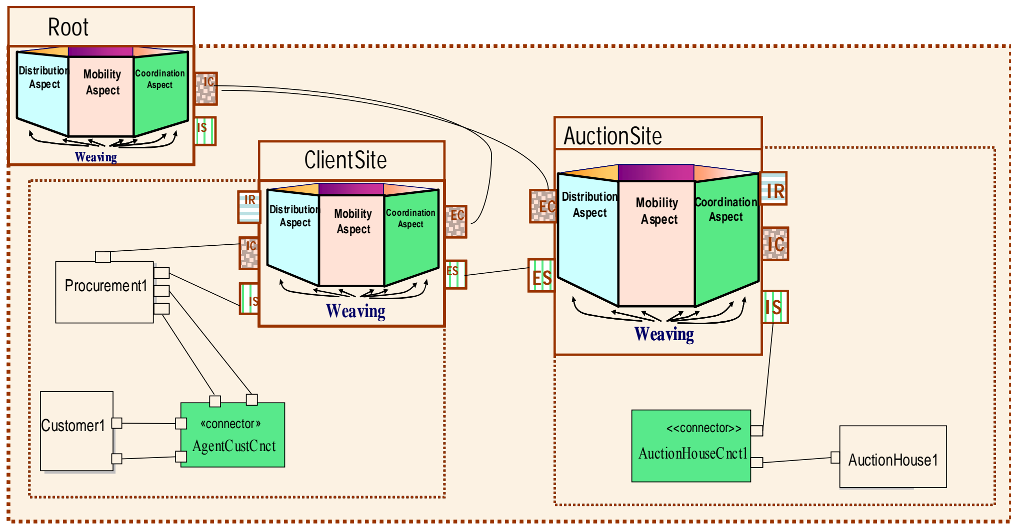


Figure 71. Procurement1 component in the ClientSite ambient

Since an ambient has five predefined ports, each port can have a type of attachment associated to it. In the following, these are explained:

- An attachment that connects the InCapabilitiesPort port of an ambient to a port called DCapPort port of a mobile architectural element located in it.

This attachment connects a mobile architectural element to its parent ambient in order to request mobility services. The mobile architectural element defines a port called DCapPort that is attached to the InCapabilitiesPort port of its parent ambient. When a mobile architectural element moves from an ambient to another, this attachment is deleted from the origin ambient and an attachment is created in order to connect the DCapPort port of the architectural element to the InCapabilitiesPort port of the new ambient. For example, when the *Procurement1* component is located in the *AuctionSite* ambient, the *Procurement1* component is connected to the *InCapabilities* port of the *AuctionSite* ambient (see Figure 70). However, when the *Procurement1* component moves to the *ClientSite* ambient, the *Procurement1* component becomes connected to the *InCapabilitiesPort* port of the *ClientSite* ambient (see Figure 71). As a result, the *Procurement1* component can always request mobility services because it is always connected to the *InCapabilitiesPort* port of its parent ambient by an attachment.

- An attachment that connects the InRoutePort port of an ambient to a port called DRoutePort port of an architectural element located in it.

This attachment connects an architectural element to its parent ambient in order to consult routes of any architectural elements. The architectural element that needs to consult routes defines a port called DRoutePort that is attached to the InRoutePort port of its parent ambient. When a mobile architectural element moves from an ambient to another, this attachment is deleted from the origin ambient and an attachment is created in order to connect the DRoutePort port of the architectural element to the InRoutePort port of the new ambient.

- An attachment that connects the InCapabilitiesPort port of an ambient to a port called RCapPort of a mobile architectural element located in it.

This attachment connects an architectural element that needs to be copied to its parent ambient. The architectural element defines a port called RCapPort that is attached to the InCapabilitiesPort port of its parent ambient. When a mobile

architectural element moves from an ambient to another, this attachment is deleted from the origin ambient and an attachment is created in order to connect the RCapPort port of the architectural element to the InCapabilitiesPort port of the new ambient.

- An attachment that connects the ECapabilitiesPort port of an ambient to the InCapabilities port of its parent ambient.

This attachment connects an ambient to its parent ambient because both need to collaborate for supporting mobility of architectural elements located in them. This attachment always exists between an ambient its parent. For example in Figure 70 and Figure 71, the ECapabilitiesPort ports of the AuctionSite ambient and the ClientSite ambient are always connected to the InCapabilitiesPort port of the Root ambient (their parent ambient).

- An attachment that connects the EServicesPort port of an ambient to the EServicesPort port of a sibling ambient.

This attachment is created between sibling ambients for communication purposes of their internal architectural elements. When two ambients do not have any internal architectural elements that need to be connected this attachment is not needed. For example in Figure 72, the EServicesPort port of the *ClientSite* ambient and the EServicesPort port of the *AuctionSite* ambient are connected through an attachment. This attachment is created in order to provide a connection between the *Procurement1* component and the *AgentCustCnct* connector. This is possible because *ClientSite* and *AuctionSite* are siblings.

- An attachment that connects the InServicesPort port of an ambient to a port of an architectural element located in it.

This attachment connects a port of an architectural element to the InServicesPort port of an ambient. This attachment transforms a service signature received to or sent by an architectural element to a *call* service in order to allow be

processed by an ambient. For example Figure 72 shows the operation of this attachment when the *Procurement1* component requests the *notifyProdInterest* service to the *AgentCustCnt* connector. The *AuctionSite* ambient receives a call of kind in (*?*) through its InServicesPort port thanks to the attachment that connects the port of the Procurement1 component to the InServicesPort port of the AuctionSite ambient. In addition, the *AgentCustCnct* connector receives a *notifyProdInterest* of kind out (*!*) through its port thanks to the attachment that connects the port of the *AgentCustCnct* connector to the InServicesPort port of the *ClientSite* ambient.

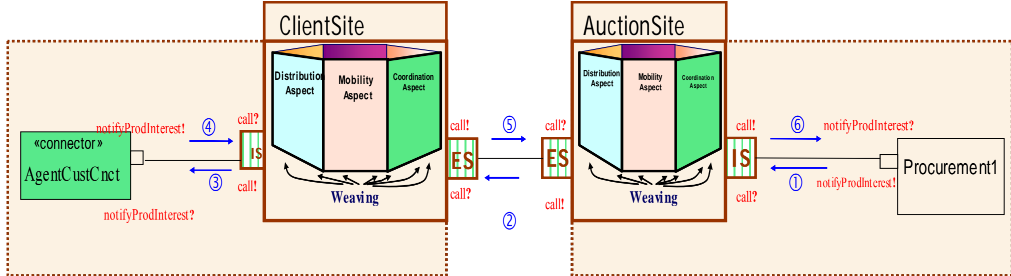


Figure 72. Operation of the attachments associated to the InServicesPort ports and the EServicesPorts of ambients

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

7.2.6 Bindings

A binding in PRISMA is a connection between a port of a system and a port of one of the architectural elements that compose it. A system architectural element can be distributed .i.e., its architectural elements can be located in different ambients. As a result, Ambient-PRISMA needs to provide a binding that connects a

port of a system which is located in an ambient to a port of one of its architectural elements which is located in another ambient.

When both a system and its architectural element are located in the same ambient, their binding is located in their ambient. For example, when the *AuctionHouse1* system and the *LotOnAuction1* are located in the *AuctionSite* ambient (see Figure 73), their binding is also located in the *AuctionSite* ambient (represented with the dotted line).

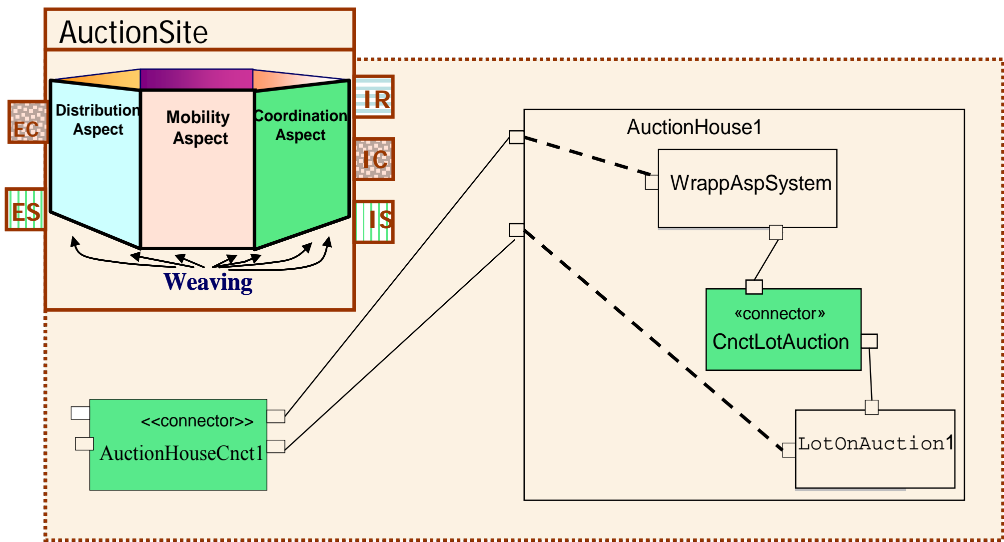


Figure 73. The architectural elements of *AuctionHouse1* system located in the AuctionSite

However, when the *AuctionHouse1* system is in the *AuctionSite* ambient and the *LotOnAuction1* component is in the *LondonSaleRoomSite* ambient as shown in Figure 74, the connection is performed by: a binding between the InServicesPort port of the *AuctionSite* and the port of the *AuctionHouse1*, an attachment between the EServicesPort port of the *AuctionSite* ambient and the EServicesPort port of the *LondonSaleRoomSite* ambienr, and an attachment between the port of the *LotOnAuction* component and the InServicesPort port of the *LondonSaleRoomSite* ambient.

The *InServicesPort* port is the only port of an ambient that can be connected to a system port through a binding. In addition, this binding exists only when the port of a system has a binding that connects an architectural element of a system that is not in its same ambient.

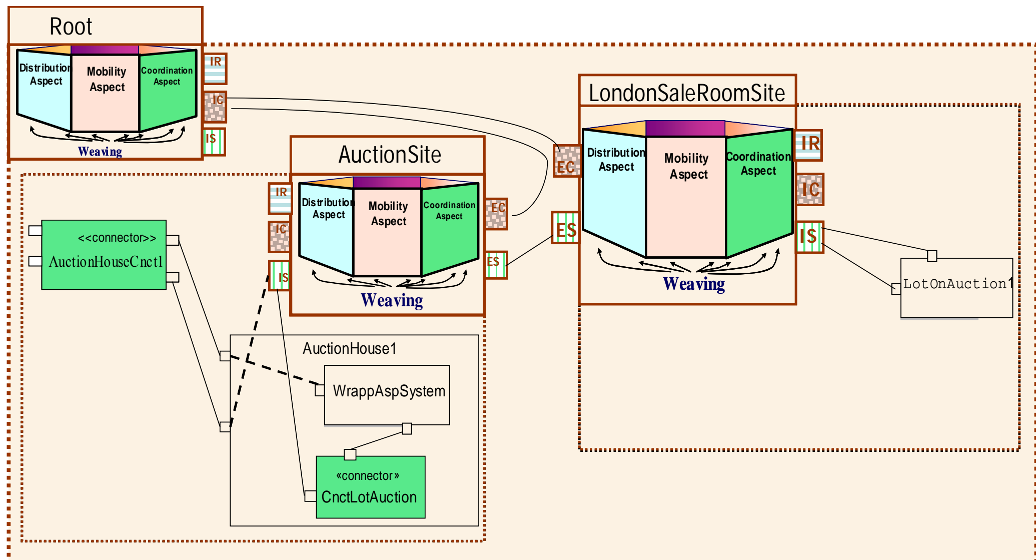


Figure 74. *AuctionHouse1* system architectural elements distributed between the AuctionSite ambient and the LondonSaleRoomSite ambient

As a result, each ambient manages the bindings that connect a local system to its architectural elements as well as the bindings between a system port and its InServicesPort port.

7.2.7 Kinds of Ambients

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

- **Site Ambients:** They represent physical locations; that is, Site ambients have a physical address. They can be devices or physical regions. A PC or a PDA are examples of ambients that can be modelled as Sites. Site ambients can only contain Components, Connectors or Group ambients. For example, the *ClientSite* and the *AuctionSite* of Figure 58 are of type Site.
- Non-Site Ambients: Non-Site Ambients do not model a physical location. They are separated into two kinds :
  - **Group Ambients**: They represent a collection of architectural elements. There utility can be different depending on the software system requirements. They can be used for grouping a set of architectural elements that share specific security politics or for moving architectural elements together. A Group ambient can lodge other Group ambients, Components or Connectors. For example, *MobileAmb* in Figure 67 is a Group Ambient that moves architectural elements together.
  - **Virtual Ambients**: They represent domains which can lodge other Virtual or Site ambients. Virtual ambients model networks or virtual spaces where sites are located. A Virtual ambient that consists of a collection of Sites represents the boundary of a network (e.g. a LAN). The modelling of a Virtual ambient hierarchy can represent the hierarchy of LANs to form a Wide Area Network (WAN). Different Virtual ambient hierarchy levels can be represented in the software architecture depending on the software architect and on the distributed requirements. In Ambient-PRISMA there is always a default Virtual Ambient that represents the root of an ambient hierarchy called Root (see Figure 58).

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

Figure 75 shows an example of how two WANs formed by two LANs have been modelled using the different kinds of ambients. Also, the LANs are formed from hosts that represent devices. Also, each host can contain other ambients. In the Figure, the kind of each ambient is indicated.

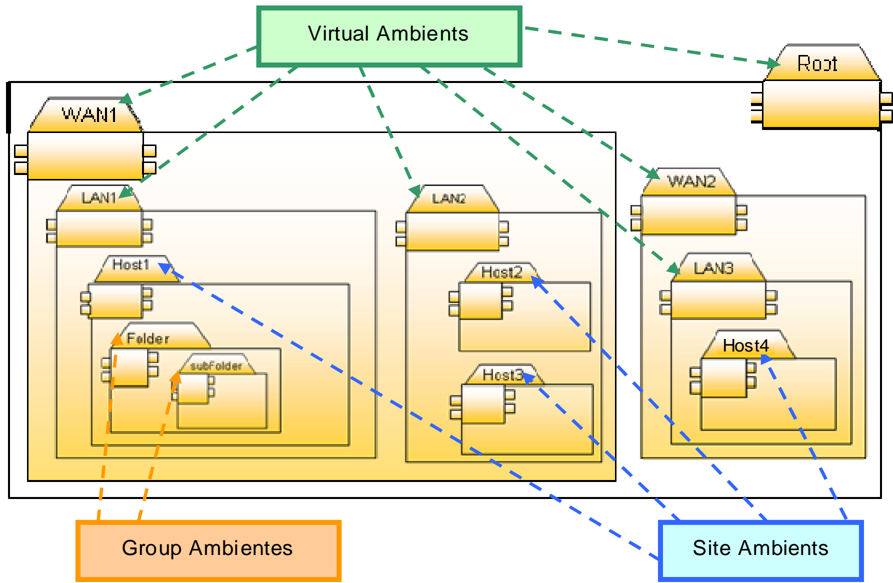


Figure 75. Kinds of ambients in a possible network

7.3 Ambient-PRISMA Metamodel

PRISMA provides a metamodel that includes a set of metaclasses and their relationships. Each concept considered in PRISMA is represented by a metaclass that defines a set of properties and services. The metamodel has been defined using the class diagram of the Unified Modelling Language (UML) version 1.5. and the Object Constraint Language (OCL) 2.0 [UML07]. CHAPTER 6 presented part of the PRISMA metamodel. Since Ambient-PRISMA enriches PRISMA with new concepts related to ambients, the metamodel of PRISMA has to be enriched.

In this section, the PRISMA metamodel is extended with concepts and properties that Ambient-PRISMA needs. These concepts have been included by adding new

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

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

7.3.1 Connector Package

The *Connector* package is extended in order to define an ambient architectural element as a specialized type of connector (see Figure 76). As a result, the *Ambient* metaclass is introduced by inheriting from the *Connector* metaclass. The connector inherits all the properties of a connector. As a result, it inherits the invariant called *coordinationAspect* that specifies that a connector must import an aspect whose concern is *coordination*.

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

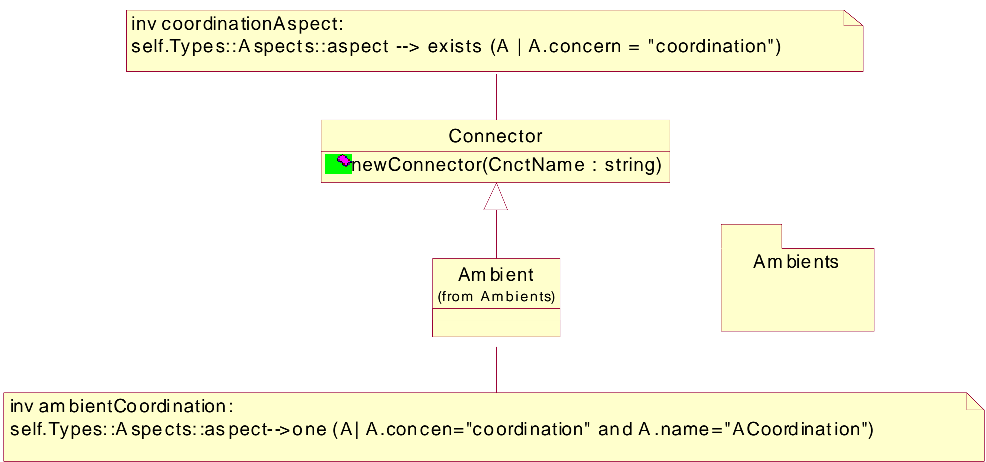


Figure 76. Connector Package of the Ambient-PRISMA metamodel

In the PRISMA metamodel, connectors were constrained to only be attached to two different components (see section 6.2.4). However, this constraint cannot be applied in Ambient-PRISMA due to the fact that an ambient can be attached to a connector, a component, or another ambient. Since ambients are connectors that can be attached to other connectors this constrain is not valid in Ambient-PRISMA. As a result, this constraint has been eliminated from the metamodel.

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

7.3.2 Ambient Package

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

An ambient locates architectural elements (which can be components, connectors, systems or other ambients), attachments that connect an ambient with architectural elements located in it and attachments that connect architectural elements of an ambient, and bindings that connect systems with architectural elements they are composed of and bindings that connect an ambient with a system located in it. As a result, the *Ambient* metaclass has three aggregation relationships: one with the *ArchitecturalElement* metaclass, one with the *Attachments* metaclass and another with the *Bindings* metaclass (see Figure 77).

The aggregation between the *Ambient* metaclass and the *ArchitecturalElement* metaclass has the *child* role and the *deployed* role. The *child* role indicates that an ambient can have from 0 to many architectural elements located in it. The *deployed* role indicates that an architectural element can be located in 0 or in only one ambient. The aggregation between the *Ambient* metaclass and the *Attachments* metaclass has the *located* role and the *locatesAttach* role. The *locatesAttach* role indicates that an ambient can have 0 or many attachments located in it. The *located* role indicates that an attachment can only be located in one ambient. The aggregation between the *Ambient* metaclass and the *Binding* metaclass indicates that a binding can only be located in an ambient and that an ambient can locate 0 or many bindings.

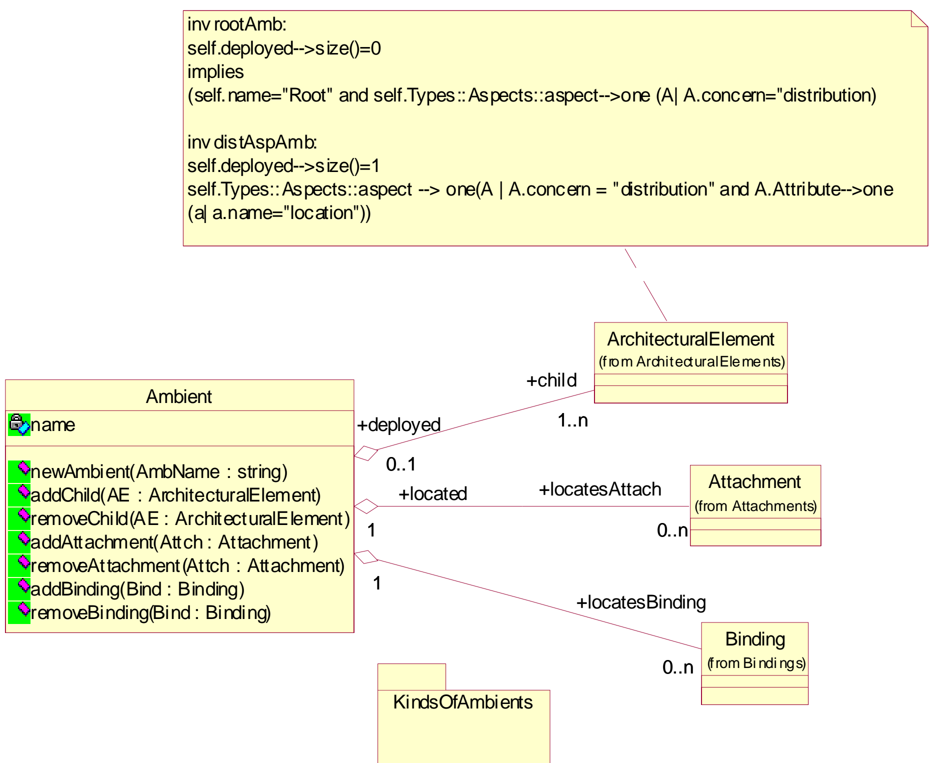


Figure 77. Ambient Package of the Ambient-PRISMA metamodel

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

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

The *Ambient* metaclass has an attribute called *name* which specifies the name of an ambient and seven services:

- *newAmbient* service creates a new ambient by providing the name of the ambient as a parameter.
- *addChild* service adds a new architectural element in the boundary of an ambient.
- *removeChild* service removes an architectural element that is located in the boundary of an ambient.
- *addAttachment* service adds an attachment that connects an ambient to one of its children or an attachment between two children of the same ambient.
- *removeAttachments* removes an attachment located in the boundary of an ambient.
- *addBinding* service adds a binding that connects an ambient to one of its system children or a binding between a system and an architectural element that it is composed of children which are located in the same ambient.
- *removeAttachments* removes a binding located in the boundary of an ambient.

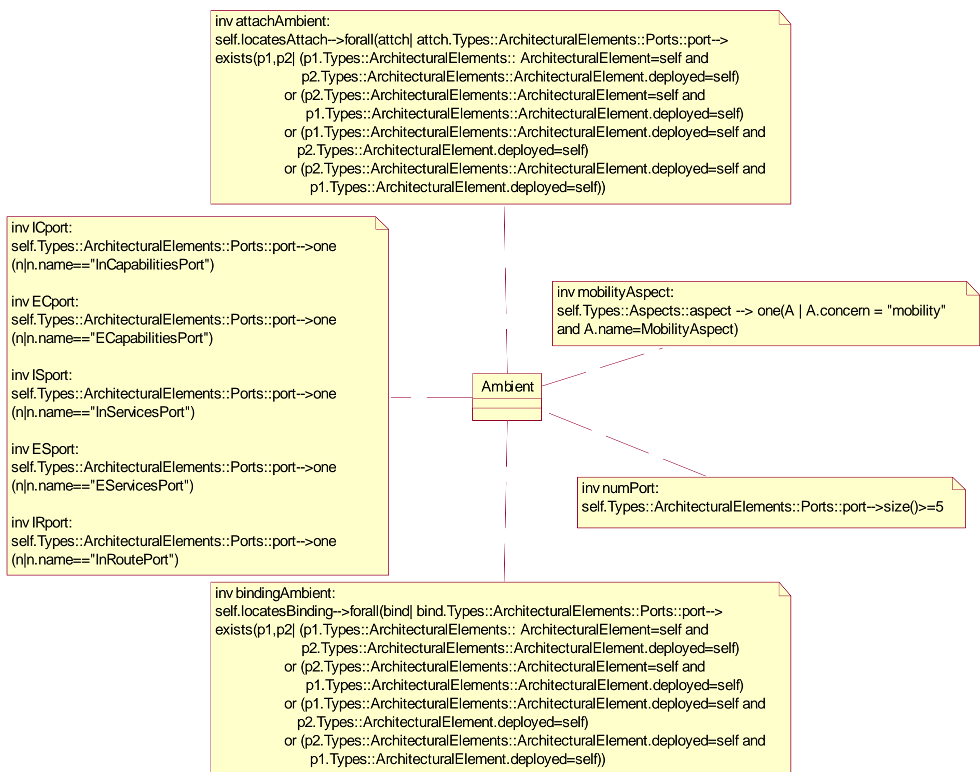


Figure 78. OCL constraints associated to the Ambient metaclass

Figure 78 shows the OCL constraints that an ambient should satisfy. In the following they are explained:

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

- The *ISport* invariant specifies that an ambient must have a port called InServicesPort.
- The *ESport* invariant specifies that an ambient must have a port called EServicesPort
- The *IRport* invariant specifies that an ambient must have a port called InRoutePort
- The *attachAmbient* invariant specifies that attachments located in an ambient are those that connect the ambient with the architectural elements located in its boundary or those that connect two architectural elements that are located in its boundary. Elements of different ambients cannot have attachments.
- The *bindingAmbient* invariant specifies that bindings located in an ambient are those that connect the ambient with a system architectural elements located in its boundary or those that connect a system and an architectural element it is composed of which are located in its boundary. A system located in an ambient and an architectural element located in another ambient cannot have a binding.

In addition, the *Ambient* package includes the *KindsOfAmbients* subpackage (see Figure 77 which defines the properties of the three kinds of ambients.

7.3.3 KindsOfAmbient Package

In Ambient-PRISMA, there are three kinds of ambients: sites, virtuals, and groups. The KindsOfAmbient package defines that the three kinds of ambients inherit the properties of the *Ambient* metaclass (see Figure 79). Each kind of ambient is represented by a metaclass and each one has a new service to create new kinds of ambients. Also, each metaclass has OCL invariants associated to it in order to differentiate each kind of ambient from the others.

The *Site* metaclass has the following OCL invariants:

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

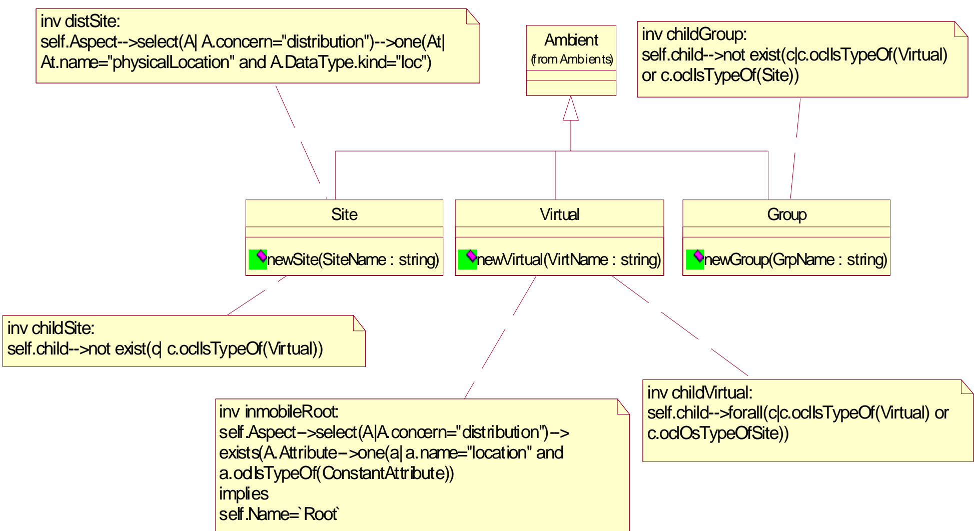


Figure 79. The *KindsOfAmbients* package of the Ambient-PRISMA metamodel

The *Virtual* metaclass has the following OCL invariants:

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

The *Group* metaclass has the following OCL invariants:

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

7.3.4 Attachments Package

The *Attachments* package of the PRISMA metamodel defines an attachment as a communication channel between a port of a component and a port of a connector (see section 6.2.5.1). The *Attachments* package in Ambient-PRISMA is extended in order to include attachments that can also connect ambients with other architectural elements. This extension has been performed by extending an OCL constraint and including three new ones. In addition, the names of the attributes of the *Attachment* metaclass have been changed. As a result, the *Attachment* package becomes as shown in Figure 80.

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

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

<<An attachment cannot connect a port of a component to a port of another component >>

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

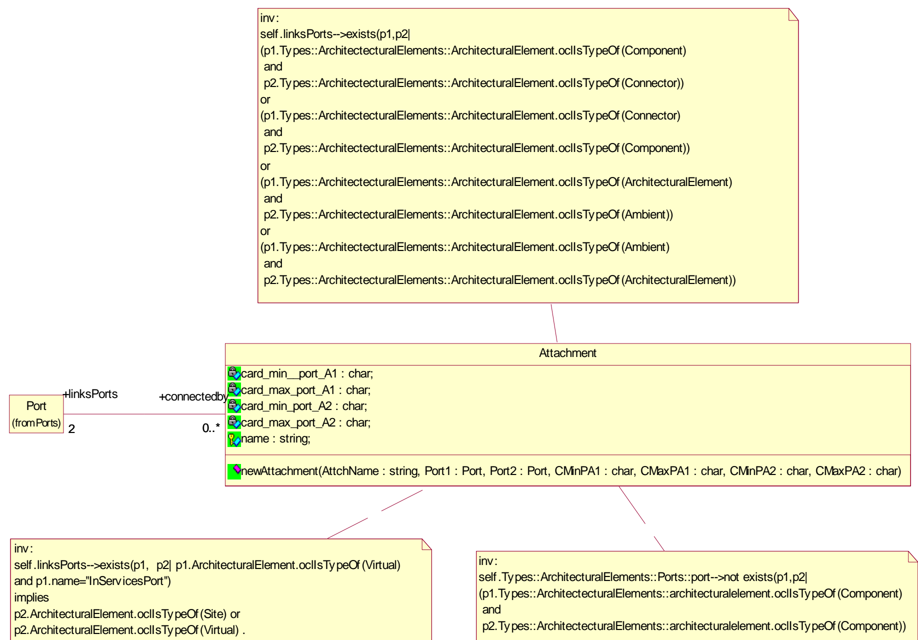


Figure 80. The *Attachments* package of the Ambient-PRISMA metamodel

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

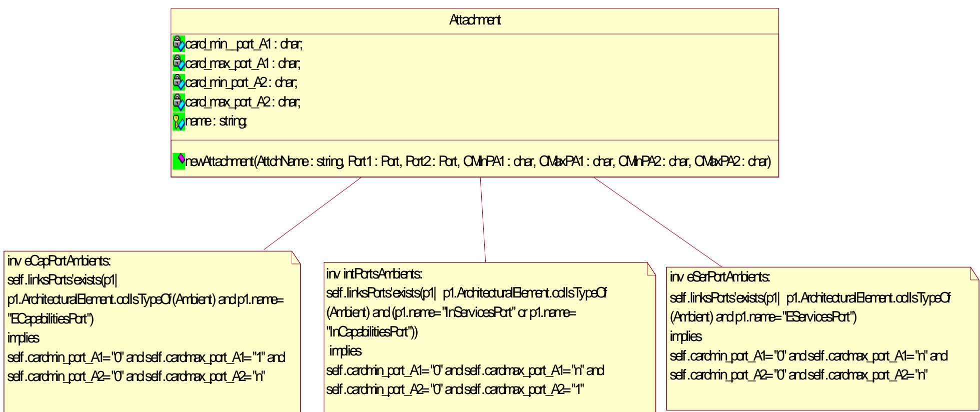


Figure 81. OCL constraints for the Attachment metaclass

In Ambient-PRISMA, an architectural element is deployed in only an ambient. An ambient also has five predefined ports. Therefore, the attachment communication patterns of an ambient are predefined. As a result, the *Attachment* metaclass has three OCL invariants to predefine the values of the cardinality attributes. In the following, these are explained (see Figure 81):

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

7.3.5 Bindings Package

In Ambient-PRISMA, a system architectural element can be composed of distributed architectural elements. Since a binding can connect a port of a system located in an ambient to a port of an architectural element of a system that is located in another ambient, the Bindings package of the PRISMA metamodel has to be extended.

As a result, a new OCL invariant has been associated to the *Binding* metaclass (see Figure 82). This invariant specifies that a binding is defined between a system port and an ambient port, the port of the ambient must be called *InServicesPort*.

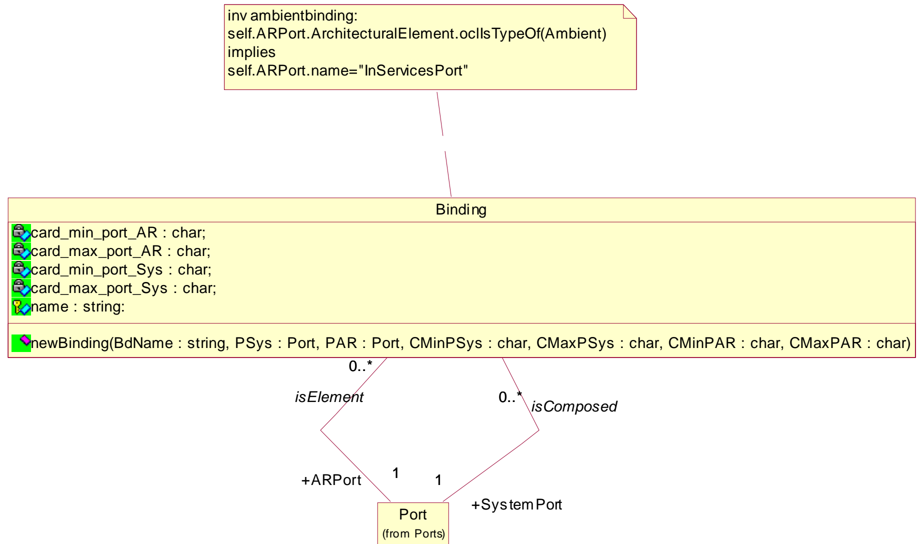


Figure 82. The Binding package in the Ambient-PRISMA metamodel

7.3.6 AmbientPRISMAArchitecture Package

The Ambient PRISMA Architecture package defines how a software architecture with ambients is defined. The *PRISMAArchitecture* metaclass that represents a software architecture (see Figure 20) has an aggregation relationship with the *Ambient* metaclass (see Figure 83). An architecture in Ambient-PRISMA is defined by

at least an ambient. Since an ambient is reusable, it can be used by many software architecture. In this way, an ambient is a first-class entity of an Ambient-PRISMA architectural model.

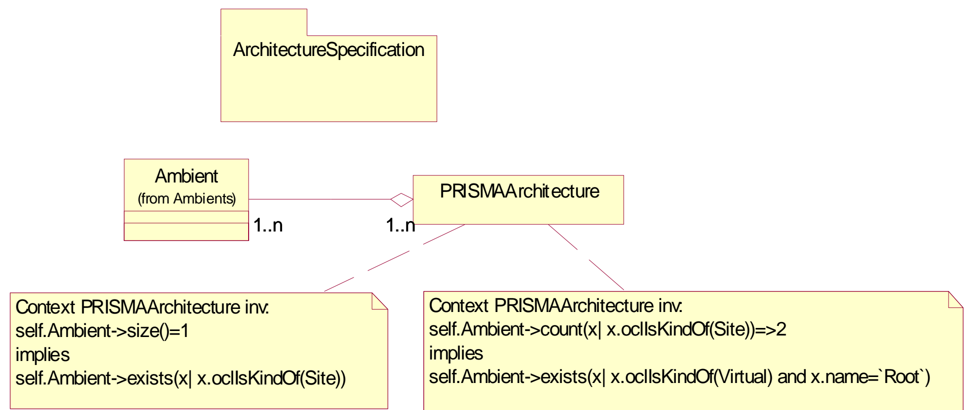


Figure 83. The *AmbientPRISMAArchitecture* package of the Ambient-PRISMA metamodel

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

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

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

7.3.7 Data Types Package

The different data types that PRISMA supports are included in the *DataTypes* package (see Figure 84). The *DataType* metaclass has an attribute called *kind*. The *kind* attribute can have different values, e.g., natural, integer, double, etc. A new kind of data type called *loc* is included in the Ambient-PRISMA *DataType* metaclass.

The *loc* data type is needed in order to model the different physical locations that form a specific distributed system.

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

Figure 84. The package *DataTypes* of Ambient-PRISMA

7.4 Conclusions

Ambient-PRISMA has been presented in this chapter. Ambient-PRISMA permits to describe PRISMA aspect-oriented architectural models of distributed and mobile software systems. This has been possible thanks to the integration of AC with PRISMA. Mainly, PRISMA has been enriched with an ambient primitive of AC and its capabilities. An ambient has been introduced as new kind of connector that coordinates architectural elements of a boundary. The capabilities of AC have been introduced as services that ambients provide to architectural elements located in their boundary. The main characteristics of Ambient-PRISMA are the following:

- An explicit notion of location has been provided. This notion of location has been introduced as a new kind of connector called ambient.
- Components, connectors, and systems can be located in ambients. Ambients can also locate other subambients. In this way, the hierarchy of a distributed system can be modelled.
- Different kinds of ambients have been defined in order to model ambients with different functionalities.
- An ambient includes coordination mechanisms related to the synchronization of architectural elements in different ambients.
- Since an ambient has been included as an architectural element, it can provide other architectural elements with distribution and mobility services.
- Mobility of architectural elements is modelled by reconfiguring the software architectures.

In addition, the PRISMA metamodel has been extended in order to permit the creation of Ambient-PRISMA architectural models in an accurate way. The Ambient-PRISMA metamodel defines the required metaclasses, their properties and services, and relationships with each other and the ones defined in PRISMA. In addition, the metamodel specifies the constraints to ensure that the definition of an architectural model is correct.

The metaclass ambient has the needed services to create ambients and add and remove architectural elements, attachments, and bindings of ambients. In this way, ambients can manage architectural elements located in them and also provide mobility in a consistent way.

The work related to this chapter has produced a set of results that have been published in the following publications:

- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, “Mobile Ambients in Aspect-Oriented Software Architectures”, IFIP Working Conference on Software Engineering Techniques (SET 2006), **Springer Series in Computer Science**, Warsaw, Poland, October 17-20, 2006.
- **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí , “Introducing Ambient Calculus in Mobile Aspect-Oriented Software Architectures”, Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), **IEEE Computer Society**, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper).
- **Nour Ali**, Isidro Ramos, Jose A. Carsí, “A Conceptual Model for Distributed Aspect-Oriented Software Architectures”, International Conference on Information Technology Coding and Computing (ITCC 2005), **IEEE Computer Society**, Las Vegas, NV, USA, 2005.
- **Nour Ali**, Jose Angel Carsi, Isidro Ramos, “Analysis of a Distribution Dimension for PRISMA”, Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2004), Málaga, 10-12 Noviembre 2004

- **Nour Ali**, Isidro Ramos, "Ambient-PRISMA: Ambients in Distributed and Mobile Aspect-Oriented Software Architectures", V Jornadas de Trabajo DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, 2006.
- **Nour Ali**, Jose Cercos, Isidro Ramos, Patricio Letelier, Jose Angel Carsi, "Distribution in PRISMA", workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- Josep Silva, **Nour Ali**, Jose Angel Carsi, Isidro Ramos, "The Distribution Aspect of PRISMA", VIII Conference on Software Engineering and Databases (JISBD), ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)
- **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "Distribution in an Aspect-Oriented Component Based Software Architecture through PRISMA", Technical Report DSIC-II/14/04, Polytechnic University of Valencia, October 2004.

CHAPTER 8

AMBIENT-PRISMA LANGUAGE

"High thoughts must have high language."

Aristophanes

8.1 Introduction

Ambient-PRISMA provides an ambient-oriented extension to the PRISMA Aspect-Oriented Architecture Description Language (AOADL) called the Ambient-PRISMA AOADL. Ambient-PRISMA AOADL permits the specification of distribution and mobility characteristics of architectural models and their configurations conforming to the Ambient-PRISMA metamodel. Given that the Ambient-PRISMA AOADL, is an extension to the PRISMA AOADL, every architectural model or configuration specified in PRISMA is also valid in Ambient-PRISMA.

The PRISMA AOADL syntax is used in Ambient-PRISMA for specifying interfaces, aspects, components, connectors, systems and attachments. Ambient-PRISMA provides the syntax for specifying ambients.

The PRISMA AOADL (see CHAPTER 6) specifies architectural elements at two levels of abstraction: the type definition level and the configuration level. At the type definition level, types are defined in order to be reusable by other types or specific architectures. At the configuration level a topology of a software architecture is designed by instantiating architectural element types and their connections. The

Ambient-PRISMA AOADL introduces ambients both at the type definition level and at the configuration level in order to define distributed software architectures. In a similar way to the rest of architectural elements, ambients are specified by using interfaces and aspects. In addition, ambients form part of the specification of an architectural model and its configuration.

This chapter presents the language constructs that have been defined in order to permit the specification of distributed and mobile aspect-oriented software architectures. The chapter is structured as follows: Section 8.2 presents the syntax for specifying an architectural model at the type definition level. In addition, constructs that have been predefined in order to give functionality to ambients are explained. Section 8.3 presents the syntax for specifying a configuration of a software architecture. Finally, section 8.4 concludes the chapter.

8.2 The Type Definition Level

In this section, constructs that are needed for defining architectural models in Ambient-PRISMA are presented in the AOADL. These constructs include specifications of predefined interfaces, aspects, ports, and weavings. These constructs have been defined using the templates of PRISMA. New syntactical constructs have been also defined by incorporating new templates.

8.2.1 Architectural Model syntax

An architectural model in Ambient-PRISMA consists of its first class entities: interfaces, aspects, components, connectors, systems, and ambients.

The template for specifying an architectural model is presented in Figure 85. An architectural model specification is preceded and ended by the reserved word *Architectural_Model* and *End_Architectural_Model*, respectively. The specification of an architectural model consists of a name and the blocks that specify each first-class entity: interfaces, aspects, components, connectors, systems, virtual ambients,

site ambients, group ambients and attachments. The blocks of the virtual ambients, the group ambients, and the systems can be optionally specified.

```
<architectural_model> ::= Architectural_Model <model_name>
                          <interface_block>
                          <aspect_block>
                          <component_block>
                          <connector_block>
                          [<system_block>]
                          [<virtual_ambient_block>]
                          <site_ambient_block>
                          [<group_ambient_block>]
                          <attachment_block>
                            End_Architectural_Model <model_name>';'
```

Figure 85. Architectural Model template in Ambient-PRISMA

For example in Figure 86, an architectural model called *MobileAgentsAuctionModel* is specified.

```
Architectural_Model MobileAgentsAuctionModel
……
End_Architectural_Model MobileAgentsAuctionModel
```

Figure 86. MobileAgentsModel Architectural Model

The blocks of each first class entity of Figure 85 are explained in the following.

8.2.2 Interfaces

Interfaces publish a set of services. Interfaces are defined in the AOADL using the template in APPENDIX A. In Ambient-PRISMA AOADL, an interface can be

specified for publishing services related to distribution or mobility. For example, an interface called *IMobility* is specified in Figure 87. The interface publishes a service called *move* which has an input parameter called *NewAmbient* in order to indicate its destination ambient.

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

Figure 87. *IMobility* Interface

8.2.3 Predefined Interfaces

Some interfaces are generic and predefined in Ambient-PRISMA. These interfaces have been defined in order to give specific functionalities to ambients. In the following, these interfaces are explained:

8.2.3.1 ICall Interface

ICall interface is used by ambients in order to receive services of architectural elements located in them in a generic way. The interface publishes a service called *call*. The call service has three parameters: *Name*, *ParamsList* and *ParamType*. The *Name* parameter stores the name of a service that needs to be invoked or provided by an architectural element located in an ambient. The *ParamsList* parameter is a list that stores the parameters of the service. The *ParamType* parameter is a list that stores the kind of each parameter (whether it is input or output). The *ParamType* stores the kind of parameters in the same order as the parameters appear in the original service.

```
Interface ICall

   call(input Name: Service, input ParamsList[]: Parameter,
        input ParamType[]: ParameterKind);

End_Interface ICall
```

Figure 88. Specification of the *ICall* Interface

This interface is only used and predefined in ambients. The user of Ambient-PRISMA is not allowed to use it in any case.

8.2.3.2 ICapability Interface

ICapability interface publishes a set of services for providing the AC capabilities: the *exit* service (in capability of AC) and the *enter* service (the out capability of AC), the *startMovement* and *finishMovement* services that are needed in order to reconfigure attachments and bindings associated to a mobile architectural element after and before the mobility process, the *changeLocation* service which changes the location of an architectural element, the accept service which allows an ambient to accept a new architectural element, and the *copy* service that replicates architectural elements of an ambient (see Figure 89).

```
Interface ICapability

   startMovement(input Name: ArchitecturalElement);
   exit (input Name: ArchitecturalElement);
   enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
   finishMovement(input Name: ArchitecturalElement);
   changeLocation(input Name: ArchitecturalElement,
                  input NewLocation: Ambient);
   accept(input NewAmbient: Ambient, input Caller:ArchitecturalElement,
          input Child:ArchitecturalElement,
          input CommuniList[]:string,
          output AcceptanceOK: boolean);
   copy(input Name: ArchitecturalElement, output Ok:boolean);

End_Interface ICapability
```

Figure 89. Specification of the *ICapability* Interface

A description for each service of the *ICapability* interface is presented in Table 5. The services of this interface are used by aspects that are imported by ambients and the other architectural elements. Ambients can be clients of the *changeLocation* service and the *accept* service through their *MobilityAspect* aspect (see 8.2.5.2). Architectural elements (including subambients) can be clients of the *startMovement*, *exit*, *enter*, and *finishMovement* services offered by their parent ambients through

their distribution aspect. Architectural elements can also be clients of the *copy* service offered by their parent ambient through their replication aspect. As a result, the user has to understand their functionality in order to specify them when distribution aspects or replication aspects of architectural elements are defined.

Table 5. Description of the *ICapability* interface services

Service	Description
`startMovement`	Given a name of an architectural element, the ambient prepares an architectural element to move.
`exit`	Given a name of an architectural element, an architectural element is allowed to exit its parent ambient.
`enter`	Given a name of an architectural element and a name of an ambient, an architectural element is provided to enter the ambient with the given name, if the ambient is a sibling of the architectural element.
`finishMovement`	Given a name of an architectural element, an ambient creates the needed attachments and bindings in its boundary.
`changeLocation`	An ambient returns its name to an architectural element that has just arrived to it.
`accept`	Given a receptor ambient, an origin ambient, an architectural element, and the list of attachments and bindings connected to the architectural element, the receptor ambient returns true to the origin ambient if it accepts to receive the architectural element
`copy`	Given a name of an architectural element, a parent ambient creates a new instance with the same state as the architectural element and returns true when the copy has been created.

8.2.3.3 IGetLocation Interface

IGetLocation interface publishes a service called *getLocation*. This interface is used in order to inform the name of a parent ambient (see Figure 90). The *getLocation* service has one parameter called *Location*. The *Location* output parameter returns the name of the parent ambient.

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

Figure 90. Specification of the *IGetLocation* Interface

The service of this interface is provided by an ambient. This service can be used by the aspects of an ambient. Architectural elements that need to know their grandparent ambient can request this service, once it is offered by an ambient.

8.2.3.4 IRoute Interface

IRoute interface publishes a service called *getRoute* (see Figure 63). This service allows an ambient to provide its internal architectural elements for consulting a route of any architectural element of a software architecture. A route of an architectural element consists of the names of ambients where the architectural element is positioned in a tree hierarchy of ambients

The *getRoute* service has two parameters called *Name* and *Route[]*. The *Name* parameter is an input parameter that indicates the name of an architectural element. The Route is an output parameter which returns a list with the route of an architectural element.

```
Interface IRoute
 getRoute(input Name:ArchitecturalElement, output Route[]: Ambient);
End_Interface IRoute
```

Figure 91. Specification of the *IRoute* Interface

The service of this interface is provided by an ambient. Architectural elements can be clients of the *getRoute* service offered by their parent ambient through their distribution aspect. As a result, the user has to understand its functionality in order to specify it when a distribution aspect of an architectural element is defined.

8.2.4 Aspects

Aspects define the behaviour of architectural elements from a specific concern of the software system. Aspects are defined in the AOADL using the template shown in Figure 92. This template includes a set of sections: Attributes, Services, Valuations, Preconditions, Constraints, Played_Roles, and Protocols. These sections have been introduced in by the PRISMA AOADL and were explained in CHAPTER 6. In addition, a section called *triggers* (see section 8.2.4.2) which comes from the OASIS language [Let98] has been introduced for specifying more expressive aspects.

```
<aspect> ::= <aspect_type> Aspect <aspect_name> [using <interface_name_list>]
                      [ <constant_attributes> ]
                      [ <variable_attributes> ]
                      [ <derived_attributes> ]
                       <services>
                       [ <preconditions> ]
                       [ <transactions> ]
                       [ <constraints> ]
                       [ <triggers> ]
                       [<played_roles>]
                        <protocol>
            End_Aspect <aspect_name>';'
<aspect_type> ::= functional | coordination | distribution | replication | mobility |
                 | quality | safety| integration| persistence | presentation
                 | navigational | context-awareness
```

Figure 92. Specification template of aspects in Ambient-PRISMA

In Ambient-PRISMA each architectural element has to import a distribution aspect. A distribution aspect which an architectural element imports has to include some specifications which are different depending on the kind of architectural elements.

In the following, the details for specifying aspects in Ambient-PRISMA are explained.

8.2.4.1 Distribution Aspects of Components, Connectors and Systems

A distribution aspect is specified with the template for aspects (see Figure 92). The sections of this template have been explained in CHAPTER 6 except for the triggers section (see section 8.2.4.2). An example of a distribution aspect called *BidderDist* is specified in Figure 93. The *BidderDist* aspect uses the *IMobility* interface (specified in Figure 87) and the *ICapability* interface (specified in Figure 89).

It can also be observed that the *BidderDist* aspect has a variable attributes called ***location***. This attribute is variable because its value can change during the life of the aspect. The ***location*** attribute is marked with bold due to the fact that location is a keyword in Ambient-PRISMA. The ***location*** attribute stores the name of the parent ambient of the architectural element that imports the aspect. The *location* attribute is also a ***NOT NULL*** attribute due to the fact that its value can never be NULL. This attribute exists in all distribution aspects of architectural elements except for the *Root* architectural element (see section 8.2.7). In addition, the *BidderDist* attribute has a constant attribute called *originLocation*. The *originLocation* is a ***NOT NULL*** attribute that saves the value of the origin ambient of the architectural element which imports this aspect.

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

```
Distribution Aspect BidderDist using IMobility, ICapability

 Attributes
  Variable
   location: Ambient NOT NULL;

  Constant
   originLocation: Ambient NOT NULL;
 Services

  begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)]
          location := ParentAmbient, originLocation:= ParentAmbient;


  //These services are concerned with mobility in order to interact with the
parent ambient.
  in changeLocation(input Name: ArchitecturalElement,
                    input NewLocation: Ambient)
        Valuations
          [changeLocation()] location:=NewLocation;
  out startMovement(input Name:ArchitecturalElement);
  out exit (Name: ArchitecturalElement);

  out finishMovement(input Name:ArchitecturalElement);

  out enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
  end();
 Preconditions
 //This precondition is concerned with the interaction with the ambient
      in  changeLocation(Name, NewLocation)
                               if {self.Name==Name};


 Played_Roles
  CapParent for ICapability ::= startMovement!(Name)→
                                exit!(Name)→
                                enter!(Name, NewAmbient)→
                                finishMovement!(Name) →
                                changeLocation?(Name, NewLocation);

  CUSTMOVESBIDDER for IMobility ::= move?(NewAmbient);

//
 TRANSACTIONS in MOVE (NewAmbient:string)
      move =  CapParent_startMovement!(self.Name)→
              MOVE1;
      MOVE1 = CapParent_exit!(self.Name)→ MOVE2;
      MOVE2 = CapParent_enter!(self.Name, NewAmbient)→MOVE3;
      MOVE3 = CapParent_finishMovement!(self.Name);
      MOVE4 = CapParent_changeLocation?(Name, NewLocation);


 Protocol
 // The mobility should be first executed by the customer and then it can
moveback to the customer location by its own.
  BIDDERDIST:= begin→ BIDDERDIST1;
```

```
  BIDDERDIST1:=(CUSTMOVESBIDDER_MOVE?(NewAmbient)+ MOVE?(originLocation)
                + end)
                →
                BIDDERDIST1;

End_Distribution Aspect BidderDist
```

Figure 93. Speification of the BidderDist distribution aspect

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

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

The *BidderDist* aspect specifies that the architectural element which imports it is a mobile architectural element. This is specified by specifying a transaction called *MOVE* that uses the services of the *CapParent* played_role. The *MOVE* transaction consists of invoking the services of the *CapParent* played_role. The *MOVE* transaction consists of requesting the *startMovement* service, receiving the result of the *startMovement* service (the *CommuniList* output parameter), requesting the *exit*, the *enter*, and the *finishMovement* services, and receiving the *changeLocation* service.

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

In addition to the sections presented in the PRISMA AOADL an aspect can include the triggers section. In the following, the triggers section is explained.

8.2.4.2 Triggers

Triggers force a service defined in an aspect to be either served or requested whenever the state of an aspect satisfies a certain condition.

The specification of triggers is preceded with the reserved word ***Triggers*** (see Figure 94). A trigger specification consists of defining a service and a formulae that the condition affects. The service is specified by describing its name, the kind of behaviour of a service, and the name of its parameters in brackets. When a service needs to be served, the service name is preceded with reserved word ?. When a service needs to be requested, the service name is preceded with the reserved word !. The formula is preceded by the reserved word ***when*** and is specified in curly brackets.

```
<triggers> ::= Triggers
                <trigger_def_block >
<trigger_def_block > ::= <service_name> <kind_channel>  '('  [<parameter_name_list> ]')'
                        when '{' formula '}' ';'
<kind_channel> ::= <input_channel> | <output_channel>
```

<input_channel> ::= **'?'**

<output_channel> ::= **'!'**

Figure 94. Specification template of triggers

Triggers can be useful in any kind of aspect in order to determine when a certain condition is satisfied, a service execution is provoked. For example, Figure 95 shows the specification of a trigger in the *BidderFunct* aspect. The trigger specifies that the bid service is requested when both the current bidding situation (attribute *biddingSituation*) is equal to "bid against you" and the current bid is less or equal to the value specified by the customer to be the maximum bid (attribute *lotMaximumBid*).

```
Functional Aspect BidderFunct using ICustBidder, IBidderAuct
… …
 Triggers
   bid!(input currentBiddingAmount, output Situation) when
        { biddingSituation=="bid against you"
          and currentBiddingAmount<=lotMaximumBid};
… …
End_Aspect BidderFunct;
```

Figure 95. Specification of a trigger in the *BidderFunct aspect*

Examples of triggers in a distribution aspect are the ones found in the *ProcurDist* aspect (see Figure 96). The triggers section specifies three triggers. The first trigger specifies that when the value stored in the *originLocation* attribute is equal to the value stored in the ***location*** attribute, then the *initializeCounter* service is executed. The next trigger specifies that when the counter attribute is equal to 1, then the *move* transaction is executed with the *NewAmbient* input parameter stores the value of the *nextAuctionSiteLoc* attribute. The last trigger specifies that when the counter attribute is equal to 2, the *move* transaction is executed with the *NewAmbient* input parameter stores the value of the *originLocation* attribute. The objective of these

triggers is to specify to which locations the architectural element that imports *ProcurDist* aspect has to move.

```
Distribution Aspect ProcurDist using IMobility, ICapability

 Attributes
  Variable
  location: Ambient NOT NULL;
  nextAuctionSiteLoc: Ambient NOT NULL;
  counter: integer(0);
  Constant
   originLocation: Ambient NOT NULL;

 Services
…

  initializeCounter()
     Valuations
      [initializeCounter ()]
          counter := 0;
 TRANSACTIONS in MOVE (NewAmbient:Ambient)
      move =  CapParent_startMovement!(self.Name)→ MOVE1;

       …..

…..
 Triggers
       initializeCounter?()  when {originLocation==location};
       move?(nextAuctionSiteLoc)   when {counter==1};
       move?(originLocation)   when {counter==2};

……

End_Distribution Aspect ProcurDist
```

Figure 96. Specification of a trigger in the *ProcurDist aspect*

8.2.5 Predefined Aspects

Some aspects are generic and predefined in Ambient-PRISMA. These are the *ACoordination* aspect and the *MobilityAspect* aspect which an ambient must import. In addition, an ambient must import a distribution aspect which is not generic or predefined for all ambients. However, a distribution aspect of an ambient has to include some predefined specifications. These aspects specification follows the sections of an aspect shown in Figure 92.

8.2.5.1 ACoordination Aspect of an ambient

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

The specification of the Coordination Aspect is shown in Figure 97. The Coordination Aspect is called *ACoordination* and uses the *ICall* interface (see Figure 88). As a result, in its *Services* section the *begin*, the *end* and the *call* services are specified. The call service is of kind *in* and *out*. This is due to the fact that the *ACoordination* aspect can receive invocations of required (out) or provided (in) services. In addition, the aspect has two played_roles: INTERIOR and EXTERIOR. Both of them specify when and how the call service of the *ICall* interface can be required or provided. The *INTERIOR* played_role is for receiving (*?*) and sending (*!*) services of architectural elements located in the boundary of an ambient. The *INTERIOR* played_role specifies that it can receive or send (indicated with + symbol) *call* services. The *EXTERIOR* played_role is for receiving and sending services of architectural elements located outside the boundary of an ambient. Also, the *EXTERIOR* played_role specifies that it can receive or send call services.

```
Coordination Aspect ACoordination using ICall

Services
   begin();
   in/out call(input Name: Service, input ParamsList[]: Parameter,
               input ParamType[]: ParameterKind);

   end;

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

Protocol
```

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

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

End_Coordination Aspect ACoordination
```

Figure 97. Specification of the ACoordination Aspect

The *ACoor* protocol of the *ACoordination* aspect orchestrates the begin service, the end service, and the *INTERIOR* and the *EXTERIOR* played roles. Basically, the protocol specifies that once the *begin* service is executed, i.e. the aspect is instantiated through an ambient, the aspect executes the *ACOOR1* process. The *ACOOR1* process specifies a nondeterministic choice among the execution of four subprocesses: *INTrecEXTsnd*, *INTsndEXTrec*, *EXTrecINTsnd*, and *EXTsndINTrec* and the *end* service. Each subprocess specifies the possible combination of the reception or sending of a call service from a played role and the reception and sending of a call service from another played role. For example, the *INTrecEXTsnd* subprocess specifies that when a *call* service of the *INTERIOR* played role is received, the aspect sends a *call* service by the *EXTERIOR* played role.

8.2.5.2 MobilityAspect aspect of an ambient

The *MobilityAspect* aspect of an ambient provides architectural elements located in its boundary with the needed services in order to be capable of moving. The

specification of this aspect cannot be modified by the user. This aspect is also reused by all ambients. The *MobilityAspect* aspect uses the *ICapability* interface which is specified in Figure 89 in order to specify the behaviour of the services which it publishes (see Figure 98).

```
1   Mobility Aspect MobilityAspect using ICapability
       …. …
    Services
        begin();
        in startMovement(input Name: ArchitecturalElement);

        in finishMovement(input Name: ArchitecturalElement,
                          input CommuniList[]:string);
        ………

      end;

2   TRANSACTIONS in EXIT(input Requested: ArchitecturalElement)
       exit  = isChild! (input Requested, output IsChildOK)→
               isChild? (input Requested, output IsChildOK)→
               EXIT1;
       EXIT1 = {IsChildOK==true} getParent(output Parent)→EXIT2;
       EXIT2 = moving! (Requested,Parent);

3   TRANSACTIONS in ENTER(input Requested: ArchitecturalElement,
                          input NewAmbient: Ambient):
      ENTER = areChildren! (input Requested, input NewAmbient,
                            output AreChildrenOK)→
              areChildren! (input Requested, input NewAmbient,
                            output AreChildrenOK)→ENTER1;
      ENTER1 = {AreChildrenOK==true} moving?(Requested, NewAmbient);

4   TRANSACTIONS in MOVING(input Requested: ArchitecturalElement,
                           input NewAmbient: Ambient):
      MOVING = movingInf!(input Requested, input self.Name,
                          input Reguested, output Type,
                          output CommuniList[]:string)→
               movingInf?(input Requested, input self.Name,
                          input Reguested, output Type,
                          output CommuniList[]:string)→ MOVING1;
      MOVING1 = accept!(input NewAmbient, input Type,
                        input Requested, input CommuniList[],
                        output AcceptanceOK)→ MOVING2;
      MOVING2 = {Acceptance0K==true}

              modifyAttachment!(Requested)→modifyBinding!(Requested)→MOVING3;
      MOVING3 =
           removeAttachments!(Requested)→ removeBindings!(Requested)→MOVING4;
      MOVING4 = removeChild!(Requested);
```

```
5   TRANSACTIONS in/out ACCEPT(input NewAmbient: Ambient,
                               input Caller:ArchitecturalElement,
                               input Child:ArchitecturalElement,
                               input CommuniList[]:string,
                               output AcceptanceOK: boolean);

      accept = addChild!(input Child, input AttachmentsList[],
                         input BindingList[], output AddedOK) →
               addChild?(input Child, input AttachmentsList[],
                          input BindingList[], output AddedOK)
                → ACCEPT1;
      ACCEPT1 = {AcceptanceOK==AddedOK} changeLocation!(Child, self.Name);
6   TRANSACTIONS in/out COPY(input Requested: ArchitecturalElement,
                             output Ok: boolean):
      Copy = replInf!(input Requested, output Instance,
                      output CommuniList[])→
             replInf?(input Requested, output Instance,
                      output CommuniList[])→ COPY1;
      COPY1 = createNew (Instance)→ COPY2;
      COPY2 = creataeNewCommunChannels(CommuniList);
7   Preconditions
          in accept(input NewAmbient, input Caller, input Child,
                    input CommuniList[], output AcceptanceOK)
             if
               {self.Name==NewAmbient};

8   Played_Role
      INTERIOR for ICapability::= (accept?(input NewAmbient, input Caller,
                                           input Child,
                                           input CommuniList[],
                                           output AcceptanceOK)
                                   →
                                   accept!(input NewAmbient, input Caller,
                                           input Child,
                                           input CommuniList[],
                                           output AcceptanceOK))
                                   +
                                   enter?(input Requested, input NewAmbient)
                                   +
                                   exit?(input Requested)
                                   +
                                   (startMovement?(input Name)
                                   +
                                   finishMovement?(input Name)
                                   +
                                   changeLocation!(input Child,
                                                   input self.Name)
                                   +
                                   (copy?( input Requested, output Ok)→
                                    copy!( input Requested, output Ok));

9   EXTERIOR for ICapability::= (accept?(input NewAmbient, input Caller,
                                         input Child,
                                         input CommuniList[],
                                         output AcceptanceOK)
                                 →
                                 accept!(input NewAmbient, input Caller,
                                         input Child,
                                         input CommuniList[],
```

```
                                    output AcceptanceOK))
                                   +
                                  (accept!(input NewAmbient, input Caller,
                                           input Child,
                                           input CommuniList[],
                                           output AcceptanceOK)
                                   →
                                   accept?(input NewAmbient, input Caller,
                                           input Child,
                                           input CommuniList[],
                                           output AcceptanceOK));

   …   …
End_Mobility Aspect MobilityAspect
```

Figure 98. Specification of the *MobilityAspect* aspect

The services *startMovement* and *finishMovement* (see Figure 98, section 1) are auxiliary services that the aspect uses in order to make special operations at the beginning and at the end of a mobility process. The *startMovement* service is invoked by an architectural element in order to indicate to an ambient that it is going to invoke a chain of enter and exit services. The *finishMovement* service is invoked by an architectural element in order to indicate to an ambient that it has finished invoking the chain of capabilities, i.e., it has finalized the mobility process. In this way, an ambient has services in order to manage the beginning and the end of a mobility process.

The rest of services are specified as transactions. This is due to the fact that they consist of the invocation of a set of services that need to be executed in an atomic way in order to ensure a consistent state of the software architecture before and after an architectural element is moved.

The *EXIT (Requested: ArchitecturalElement)* transaction (see Figure 98, section 2) is in charge of serving an exit requested by architectural elements of an ambient. This transaction checks whether the architectural element that needs to exit is a child of the aspects ambient or not. This check is performed through the *isChild* service. The *ischild* service returns true through the *isChildOK* output parameter if the requested architectural element is a child of the ambient. If *ischild* returns true,

then the transaction proceeds to execute *getParent* service. The *getParent* service returns in the *Parent* parameter the name of the parent ambient of the *MobilityAspect* ambient. This is needed in order to indicate the destination of the exiting architectural element. Then the *moving* transaction is executed in order to move the architectural element.

The *ENTER (Requested: ArchitecturalElement, NewAmbient: Ambient)* transaction (see Figure 98, section 3) is in charge of serving an enter requested by an architectural element of an ambient that needs to enter a sibling ambient. This transaction checks whether both the mobile architectural element and the destination ambient are siblings, i.e., both are children of the requested ambient. If this is fulfilled, the moving transaction is executed in order to move the architectural element.

The *MOVING (Requested: ArchitecturalElement, NewAmbient: Ambient)* transaction (see Figure 98, section 4) is in charge of performing all the steps needed in order to reconfigure the architecture in the origin ambient. Initially, the information about the attachments connected to the mobile architectural element is obtained. This information is encapsulated in a list in order to enable their recreation in the destination ambient. The accept service of the destination ambient is invoked and the information of the attachments and bindings is sent. In any case, the origin ambient is attached to the destination ambient. If the destination ambient accepts to receive the mobile architectural element, the origin ambient concludes the mobility process by means of modifying the attachments. Then, the origin ambient removes all the attachments that connect the mobile architectural element to the parent ambient or to other architectural elements in the ambient, creates attachments between its *InServicesPort* port and the elements that were connected to the mobile architectural element and, finally, deletes the architectural element instance.

The *ACCEPT* transaction (see Figure 98, section 5) is invoked by an origin ambient to a destination ambient in order to request to the destination ambient to accept a new architectural element. This transaction adds a new element to its list of

elements, correctly creates the needed attachments thanks to the list of information saved about the attachments that it receives, and finally requests the *changeLocation* service of the new element, so that it updates its location stored in the distribution aspect. The accept transaction is in charge of performing all steps needed in order to reconfigure the architecture in the destination ambient. This transaction has a precondition associated to it (see Figure 98, section 7). The precondition indicates that the accept transaction is only executed if the service is directed to the ambient of the *MobilityAspect* aspect.

The *COPY* transaction (see Figure 98, section 5) is invoked by an architectural element that needs to be replicated. First, the ambient collects the information needed in order to replicate an architectural element. This information consists of the instance state, the attachments located in the ambient and the bindings located in the ambient. Then, a new instance is created with the state of the requested architectural element and its attachments and bindings.

Finally, the *INTERIOR* (see Figure 98, section 8) and the *EXTERIOR* (see Figure 98, section 9) played_roles are described. The *INTERIOR* played role allows an ambient to offer the services of the *ICapabily* interface to the elements located in its boundary. The *EXTERIOR* played role allows the ambient to offer and request the accept service of the *ICapability* interface to the elements located outside the boundary of the ambient. The architectural elements located outside the boundary of an ambient can only invoke the *accept* service.

8.2.5.3 Distribution Aspects of Ambients

Each distribution aspect of an ambient must save the name of its parent ambient, must provide a service which allows other aspects of the ambient to consult its current location, and provide a service which allows architectural elements located in it to consult routes of architectural elements. Therefore, a distribution aspect uses the *IGetLocation* interface (see Figure 90) and the *IGetRoute* interface (see Figure 63).

The specification of distribution aspects of the kinds of ambients have to include a minimum specification. This specification is the same for the distribution aspect of a Group ambient and a Virtual ambient (see Figure 100). The minimum specification for the distribution aspect of a Site ambient is slightly different (see Figure 99).

All distribution aspects of ambients store the name of the parent ambient of the ambient which imports them through the *location* attribute. The aspects specify that the *getLocation* service is of kind *in/out* i.e., the service first receives the request (*in*), then the value of the *location* attribute is assigned to the *Location* parameter through the *Valuation*. Finally, the service sends the *Location* parameter (*out*). In addition, aspects specify that the *getRoute* service is also of kind in/out.

A distribution aspect of a Site ambient must include at least the specification shown in Figure 99. A distribution aspect of a Site ambient includes an attribute called *physicalLocation* of type *loc*. The *physicalLocation* attribute stores the physical location that the site represents. This attribute is a *NOT NULL* attribute because it always has to have a value. As a result, the *begin* service must have a parameter that gives a value to the *physicalLocation* attribute as well as to the *location* attribute when the aspect starts executing.

```
Distribution Aspect SiteAmbient using IGetLocation, IGetRoute
 Attributes
  …
  location : Ambient NOT NULL;
  physicalLocation: loc NOT NULL;

  …
 Services
  …

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

  in/out getLocation( output Location:Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;
  in/out getRoute(input Name:ArchitecturalElement, output Route[]: Ambient)
    Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);
```

```
 Played_Roles
   INTROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);

 ……
 Protocol
  DIST:= begin(ParentAmbient, PhysicalLocation) → DIST1;
  DIST1 :=(getLocation?(Location)→ getLocation!(Location))+
          (INTROUTE.getRoute?(Name, Route)→ INTROUTE.getRoute!(Name, Route))+
           end;

End_Distribution Aspect SiteDist
```

Figure 99. A distribution aspect of a Site ambient

Figure 100 shows the minimum specification of a distribution aspect that is imported by a Group ambient or a Virtual ambient. It can be observed, that the aspect has the *location* attribute as *NOT NULL*.

```
Distribution Aspect GroupVirtualAmbient using IGetLocation, IGetRoute
 Attributes
  …
  location : Ambient NOT NULL;

  …
 Services
  …

  begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)]
                      location := ParentAmbient,


  in/out getLocation( output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;
  in/out getRoute(input Name: ArchitecturalElement, output Route[]: Ambient)
    Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);

 Played_Roles
   INTROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);

 Protocol
  DIST:= begin→ DIST1;
  DIST1:= (getLocation?(Location)→ getLocation!(Location))+
          (INTROUTE.getRoute?(Name, Route)→ INTROUTE.getRoute!(Name, Route))+
           end;


End_Distribution Aspect GroupVirtaulDist
```

Figure 100. A distribution aspect of a Group or a Virtual ambient

Figure 101 shows the minimum specification of a distribution aspect imported by the Root ambient. It can be observed that the *location* attribute always has a value equal to NULL. This is due to the fact that a *Root* ambient does not have a parent ambient. As a result, the *begin* service of the aspect can have no parameters and no valuations.

```
Distribution Aspect RootAmbient using IGetLocation, IGetRoute
 Attributes
  Constant
  location: Ambient (NULL);

  …
 Services
  …

  begin()


  in/out getLocation( output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := NULL;
  in/out getRoute(input Name:ArchitecturalElement, output Route[]: Ambient)
    Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);

 Played_Roles
   INTROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);

 Protocol
  DIST:= begin→ DIST1;
  DIST1:= (getLocation?(Location)→ getLocation!(Location))+
          (INTROUTE.getRoute?(Name, Route)→ INTROUTE.getRoute!(Name, Route))+
           end;


End_Distribution Aspect RootDist
```

Figure 101. A distribution aspect of the Root ambient

Moreover, a distribution aspect of an ambient can include the different sections of an aspect (see Figure 92). These sections can specify if an ambient is mobile or not and the mobility strategies of an ambient. For example, Figure 102 shows a possible distribution aspect called *MobileAmbDist* of a Group ambient. It can be noticed that *MobileAmbDist* aspect provides an ambient to be mobile because it has a transaction called *MOVE* which invokes mobility services offered by its parent ambient.

```
Distribution Aspect MobileAmbDist using IMobility, ICapability, IGetLocation,
                                IGetRoute
 Attributes
  location : Ambient NOT NULL;

Services


   begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)]
                        location := ParentAmbient,

 … …
   in/out getLocation( output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;
  … …
  TRANSACTIONS in MOVE (NewAmbient: Ambient)
      move =  CapParent_startMovement!(self.Name)→ MOVE1;
      MOVE1 = CapParent_exit!(self.Name,location)→ MOVE2;
      MOVE2 = CapParent_enter!(self.Name, NewAmbient)→MOVE3;
      MOVE3 = CapParent_finishMovement!(self.Name);
      MOVE4 = CapParent_changeLocation?(Name, NewLocation);

  Preconditions
  ….

End_Distribution Aspect MobileAmbDist
```

Figure 102. A distribution aspect called MobileAmbDist aspect

8.2.6 Components, Connectors and Systems

Components, connectors and systems are specified in Ambient-PRISMA using their respective templates in APPENDIX A. However, each architectural element in Ambient-PRISMA has to import a distribution aspect.

Figure 103 shows the specification of a *Bidder* component. It can be observed, that the *Bidder* component imports the *BidderDist* distribution aspect (see Figure 103) and the *BidderFunct* functional aspect. These two aspects are coordinated through a weaving which specifies that after the execution of the *finishedAuction* service of the *BidderFunct* aspect, the move service of the *BidderDist* aspect is executed. The *Bidder* also specifies four ports: *DCapPort*, *DMovingPort*, *CustBidderPort*, and *BidderAuctPort*. The *DCapPort* port has to be specified in order

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

Each port of the *Bidder* component exports the services of different interfaces and has a different played_role defined for the interface. The played_roles of the *CustBidderPort* and the *BidderAuctPort* ports are defined by the *BidderFunct* aspect. The played_roles of the *DCapPort* and the *DMovingPort* ports are defined by the *BidderDist* aspect. After the port definitions, the creation section is specified by invoking the begin services of the *BidderDist* and *BidderFunct* aspects. Finally, the destroy service of the component is specified by invoking the end services of both aspects.

```
 Component_type Bidder
  Import Distribution Aspect BidderDist;
  Import Functional Aspect BidderFunct;

  Weavings
   //This weaving is for moving to the customer location after the auction
     has finished.

   BidderDist.move(CustLocation) after BidderFunct. finishedAuction(BidWon)
  End_Weavings

  Ports
     DCapPort: ICapability Played_Role BidderDist. CapParent;
     DMovingPort: IMobility Played_Role BidderDist.CUSTMOVESBIDDER;
     CustBidderPort: ICustBidder Played_Role BidderFunct.CUSTBIDDER;
     BidderAuctPort: IBidderAuct Played_Role BidderFunct. BIDDERAUCT;
  End_Ports


  new(input ParentAmbient: string)
       {
        BidderFunct. begin();
        BidderDist.begin(ParentAmbient);
        }
  destroy()
        {
             BidderDist.end();
             BidderFunct.end();
         }
End Component_type Bidder;
```

Figure 103. Specification of the *Bidder* component

It can be noticed from the specification of Figure 103 that an architectural element needs to import a distribution aspect in order to indicate where an instance is located at instantiation time (see new in Figure 103). In this way, each time an architectural element is instantiated, the name of the ambient where the instance is located has to be indicated.

Another example of the specification of an architectural element is the *AuctionHouse* system (see Figure 104). It can be observed, that it imports a distribution aspect called *Dist*, i.e., Ambient-PRISMA systems also have to import a distribution aspect.

It can be also observed that the attachments and the bindings specified in the system type are the same ones as specified in PRISMA (see Figure 48.). This is due to the fact that when a system is defined at the type level, it is not known where the different architectural elements it is composed of are located. As a result, the attachments that are specified are those that connect components to connectors and the bindings that are specified are those that connect systems to their components or connectors. In this way, attachments and bindings are defined in Ambient-PRISMA independently of where the architectural elements they connect are located.

```
System_type AuctionHouse
 Import Distribution Aspect Dist;
 Import Functional Aspect AuctHseFunct;
 Ports
     ProcurAuctPort: IProcurAuction Played_Role AuctHseFunct.PROCURAUCT;
     SysBidderPort: IBidderAuct Played_Role lotFunct.BIDDERAUCT;
 End_Ports

Import Architectural Elements WrappAspSystem, LotOnAuction, CnctLotAuction;


Attachments
 AttchAWrapCnct:
           CnctLotAuction.CoorWPort(1,1) ←→ WrappAspSystem.LotAuctPort(1,1);
 AttchCnctLot:
           CnctLotAuction.CoorLPort (1,n) ←→ LotOnAuction.LotAuctPort (1,1);
End_Attachements;

Bindings
  BndWrap: ProcurAuctPort(1,1)<--> WrappAspSystem.ProcurAuctPort(1,1);
```

```
  BndL: SysBidderPort(1,n)<--> LotOnAuction.LotAuctPort(1,1)
End_Bindings;
…….
```

Figure 104. Partial Specification of the *AuctionHouse* system

8.2.7 Ambients

An ambient is specified by the set of aspects it is formed of, its aspect weavings and its ports. Figure 105, shows the template of ambients. All the kinds of ambients are specified in the same way. The only difference is the reserved words that precede and end their specifications. These reserved words are *Virtual…End_Virtual*, *Site…End_Site,* and *Group…End_Group* for the specification of a virtual ambient, a site ambient and a group ambient, respectively.

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

<Site> ::= Site <site_name>
                          <aspects_importation_seq>
                          [<weavings>]
                          <ports>
                          <creation>
                          <destruction>
           End_Site <site_name>';'

<Group> ::= Group <group_name>
                          <aspects_importation_seq>
                          [<weavings>]
```

```
                                   <ports>
                                   <creation>
                                   <destruction>
                    End_Group <group_name>';'
<aspects_importation> ::= <concern> Aspect  Import <aspect_name>
<creation> ::= new'(' [<param_service_list>]')' '{' <start_aspects_seq> '}'
<destruction> ::= destroy'(' ')' '{' <stop_aspects_seq> '}'
<start_aspects> ::= <aspect_name>'.'begin'(' [<parameter_name_list>]')'
<stop_aspects> ::= <aspect_name>'.'end'(' ')'
```

Figure 105. Specification Template of Ambients

Figure 106 shows the specification of a site ambient called *HostSite*. The ambient imports the *MobilityAspect* aspect, the *ACoordination* aspect, and the *ADist* aspect. The five predefined ports of an ambient are specified with their played roles. The predefined weaving which triggers the *getLocation* service of the *ADist* aspect instead of the *getParent* service of the *MobilityAspect* aspect is also specified.

```
Ambient_Site type HostSite
 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Weavings
     ADist.getLocation(Location)instead
                                  MobilityAspect.getParent(Parent);
 End_Weavings
 Ports
     InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
     ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
     EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
     InServicesPort: ICall Played_Role ACoordination.INTERIOR;
     InRoutePort: IGetRoute Played_Role ADist.INTROUTE;
 End_Ports
End Ambient_Site type HostSite;
```

Figure 106. Specification of HostSite ambient

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

created automatically. In this way, an ambient can be reused for different architectural models with different architectural elements.

8.2.8 Attachments

In Ambient-PRISMA attachments connect a port of a component to a port of a connector, a port of a connector to a port of an ambient, and a port of an ambient to a port of another ambient. However, the only attachments that are specified in the AOADL language are the ones that connect a port of a component (or system) to a port of a connector, or attachments that connect additional ports of an ambient to a port of a connector. Additional ports of an ambient are those that are not predefined. In this way, the specification is simpler to the user, since he/she only has to define which components and connectors need a connection independently of their locations. Once an ambient is added to an architectural model, attachment types are created in order to connect an ambient with all the architectural element types of the architectural model. However, this is transparent to the user.

It can be observed from the specification of Figure 107, that only attachments that connect components (or systems) to connectors. These are the same attachments specified in the PRISMA architectural model (see Figure 42). In this way, the attachments specified in the Ambient-PRISMA AOADL are the ones specified in the PRISMA AOADL.

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

Figure 107. Some attachments of the *MobileAgentsAuctionModel* architectural model

8.3 The Configuration Level

At the configuration level, a specific software architecture is defined. As a result, types of ambients, components, connectors, and systems are instantiated by specifying where the instances are located. Then, attachments and bindings instances are created in order to connect architectural element instances together. The user only creates attachment instances that connect component to connectors instances and binding instances that connect systems to components or connectors that compose it.

The template for specifying a configuration of an architectural model is shown in Figure 108. The architectural model specification is preceded by the reserved word *Architectural_ModelConfiguration* and the name of the architectural model configuration. The possible physical locations of the software architecture are instantiated by the *loc* data type. Then ambients are instantiated. First, virtual ambients are instantiated by indicating their parent ambient with the exception of the Root ambient which does not have a parent ambient. Second, each site ambient is instantiated by indicating both its parent ambient and an instance of the *loc* data type. Finally, group ambients are instantiated by indicating their parent ambient.

Then, possible constraints to ambients connections can be specified. These are specified by indicating which specific ports of an ambient cannot be connected to specific ports of other ambients. This allows the user of the AOADL to indicate that certain ambients are not allowed to communicate. Afterwards, components and connectors are instantiated by indicating where they are located (their parent ambients). Finally, attachments are instantiated.

```
<architectural_model_configuration> ::=
        Architectural_Model_Configuration  <configuration_name> '='
            new <model_name> '{'<loc_instantiation_seq>
                                 [<virtualAmbient_instantiation_seq>]
```

```
                                        <siteAmbient_instantiation_seq>
                                        [<groupAmbient_instantiation_seq>]
                                        <components_instantiation_seq>
                                        [<systems_instantiation_seq>]
                                        <connectors_instantiation_seq>
                                        <attachments_instantiation_seq> '}'
<LOC_instantiation> ::= <loc_name> '=' new
                                 loc'(' [<param_value _list> ] ')
<virtualAmbient_instantiation> ::= <virtualAmbient_instance_name> '=' new
                                 <virtualAmbient_name>'(' [<param_value _list> ] ')
<siteAmbient_instantiation> ::= <siteAmbient_instance_name> '=' new
                                 <siteAmbient_name>'(' [<param_value _list> ] ')
<groupAmbient_instantiation> ::= <groupAmbient_instance_name> '=' new
                                 <groupAmbient_name>'(' [<param_value _list> ] ')
<ambient_Constraints> ::= <ambient_instance_name> '.' <port_name> '<←X→'
                          <ambient_instance_name> '.' <port_name>

<components_instantiation> ::= <component_instance_name> '=' new
                                 <component_name>'(' [<param_value _list> ] ')
<connectors_instantiation> ::= <connector_instance_name> '=' new
                                 <connector_name> '(' [<param_value_list> ] ')
<attachments_instantiation> ::= <attachment_instance_name> '=' new
                                <attachment_name> '('<param_attachment_value>')'
<systems_instantiation> ::= <system_instance_name> '=' new <system_name>
      '('[<param_service_value_list>','] <architectural_element_number_value_list>,
        [<attachment_number_value_list>',' <binding_number_value_list>] ')'
            '{' [<start_aspects_seq>] <architectural_elements_instantiation_seq>
              <attachments_instantiation_seq> <bindings_instantiation_seq> '}'
<architectural_element_instantiation > ::= <components_instantiation> |
                                           <connectors_instantiation> |
                                           <systems_instantiation>
```

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

Figure 108. Specification Template of configurations of ambient-PRISMA architectural models

Figure 109, shows the specification of the configuration called *MobileAgentsAuctionConf* of the *MobileAgentsAuction* architectural model. The configuration consists of giving the possible physical locations by instantiating the loc data type. Then the ambients are instantiated. First, the *Root* ambient is instantiated. It can be noticed that the constructor of the *Root* ambient does not have any parameters. This is due to that fact the *Root* ambient is not located in any ambient. Then, site ambients are instantiated by indicating in their constructor the name of their parent ambient (their location) and the physical address they represent. For example, an ambient instance called *ClientSite* is created from the *HostSite* type ambient (see Figure 106). The constructor of the *ClientSite* instance indicates that it is located in the *Root* ambient and that its physical location is *IP1*. Then an ambient constraint is specified. This constraint indicates that all the ports of the *ClientSite* ambient and the *LondonSaleRoomSite* ambient cannot be connected. Then, components, connectors and systems are instantiated by indicating their locations in their constructors. In addition, the attachments and binding relationships are also instantiated by connecting the instances created from the components, connectors and systems types.

```
Architectural_Model_Configuration MobileAgentsAuctionConf =

New MobileAgentsAuction
{
       IP1 = new loc(ip.of.host.1);
       IP2 = new loc(ip.of.host.2);
       IP3 = new loc(ip.of.host.3);

       ROOT = new Root() ;

       ClientSite = new HostSite(ROOT, IP1);
       AuctionSite = new HostSite(ROOT, IP2);
       LondonSaleRoomSite = new HostSite(Root, IP3);

       ClientSite.* ←X→ LondonSaleRoomSite.*;
```

```
AuctionHouse1 = new AuctionHouse (1,1,1,1,1,1,1, "AuctionSite")
                { WrapAspSys = new WrapAspSys();
                  LotOnAuction1 =new (1235, London, 1678, 17/10/2008,
                                    "Painting", 500, 18/10/2008);
                  CnctLotAuction = new CnctLotAuction();
                  …..
                }
Customer1 = new Customer("ClientSite");
Procurement1 = new Procurement ("ClientSite");
Bidder1 = new Bidder("painting", "3 Jul", "ClientSite");
AgentCustCnct1 = new AgentCustCnct("ClientSite");
AuctionHouseCnct1 = new AuctionHouseCnct("AuctionSite");
AttchAuct1Cnct = new AttchAuctCnct (AuctionHouseCnct1,
                                    CnctAuctPortBidder,
                                    AuctionHouse1,
                                    BidderAuctPort);
AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                              AuctionHouseCnct1, CustPortAuct);
AttchProc1Auc1 = new AttchCustAuc (Procurement1, PROCURAUCTPort,
                              AuctionHouseCnct1, ProcurPortAuct);

 …….
```

Figure 109. Specification of the MobileAgentsAuctionConf configuration

The configuration of an architectural model in Ambient-PRISMA is defined by instantiating ambients and assigning each architectural element instance in an ambient. In this way, the hierarchy of a distributed system can be modelled in Ambient-PRISMA.

8.4 Conclusions

This chapter has presented the Ambient-PRISMA AOADL. The Ambient-PRISMA AOADL enriches the PRISMA AOADL with new constructs in order to specify aspect-oriented software architectures of distributed and mobile software systems. The Ambient-PRISMA AOADL includes the syntactical constructs needed for specifying

ambients and architectural elements that are deployed in ambients which can be mobile.

The Ambient-PRISMA AOADL takes advantage of defining software architectures at two levels of abstraction: the type definition level and the configuration level. The type definition level specifies a software architecture by using interfaces, aspects, components, connectors, systems and ambients types. In this way, a software architecture can be specified by reusing distribution and mobility properties. For example, the properties specified in a distribution aspect can be reused in different architectural elements. In addition, maintainability is achieved since the distribution and mobility properties are separated from the rest of properties of a software system. Ambient types can be also defined by reusing interfaces and aspects as well as ambients can be reused in different architectural models. This is achieved due to the fact that ambient types are defined independently of the architectural elements that they locate. Also, at the type definition level, architectural element types are defined independently of the ambients they are located. At the configuration level, ambient types are instantiated and the hierarchy of a tree of ambients is determined by locating subambients in ambients. Each component, connector, and system is instantiated and deployed in an ambient. Attachment and binding instances are created in a transparent way due to the fact that the user does not have to consider attachments or bindings between architectural elements and ambients. In this way, distributed communication is transparent.

The Ambient-PRISMA AOADL has predefined constructs that provide the definition of distributed and mobile software architectures in a transparent way. These constructs offer predefined functionalities that are used in the definition of distributed and mobile architectural elements.

Finally, the Ambient-PRISMA AOADL is a technology independent language that facilitates the automatic code generation of distributed and mobile aspect-oriented architectural models. As a result, the development time is reduced and the traceability is preserved between an architectural model and its application code.

The work related to the Ambient-PRISMA AOADL has produced a set of results that have been materialized in the following publications:

- **Nour Ali**, Jennifer Perez, Isidro Ramos, High Level Specification of Distributed and Mobile Information Systems, Second International Symposium on Innovation in Information & Communication Technology (ISSICT 2004), April, Amman, Jordan, 2004.
- **Nour Ali**, Josep Silva, Javier Jaén, Isidro Ramos, José Ángel Carsí, Jennifer Pérez, "Mobility and Replication Patterns in Aspect-oriented component Based Software Architectures", 15th IASTED International Conference, PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS~PDCS 2003~, November 3-5, Marina del Rey, CA, USA, 2003.
- **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "Distribution in an Aspect-Oriented Component Based Software Architecture through PRISMA", Technical Report DSIC-II/14/04, Polytechnic University of Valencia, October 2004.
- **Nour H. Ali**, Josep F. Silva, Javier Jaen, Isidro Ramos, Jose Á. Carsí, Jennifer Pérez, "Distribution Patterns in Aspect-Oriented Component-Based Software Architectures", IV Workshop Distributed Objects, Languages, Methods and Environments (DOLMEN), pp.74-80, Alicante, November, 2003.

CHAPTER 9

AMBIENT-PRISMA FORMALIZATION

"The opposite of a correct statement is a false statement. But the opposite of a profound truth may well be another profound truth."

Niels Bohr

9.1 Introduction

Formal models are important for specifying accurate designs. As a result, Ambient-PRISMA concepts are formalized in process algebra in order to express in an accurate way the behaviour captured in its specifications. In this way, software architecture configurations specified using the Ambient-PRISMA language are formal and loose ambiguity.

The formalism which is used for formalizing Ambient-PRISMA is Channel Ambient Calculus (ChAC) [Phi05]. The ChAC is a process algebra for specifying mobile applications that provides channels and ambients as first-class citizens. The choice of ChAC as the target formalism allows us to take advantage of the previous formalization of PRISMA performed in the polyadic π-calculus [Per06b]. Indeed, Ambient-PRISMA is a conservative extension of PRISMA, hence the common constructs of PRISMA and Ambient-PRISMA are mapped to ChAC by using the mapping from PRISMA to π-calculus proposed in [Per06b] and the mapping from π-calculus to ChAC presented in [Phi05].

This chapter is structured as follows: Section 9.2 motivates the use of ChAC for formalizing Ambient-PRISMA and presents the notation used in the formalization. Section 9.3 presents how the concepts of Ambient-PRISMA are formalized. Finally, section 9.4 presents some conclusions.

9.2 Channel Ambient Calculus

π-calculus [Mil99] is a process calculus for modelling concurrent interacting computations. It is based on the notion of naming. Names can be channels or data. The basic notion of action that can be performed is interaction between processes. Interaction is achieved by sending or receiving values (names) through channels (names). The variant of π-calculus called polyadic π-calculus allows a channel to send or receive many names.

π-calculus allows channels to be transferred from a process to another and channels between processes can change as they communicate. In this way, π-calculus provides modelling mobility and reconfiguration (evolution of structure). It is important to emphasize that the mobility which can be modelled in π-calculus is called virtual mobility (as denominated by Milner). In virtual mobility, movement is represented by the change of links among processes i.e., channels between processes move but processes do not. On the other hand, other forms of mobility exist such as physical mobility. In physical mobility, processes move from one physical space to another. This type of mobility is the one supported by Ambient Calculus [Car98a]. In Ambient Calculus (see section 5.2), names are ambients and the basic notion of action is movement from one ambient to another (instead of interaction as in π-calculus).

To model real distributed and mobile systems both interaction and mobility are essential. This is due to the fact that components need to interact among each other while they are in different physical spaces and in some cases they need to move from one physical space to another. In addition, virtual mobility and physical mobility are important because physical mobility can cause the change of links among

processes. As a result, a process algebra that provides both ambients and channels is of great interest.

ChAC [Phi05] is a process algebra that specifies mobile applications. It is inspired by the π-calculus [Mil99], the Nomadic π-calculus [Woj00], the Ambient Calculus [Car98a] (see section 5.2), and the Boxed Ambient Calculus [Bug01]. ChAC defines channels as first-class citizens in order to allow ambients to communicate with each other. Also, ChAC uses the notion of ambient, as introduced in Ambient Calculus, to model the entities of a mobile application.

ChAC is well-suited to formalize Ambient-PRISMA for the following reasons:

- It is suitable to model the mobility and communication mechanisms of Ambient-PRISMA, since it provides channels for communication as well as ambient capabilities for mobility in a way similar to Ambient-PRISMA.
- The formalization of PRISMA [Per06b] without distribution and mobility has been performed in π-calculus. Since ChAC is inspired by the π-calculus, the PRISMA formalization only has to be extended to incorporate the mobility and distribution characteristics of Ambient-PRISMA. In addition, π-calculus primitives can be easily encoded in ChAC.

Table 6. The notations used in this chapter

x, y, z		Names
$\vec{x}, \vec{y}, \vec{z}$		tuple of names
$x@\vec{x}$		x is an element of the tuple $\vec{x}$
P, Q ::=	$\emptyset$	Null
	$P \mid Q$	Parallel Composition
	$P+Q$	Non-deterministic choice (
	$\upsilon n\ P$	Restriction
	$a[P]$	Ambient
	$\alpha .P$	Action
α ::=	$a.\ x\langle v\rangle$	Sibling Output
	$x^{\uparrow}\langle v\rangle$	Parent Output
	$x(u)$	Internal Input

	$x^{\uparrow}(u)$	External Input
	$x\langle v\rangle$	Local Output
	$a/x\langle v\rangle$	Child Output
	$in\ a \cdot x$	Enter
	$out\ x$	Exit
	$\overline{in}\ x$	Accept
	$\overline{out}\ x$	Release
$A(\vec{x}) \stackrel{def}{=} P_A$		Process definition
$\prod_{i=1}^{n} P_i$		$P_1 \mid \ldots \mid P_n$
if *v* ***then*** P_1 ***else*** P_2		Conditional (derived operator)
$[x=b]$		Comparison
$a?[b]P \div Q$		$a(x)$ ***if*** $x=b$ ***then*** P ***else*** Q

Table 6 shows the notations used for formalizing Ambient-PRISMA. A name can be a name of an ambient or a channel. For example, x, y, z are names. A tuple of names can be defined as $\vec{x}, \vec{y}, \vec{z}$. A tuple that contains a specific name can be indicated as $x@\vec{x}$.

A process can be the inactive process $\varnothing$, a parallel composition of processes $(P|Q)$, a non deterministic choice of processes $(P+Q)$ i.e. either P or Q is executed, a process with a restricted (or local) name $(\upsilon n\ P)$, a process localized in an ambient $(a[P])$, or a process prefixed by an action $(\alpha .P)$. An action can be either a communication or a migration. The communication actions allow ambient processes to send and receive values to and from processes of other ambients using channels. The migration actions allow an ambient to move in and out of each other over channels.

The $a.\ x\langle v\rangle$ sends a value v on channel x to a sibling ambient a. The $x^{\uparrow}\langle v\rangle$ sends a value v on channel x to the parent ambient. The $x(u)$ receives a value u on channel x from inside the current ambient. The $x^{\uparrow}(u)$ receives a value u on channel x from

outside the current ambient. The $x\langle v\rangle$ sends a value v on channel x locally. The $a/x\langle v\rangle$ sends a value v on channel x to a child ambient.

The $in\ a \cdot x$ enters a sibling ambient over a channel x. The $out\ x$ leaves a parent ambient over channel x. The $\overline{in}\ x$ accepts a sibling ambient over x. The $\overline{out}\ x$ releases a child ambient over channel x.

Every process identifier A has a defining equation $A(\vec{x}) \stackrel{def}{=} P_A$ where P_A is a process and $\vec{x} = x_1 \ldots, x_n$ (all distinct). The notation $\prod_{i=1}^{n} P_i$ is used to indicate the parallel composition of a set of processes. The operation $[x=b]$ introduces the mechanism of comparing two names and only executes a process P when the names are equal.

9.3 Formalization of Ambient-PRISMA Concepts

In this section, concepts of Ambient-PRISMA are formalized by translating them into the following:

- components and connectors into ChAC ambients
- ports into processes
- aspects into processes
- ambients into ChAC ambients (of course, the former are in Ambient-PRISMA while the latter are in ChAC ambients)
- attachments are mapped into processes
- bindings are mapped into processes

In the following sections the details of the mappings for each Ambient-PRISMA construct is presented.

9.3.1 Simple Architectural Elements

Simple architectural elements are components and connectors. These are formed by a set of aspects, their weaving relationships and one or more ports.

Each architectural element is formed by the following process:

$$P_{IdAE}(\overrightarrow{Asp}, \overrightarrow{W}, \overrightarrow{Port}) \stackrel{def}{=} \prod_{i=1}^{a} Asp_i \mid \prod_{i=1}^{w} W_i \mid \prod_{i=1}^{p} Port_i$$

where *IdAE* is the identifier of an architectural element, $\overrightarrow{Asp}$ is a tuple of aspects (of length *a*), $\overrightarrow{W}$ is a tuple of weavings (of length *w*), and $\overrightarrow{Port}$ is a tuple of ports(of length *p*.

Each architectural element consists of the parallel composition of the processes of its ports (see section 9.3.4), weavings (see [Per06b]) and aspects (see section 9.3.6).

9.3.2 Components

A component is formed by a set of aspects, their weaving relationships and one or more ports. A component in Ambient-PRISMA can be a mobile element. The movement of a component involves moving its aspects, weavings and ports. As a result a component is modelled as a ChAC ambient where the P_{IdAE} process (see section 9.3.1) executes. The definition of a component is as follows:

$$Id_{Comp}(\overrightarrow{Asp}, \overrightarrow{W}, \overrightarrow{Port}) \stackrel{def}{=} Id_{Comp}\,[P_{IdAE}(\overrightarrow{Asp}, \overrightarrow{W}, \overrightarrow{Port})]$$

A component which has Id_{Comp} as its identifier is mapped to a ChAC ambient called Id_{Comp}. The identifier of a component is its name. The Id_{Comp} ambient contains the P_{IdAE} process which models the parallel composition of its aspects, its set of weavings, and its set of ports.

For example, consider the *Bidder* component given in Figure 103. The component has a name (*Bidder*), a distribution aspect (*BidderDist*), a functional aspect (*BidderFunct*), a weaving process and a set of ports. Its formalization is $Bidder[P_{BidderDist} \mid P_{BidderFunct} \mid P_W \mid P_{DCapPort} \mid P_{DMovingPort} \mid P_{CustBidderPort} \mid P_{BidderAuctPort}]$.

```
 Component_type Bidder
  Import Distribution Aspect BidderDist;
  Import Functional Aspect BidderFunct;

  Weavings
   BidderDist.move(CustLocation) after BidderFunct. finishedAuction(BidWon)
  End_Weavings

  Ports
     DCapPort: ICapability Played_Role BidderDist. CapParent;
     DMovingPort: IMobility Played_Role BidderDist.CUSTMOVESBIDDER;
     CustBidderPort: ICustBidder Played_Role BidderFunct.CUSTBIDDER;
     BidderAuctPort: IBidderAuct Played_Role BidderFunct. BIDDERAUCT;
  End_Ports

.....
      }
End Component_type Bidder;
```

Figure 110. Specification of the *Bidder* component

9.3.3 Connectors

A connector is formed by a set of aspects, their weaving relationships and one or more ports. A connector in Ambient-PRISMA can also be a mobile element. The movement of a connector involves moving its aspects, weavings and ports. As a result a connector is modelled as a ChAC ambient where the P_{IdAE} process (see section 9.3.1) executes. as follows:

$$Id_{Con}(\vec{Asp}, \vec{W}, \vec{Port}) \stackrel{def}{=} Id_{Con}\,[P_{IdAE}(\vec{Asp}, \vec{W}, \vec{Port})]$$

A connector which has Id_{Con} as its identifier is mapped to a ChAC ambient called Id_{Con}. The identifier of a connector is its name. The Id_{Coon} ambient contains the P_{IdAE} process which models the parallel composition of its aspects, its set of weavings, and its set of ports.

For example, consider the *AuctionHouseCnct* connector given in Figure 111. The connector has a name (*AuctionHouseCnct*), a distribution aspect (*Dist*), a functional aspect (*AuctCnctrCoor*), and a set of ports. Its formalization is $AuctionHouseCnct[P_{Dist} \mid P_{AuctCnctrCoor} \mid P_{AuctPortCust} \mid P_{AuctPortCust} \mid P_{CustPortAuct} \mid P_{CnctAuctPortBidder} \mid P_{BidderPortAuct} \mid P_{BidderPortAuct} \mid P_{ProcurPortAuct} \mid P_{AuctPortProcur}]$.

```
Connector_type AuctionHouseCnct
 Import Distribution Aspect Dist;
 Import Coordination Aspect AuctCnctrCoor;
 Ports
    AuctPortCust: ICustAuct;
    CustPortAuct: ICustAuct;
    CnctAuctPortBidder: IBidderAuct;
    BidderPortAuct: IBidderAuct;
    ProcurPortAuct: IProcurAuction;
    AuctPortProcur: IProcurAuction;
 End_Ports
……..

End Connector_type AuctionHouseCnct;
```

Figure 111. Specification of the *AuctionHouseCnct* connector

9.3.4 Ports

Ports are points where the architectural element receives and sends a set of services. The services that a port can receive or send are specified in an interface. Services have *input* parameters which are the arguments, and *output* parameters which are the return values.

For example, the *DMovingPort* port of Figure 103 can receive and send services of an interface called *IMobility*. The *IMobility* interface is specified in Figure 112 and it consists of a service called *move* that takes a string input.

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

Figure 112. Specification of the *IMobility* interface

A port can send requests or receive invocations of services published by an interface. As a result, a port is mapped into the following process P_{Port}:

$$P_{Port}(n, \vec{x}, \vec{y}, \vec{s}) \overset{def}{=} P_{PortSender}(\text{is}, \vec{x}, \vec{r_y}, \vec{s}) \mid P_{PortRec}(n, \vec{x}, \vec{y}, \vec{s})$$

where *Port* is the name of a port, n is a name of an invoked or a requested service, $\vec{x}$ is a tuple of channels for the *input* parameters $x_{1,\ldots,} x_n$, $\vec{y}$ is a tuple of channels for the output parameters $y_{1,\ldots,} y_n$, and $\vec{s}$ is a tuple of service names $s_{1,\ldots,} s_n$, $P_{PortSender}$ sends requests of services and $P_{PortRec}$ receives invocations of services in $\vec{s}$.

The port process consists of the parallel composition of the $P_{PortSender}$ process and the $P_{PortRec}$ process. This is due to the fact that an architectural element can receive and send services at the same time.

$P_{PortSender}$ process is defined as follows assuming there is a mismatch operator:

$$
\begin{aligned}
P_{PortSender}(is, \vec{x}, \vec{r_y}, \vec{s}) \overset{def}{=} \; & is(\vec{x}, \vec{r_y}) \\
& (\textbf{\textit{if}}\ [is = s_1] \\
& \qquad \textbf{\textit{then}}\ Request(s_1, \vec{x}, \vec{r_y}) \\
& \textbf{\textit{else}}\ \ldots. \\
& \textbf{\textit{else}}\ \textbf{\textit{if}}\ [is = s_n] \\
& \qquad \textbf{\textit{then}}\ Request(s_n, \vec{x}, \vec{r_y})).\ P_{PortSender}(\text{is}, \vec{x}, \vec{y}, \vec{s})
\end{aligned}
$$

$$Request(s, \vec{x}, \vec{r_y}) \overset{def}{=} s^{\uparrow} \langle \vec{x}, \vec{r_y} \rangle \, . \, r_{y1}^{\uparrow} (y_0) \ldots r_{ym}^{\uparrow} (y_m))$$

where *is* is a channel of the invoked services from an aspect, $\vec{x}$ a tuple of channels for the *input* parameters, $\vec{r_y}$ is a tuple of channels for the *output* parameters $r_{y1}, \ldots, r_{ym}$, and $\vec{s}$ is a tuple of service names $s_{1,\ldots,} s_n$.

The port requests a service only when an aspect of an architectural element requests it through an *is* channel (the channel that represents a service of an

aspect). The $P_{PortSender}$ process compares the *is* channel with the $\vec{s}$ channels in order to match it with an s channel. Once the *is* channel is matched with a s channel, the process forwards the request on a channel *s* that sends the input parameters and the channels created for the output parameters to a process outside the ambient which represents the architectural element (it sends it to the attachments, see section 9.3.8). Then the process waits in order to receive the results of the output parameters $y_0,\ldots, y_m$ through the $r_{y1}, \ldots, r_{ym}$ channels. Finally, the $P_{PortSender}$ process is prepared to receive a new service request.

For example, the $P_{MovePortSender}$ of the *MovePort* port in Figure 103 is formalized as follows:

$$imove(NewAmbient, ack).move^{\uparrow}\langle NewAmbient, ack\rangle.ack^{\uparrow}(true))$$

where *true* is the result of the acknowledgement (or return value).

The $P_{PortRec}$ process is defined as follows:

$$P_{PortRec}(s, \vec{x}, \vec{y}, \vec{s}) \stackrel{def}{=} s^{\uparrow}(\vec{x}, \vec{y}).\upsilon(is_{1,\ldots,}is_n).$$

$$(\textbf{ if } [s = s_1] \textbf{ then } Invoke(is_1, \vec{x}, \vec{y})$$
$$\textbf{else } \ldots.$$
$$\textbf{else if } [s = s_n] \textbf{ then } Invoke(is_n, \vec{x}, \vec{y})).P_{PortRec}(n, \vec{x}, \vec{y}, \vec{s})$$

$$Invoke(is, \vec{x}, \vec{y}) \stackrel{def}{=} is(\vec{x}, \vec{y}).\vec{y}^{\uparrow}\langle v_{0,\ldots,}v_m\rangle$$

where s is the channel of the invoked service, $\vec{x}$ a tuple of channels for the *input* parameters, $\vec{y}$ is a tuple of channels for the *output* parameters , and $\vec{s}$ is a tuple of service names $s_{1,\ldots,}s_n$.

The $P_{PortRec}$ process receives the service invocation ($s^{\uparrow}(\vec{x}, \vec{y})$) with the $\vec{x}$ input parameters and the $\vec{y}$ channels for returning the output parameters from outside of the ambient which represents the architectural element (it receives the invocation from the attachments see section 9.3.8). Then the $P_{PortRec}$ process creates channels which represent the services in an aspect. The $P_{PortRec}$ process matches the s channel with one of the is channels of an aspect. Once the service is matched, the process sends the service invocation to an aspect of an architectural element and waits to send the return values using the output channels. Finally, the $P_{PortRec}$ process is prepared to receive a new service invocation.

For example, the $P_{MovePortRec}$ of the *MovePort* port in Figure 103 is formalized as follows:

$$move^{\uparrow}(NewAmbient, ack).\ \upsilon(imove).imove(NewAmbient, ack).ack^{\uparrow}\langle true \rangle$$

where *true* is the result of the acknowledgement (or return value).

9.3.5 *DCapPort* Port

A mobile architectural element needs to specify a port called *DCapPort*. The *DCapPort* port allows an architectural element to invoke mobility services of the *ICapability* interface, including the *exit* and the *enter* services (see Figure 113). The exit service is invoked by an architectural element to its ambient in order to allow it to leave. The enter service is invoked by an architectural element to its ambient in order to allow it to be accepted in a sibling ambient.

```
Interface ICapability
   …
   exit (input Name: ArchitecturalElement);
   enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
   ….
End_Interface ICapability
```

Figure 113. Partial Specification of ICapability Interface

The *exit* and *enter* services are different from the rest of services. This is due to the fact that the satisfaction of the exit request means that the architectural element can exit its parent ambient through a channel and that the satisfaction of the enter request means that the architectural element can enter a sibling ambient. As a result, the exit and enter services receive an exit channel or a enter channel, respectively through their return channels. In this way, an architectural element can execute the *in* and *out* actions of ChAC.

As a result, the $P_{PortSender}$ process of the *DCapPort* port is mapped as follows:

$$P_{DCapPortSender}(iexit, ienter, ..., Name, NewAmbient, ..., \vec{r}_y, exit, enter, ...) \stackrel{def}{=}$$

$$\begin{array}{l} \textbf{if}\ iexit(Name, r_y). \\ \quad \textbf{then}\ EXIT(exit, Name, r_y) \\ \textbf{if}\ ienter(Name, NewAmbient, r_y) \\ \quad \textbf{then}\ ENTER(enter\ Name, NewAmbient, r_y) \\ \\ \textbf{else}\ .... \\ \textbf{else}\ .... \\ \qquad \textbf{then}\ Request(s_n, \vec{x}, \vec{r}_y)).\ P_{DCapPortSender}(is, \vec{x}, \vec{y}, \vec{s}) \end{array}$$

$$EXIT(exit, Name, r_y) \stackrel{def}{=} exit^{\uparrow} \langle exit, Name, r_y \rangle . r_y^{\uparrow} (y_1).out\ y_1$$

$$ENTER(enter, Name, NewAmbient, r_y) \stackrel{def}{=} enter^{\uparrow} \langle enter, Name, NewAmbient, r_y \rangle . r_y^{\uparrow} (y_1). in\ NewAmbient \cdot y_1$$

where *iexit* and *ienter* are channels of the requested services received from a distribution aspect, *Name, NewAmbient* are tuple of channels for the *input* parameters, $\vec{r}_y$ is a tuple of channels for the *output* parameters $r_{y1}, ..., r_{ym}$, and *exit* and *enter* are the channels for services for sending the requests.

It is assumed that each ambient has two unique ambient channels: one for accepting ambients and the other for releasing ambients. In this way, when an architectural element requests the exit of an ambient and the ambient accepts to

release it, an architectural element receives through the r_y channel the name of a channel where it can leave an ambient (*out* y_1). Also, when an architectural element requests to enter an ambient and the ambient accepts it, it receives through the r_y channel a name of a channel where it can enter an ambient ($in\ NewAmbient \cdot y_{1y}$).

9.3.6 Aspects

Aspects define the state and behaviour of an architectural element from a specific concern of a software system. The state is defined by an aspects attributes and the behaviour is defined by services. An aspect has been formalized in [Per06b] as follows:

- A: a set of attributes
- X: a set of services
- Φ: a set of formulae in modal logic of actions. These formulae define how and when the services of an aspect change the value of the attributes of the aspect.
- π: a set of terms in π-calculus

It can be noticed that an aspect is formalized using both modal logic of actions and π-calculus. In this section, an aspect is formalized entirely in π-calculus. In this way, the formalization of an aspect is performed in the same formalism.

The distribution aspect has been chosen in order to demonstrate the π-calculus formalization of aspects due to the fact that it has a predefined attribute called *location* which stores the parent ambient name. In this way, the formalization can be performed by showing how services change the value of the *location* attribute.

The definition of a Distribution Aspect process in π-calculus is the following:

$$Did_{DA}(location, begin, end, \vec{X} \# \vec{S}, \vec{Y} \# \tau) \overset{def}{=} begin(Initial).(((S_1| \ldots |S_n)| \Pi \\ | Loc(location,initial)) + end().0)$$

where *Did* is the identifier of a distribution aspect, *location* the variable attribute that stores the parent ambient name, *begin* the initializing service, *end* the destruction service, $\vec{X}\#\vec{S}$ the set of services names that are invoked by a played_role and $\vec{Y}\#\tau$ the name of a transaction.

The process of a distribution aspect starts by receiving the initialization parameters through the *begin* channel. Then, the process can in parallel execute a set of services, a set of π-calculus formulae (valuations and protocols), and a process that changes a value of a *location* attribute or receives an invocation of the end service which terminates the process.

The formalization of the sections of a Distribution Aspect is the following:

9.3.6.1 Attributes

The location attribute is translated into a π-calculus process called *Loc*. The definition of *Loc* is as follows:

$$
\begin{aligned}
Loc(location, initial) \overset{def}{=} & (\upsilon\ val)(location(op,r) \\
& \textbf{if}\ op{=}set \\
& \quad \textbf{then}\ val(x).\overline{val}\langle r \rangle .Loc(location,r) \\
& \textbf{else}\ val(x).\overline{r}\langle x \rangle \mid \overline{val}\langle x \rangle .Loc(location,x) \\
& \quad \mid \overline{val}\langle initial \rangle
\end{aligned}
$$

where *location* is a channel which receives whether a service is a *set* (changes the value of the location attribute) or a *get* (consults the value of the location attribute), and *initial* is the initial value stored in the location attribute.

The *Loc* process consists of creating a private channel called *val* each time a service is invoked on the *location* channel. If the service is a *set* operation that changes the value of the location attribute, then the previous value of *val* is cancelled and the value of the *r* parameter is sent to *Loc*. If the service is a *get* operation, then the

previous value is cancelled and the value of *location* is sent on the return channel *r*. After that, the value of location is sent again.

9.3.6.2 Valuations

Valuations define the change of the state of an aspect when one of its services is executed. In modal logic of actions, they are formalized by the following formula:

$$\psi \rightarrow [a]\ \varphi$$

This means that if ψ is satisfied (a precondition) before the execution of a, φ must be satisfied (a postcondition) after the execution of a. ψ and φ are wff that make reference to the states before and after the execution of *a*, respectively.

The φ postcondition in π-calculus is a process defined as follows:

$$P\varphi \stackrel{def}{=} (\upsilon\ r)\ (\mathrm{E}_r\ (\varphi) \mid r(v).location\langle set,\ v\rangle\ )$$

$$\mathrm{E}_r\ (\varphi) \stackrel{def}{=} location\ \langle get,r\rangle\ .0$$

The process consists of creating a name called r . r is used as the value of the location attribute which is sent to the *Loc* process when the service is a *get* and it is used as a channel in order to receive the value of the attribute from the *Loc* process when the service is a *set*. The cases where φ is a complex expression are left unspecified.

9.3.6.3 Service Execution

The execution of a service is defined as a process called *S* as follows:

$$S(X\#S,\ X\#S_status,\ \vec{p}\ ) \stackrel{def}{=} X\#S(\ \overrightarrow{x}\ ,\ r_0 @ \overrightarrow{r}_y\ ).\ X\#S_status(check)$$

if *check*
then
  if $\mathrm{E}\ (\psi)$
    then $r_0\ \langle ok\rangle\ .\ r_0\ (ok)(\ P\varphi$

$$
\begin{aligned}
&|S(X\#S,\ X\#S_status,\ \vec{p}\,)) \\
&\div S(X\#S,\ X\#S_status,\ \vec{p}\,) \\
&\textbf{\textit{else}}\ r_0 \langle ko \rangle . r_0 ().S(X\#S,\ X\#S_status,\ \vec{p}\,) \\
\textbf{\textit{else}}\ &r_0 \langle ko \rangle . r_0 ().S(X\#S,\ X\#S_status,\ \vec{p}\,)
\end{aligned}
$$

where *X#S* is the name of a services that is invoked by a played_role, *X#S _status* is a channel for checking the protocol status, and $\vec{p}$ are the input ($\vec{x}$) and output ($r_0 @ \vec{r}_y$) parameters of the service.

The *S* process consists of the following:

- When an invocation is received on the service channel from a distribution aspect played role, the protocol of the distribution aspect sends if the service invocation is on a valid state of the protocol. If the protocol answers then this means that the service can be invoked. If it does not answer, the process sends to the transaction (that originated the service) that there a failure has occurred by using a special return channel and the next service is invoked.
- When the protocol answers that the service satisfies a valid state, the precondition of the valuation is checked. If it is satisfied: on a special return channel an *ok* is sent to the transaction that originated the service (if the service is part of a transaction), then it receives from the protocol an acknowledgment. After that, the assignment of the valuation is performed $P\varphi$ and in parallel the protocol moves to the next state.

9.3.6.4 Protocols

A protocol specifies which services are permitted to be executed on a determined state of an aspect. As a result, a protocol determines when a service can be executed.

Protocols are translated to a process which is defined as follows:

$$PR \stackrel{def}{=} PR_{cl} + PR_s$$

$$PR_{cl}(X\#S, \vec{x}, \vec{r}) \stackrel{def}{=} (\upsilon r_0)(X\#S \langle \vec{x}, r_0 @ \vec{r}_y \rangle . r_0(ack) \text{ if } ack = ok \text{ then } r_0 \langle ok \rangle \text{ else } r_0 \langle \, \rangle$$

$$PR_{cl}(X\#S, \vec{x}, \vec{r}) \stackrel{def}{=} X\#S_status! \langle \, \rangle$$

The protocol of the client creates a special return channel called r_0 each time it invokes a service. On the channel that represents the service name (*X#S*), it sends the input parameters and the r_0 and the output channels created for the output parameters. After that, it waits to receive an acknowledgement that the service has been satisfactory executed. The protocol of the client sends to the *S* process whether the service can be executed or not in the current state of an aspect.

9.3.7 Ambients

Ambient-PRISMA ambients are architectural elements that can locate other architectural elements. They are distinguished from other architectural elements because they can release and accept new architectural elements in them.

In the following we show the definition of an ambient in ChAC:

$$Id_{Amb}(\vec{Amb}, \vec{Comp}, \vec{Con}, \vec{ATCH}, \vec{BIND}, \vec{Asp}, \vec{W}, \vec{Port}) \stackrel{def}{=} Id_{Amb}[\prod_{i=1}^{a} IdAmb_i \mid \prod_{i=1}^{c} IdComp_i \mid \prod_{i=1}^{n} IdCon_i \mid \prod_{i=1}^{t} PATCH_i \mid \prod_{i=1}^{t} PBIND_i \; P_{IdAE}(\vec{Asp}, \vec{W}, \vec{Port})]$$

where $\overrightarrow{Amb}$ is the tuple of subambients (of length a), $\overrightarrow{Comp}$ is a tuple of components (of length c), $\overrightarrow{Con}$ is a tuple of connectors, $\overrightarrow{ATCH}$ is the tuple of attachments where an attachment can connect a port of one of the architectural elements located in it to a port of another architectural element located in it or an attachment can connect a port of an architectural element located in it to a port of the ambient, $\overrightarrow{BIND}$ is the tuple of bindings that connect a system to one of its architectural elements or that connect a system to its parent ambient, $\overrightarrow{Asp}$ is the tuple of aspects of the ambient, $\overrightarrow{W}$ is the tuple of weavings of the ambient, and $\overrightarrow{Port}$ is the tuple of ports of the ambient. As a result, $_{Amb\,1} \ldots {}_{Amb\,n}$ does not include $_{Amb}$.

Each ambient is defined by a ChAC ambient that contains a set of subambients, components, connectors, attachments (for the formalization of attachments in ChAC see section 3.2.4), and the process definition of its own aspects, weavings and ports. All these exist in a parallel composition.

In addition, the $P_{MobilityAspect}$ process which must form part of the $\prod_{i=1}^{a} Asp_i$ of an ambient has to perform the following process in order to create a channel for releasing and accepting ambients and perform the accepting and releasing actions as follows:

$$\upsilon\ exitingCh.\ \overline{out}\ \ exitingCh\ \ + \upsilon\ enteringCh.\ \overline{out}\ \ enteringCh$$

Figure 106 shows the *HostSite* ambient specification in the AOADL. As can be observed, an ambient is specified as an architectural element (similarly to a component and a connector), but it also has attachments that connect ports of architectural elements (components, connectors, and subambients) and attachments that connect architectural elements with its ports.

```
Ambient_Site type HostSite
```

```
 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Weavings
     ADist.getLocation(Location)instead
                              MobilityAspect.getParent(Parent);
 End_Weavings
 Ports
     InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
     ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
     EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
     InServicesPort: ICall Played_Role ACoordination.INTERIOR;
     InRoutePort: IGetRoute Played_Role ADist.INTROUTE;
 End_Ports

End Ambient_Site type HostSite;
```

Figure 114. Specification of HostSite ambient

For example, the *HostSite* ambient in Figure 106 is formalized as follows:

$$ClientSite\ [Customer \mid AgentCustCnctr \mid AgentAP[Collector \mid Purchaser\ P_{ATCH1} \mid \ldots \mid P_{ATCHm} \mid P_{AgentAP}] \mid P_{ATCH1} \mid \ldots \mid P_{ATCHm} \mid P_{ClientSite}\ ]$$

where the attachments are left unspecified.

9.3.8 Attachments

Attachments are channels that connect a port of an architectural element to a port of another architectural element. Attachments can only connect two architectural elements that are located in the same ambient or connect an architectural element to its parent ambient.

There are two options in mapping attachments in ChAC:

- An ambient: When an attachment is mapped into a ChAC ambient, the ambient which represents an attachment which connects for example, a component to a connector would be a sibling ambient of the component ambient $Id_{Comp}[P_{comp}]$ and the connector ambient $Id_{Con}[P_{con}]$ as the following:

$$Id_{Comp}[P_{comp}] \mid \ldots \mid atch[P_{ATCH}] \mid Id_{Con}\ [P_{con}\ ]$$

In this case, the P_{comp} and the P_{con} need to know the names of the attachments associated to their ports. This is due to the fact that in ChAC a sender ambient needs to know the name of the receptor ambient.

- A Process: In this case, the attachment would be a process of the parent ambient of the component and connector ambient. In this way, the component and connector processes do not need to know the name of the receiver.

As a result, an attachment is formalized as a process which exists in the same ambient where the component and connector that it connects are located. In addition, an attachment is not a mobile entity of the software architecture which emphasizes that an attachment does not need to be an ambient. In the following, the P_{atch} process is shown in parallel composition with the ambients which represent the component and connector that it connects:

$$Id_{Comp}\ [P_{comp}] \mid \ldots \mid P_{ATCH} \mid Id_{Con}\ [P_{con}\ ]$$

The formalization of an attachment process has been performed depending on the following classification:

- **Attachments that connect a port of an architectural element located in an ambient to a port of an architectural element located in the same ambient (ports of two sibling architectural elements)**

An attachment of this kind can connect a port of a component to a port of a sibling connector, or a port of an ambient to a port of another sibling ambient. In this case, an attachment is a process that connects two processes of different sibling ambients as follows:

$$a[Id_{AE1}\ [P_{AE1}] \mid \ldots \mid P_{ATCH} \mid Id_{AE2}\ [P_{AE2}\ ] \ldots .]$$

where Id_{AE1} and Id_{AE2} are the ambients which represent the architectural elements and a is the ambient where these architectural elements are executing.

The process that models an attachment when it receives a service from a client architectural element is shown as below:

$$P_{ATCHCl}(Id_{AE1}, Id_{AE2}, \vec{s}\,) \stackrel{def}{=} s_1(\vec{x}, \vec{r}_y).(\upsilon\, \vec{y}\,)(Id_{AE2}/s_1\langle \vec{x}, \vec{y}\,\rangle).$$

$$\vec{y}\,(y_0,\ldots,y_m).((Id_{ae1}/\vec{r}_y)\langle y_0,\ldots,y_m\rangle)$$

$$+\ldots\ldots+$$

$$s_n(\vec{x}, \vec{r}_y).(\upsilon\, \vec{y}\,)(Id_{AE2}/s_n\langle \vec{x}, \vec{y}\,\rangle).$$

$$\vec{y}\,(y_0,\ldots,y_m).(Id_{AE1}/\vec{r}_y\langle y_0,\ldots,y_m\rangle)$$

where Id_{AE1} , Id_{AE2} are names of architectural elements, and where $Id_{AE1} \neq Id_{AE2}$ and $\vec{s}$ is a tuple of services that the ports of the architectural elements publish.

The P_{ATCH} process listens to the s_n channel in order to receive the **input** parameters and the names of the channels that are needed to return the **output** parameters. Once, it receives this data, it creates a channel ($\vec{y}$) to receive the **output** parameters (or the acknowledgements). Then, it sends the information to the server architectural element (Id_{AE2}), which is a ChAC ambient). It then receives the **output** parameters and sends them to the Id_{AE1} ambient.

However, since an attachment in Ambient-PRISMA is bidirectional (i.e. it represents the communication when Id_{AE1} is a client and also when Id_{AE2} is a client), the attachment process is modeled as follows:

$$P_{ATCH}(Id_{AE1}, Id_{AE2}, \vec{s}\,) \stackrel{def}{=} P_{ATCHCl}(Id_{AE1},, Id_{AE2}, \vec{s}\,) \mid P_{ATCHCl}(Id_{AE2}, Id_{AE1}, \vec{s}\,)$$

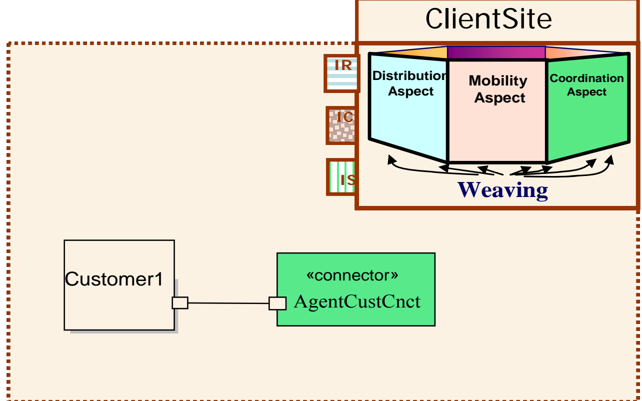


Figure 115. Customer1 component and AgentCustCnct connector connected in ClientSite ambient

For example, this can be instantiated into the attachment between the *Customer1* component and the *AgentCustCnct* connector when they are located in the *ClientSite* ambient as follows (see Figure 115):

ATT1 = new CustCnctrAtt(Customer, AgentsPort, CustAgentPort, AgentCnctr);

- **Attachments that connect a port of an ambient (excluding the InServicesPort) to a port of one of its children architectural element**

An attachment of this kind can connect a port of an ambient to a port of one of its children architectural elements. An example of an attachment of this kind can be the one that connects the DCapPort port of the *Procurement1* component to the InCapabilities port of the *ClientSite* ambient (see Figure 116). In this case, an attachment is a process that connects a local process to a child ambient as follows:

$$Id_A\ [Id_{AE1}\ [P_{AE1}] |\ ... |\ P_{ATCH}\ |\ P_{Port}\ .....]$$

where Id_{AE1} is the ambient which represents the architectural element and *Port* is the name of the port of an Ambient-PRISMA ambient called Id_A.

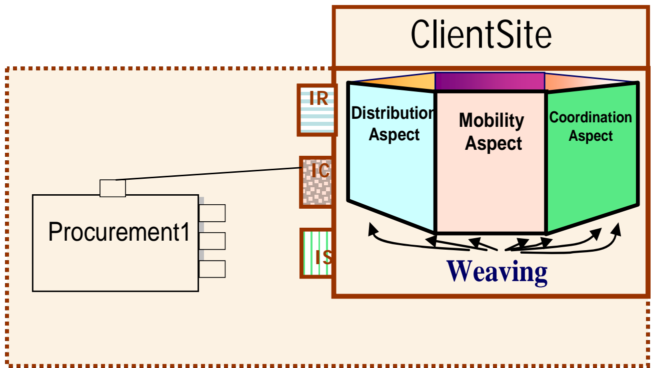


Figure 116. An attachment that connects a port of an ambient to a port of a child architectural element

The process of an attachment between an ambient and its child architectural element is defined as follows:

$$P_{ATCH}(Id_{AE1}, Id_{AE2}, \vec{s}\,) \overset{def}{=} P_{ATCH1}(Id_{AE}, Id_{AE2}, \vec{s}\,) \mid P_{ATCH2}(Id_{AE2}, Id_{AE}, \vec{s}\,)$$

where P_{ATCH1} is the process that models an attachment when the invocation of services is executed from a parent ambient to its child architectural element and P_{ATCH2} is the process that models an attachment when the invocation of services is executed from a child architectural element to its parent ambient

$$\begin{aligned} P_{ATCH1}(Id_{AE}, Id_{AE2}, \vec{s}\,) \overset{def}{=}\ & s_1\,(\vec{x}, \vec{r}_y).\ (\upsilon\ \vec{y}\ )\ (Id_{AE2}/\ s_1 \langle \vec{x}, \vec{y}\ \rangle).\\ & \vec{y}\ (y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle \\ & + \ldots\ldots + \\ & s_n\,(\vec{x}, \vec{r}_y).\ (\upsilon\ \vec{y}\ )\ (Id_{AE2}/\ s_n \langle \vec{x}, \vec{y}\ \rangle).\\ & \vec{y}\ (y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle \end{aligned}$$

The process that models an attachment when the invocation of services is executed from a child architectural element to its parent ambient is shown below:

$$\begin{aligned} P_{ATCH2}(Id_{AE2}, Id_{AE}, \vec{s}\,) \overset{def}{=}\ & s_1\,(\vec{x}, \vec{r}_y).\ (\upsilon\ \vec{y}\ )\ s_1 \langle \vec{x}, \vec{y}\ \rangle.\\ & \vec{y}\ (y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle \\ & + \ldots\ldots + \end{aligned}$$

$$s_n(\vec{x}, r_y).(\upsilon\ \vec{y})\ s_n\langle \vec{x}, \vec{y}\rangle.$$
$$\vec{y}\ (y_0, \ldots, y_m).\ r_y\langle y_0, \ldots, y_m\rangle$$

- **Attachments that connect the InServicesPort or the EServicesPort ports of an ambient to a port of a component or connector.**

An attachment of this kind connects an InServicesPort port or an EServicesPort port of an ambient to a port of a component or a connector. This kind of attachment is distinguished from the rest of attachments because the InServicesPort port and the EServicesPort port of an ambient do not receive or send an invocation of a service. Instead, these ports receive or send a call invocation.

For example Figure 117 shows that the attachment between the InServicesPort port (marked with IS) of *AuctionSite* ambient and the port of the *Procurement1* component operates. When the *Procurement1* component invokes the *notifyProdInterest* service (!), the *AuctionSite* ambient receives a call service (?). When the *Procurement1* component sends a call service (!), the *Procurement1* component receives a *notifyProdInterest* service (?). As a result, it is the responsibility of an attachment to perform this functionality.

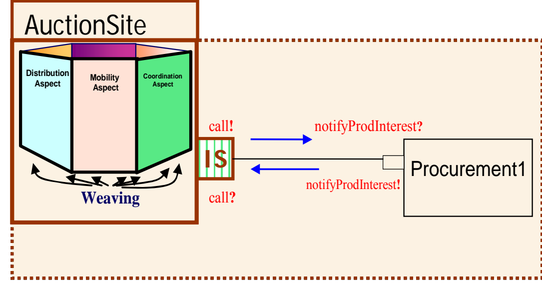


Figure 117. Operation of the attachments associated to the InServicesPort port

$$P_{ATCH}(Id_{AE1}, Id_{AE2}, \vec{s}) \overset{def}{=} P_{ATCH1}(Id_{AE}, Id_{AE2}, \vec{s}) \mid P_{ATCH2}(Id_{AE2}, Id_{AE}, \vec{s})$$

where P_{ATCH1} is the process that models an attachment when the invocation of services is executed from a parent ambient to its child architectural element and P_{ATCH2} is the process that models an attachment when the invocation of services is executed from a child architectural element to its parent ambient

The process that models an attachment when the invocation of services is executed from a parent ambient to its child architectural element is shown below:

$$P_{ATCH1}(Id_{AE}, Id_{AE2}, call@\vec{s}) \stackrel{def}{=} call\,(s_1, \vec{x}, \vec{r}_y).\,(\upsilon\ \vec{y}\,)\,(Id_{AE2}/\,s_1 \langle \vec{x}, \vec{y} \rangle).\ \vec{y}\,(y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle$$
$$+ \ldots\ldots +$$
$$call\,(s_1, \vec{x}, \vec{r}_y)..\,(\upsilon\ \vec{y}\,)\,(Id_{AE2}/\,call \langle \vec{x}, \vec{y} \rangle).\ \vec{y}\,(y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle$$

The process that models an attachment when the invocation of services is executed from a child architectural element to its parent ambient is shown below:

$$P_{ATCH2}(Id_{AE2}, Id_{AE}, call@\vec{s}) \stackrel{def}{=} s_1\,(\vec{x}, \vec{r}_y).\,(\upsilon\ \vec{y}\,)\,call \langle s_1, \vec{x}, \vec{y} \rangle.\ \vec{y}\,(y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle$$
$$+ \ldots\ldots +$$
$$s_n\,(\vec{x}, \vec{r}_y).\,(\upsilon\ \vec{y}\,)\,call \langle s_n, \vec{x}, \vec{y} \rangle.\ \vec{y}\,(y_0, \ldots, y_m).\ \vec{r}_y \langle y_0, \ldots, y_m \rangle$$

9.3.9 Bindings

A binding connects a port of a system to a port of one of its architectural elements. A binding redirects the invocations and results of services from one port to another. In Ambient-PRISMA, a binding only connects a system to one of its architectural elements when the system and the architectural element are in the same ambient. When the system and architectural element are distributed in

different ambients, a binding redirects the invocations and results of services to/from the port of a system to the *InServicesPort* of an ambient.

A binding is formalized as a process which exists in the same ambient where the system and architectural element that it connects are located. The formalization of a binding process has been performed depending on the following classification:

- **Bindings that connect a port of a system located in an ambient to a port of one of its architectural elements located in the same ambient**

The process of a binding between a system and its architectural element is defined as follows:

$$P_{BIND}(Id_{SYS}, Id_{AE2}, \vec{s}_{sy}, \vec{s}_{AE}) \overset{def}{=} P_{BIND1}(Id_{SYS}, Id_{AE2}, \vec{s}_{sy}, \vec{s}_{AE}) \mid P_{BIND2}(Id_{AE2}, Id_{SYS}, \vec{s}_{sy}, \vec{s}_{AE})$$

where P_{BIND1} is the process that models a binding when the invocation of a service arrives to a system, P_{BIND2} is the process that models a binding when the invocation of services is executed from an architectural element, $\vec{s}_{sy}$ is the tuple of services that a port of a system publishes, and $\vec{s}_{AE}$ is the tuple of services that a port of an architectural element publishes.

The process that models a binding when an invocation of a service arrives to a system consists of the parallel composition of three processes as shown below:

$$P_{BIND1}(Id_{SYS}, Id_{AE2}, \vec{s}_{sy}, \vec{s}_{AE}) \overset{def}{=} P_{BR} \mid P_{IB} \mid P_{BS}$$

The P_{BR} receives invocations of services that arrive to the port of Id_{SYS} and redirects invocations to P_{IB} :

$$P_{BR} \overset{def}{=} s_{sy1}(\vec{x}, \vec{rs}_y).(\upsilon\ \vec{rbs}_y).(s_{b1}\langle \vec{x}, \vec{rbs}_y \rangle. rbs_{y0}(y_0) \ldots rbs_{ym}(y_m)).rs_{y0}\langle y_0 \rangle \ldots rs_{ym}\langle y_m \rangle$$

$$+ \ldots +$$
$$s_{sym}(\vec{x}, \vec{rs}_y).(\upsilon \vec{rbs}_y).(s_{bm}\langle \vec{x}, \vec{rbs}_y \rangle.$$
$$rbs_{y0}(y_0) \ldots rbs_{ym}(y_m)).rs_{y0}\langle y_0 \rangle \ldots rs_{ym}\langle y_m \rangle$$

The P_{IB} process receives invocations and redirects them to P_{BS} :

$$P_{IB} \stackrel{def}{=} s_{b1}(\vec{x}, \vec{rbs}_y).(\upsilon \vec{r}_{AEsy})\ (s_{AE1}\ \langle \vec{x}, \vec{r}_{AEsy} \rangle.$$
$$r_{AEs\,y0}?\ (y_0) \ldots r_{AEs\,ym}?\ (y_m)).\ rbs_{y0}\langle y_0 \rangle \ldots rbs_{ym}\ \langle y_m \rangle$$
$$+ \ldots +$$
$$s_{bm}(\vec{x}, \vec{rbs}_y).(\upsilon \vec{r}_{AEsy})\ (s_{AEm}\langle \vec{x}, \vec{r}_{AEsy} \rangle.$$
$$r_{AEs\,y0}?\ (y_0) \ldots r_{AEs\,ym}?\ (y_m)).\ rbs_{y0}\langle y_0 \rangle \ldots rbs_{ym}\langle y_m \rangle$$

P_{BS} receives invocations from P_{IB} and sends them to the port of the architectural element of the system:

$$P_{BS} \stackrel{def}{=} s_{AE1}(\vec{x}, \vec{r}_{AEsy}).(\mathrm{Id}_{AE2}/\,r_{AEsy}\langle y_0 \rangle \ \ldots\ \mathrm{Id}_{AE2}/\,r_{AEs}\langle y_0 \rangle)$$
$$+ \ldots +$$
$$s_{AEm}(\vec{x}, \vec{r}_{AEsy}).(Id_{AE2}/\,r_{AEsy}\langle y_0 \rangle \ \ldots\ Id_{AE2}/\,r_{AEs}\langle y_0 \rangle)$$

- **Bindings that connect the InServicesPort or the EServicesPort ports of an ambient to a port of a system.**

The process of a binding between a system and InServicesPort or the EServicesPort ports of an ambient is defined as follows:

$$P_{BIND}(Id_{SYS}, Id_{AMB}, \vec{s}_{sy}, call) \stackrel{def}{=} P_{BIND1}(Id_{SYS}, Id_{AMB}, \vec{s}_{sy}, call)$$
$$|\ P_{BIND2}(Id_{AMB}, Id_{SYS}, \vec{s}_{sy}, call)$$

where P_{BIND1} is the process that models a binding when the invocation of a service arrives to a system, P_{BIND2} is the process that models a binding when the invocation of services is arrived to an ambient, $\vec{s}_{sy}$ is the tuple of services that a port of a system

publishes, and *call* is the service the InServicesPort port or the EServicesPort port of an ambient publishes.

The process that models a binding when an invocation of a service arrives to a system consists of the parallel composition of three processes as shown below:

$$P_{BIND1}(Id_{SYS}, Id_{AMB}, \vec{s}_{sy}, call) \stackrel{def}{=} P_{BR} \mid P_{IB} \mid P_{BS}$$

The P_{BR} receives invocations of services that arrive to the port of Id_{AMB} and redirects invocations to P_{IB} :

$$\begin{aligned} P_{BR} \stackrel{def}{=} & \; call\,(s_{b1,}\, \vec{x}\,,\ \vec{rs}_{y}\,).(\upsilon\ \vec{rbs}_{y}\,).\ (s_{b1}\langle\ \vec{x}\,,\ \vec{rbs}_{y}\ \rangle\,. \\ & rbs_{y0}(y_0)\ldots\ rbs_{ym}(y_m))\ .rs_{y0}\ \langle\ y_0\ \rangle\ \ldots rs_{ym}\langle\ y_m\rangle \\ & + \ldots + \\ & call\,(s_{b1,}\, \vec{x}\,,\ \vec{rs}_{y}\,).(\upsilon\ \vec{rbs}_{y}\,).\ (s_{bm}\langle\ \vec{x}\,,\ \vec{rbs}_{y}\ \rangle\,. \\ & rbs_{y0}(y_0)\ldots\ rbs_{ym}(y_m))\ .rs_{y0}\ \langle\ y_0\ \rangle\ \ldots rs_{ym}\langle\ y_m\rangle \end{aligned}$$

The P_{IB} process receives invocations and redirects them to P_{BS} :

$$\begin{aligned} P_{IB} \stackrel{def}{=} & \; s_{b1}(\,\vec{x}\,,\ \vec{rbs}_{y}\,).\ (\upsilon\ \vec{r}_{sy}\,)\ \ (s_{sy1}\ \langle\ \vec{x}\,,\ \vec{r}_{sy}\ \rangle\,. \\ & r_{s\,y0}?\ (y_0)\ldots r_{s\,ym}?\ (y_m)).\ rbs_{y0}\langle\ y_0\rangle\ \ldots\ rbs_{ym}\ \langle\ y_m\rangle \\ & + \ldots + \\ & s_{bm}(\,\vec{x}\,,\vec{rbs}_{y}\,).(\upsilon\ \vec{r}_{sy}\,)\ \ (s_{sym}\ \langle\ \vec{x}\,,\ \vec{r}_{sy}\ \rangle\,. \\ & r_{s\,y0}?\ (y_0)\ldots r_{s\,ym}?\ (y_m)).\ rbs_{y0}\langle\ y_0\rangle\ \ldots\ rbs_{ym}\langle\ y_m\rangle \end{aligned}$$

P_{BS} receives invocations from P_{IB} and sends them to the port of the system:

$$\begin{aligned} P_{BS} \stackrel{def}{=} & \; s_{sy1}(\ \vec{x}\,,\ \vec{r}_{sy}\,)\,.(\,Id_{SYS}\,/\,r_{sy}\ \langle\ y_0\rangle\ \ldots\ Id_{SYS}\,/\,r_{sy}\ \langle\ y_0\rangle\,) \\ & + \ldots + \\ & S_{sym}(\ \vec{x}\,,\ \vec{r}_{sy}\,)\,.(\,Id_{SYS}\,/\,r_{sy}\ \langle\ y_0\rangle\ \ldots\ Id_{SYS}\,/\,r_{sy}\ \langle\ y_0\rangle\,) \end{aligned}$$

9.3.10 Architectural Model Configuration

In Ambient-PRISMA, an architectural model Id_{AM} is defined as follows:

$$Id_{AM}\ (Id_{Root}) \overset{def}{=} Id_{Root}(\overrightarrow{Amb}, \overrightarrow{Comp}, \overrightarrow{Con}, \overrightarrow{ATCH}, \overrightarrow{Asp}, \overrightarrow{W}, \overrightarrow{Port})$$

Each Ambient-PRISMA architectural model is represented by an ambient called *Root*.

Figure 109 shows a fragment of the AOADL that specifies the *MobileAgentsAuctionConf* architectural model. It can be also observed that each ambient (*Root*, *ClientSite*, *AuctionSite*, *LondonSaleRoomSite*), component (*Customer1*, *Procurement1*, *Bidder1*), connectors, and attachments are instantes. The parent ambient is assigned at the creation of each element. For example, the *Customer1* component has *ClientSite* as its parent ambient.

```
Architectural_Model_Configuration MobileAgentsAuctionConf =

New MobileAgentsAuction
{
       ….

       ROOT = new Root() ;

       ClientSite = new HostSite(ROOT, IP1);
       AuctionSite = new HostSite(ROOT, IP2);
       LondonSaleRoomSite = new HostSite(Root, IP3);

       ….
       Customer1 = new Customer("ClientSite");
       Procurement1 = new Procurement ("ClientSite");
       Bidder1 = new Bidder("painting", "3 Jul", "ClientSite");
       AgentCustCnct1 = new AgentCustCnct("ClientSite");
       AuctionHouseCnct1 = new AuctionHouseCnct("AuctionSite");
       AttchAuct1Cnct = new AttchAuctCnct (AuctionCnct1,
                                           CnctAuctPortBidder,
                                           AuctionHouse1,
                                           BidderAuctPort);
        AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                                     AuctionCnct1, CustPortAuct);
    ……..
```

Figure 118. Specification of the MobileAgentsAuctionConf configuration

The *MobileAgentsAuctionConf* architectural model is formalized as follows:

Root [ClientSite [Procurement1 | Customer1 | Bidder1 | AgentCustCnct1 ....]
AuctionSite[AuctionHouse1 | AuctionHouseCnct1 | P_{ATT1} | ... | P_{ATT13} | $P_{AuctionSite}$ *]* |
LondonSaleRoomSite[....]
P_{ATCH1} | ... | P_{ATCHm} | P_{Root}*]*

where some of the attachments are left unspecified.

9.4 Conclusions

This chapter has formalized Ambient-PRISMA in ChAC. The formalization shows that ChAC provides the primitives that are necessary to model Ambient-PRISMA. At the same time, the concepts of Ambient-PRISMA model and language can be defined in a precise way by transforming them into ChAC primitives.

The formalization defines the properties of the Ambient-PRISMA and provides a formal language that is independent of technology for specifying distributed and mobile software architectures and to automatically generate code from its specifications.

PART V. TOOL AND METHODOLOGY

CHAPTER 10

AMBIENT-PRISMA CASE TOOL

"Science is one thing, wisdom is another. Science is an edged tool, with which men play like children, and cut their own fingers."

Sir Arthur Eddington

10.1 Introduction

Ambient-PRISMA follows Model Driven Engineering, in the same way as PRISMA does. The objective of Ambient-PRISMA is to develop distributed and mobile software systems in a technology-independent way using aspect-oriented architectural models and compile them to different technological platforms. As a result, the PRISMA Case Tool (see CHAPTER 6) has been extended in order to provide the graphical modelling of Ambient-PRISMA specifications and the execution of these specifications in the .NET.

The current version of the Case Tool is a prototype which provides the graphical modelling of Ambient-PRISMA specifications at a Type Definition Level. In addition, Ambient-PRISMA specifications have been manually generated using a middleware called Ambient-PRISMANET. Ambient-PRISMANET extends PRISMANET middleware in order to provide the needed primitives for executing distributed and mobile applications. However, the automatic code generation of applications has been postponed for further work.

This chapter is structured as follows: Section 10.2 presents how the PRISMA Modelling Tool has been extended in order to support the modelling of Ambient-PRISMA specifications in a graphical and friendly way. Section 10.3 presents the Ambient-PRISMANET middleware in order to show how the execution of Ambient-PRISMA is supported in the .NET platform. Finally, section 10.4 presents conclusions.

10.2 The Modelling Tool for Ambient-PRISMA

The PRISMA Case Tool has been developed using the Domain Specific Languages Tools (DSL Tools) which provides the DomainModel project and the Designer project (see CHAPTER 6). The PRISMA Case Tool uses the DomainModel project in order to support the PRISMA metamodel concepts and the Designer project in order to provide a graphical metaphor for each concept of the metamodel.

The PRISMA metamodel introduced in the DomainModel project (see CHAPTER 6) has been extended in order to include the metaclasses needed for supporting ambients (see Figure 119). An *Ambient* metaclass which extends the *Connector* metaclass has been introduced. The *Virtual*, *Site*, and *Group* metaclasses which represent Virtual, Site, and Group ambients respectively, inherit from the *Ambient* metaclass.

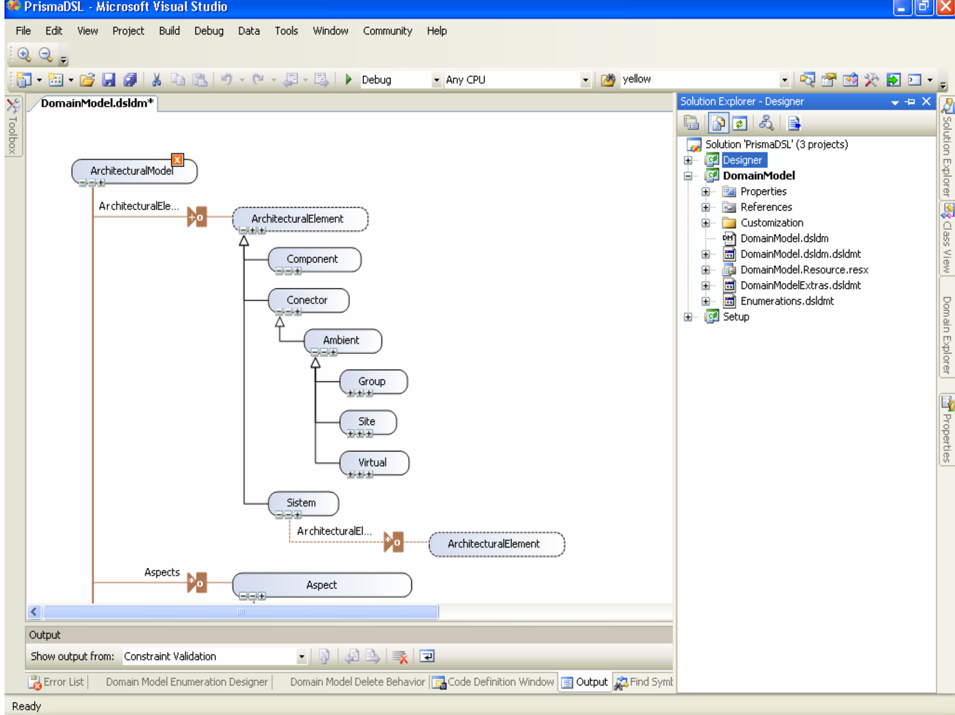


Figure 119. Definition of Ambients in the DomainModel

The Designer project of PRISMA has been also extended in order to create a graphical modelling tool for modelling PRISMA aspect oriented software architectures with ambients. This has been performed in the following two steps:

- STEP1: Associate a graphical icon for each ambient metaclass introduced in the DomianModel project for creating a toolbox. As a result, the Modelling Tool has the toolbox shown in Figure 120.

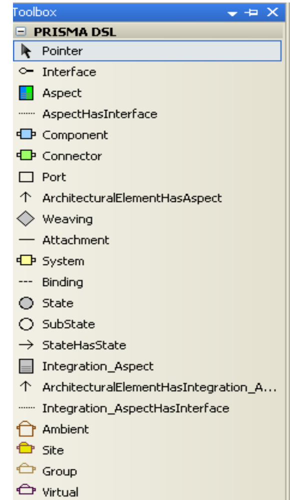


Figure 120. Tool box with ambients

- STEP2: Implement partial classes which describe how the graphical metaphor associated to an ambient appears on the drawing sheet of the Modelling tool once the icons of the toolbox are dragged and dropped.
- STEP3: Implement partial classes which verify that the drawn models do not violate the metamodel. Two types of verifications are distinguished: *hardconstraints* that must always be satisfied, and *verification rules* that must be satisfied once the model has been completely finished.

The Modelling tool is generated from the DomainModel and the Design project. The user of the Modelling tool drags and drops icons from the toolbox to the drawing sheet. Figure 121 shows the appearance of a Site ambient on the drawing sheet when a user drags and drops its icon from the Tool box.

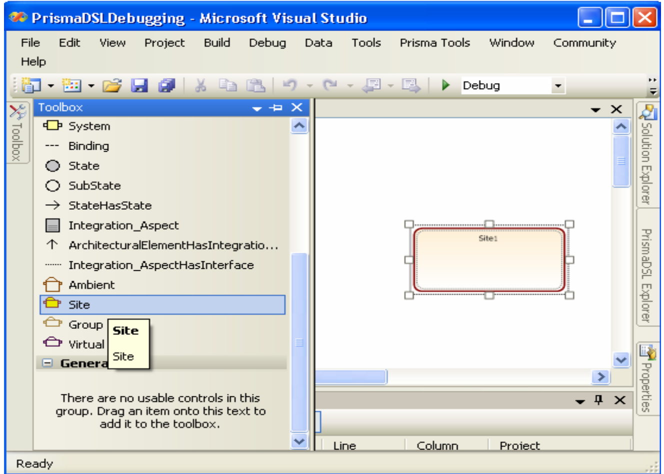


Figure 121. Site ambient shape

While the user is building its model, the hard constraints are checked. An example of a hard constraint in Ambient-PRISMA is that ambients cannot have a functional aspect. As a result, if the user tries to define an ambient which imports a functional aspect, the tool must not allow him/her. Figure 122 shows that when a user tries to define a Group ambient that imports a functional aspect, the tool shows a message in order to indicate to the user that a Group ambient cannot have a functional aspect as well as prohibiting him/her from relating the ambient with the functional aspect.

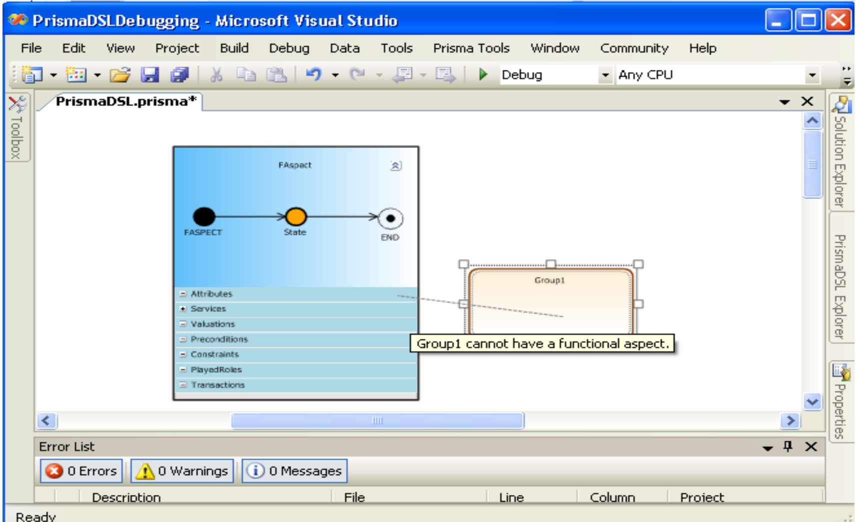


Figure 122. Modelling Tool indicating a violation of a hard constraint for an ambient

The Modelling Tool allows the user to define ambients using interfaces, aspects, ports, and weavings. In addition, ambients can be related to other architectural elements of the software architecture. For example, Figure 123 shows how a Group ambient called *MobileAmb* is defined by importing a distribution aspect which allows it to be mobile. The distribution aspect is called *MobileAmbDist* aspect and it uses an interface called *IMobility*. The MobileAmb ambient has a port which is connected through an attachment to a port of a connector called *AgentCustCnct*. It can be noticed that the predefined aspects of the MobileAmb ambient i.e., the *MobilityAspect* aspect and the *ACoordination* aspect (see CHAPTER 8) do not have to be defined by the user.

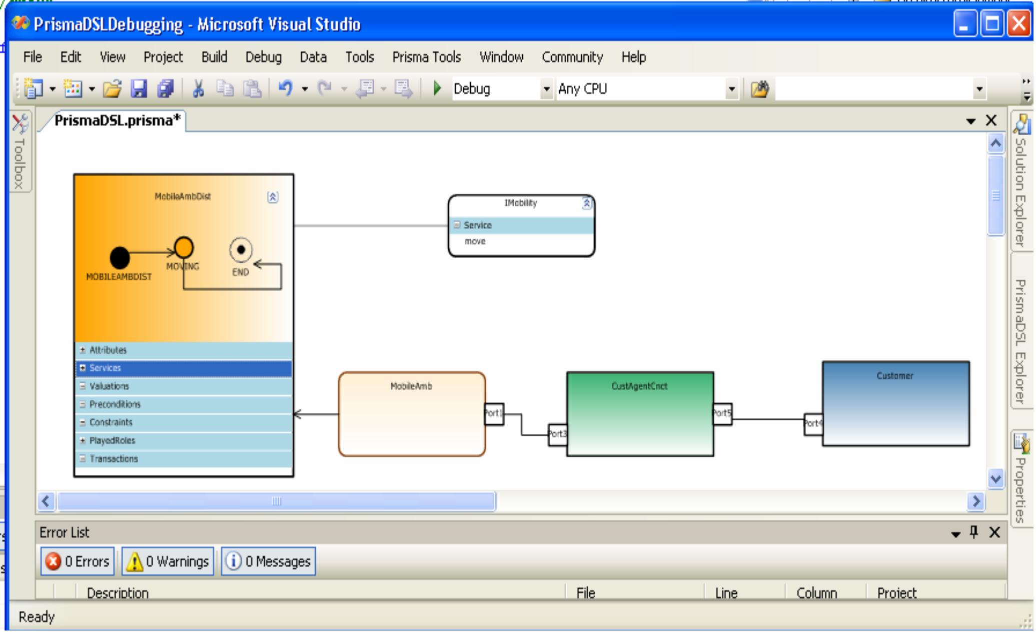


Figure 123. Modelling of a Group ambient at the Type Definition Level

10.3 Ambient-PRISMANET

PRISMANET middleware implements PRISMA in C# in order to permit PRISMA software architectures to be developed and executed on the .NET platform (see section 6.4.3). PRISMANET allows the execution of aspects, the concurrent execution of aspects and architectural elements, the loading of architectural elements, the creation of execution threads, and the management of the local components.

Since Ambient-PRISMA permits specifying PRISMA aspect-oriented software architectures of distributed and mobile software systems, the purpose of Ambient-PRISMANET is to provide the needed primitives for allowing the execution of Ambient-PRISMA distributed and mobile software systems. As a result, PRISMANET middleware has to be extended for supporting the distribution and mobility characteristics of Ambient-PRISMA. The implementation of the distributed

characteristics needed for Ambient-PRISMANET have been performed using .NET Remoting technology [Mcl03] which provides mechanisms for accessing objects in a remote way as well as serializing them for providing mobility.

In future works, the Ambient-PRISMANET middleware is going to be used in order to implement the automatic code generation patterns and be launched from the Ambient-PRISMA Case Tool.

10.3.1 Ambient-PRISMANET Architecture

Ambient-PRISMANET provides a distributed runtime environment for Ambient-PRISMA applications (see Figure 124). As a result, each host which forms part of the runtime environment needs to have an Ambient-PRISMANET instance executing on it. Each middleware instance is in charge of managing architectural elements running on it and to collaborate with the rest of middlewares executing on other hosts in order to provide architectural elements with distribution and mobility. The runtime environment has a distributed Domain Name Server (DNS) which all middlewares on hosts can access. In this way, a middleware can access any distributed architectural element.

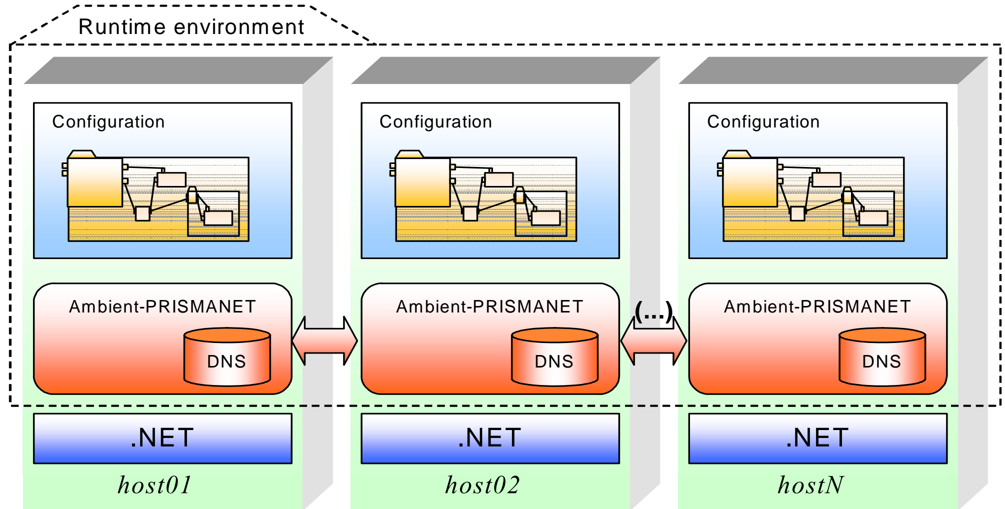


Figure 124. Distributed Run-time Environment of Ambient-PRISMANET

The Ambient-PRISMANET architecture consists of the following layers (see Figure 125):

- **Ambient-PRISMA Execution Model**: This layer extends the PRISMA execution module (see 6.4.3) in order to implement the primitives which are introduced by Ambient-PRISMA. Each Ambient-PRISMA primitive is implemented in a class. As a result, the implementation of an Ambient-PRISMA application is achieved by extending the classes
- **Element Management Layer**: This layer is in charge of instantiating architectural elements (their aspects, ports, and weavings), attachments, and bindings and storing each created instance in lists in order to be able to manage the instances.
- **Middleware2Middleware Layer**: This layer is in charge of establishing communication among the different distributed middleware that form part of a distributed environment. This layer acts as a proxy offering the needed services of other layers of a middleware to the other middlewares of the distributed environment.
- **Middleware's List Management layer**: This layer is in charge of maintaining a list which each middleware shares with other middleware which form part of a distributed environment. It is also responsible of providing each middleware with the capability of detecting failures and recoveries of other middleware (see section 10.3.2).
- **DNS Layer**: This layer is in charge of resolving locations of architectural elements. It receives local requests in order to find the locations of distributed architectural elements and it receives distributed requests in order to consult whether an architectural element is executing on the host of the DNS layer or not (see section 10.3.3).
- **Distribution Management**: This layer is in charge of implementing the services needed for moving architectural elements from one middleware on a host to a new middleware on another host in a transparent way.

- **Transaction Manager**: This layer is in charge of executing transactions in a correct way. It provides mechanisms that allow services of transactions to be offered by distributed architectural elements as well as mechanisms for recovering the state of the software architecture when a transaction fails (see section 10.3.5).

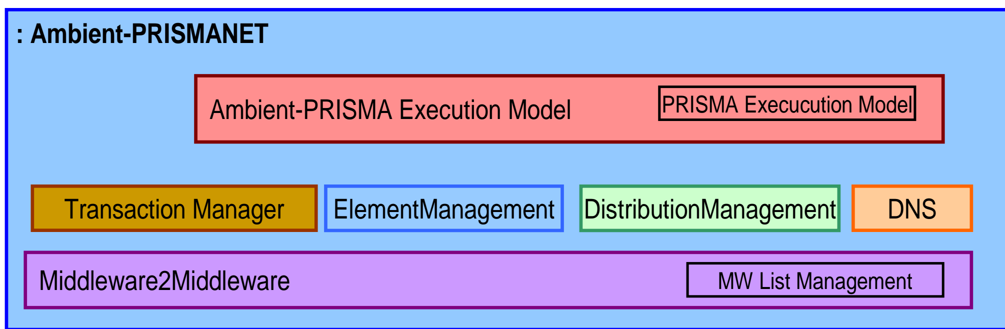


Figure 125. Ambient-PRISMANET architecture

In the following sections, the implementation of the Ambient-PRISMANET layers is explained.

10.3.2 Middleware's List Management

The runtime environment of Ambient-PRISMA is formed by a group of distributed middleware which work in a collaborative way for providing architectural elements of a distributed software architecture with a correct functioning. All middlewares which form a runtime environment have to know each other for performing collaborating tasks. As a result, each Ambient-PRISMANET middleware needs to have an updated list which stores the references to hosts where an Ambient-PRISMA middleware is executing. This list has to store exactly in its elements the same references to hosts of all middlewares. In this way, this list can be considered a virtual list of middlewares for the entire execution environment. The use of this list allows each middleware to be aware of its environment and optimizes the implementation of a distributed DNS.

The Middleware's List Management layer of each middleware is responsible for maintaining a list with a consistent state in all middlewares by updating it when a host fails or a new host is added. These changes can be active or passive. An active change happens when the list of middlewares is modified by the user. This can happen when the user wants to add or remove a host to/from the execution environment. The user updates the list of middlewares from the console on its host and this change is propagated to the lists on the rest of middlewares.

The passive changes on the middlewares happen when the a host is added or removed without control on the user such as when a middleware of a host fails for any reason, when a failed host returns to operate, or when a machine is restarted. The Middleware's List Management layer is also responsible for implementing the mechanisms needed for detecting passive changes.

A Fail-stop Fault [Tre05] has been considered in the implementation of passive changes. In a Fail-stop Fault, the failed hosts stop emitting results and interacting with the rest of hosts. The detection of these faults is performed by monitoring heartbeats [Kal99]. As a result, a middleware can adopt either two roles: a Notifier or a Leader. A Notifier middleware emits a heartbeat to a leader middleware during a frame of time. A Leader middleware monitors and determines which Notifier middlewares are answering and which are not. As a result, heartbeats are periodical calls which Notifier middlewares performs to a method of a Leader middleware. A Leader middleware does not need to send any heartbeats because its existence guarantees it's functioning.

It can be noticed that the management of the distributed environment is centralized in only one middleware (the Leader). This allows the Leader to control the timeout and not depend on the clocks of different machines. The Leader saves the last time a determined middleware notifies it as well as checks depending on the saved time whether a middleware expires a timeout or not.

Each Notifier middleware has a reference to the Leader middleware and a thread which periodically calls a method of the leader called *ImAlive* (see Figure 126). The method has a parameter of data type **loc** which indicates the host of a Notifier middleware caller. The Leader middleware listens to the notifications received from the rest of middlewares and saves a list called knownMWList the middleware and the time which it received the heartbeat through a timestamp. If it receives a notification from a middleware which is not in the list, it adds it in the list.

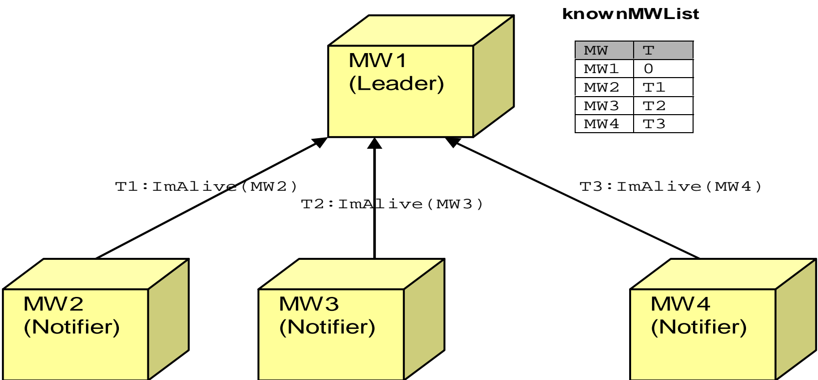


Figure 126. Leader-Notifier Interaction

The active changes of middlewares i.e., adding and removing middlewares are performed by the user. As a result, the GUI of the Ambient-PRISMA console has a panel which allows the user to introduce these changes or consult the current state of the distributed executing environment.

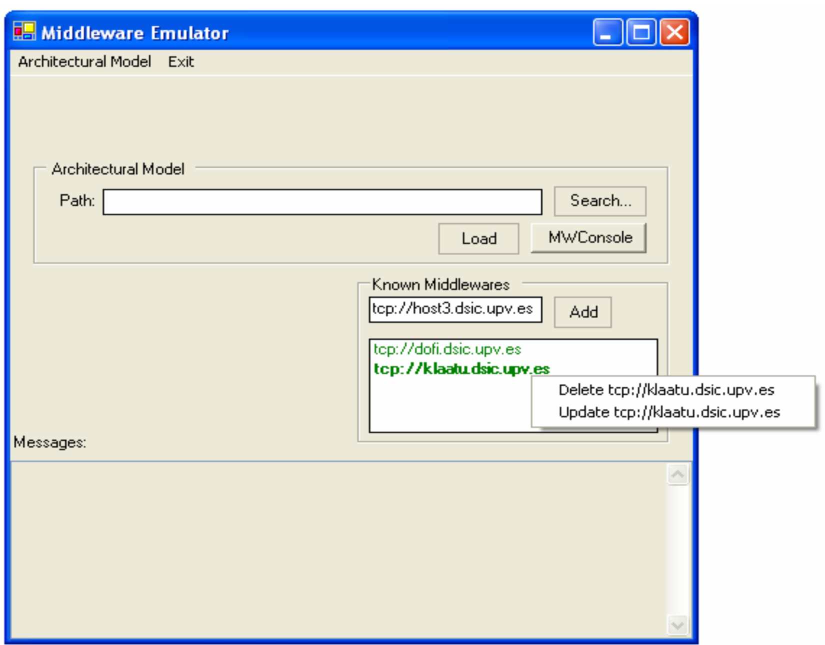


Figure 127. The Panel of Known Middlewares

This panel shows the Ambient-PRISMANET middlewares that are implicated in the execution environment (see Figure 127). The ones in green are the ones which are currently executing and the ones in red are the ones that are not responding. The one in bold represents the middleware where the user is interacting i.e.,where the panel is executing. If a middleware is removed, it will appear in the panel but with a line. In this way, if the removed middleware returns to be executed the panel refreshes the list. When the user presses above a middleware which is executing, a Delete option appears to allow him/her to remove it and the middleware is a removed one, an Add option appears to allow the user to add it. Table 7 shows the significance of the representation of the middlewares using the panel.

Table 7. Representation of the middlewares in the Panel for Known Middleware

Representation	Significance
tcp://url.del.host	A host with an executing Ambient-PRISMANET middleware

Representation	Significance
tcp://url.del.host	A host where an Ambient-PRISMANET middleware was executing and has been removed by a user.
~~tcp://url.del.host~~	A host where an Ambient-PRISMANET middleware was executing and failed.
tcp://url.del.host	A host where the panel is executing.

In the following, how the Middleware's List Management layer reacts to passive changes of middleware execution is explained.

10.3.2.1 Detecting Failures and Recoveries of Notifiers

The thread of the Leader searches in the knownMWList list in order to check whether the difference between the current time and the timestamp of the last notification received of a middleware is larger than a timeout value which determines that a middleware has failed. The timeout value is assigned by the user and depends on the kind of network which the middlewares are executing.

If the difference between the current time and the last value annotated is larger than the timeout, the middleware is added to a list of removed middlewares. Once the Leader finishes checking all the elements of the knownMWList list, the Leader broadcasts to all Notifier middlewares the list of removed middlewares through a method called *BroadcastDeletedList* which indicates Notifier middlewares to remove the failed middlewares from their knownMWList lists (see Figure 128). In a similar way, the new middlewares that have recovered from failures are added to a list of added middlewares and is sent to the Notifier middlewares in order to update their knownMWList lists. Once the Leader sends the lists of removed and added middleware, the thread of the Leader sleeps for a while and then it returns to check its knownMWList list.

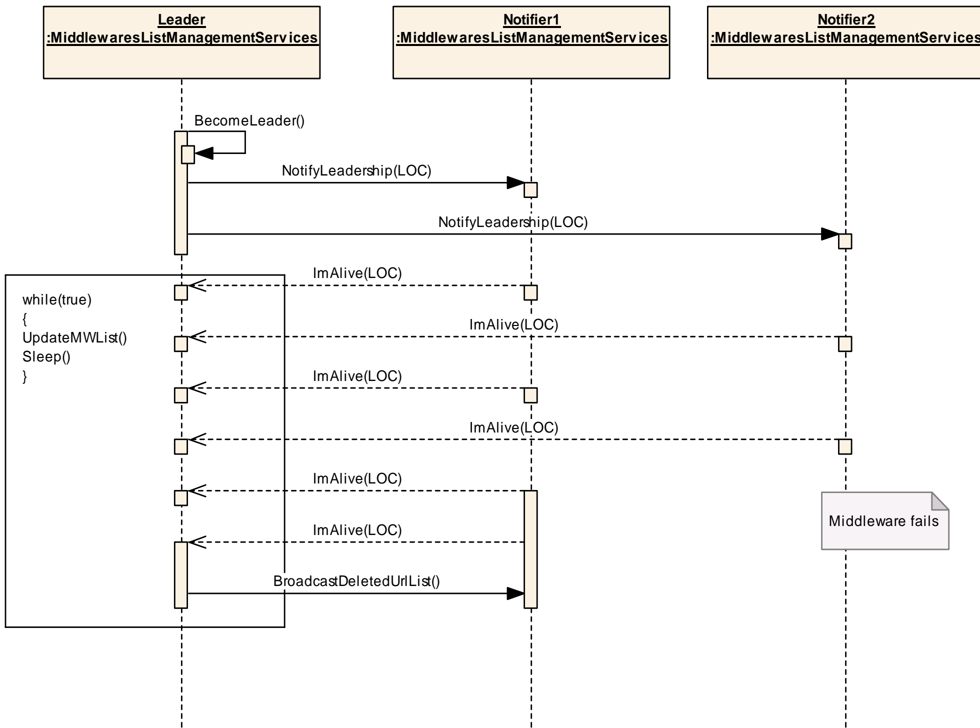


Figure 128. Sequence diagram when a failure of a middleware occurs

10.3.2.2 A Notifier recovering from a failure

When a failed Notifier returns to execute, it needs to know the other Notifier middlewares and the Leader middleware which are executing at that moment. As a result, each middleware stores in a file in secondary memory the state of the knownMWList list. In this way, if a middleware fails for any reason, the last state of the knownMWList list is known. When a middleware finishes executing not as a result of a failure, the file is emptied.

When the Middleware's List Management layer begins to execute after a failure, checks the file where the state of the knownMWList list is saved. If this file is not empty, i.e., it has recovered from a failure, the Middleware's List Management layer asks the middlewares saved in the file about their Leader middleware. If any of the

middlewares saved are executing, it will return to the recovered middleware the URL of the Leader. In this way, the recovered Middleware's List Management layer takes a Notifier role and notifies its Leader about its recovery.

If none of the middlewares saved in the file answers, then the Middleware's List Management layer does not have any way in knowing its Leader middleware. As a result, the recovered middleware becomes a Leader because it believes that it is the only middleware executing.

Figure 129 shows a fragment of the pseudocode which describes what a Notifier middleware does when it recoveries from a failure. First, it obtains the list of known middlewares stored in the file, and asks the middlewares one by one about the Leader middleware. If the middlewares which formed part of its previous environment do not answer, the recovered middleware becomes a Leader.

```
LastSessionList = ReadMWFile();
if LastSessionList has elements then
       foreach middleware in LastSessionList
              try
                     myLeader = middleware.GetLeader()
              catch
                     next
       endforeach
endif
if MmyLeader is null then BecomeLeader()
```

Figure 129. Pseudocode Fragment of the initialization of a middleware

10.3.2.3 The failure of a Leader

The knownMWList list stores in its first element the name of the Leader middleware of the execution environment. This is possible because the knownMWList list is a sorted list which all middlewares have the same elements stored in the same order. The knownMWList list stores the URL of the middlewares ordered in alphabetical.

When the Leader fails, the Notifiers obtain a Remoting exception when they send their heartbeats. As a result, the Notifiers have to perform the following:

- The Notifiers remove their previous failed Leader from their knownMWList list.
- The next middleware stored in the knownMWList list becomes a Leader. The new Leader destroys its thread as Notifier and creates a thread as a Leader.
- The rest of middlewares also have the new Leader in the first element of the knownMWList list (since they also have removed the failed Leader from it) and start sending it their heartbeats.
-

10.3.3 Distributed Domain Name Server (DNS)

A DNS is useful for the performance of a middleware. It allows consulting the locations of distributed architectural elements. Some of the cases where a DNS is important are explained in the following:

- When an architectural element is moved from an origin host to another host, the DNS removes the moved architectural element from the origin host and registers it in the new host.
- To detect that an architectural element is distributed and where the architectural element is located.
- When a new architectural element instance is created, the DNS is used in order to check that the new instance has a unique name.
- When an architectural element needs to be deleted, and the architectural element is not located in the current host, the DNS can redirect the request of deleting it to its current middleware.

It can be noticed that the DNS has an important functionality. A centralized implementation of a DNS is not appropriate due to the fact that if the host where the DNS is executing fails all the registers of the DNS are lost. As a result, Ambient-PRISMANET middleware implements a decentralized solution to be more trustworthy.

Each Ambient-PRISMANET middleware has a layer called *DNSLayer*. Each DNSLayer on a host is in charge of receiving local petitions of a method called *GetLocationOfComponent* for looking for distributed architectural elements on other hosts through the Middleware2Middleware layer. The DNSLayer is also in charge of receiving distributed petitions of a published method called *GetURL* through the Middleware2Middleware layer and searching in its local middleware in order to answer whether a determined architectural element is localized on its host or not. The DNS layer requests a method called *GetComponent* offered by the ElementManagement layer in order to consult the existence of architectural elements which are executing on the host of the DNS layer (see Figure 130).

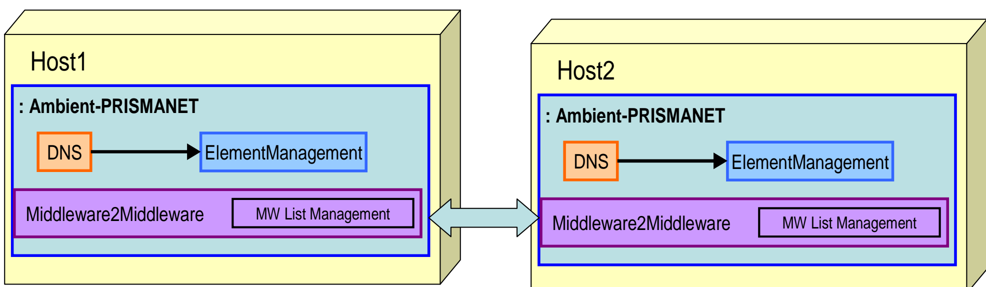


Figure 130. Decentralized DNS in Ambient-PRISMANET

In addition, each DNS layer has a cache which saves the last resolutions performed in order to make the search quicker and not to overload the network instead of asking each DNS layer of each host. The cache is represented by a hash table which stores a pair (the name of an architectural element and its corresponding URL) which is indexed by the name of an architectural element. The hash table is consulted before sending requests to DNS layers of distributed middlewares. It is also updated each time a DNS layer founds out that an architectural element has changed its host. When the DNS layer consults the hash table, the URL of an architectural element which is obtained is verified by sending to the URL found whether an architectural element exists in it or not.

The DNS layer consists of a class called *DNSServices* (see Figure 131). The DNSSErvices class has the two relevant methods: *HasComponent* and *GetLocationOfComponent*. The *HasComponent* method returns a boolean in or indicate whether an architectural element is executing on the current host or not. This method calls a public method of the ElementManagement layer in order to consult the list which stores the names of architectural elements executing on the current host.

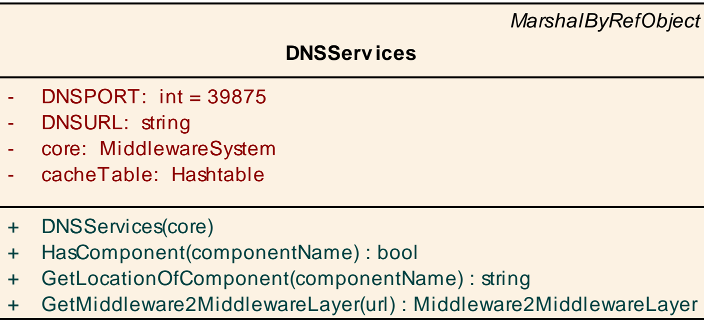


Figure 131. DNSServices class of the DNS layer

The *GetLocationOfComponent* method localizes architectural elements located in any host which forms part of the distributed execution environment. The method performs the following steps in order to localize an architectural element:

1. It checks whether an architectural elements exists on its same middleware. If an architectural element is found in the current middleware, the method performs step 4.
2. It looks for an architectural element in the hash table of architectural elements. If it finds the architectural element, it calls the middleware with the obtained URL. If it finds the architectural element in the URL, then it performs step 4.
3. It looks for an architectural element in the rest of middlewares. The method obtains a subset of known middlewares from its knownMWList list (see section 10.3.2) excluding itself and the

middleware stored in the hash table (in the case the architectural element had been stored in the cache). If it finds the architectural element, it obtains the URL of the middleware which has answered and updates the cache. If it does not find it, it saves in the cache an empty string.

4. It returns the URL of the host where the architectural element has been found (with an empty string if it has not been found).

In this way, a DNS which has been distributed among the different middlewares which form the distributed environment has been implemented. In addition, an optimization of the localization of architectural elements has been provided. The DNS layer of each middleware interacts with the Element Management layer and the Middleware List Management Layer (see section 10.3.2) in order to resolve the locations of architectural elements.

10.3.4 Distribution Management

The Distribution Management Layer is in charge of moving architectural elements from a host to another. It has two fundamental services:

- *MOVE(place, sourceObject)*: This service performs the needed task in order to ensue that the mobile architectural element is moved in a consistent state i.e., when it finishes executing the services of its aspects, it obtains the reference to the destination middleware, and invokes the *TransferComponent* service of the destination middleware.
- *TransferComponent(targetComponent)*: It is executed at the destination middleware in order to perform the serialization of the mobile architectural element. It performs the needed tasks for starting the execution of the mobile architectural element at the new destination.

The *MOVE* service is invoked by a Virtual Ambient in order to allow an architectural element to move from a host to another. As a result, the Distribution Management Layer sets a flag of the architectural element called *IsMoving* to true.

When the *IsMoving* flag is set to true, the architectural element first stops to receive service requests from its ports, then finishes processing the services that are on execution in its aspects, and finally it stops the execution of its threads. Once the aspect stops its execution, the *DistributionManagement* layer uses the *Middleware2Middleware* Layer in order to invoke the *TransferComponent* service of the destination middleware (see Figure 132). As a result, the architectural element is serialized i.e., its ports, aspects and weavings. In Ambient-PRISMANET, all the classes that form an architectural element are marked with the *[Serializable]* attribute in order to allow them to be serialized.

Afterwards, the new middleware adds the new architectural element to its list of architectural elements through the *ElementManagement* Layer, and starts the execution of the architectural element with the state of its aspects since it was stopped. In this way, Ambient-PRISMANET provides Weak Mobility due to the fact that the threads of the architectural element were stopped in order to be moved. Finally, the origin middleware deletes the architectural element from its middleware once it is notified that the mobility of the architectural element has been satisfied.

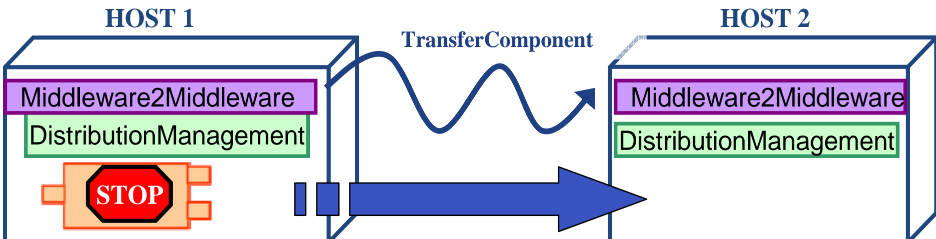


Figure 132. Executing model for mobility

When a Group ambient needs to be moved, the architectural elements located in it also move with it. As a result, the Distribution Management layer checks whether the mobile architectural element is a Group ambient or not. If the mobile architectural element is a Group ambient, the Distribution Management Layer obtains a list of architectural elements located in the mobile Group ambient and invokes a *Move* for each of its architectural elements.

10.3.5 Distributed Transactions

Transactions provide a set of services to work in a unique unit of work. In a transaction either all involved services are executed or none services are executed. Transactions are described through the Atomicity, Consistency, Isolation, and Durability (ACID) properties.

In Ambient-PRISMA transactions can be categorized as:

- Homogeneous: A homogeneous transaction consists of requesting services which are offered by the same aspect where the transaction is defined.
- Heterogeneous: A heterogeneous transaction consists of requesting services of different aspects of the same architectural element or of different architectural elements which can be executing on a different middleware of the aspect where the transaction has been defined.

The services that are involved in a transaction can be offered through the aspects of an architectural element or through other architectural elements of a software architecture. As a result, depending on the reach ability of a transaction the changes that have to be confirmed or cancelled can affect more than an architectural element which can also be distributed. These changes can affect the state of an architectural element, a set of architectural elements or the software architecture.

In order to cope with this need, the concept of transactional context has been adopted in order to define the limits of a transaction. In this way, each transaction is linked to a transactional context which represents the transaction outside the limits where the transaction has been initiated.

The state of an aspect which participates in a transaction can be undone if the transaction fails. As a result, an aspect has to be informed if it receives a service which participates in order to save its state before executing this service. An aspect which participates in a transaction is blocked until the transaction is ended in order

to guarantee its isolation. In this way, an aspect can only receive services that form part of a transactional context.

It is important to take into account that transactions can be nested. A nested transaction at least consists of requesting a service which is a transaction. In this way, a tree hierarchy of transactions can be formed where the root is the transaction which invoked the rest of transactions. When a requested transaction (subtransaction) of a nested transaction fails, all the nested transaction fails. When a subtransaction of a nested transaction commits, the only transactions which can use the result of the committed transaction are the ones involved in its nested transaction. The commit of a subtransaction can be cancelled since it may fail if any other transaction fails. When the root transaction commits, the nested transaction is considered to be satisfied.

When the transactional context of a transaction is among many architectural elements (which can be distributed), a mechanism has to be provided in order to inform the root transaction about any failure that can occur locally or remotely. As a result, the Transaction Manager layer is introduced in the Ambient-PRISMANET middleware. The Transaction Manager layer coordinates the distributed middlewares which participate in a transactional context in order to obtain a consistent state of the distributed software architecture.

In the following, the execution model of transactions is explained. Then, transactions which change the state of an architectural element are explained. Afterwards, transactions which change the state of a distributed software architecture is also explained.

10.3.6 Execution Model of Transactions

When a transaction is executed in an aspect, the middleware is notified. As a result, the Transaction Manager stores in a list called *contextsList* the information of a transaction. The elements of the contextList list are objects of a class called

TransactionalContext (see Figure 133). The Transaction Manager class saves the route of a file called Log where it saves all the actions of the transactional process.

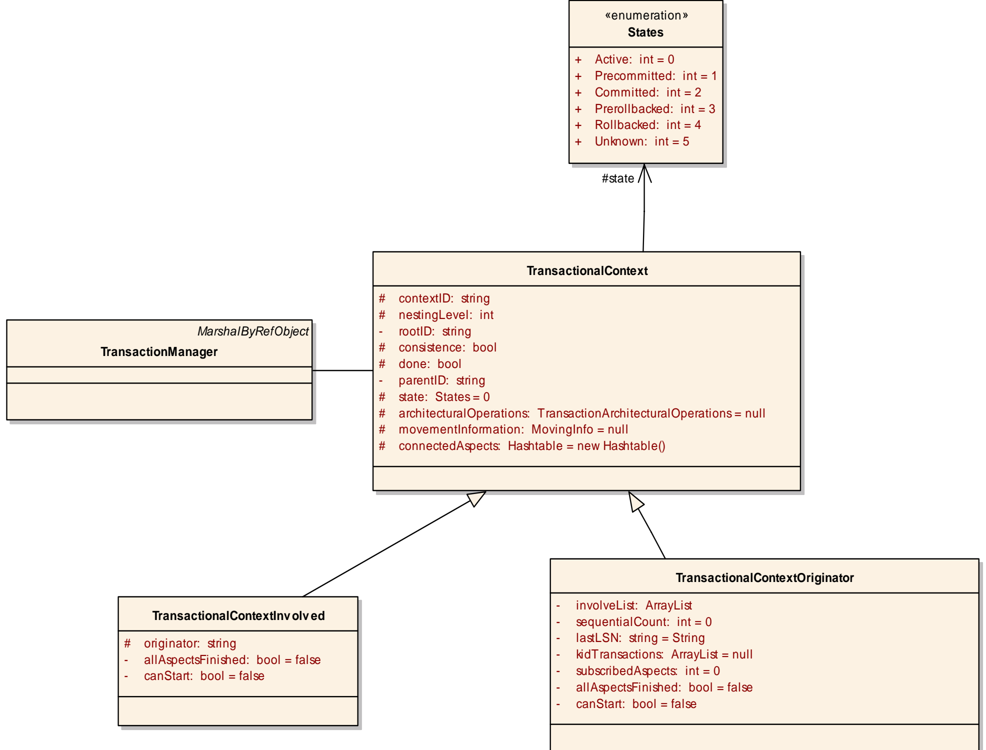


Figure 133. The Transactional Manager and the Transactional Context classes

The *TransactionalContext* class has the following attributes:

- contextID: It saves the context identifier of a transaction. The contextId saves a string which is the concatenation of the URL where the transaction is initiated and an auto incremental number.
- consistence: It is saves true if there are no failures in the transactions and false if an exception is caused during the execution of the transaction.
- done: It saves the state of the transactional context.
- parentId: It stores the contextID of the parent transaction if the transaction forms part of a nested one.

- nestingLevel: It stores the depth level of the transactional context with respect to the root transaction.
- rootId: It stores the contextId of the root transaction which initiated the transaction.
- architecturalOperations: It stores a reference to the process stored in a Log which contains the inverse actions
- state: It is an enumeration which stores the possible state a transactional context can have.

A transaction can have one of the possible states (see Figure 134):

- ACTIVE : The transaction is being executed
- PRECOMMITTED : The transaction has finished. However, since it belongs to a nested transaction, it must wait for the end of all the transactions of the hierarchy in order to confirm the commit or execute a rollback..
- PREROLLBACK : The transaction has finished. However, since it belongs to a transactional hierarchy, it must wait for the end of the root transaction because it establishes the execution order of the transitions that it nests.
- COMMITTED : The transaction has finished successfully.
- ROLLBACKED : The transaction has finished due to a failure.

A transactional context limit can be extended to different middlewares due to the fact that services of distributed architectural elements can be invoked. As a result, two classes called *TransactionalContextOriginator* and *TransactionalContextInvolved* inherit from the *TransactionalContext* class. The *TransactionalContextOriginator* class represents a middleware which originates a transaction and it stores the URL of the middlewares that participate in a transaction. The *TransactionalContextInvolved* class represents a middleware which is involved in a transaction and it stores the URL of the originator transaction.

10.3.7 Transactional Context at an architectural element level

The state of an architectural element is formed by the set of states of its aspects. Since a failure of a transaction causes the state of an architectural element to return to the state before the execution of a transaction, the state of an architectural element has to be saved before a transaction execution. In addition, the services of a transaction can not only change the state of the aspect where the transaction is originated but also can change the state of architectural element aspects which are located on different middlewares.

A pattern called Memento [Gam95] is used in order to save the state of an aspect before the execution of a transaction. The Memento pattern saves the state of an object in order to be recovered. In Ambient-PRISMA, a Memento object saves the state of an aspect which executes a service of a transactional context. If the transaction is cancelled, the saved state is recovered. If the transaction is satisfied, the Memento object is destroyed.

In Ambient-PRISMA, the class which implements aspects called AspectBase is modified by adding a new object of type *StateCareTaker*. A *StateCareTaker* object is in charge of encapsulating a *Memento* object, and recovering and destroying the Memento object. The *StateCareTaker* class is the intermediate between the *TransactionManager* class and the aspect during the transaction process. Each *StateCareTaker* of an aspect saves a list of *Memento* objects where each *Memento* object stores a state of an aspect in a different transactional context. This is due to the fact that an aspect can be involved in many transactions of the same nested transaction.

The state of an aspect is saved in a multidimensional array called *AspectMemento*. This array is filled at execution time using reflection operators of the System.Reflection namespace of .NET. Each tuple of the *AspectMemento* array stores the information of an aspect attribute.

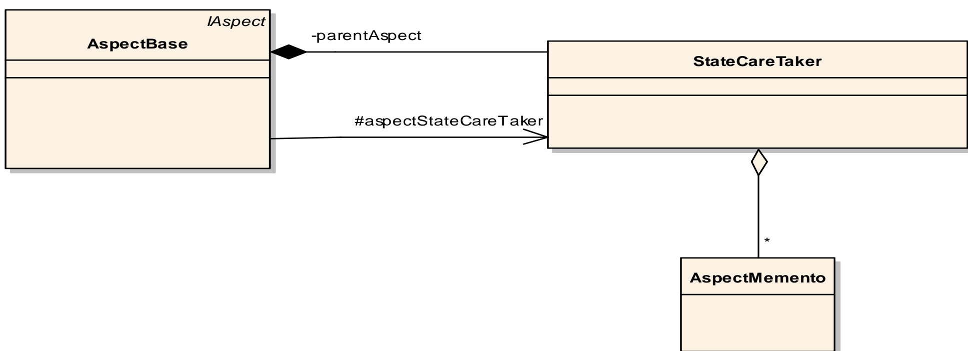


Figure 134. The StateCareTaker and the AspectMemnto classes

10.3.8 Transactional Context at a distributed architectural level

A transaction can cause a reconfiguration of the software architecture. Architectural elements, attachments and bindings can be added or deleted. If a transaction which changes the software architecture fails, the software architecture should be recovered to the same state as it was before the transaction execution. For example, moving a component through a transaction causes deleting the component, creating the component at destination, and creating and deleting attachments or bindings. If the mobility transaction fails (Rollbacks), all these changes have to be undone.

It is complicated to save the state of a distributed software architecture as it has been done to save the state of an aspect (see section 10.3.7) in order to recover from a transaction failure. Instead, it has been chosen to invoke the inverse services of a transaction in an appropriate order. For example, when a transaction called move, which indicates its destination as a parameter is executed by a component, fails, then the middleware has to invoke the move of the component by indicating its origin.

As a result, a transactional Log has been used as a mechanism to recover from a failure. Logs have been in databases in order to represent histories which save

actions performed by the database administrator [Mos87]. A *log* has a Log Sequence Number (LSN) which identifies a log register. Previous Log Sequence Number (prevLSN) links a register to its previous one. Finally, each register is identified by the transaction ID which has wrote the register.

The *log* concept has been applied in Ambient-PRISMA as follows:

- A *log* exists for each transactional context. This *log* is distributed among the different middlewares involved in the transactional context.
- Each log entrance has an LSN initiated by the originator middleware and a reference to the prevLSN. Since the *log* can be distributed, the prevLSN can be located on another middleware.
- The register is completed with the description of the inverse operation.

An example of a writing an inverse operation is shown in Figure 135. The operation consists of creating a new component in a middleware. When the CreateNewComponent method is detected to be requested by a transactional context, the inverse operation is written. The inverse operation consists of removing the created component. As a result, the string which represents removing the component is written. This string as well as the rest of operations is compiled in a dynamic class through CodeDOM technology [Sco03].

```
private void CreateNewComponent(Type componentType,
                                string componentName,
                                params object[] args,
                                string txID)
{

(...)

if (txID.Length > 0)
// this operation belongs to a transaction
// invert operation -> RemoveComponent
string operation = "core.ElementManagementLayer.RemoveComponent(\""
+ componentName +"\",true);";
// add the operation to the transactional context using its txID
core.TransactionMngr.AddArchitecturalOperation(txID, operation);
}
(...)
}
```

Figure 135. An inverse operation

On the other hand, the *TransactionManager* class redirects each message with the information of the inverse operation to the *TransactionalContext* class. The *TransactionalContext* class writes the inverse operation in its *log*. In this way, each transactional context has a file where it writes the operations which has to execute if the transaction fails. If the root transaction is satisfied, the log is eliminated. If the transaction fails, the middleware of the originator transaction executes the in inverse order the written operations.

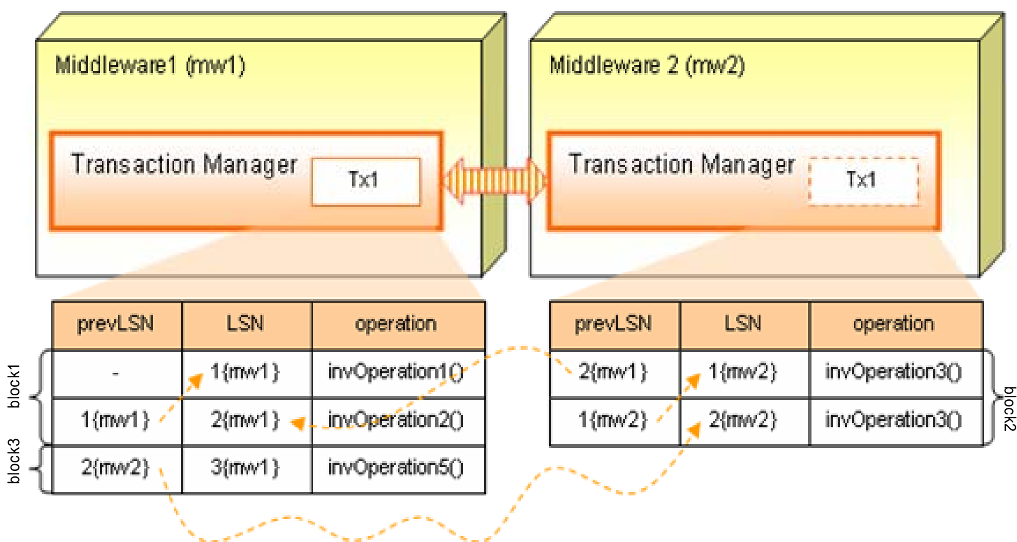


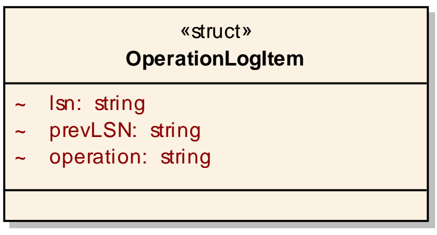
Figure 136. A log functioning

Figure 136 shows how two Transaction Managers of different middlewares create a log. If an operation which modifies the state of the architecture is executed during the *Tx1* transaction and its inverse operation is *invOperation1()*, the Transaction Manager is notified in order to write in a transactional *log*. The Transaction Manager generates a LSN for writing and since it is the first operation it leaves the prevLSN record empty. If new services that modify the state of the architecture, then new writings can be made during the execution of the transaction. If the limit of the transactional context is extended to another middleware and

performs operations that change the state of the architecture, then the inverse operations are written in the log of the *TransactionalContext* of the other middleware. The remote Transaction Manager creates an LSN and asks the originator middleware about the prevLSN. For example, when an operation whose inverse operation is *invOperation3()* is executed in *Middleware2*, the *Transaction Manager* of *Middleware2* asks *Middleware1* about the *prevLSN*. As a result, Middleware1 answers it and asks it about the LSN that it will write.

In the case a transaction fails, the inverse operations have to be executed in an inverse order as they were written. This is achieved by following in an inverse order the prevLSN and starting from the originator middleware. The operations are executed in blocks in each middleware. For example in Figure 136, three blocks have been marked. Each block represents a jump of the transaction from one middleware to another. In the recovering process, first the inverse operations of *block3* are executed, then the inverse operations of *block2*, and finally the ones of *block1*.

Three classes have been implemented for supporting the mechanism described in this section (see Figure 137). The *TransactionalArchitecturalOperations* class implements the log, the *OperationLogItem* class implements an inverse operation, and the *ArchitecturalOperationsUndoer* class executes the inverse operations. The *OperationLogItem* class has three attributes: *lsn* for saving a LSN, *prevLSN* for pointing to the previous operation, and *operation* for storing the name of the operation.



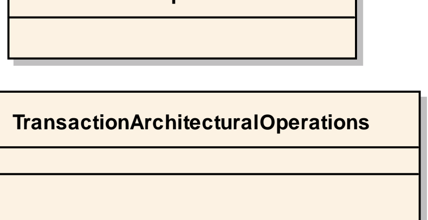


Figure 137. Classes which implement the *log*

10.3.9 Ambient-PRISMA Execution Model

The Ambient-PRISMA Execution Model layer implements the needed constructs for executing ambients. In the following, the primitives which are supported by this layer are explained.

10.3.9.1 Ambient Construct

The PRISMANET execution model includes the *ComponentBase* class. This class is responsible for implementing architectural elements by defining aspects, weavings and ports. Since ambients are also defined in this way, the `Ambient` class has been implemented by extending the *ComponentBase* class (see Figure 138). The *Ambient* class inherits the *ComponentBase* properties and methods and also defines its own. A characteristic of all ambients is that they have the predefined Mobility and Coordination aspects as well as their five predefined ports. Figure 11 shows that the predefined aspects have been included by using the *AddAspect()* method in the *Ambient* constructor as well as the predefined ports by the *Add()*method.

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

Figure 138. A segment of the `Ambient` class constructor

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

10.3.9.2 The Mobility Aspect

The Mobility Aspect allows an ambient to provide Ambient Calculus capabilities to mobile architectural elements as well as other services in order to provide a consistent state of the software architecture during and after a mobility process. Figure 139 shows that the *MobilityAspect* class extends the *AspectBase* class (the class that represents aspects in PRISMANET) and implements the *ICapability* interface. The *MobilityAspect* is a C# sealed class, i.e., the class cannot be extended. This is to emphasize that this aspect is predefined and the user does not have to include extra functionality.

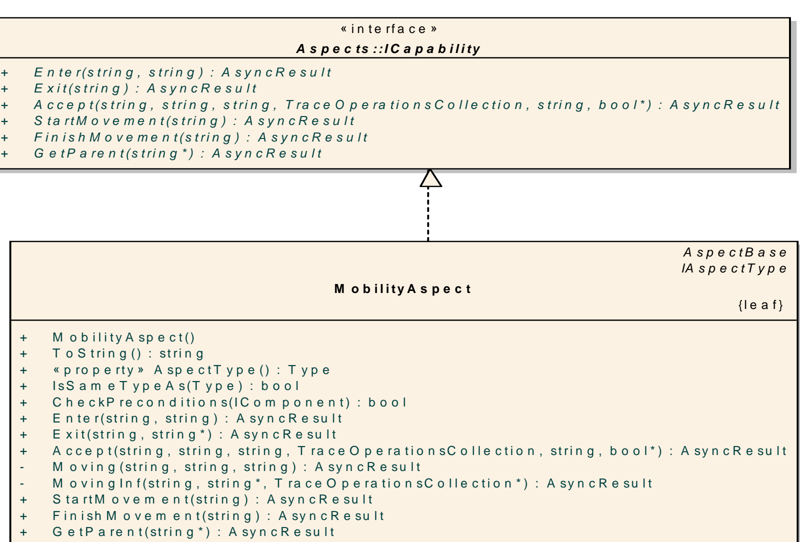


Figure 139. **The *MobilityAspect* class and the *ICapability* interface**

In the following, the *MobilityAspect* services are explained. These services have been implemented following their specification in the AOADL (see CHAPTER 8):

- *StartMovement(name)* service is in charge of managing the tasks needed for preparing an architectural element to move given its name. Basically, the

service prepares the attachments connected to the mobile element to be reconfigured (i.e., added or removed).

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

service of the moved architectural element in order to update its location value to the new parent ambient.

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

10.3.9.3 The Coordination Aspect

The Coordination Aspect provides the services of architectural elements that are located in an ambient to architectural elements that are located outside it. The Coordination Aspect also resends requests of architectural elements in an ambient to the parent ambient. The Coordination Aspect has been implemented in the *AmbientCoordinationAspect* class (Figure 140). This class extends the *AspectBase* class and implements the *ICall* interface. Like the *MobilityAspect* class, the *AmbientCoordinationAspect* class is a sealed class.

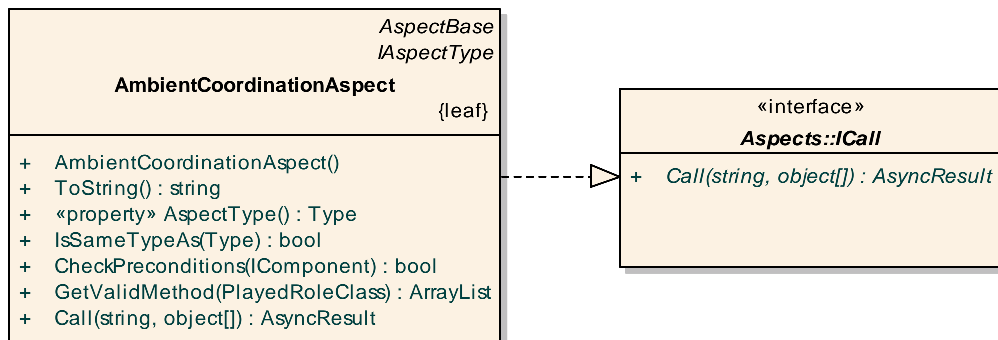


Figure 140. The AmbientCoordinationAspect class and the ICall interface

The *ICall* interface encapsulates requests in a generic invocation method called *Call* (see Figure 140). The *Call* service has two parameters: the name of the service as a `string` and the set of parameters of the service in an array of *objects*. In this way, all the petitions that the Coordination Aspect processes must be extracted in a `Call()` service. These must then be transformed into the original services when the petitions are sent to the architectural element ports.

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

10.3.9.4 The three kinds of Ambients

The three kinds of ambients: Group, Site, and Virtual all share the characteristics described in section 10.3.3. However, each ambient kind has its own characteristics. In Figure 141, shows how the three ambients extend the `Ambient` class.

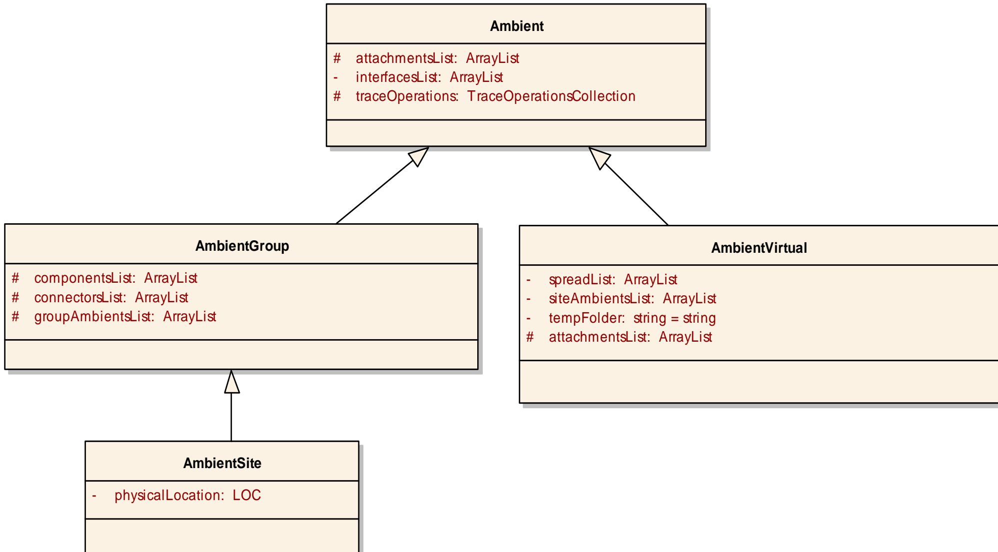


Figure 141. The three kinds of ambients in Ambient-PRISMANET

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

10.3.9.5 Group Ambients

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

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

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

10.3.9.6 Site Ambients

Site Ambients are implemented in the *AmbientSite* class. For example, the *HostSite* ambient that is specified in Figure 142 must inherit from the *AmbientSite* class in order to be executed. This kind of an ambient can contain the same architectural elements as a Group ambient. As a result, the *AmbientSite* class extends the *AmbientGroup* class in order to inherit its lists and the methods that manage them. The only difference is the *GetAmbientType*()method which must be overridden in order to return the string "AmbientSite".

```
Ambient_Site type HostSite
 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Weavings
     ADist.getLocation(Location)instead
                                MobilityAspect.getParent(Parent);
 End_Weavings
 Ports
     InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
     ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
     EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
     InServicesPort: ICall Played_Role ACoordination.INTERIOR;
     InRoutePort: IGetRoute Played_Role ADist.INTROUTE;
 End_Ports
End Ambient_Site type HostSite;
```

Figure 142. . Specification of the HostSite ambient

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

In other words, a Site ambient can be considered to be a bridge between an ambient hierarchy and the middleware where other architectural elements execute. Each time an element exits a Site, the element moves into a new host and starts

executing on another Ambient-PRISMANET middleware. However, this is transparent to the element that is moving, since it is the Site ambient that is in charge of performing these changes.

10.3.9.7 Virtual Ambients

Virtual Ambients are implemented in the *AmbientVirtual* class. For example, the *Root* ambient in Figure 143 is instantiated from the *AmbientVirtual* class in order to be executed. These kinds of ambients are the parents of Site ambients and represent the Ambient-PRISMANET execution environment that is made up of the different middlewares executing on the distributed hosts.

```
Ambient_Virtual type Root

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

 Weavings
     RootDist.getLocation(Location)instead
                                MobilityAspect.getParent(Parent);
 End_Weavings


 Ports
     InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
     ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
     EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
     InServicesPort: ICall Played_Role ACoordination.INTERIOR;
     InRoutePort: IGetRoute Played_Role RootDist.INTROUTE;

 End_Ports
End Ambient_Virtual type Root;
```

Figure 143. Specification of the Root ambient

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

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

The *AmbientVirtual* class stores the name of Site ambients that it locates in a list called *siteAmbientsList* (see Figure 141). Also, an element that is temporally passing through a Virtual ambient is stored in *tempFolder*. The methods of *AmbientVirtual* also override the methods of *Ambient* in a similar way to *AmbientGroup*. Most of the methods manage the list *siteAmbientsList*. The *AddLocalAttachment* method of the *AmbientVirtual* class creates: attachments between the ports of architectural elements (that are temporally located in the Virtual ambient) and the Virtual ambient ports or attachments between the ports of Site ambients and the ones of the Virtual ambient. The attachments that connect two Site ambients are distributed. In this case, the location of the distributed ambients must be identified using the Ambient-PRISMA DNS and be connected by .NET Remoting.

The Mobility Aspect of Virtual ambients throws an exception when the *startMovement* and *finishMovement* services are invoked by architectural elements located in it. This is due to the fact that the current implementation does not support the movement of Site ambients which are the only architectural elements located in Virtual ambients (other architectural elements are only temporally located).

10.3.10 Distributed Communication

In Ambient-PRISMA, communication among architectural elements is performed by the attachments. As explained in CHAPTER 8, the user is not aware of the creation of a complex attachment that is formed by other attachments. Ambient-PRISMANET

is responsible for creating the attachments that communicate two distributed architectural element instances.

10.3.10.1 Extending PRISMA Attachments

In PRISMA, a communication channel is formed by an attachment that connects a port of a component instance to a port of a connector instance. In Ambient-PRISMA, the distributed communication channel is not only formed by an attachment but by many attachments. An attachment can connect components and connectors to ambients, ambients to ambients, and components to connectors located in the same ambient. An attachment can also be used by many communication channels. For example in Figure 59, the attachment between the *AuctionSite* and the *ClientSite* can be used by any communication channel that connects an instance located in *AuctionSite* and an instance located in *ClientSite*.

```
Architectural_Model_Configuration MobileAgentsAuctionConf =

New MobileAgentsAuction
{
       ….
       AttchAuct1Cnct = new AttchAuctCnct (AuctionHouseCnct1,
                                           CnctAuctPortBidder,
                                           AuctionHouse1, BidderAuctPort);

        AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                                      AuctionHouseCnct1, CustPortAuct);
        AttchProc1Auc1 = new AttchCustAuc (Procurement1, PROCURAUCTPort,
                                      AuctionHouseCnct1, ProcurPortAuct);
…
};
```

Figure 144. Specification of attachments by the user in the AOADL

From now on, the term attachment is used to refer to a simple attachment and the term communication channel (an attachment defined by the user) is used to refer to an attachment in PRISMA. For example, the attachments shown in Figure 59 are referred to as attachments and the *attachments* specified by the user in

Figure 144 are *communication channels*. Thus, *attachments* can be added or removed when one of the instances they connect moves.

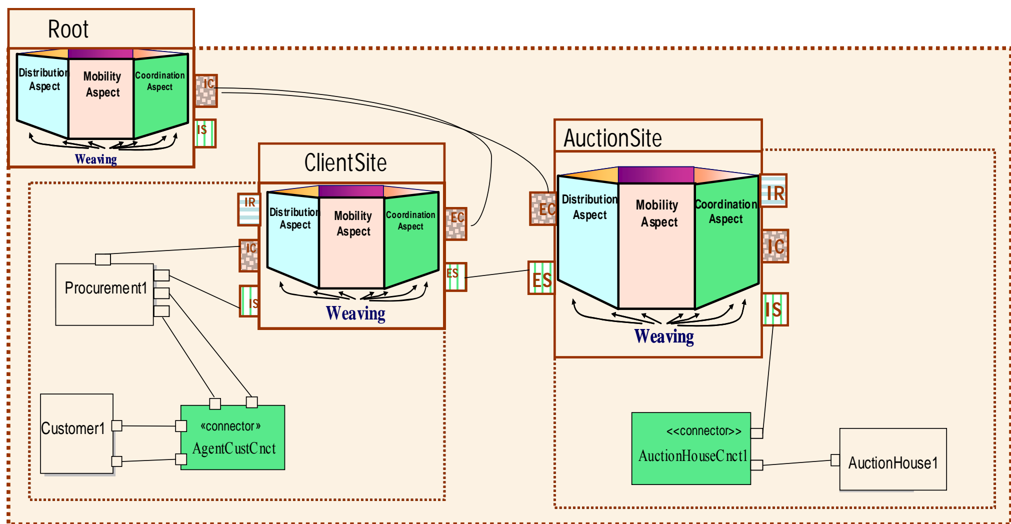


Figure 145. *Procurement1* component in the *ClientSite* ambient

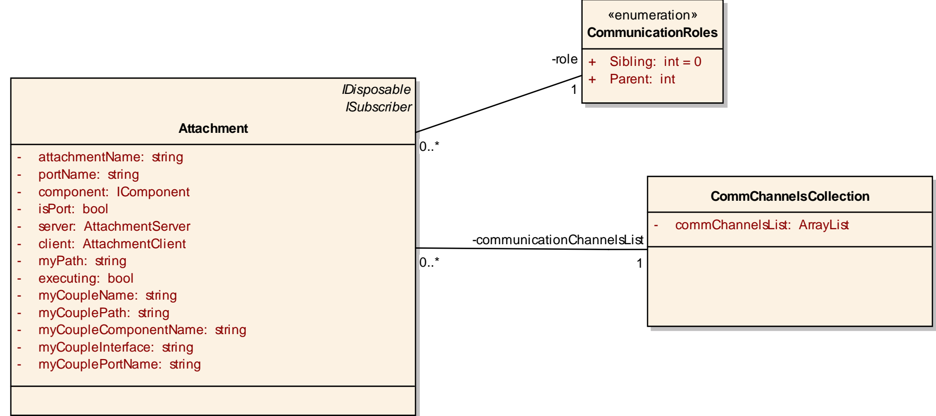


Figure 146. The *Attachment* class in Ambient-PRISMANET

To implement an attachment in Ambient-PRISMA, the *Attachment* class (see Figure 146) has a collection that stores the names of the communication channels that use an attachment instance (*communicationChannelsList*). As a result, the

Attachment class has methods for consulting and modifying the communication channel collection. Also, the *Attachment* has a property called *role* to indicate whether an attachment instance connects two siblings (*Sibling*) or it connects a parent ambient and an instance located in it (*Parent*). This information is needed to configure attachments when an architectural element is moved or for distributed communication.

10.3.10.2 Initial Configuration of Attachments

The initial configuration of the attachments is obtained as follows: Once the user specifies the communication channels in the configuration as shown in Figure 144, the Element Management Layer of the Ambient-PRISMANET middleware uses the DNS Layer in order to know on which host the instances that need to be connected are located.

Then a bottom-up search algorithm is applied to obtain the route of each instance in a tree hierarchy of ambients (i.e., from the instance passing through the intermediate ambients until the Site ambient is reached). A bottom-up search is more efficient because an instance knows its direct parent ambient, whereas an ambient has many children. Figure 147 explains how the algorithm works for getting the route of a component called *Account2*. As it can be observed, the *ElementManagement* Layer directly calls the *Account2* component. Then, a set of consecutive calls are performed until the Site ambient which represents the middleware where it is executing.

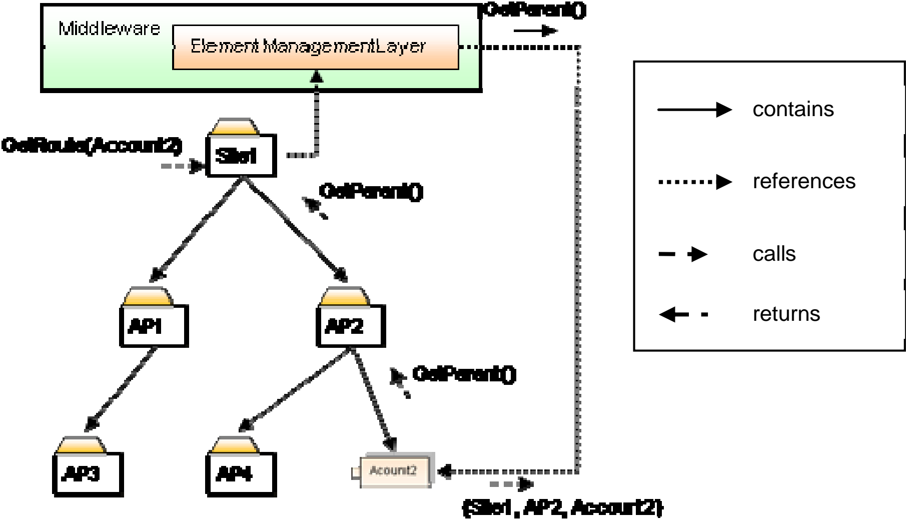


Figure 147. Bottom-Up algorithm for searching routes

Once the route of both instances is obtained in an ordered list, the names of ambients common to both routes are identified. An attachment with sibling role is created between the next elements on both lists, and the remaining attachments are created with role parent. For example, to create the communication channel between the *Procurement1* component and the *AuctionHouseCnct1* connector called *AttchProc1Auc1* (see Figure 144): the route obtained for the *Procurement1* is Root\ClientSite\Procurement1, and the route obtained for the *AuctionHouseCnct1* is Root\AuctionSite\AuctionHouseCnct1. Ambient-PRISMANET creates an attachment between *AuctionSite* and *ClientSite* with a sibling role, an attachment between *Procurement1* and *ClientSite* with a parent role, and an attachment between *AuctionHouseCnct1* and *AuctionSite* with a parent role (see Figure 148).

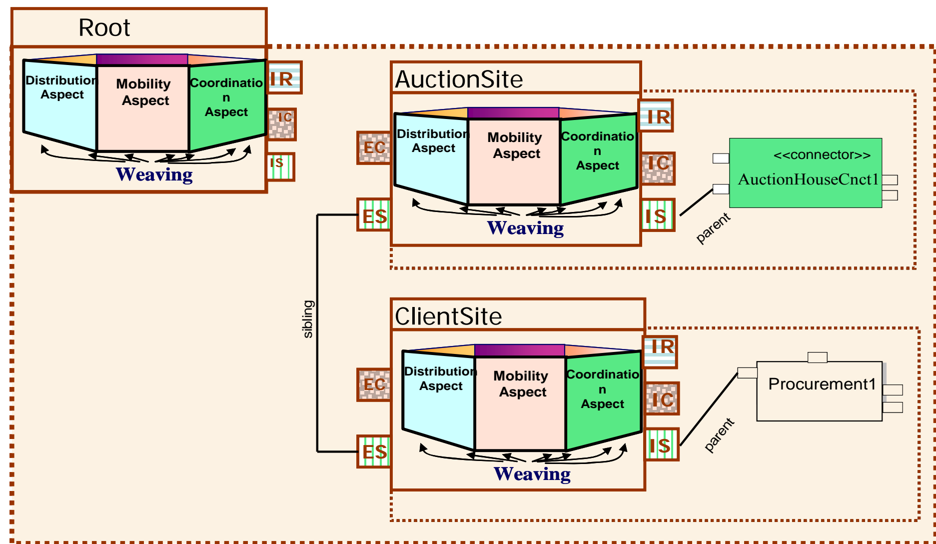


Figure 148. Creation of attachments

In this way, attachments are created in a transparent way. The user only specifies which architectural elements are connected and the Ambient-PRISMANET middleware automatically creates the attachments.

10.3.10.3 Reconfiguration of Attachments for Mobility

The attachments that form a communication channel can be modified at execution time due to the fact that an architectural element they connect can move from an ambient to another. As a result, each time an architectural element moves the set of attachments that connect it to the rest of architectural elements change.

When an architectural element exits or enters an ambient, the reconfiguration of the architecture only affects on the origin and the destination ambients i.e., the changes are transparent to the rest of the architecture. Many times a mobile architectural element has to move through intermediate ambients to reach its final destination. As a result, the attachments needed for the proper functionality of the moving architectural element in the intermediate ambients are only generated in order to simplify the attachments reconfiguration. For example, when the

Procurement1 component of Figure 148 needs to move to the *AuctionSite* ambient, the *Procurement1* component has to move through the *Root* ambient as an intermediate ambient. When the *Procurment1* component is in the *Root* ambient the attachments needed to connect the *Procurement1* component are not needed due to the fact that the *Procurement1* is going to enter *AuctionSite* and these need to be deleted. However, the attachment that connects the Procurement1 to the *InCapabilities* port (marked with IC) of the *Root* ambient is needed due to the fact that the Procurement1 needs to invoke mobility services.

As a result, an architectural element that needs to move has to invoke the *startMovement* service of its parent ambient (see section 10.3.9.2) in order to indicate its parent ambient that it is going to move and invoke the *finishMovement* service to its final destination ambient (see section 10.3.9.2). The *finishMovement* service permits an ambient to create all the attachments of a mobile architectural element. The *startMovement* service permits the origin ambient to create a list of structures that store the information of the current attachments of the architectural element. This structure is called *TraceOperation*. The list of *TraceOperation* structures is sent by an origin ambient to the destination ambient through the accept service (see section 10.3.9.2) . When an architectural element moves from an ambient, the ambient deletes the *TraceOperation* structures related to the mobile architectural element.

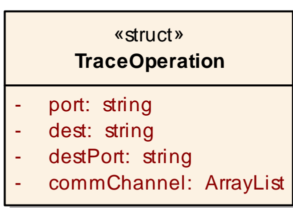


Figure 149. TraceOperation Structure

Each *TraceOperation* structure (see Figure 148) stores the name of the port where the mobile architectural element is connected (*port*), the name of the

architectural element which is connected to the mobile architectural element (*dest*), the name of the port of the connected architectural element (*destport*), and a list of communication channels names which are sharing the attachment (*commChannel*).

The trace of mobility of the *Procurement1* component of Figure 59 from the *ClientSite* ambient to the *AuctionSite* ambient is explained in order to illustrate in detail the reconfiguration process of attachments. When the *Procurment1* component needs to move, the *MOVE* transaction specified in Figure 150 is executed. The Procurement1 component invokes move(AuctionSite1) in order o indicate that it is moving to the *AuctionSite* ambient.

```
Distribution Aspect ProcurDist using IMobility, ICapability
…
TRANSACTIONS in MOVE (NewAmbient: Ambient)
      move =  CapParent_startMovement!(self.Name)→
              MOVE1;
      MOVE1 = CapParent_exit!(self.Name)→ MOVE2;
      MOVE2 = CapParent_enter!(self.Name, NewAmbient)→MOVE3;
      MOVE3 = CapParent_finishMovement!(self.Name);
      MOVE4 = CapParent_changeLocation?(Name, NewLocation);

…
End_Distribution Aspect ProcurDist
```

Figure 150. The move transaction of the *ProcurDist* aspect in the AOADL

The first service invoked by the *Procurement1* to the *ClientSite* ambient is the *startMovement(Procurement1).* The *ClientSite* ambient obtains all the attachments that connect the ports of the *Procurement1* component. The *ClientSite* ambient creates the list of *TraceOperation* structures called TO1 (see Table 8). Then, the *ClientSite* ambient removes all the attachments of the *Procurement1* component except for the attachment that connects the *Procurement1* component to its *InCapabilities* port.

Table 8. List of *TraceOperation* structure in ClientSite

TO1			
port	dest	destPort	commChannel[1]
DCapPort	ClientSite	InCapabilitiesPort	Procurement1 ←→ ClientSite
DMovingPort	AgentCustCnct	CustPortProcurMo	Procurement1 ←→ AgentCustCnct
CUSTPROCPort	AgentCustCnct	ProcurPortCust	AgentCustCnct←→Procurement1
PROCURAUCTPort	AuctionHouse1Cnct1	ProcurPortAuct	Procurement1 ←→ AuctionHouse1Cnct1

Then, the *Procurement1* component requests the *exit* service. As a result, the *ClientSite* ambient sends to the *Root* ambient the list of *TraceOperation* structures through the accept service. The Root ambient receives the list and generates another list of *TraceOperations* structures called TO2, which stores the attachments that the Root has to create (see Table 9). In addition, the Root creates an attachment between the DCapPort port of the *Procurement1* component and its InCapabilities port.

Table 9. List of *TraceOperation* structure in Root

TO2			
port	dest	destPort	commChannel
DCapPort	Root	InCapabilitiesPort	Procurement1 ←→ Root
DMovingPort	AgentCustCnct	CustPortProcurMo	Procurement1 ←→ AgentCustCnct
CUSTPROCPort	AgentCustCnct	ProcurPortCust	AgentCustCnct←→Procurement1
PROCURAUCTPort	AuctionHouse1Cnct1	ProcurPortAuct	Procurement1 ←→ AuctionHouse1Cnct1

Once the *Procurement1* component is in the *Root* ambient, the *ClientSite* modifies the attachments in its boundary that connected the Procurement1 using TO1 and deletes the attachment that connected the Procurement1 to the InCapabilitiesPort port of the *ClientSite* ambient. Finally, the *ClientSite* deletes the TO1. Figure 151 shows the state of the configuration after that the *Procurement1* component has

---

[1] The text of the commChannel should also include the ports of the connected architectural elements. For brevity they have been ommited.

exited from the *ClientSite* ambient. As it can be noticed, new attachments have been created in *ClientSite* that provide architectural elements connected to *Procurement1* to remain connected to it although it has moved.

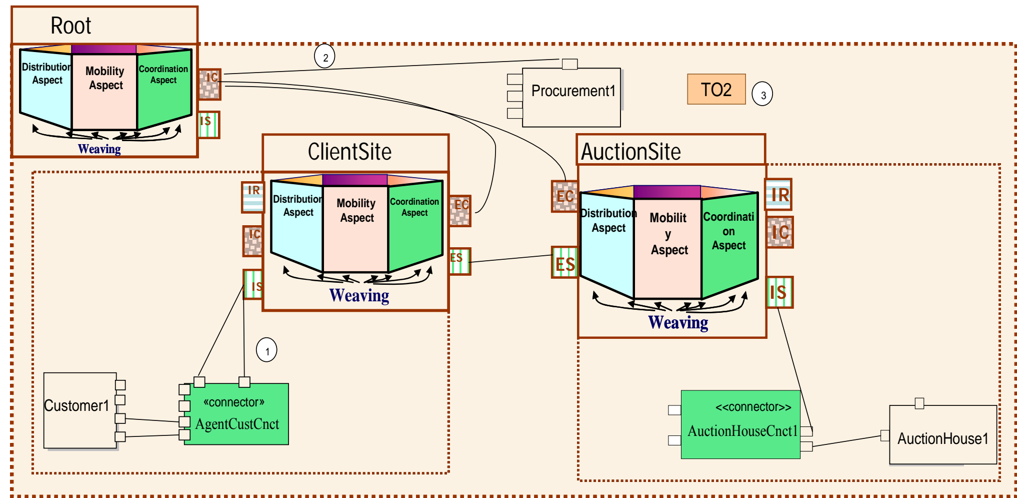


Figure 151. Configuration after that Procurement1 has exited ClientSite

Then, the *Procurment1* component requests the enter(Procurement1, AuctionSite) service to the *Root* ambient. In a similar way to the exit, the *Root* sends the TO2 list to the *AuctionSite* ambient, the *AuctionSite* ambient creates the TO3 list, and the *AuctionSite* creates an attachment between the DCapPort of Procurement1 and its InCapabilitiesPort port. Then the *Procurement1* enters the *AuctionSite* ambient. Afterwards, the Root checks whether it needs to make any attachments modifications in its boundary. In this case, the *Root* only deletes the attachment between the *Procurement1* and its InCapabilitiesPort port. Thus, the attachment between the *AuctionSite* ambient and the *ClientSite* ambient is used to connect the *Procurement1* component to the *AgentCustCnct* connector.

Finally, the *Procurement1* component requests the *finishMovement(Procurement1)* service from the *AuctionSite* ambient. As a result, the *Root* uses the TO3 to create the attachments needed for connecting Procurement1 (see Figure 152).

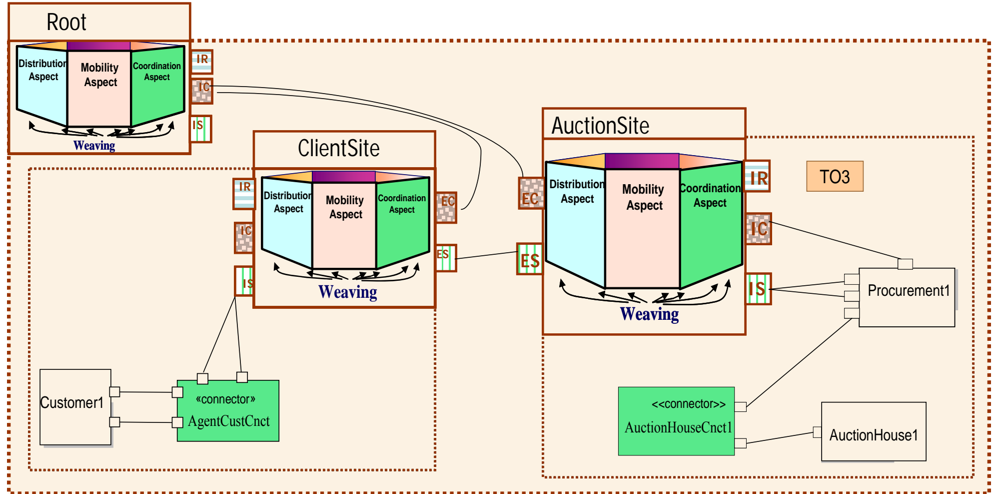


Figure 152. Configuration after that Procurement1 has requested the finishMovement service to AuctionSite

10.3.11 Mobility through Ambients

The different entities that play an important role in mobility have been explained in previous sections of this chapter (see sections 10.3.4, 10.3.9.2, and 10.3.10.3). In this section, the role of ambients in the implementation of mobility is explained.

Basically, architectural elements are provided with mobility through the MobilityAspect aspect of ambients. The MobilityAspect aspect provides a set of services (see section 10.3.9.2). The *startMovement* and the *finsihMovement* services are important for reconfiguring the attachments of a mobile architectural element, as explained in section 10.3.10.3. The *exit* and *enter* services are invoked by an architectural element to an ambient in order to allow it to be released from an origin ambient and be received in a destination ambient. When an architectural element invokes the exit and enter services, an ambient executes the *moving* and the *accept* services.

The *moving* service provides the origin ambient to manage the reconfiguration of its internal architecture. The *accept* service provides a destination ambient to

manage the reconfiguration of its internal architecture. The *moving* and the *accept* services are in common in all ambients. However, their internal implementation is different depending on whether the ambient is of kind Virtual or not. In the following, these differences are explained in detail.

10.3.11.1 The Moving service

The Moving service of an ambient is a transaction which consists of executing a set of services. The moving transaction executes the accept service of the destination ambient (the parent ambient or child ambient of the origin ambient) . As a result, the origin ambient executes a service called *RecomposeCommChannels*. If the *RecomposeCommChannels* service has been invoked after an invocation of an exit service, then the *MovingOnEntering (requested, newAmbient)* service is executed. If the *RecomposeCommChannels* service has been invoked after an invocation of an enter service, then the *MovingOnExiting (requested, newAmbient)* service is executed.

- *MovingOnEntering (requested, newAmbient)*: The service uses the list of *TraceOperation* structures (see section 10.3.10.3) to modify the attachments. If the mobile architectural element is connected to architectural elements located in the origin ambient, then attachments are created between the ports of these architectural elements and the EServicesPort port of the destination ambient.
- *MovingOnExiting (requested, newAmbient)*: The service uses the list of *TraceOperation* structures to modify the attachments. If the mobile architectural element is connected to architectural elements located in the origin ambient, then attachments are created between the ports of these architectural elements and its InServicesPort port.

The previous explanation is common to Site and Group ambients. However, Virtual ambients do not have the *MovingOnExiting* service implemented since the current version does not support many Virtual ambients. In addition, the *MovingOnEntering* service of a Virtual ambient not only modifies attachments for the proper communication of the mobile architectural element but it also invokes the

MOVE service (see section 10.3.4) of the Distribution Management Layer in order to serialize the architectural element.

10.3.11.2 The Accept service

The accept service is a transaction which consists of executing the *AddChild* service at the destination ambient. The *AddChild* service allows an architectural element to be located in an ambient. As a result, it adds the mobile architectural element to its list of elements and Similarly to the *Moving* service (see section 10.3.11.1), *AddChild* invokes one of the following services:

- *AcceptOnEntering (newAmbient, requested, origTraceOp)*: This service is invoked when the mobile architectural element originates from the parent of the destination ambient. As a result, the service uses the list *origTraceOp* in order to generate its own list of *TraceOperation* structures and modifies the attachments in the destination ambient. This service creates an attachment between the DCapPort port of the mobile architectural element and the InCapabilitiesPort port of the destination ambient. Then, if an attachment of the mobile architectural element has a parent role i.e., it was connected to the origin ambient, then the destination ambient creates an attachment between the port of the mobile architectural element and its InServicesPort port. If an attachment of the mobile architectural element has a sibling role, then the destination ambient checks whether this attachment used to connect the mobile architectural element to one of its ports. If it did, then it connects the mobile architectural element to one of the architectural elements located in it. If it didn't, then it connects the mobile architectural element to the InServicesPort port of the destination ambient.
- *AcceptOnExiting (newAmbient, requested, origTraceOp)*: This service is invoked when the mobile architectural element originates from a child of the destination ambient. As a result, the service uses the list *origTraceOp* in order to generate its own list of *TraceOperation* structures and modifies the attachments in the destination ambient. This service creates an attachment

between the DCapPort port of the mobile architectural element and the InCapabilitiesPort port of the destination ambient. Then, if an attachment of the mobile architectural element has a sibling role, then the destination ambient creates an attachment between the port of the mobile architectural element and its EServicesPort port. If an attachment of the mobile architectural element has a parent role, then the destination ambient it connects the mobile architectural element to one of the architectural elements located in it.

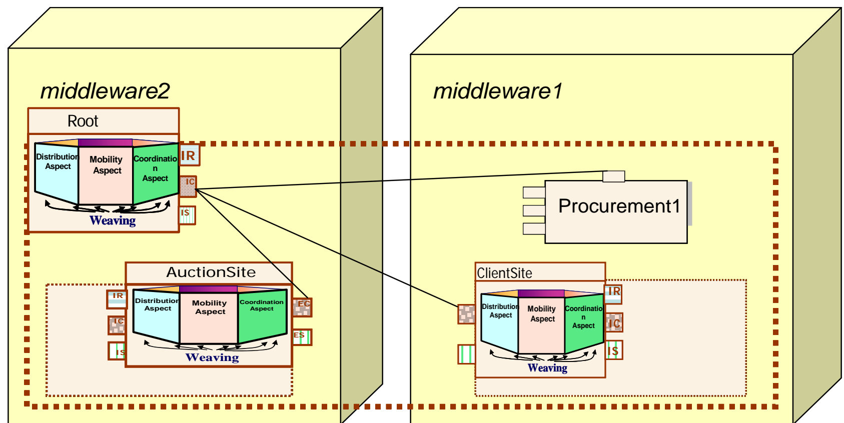


Figure 153. The Procurement1 component in the Root ambient

The previous explanation is common to Site and Group ambients. However, Virtual ambients do not have the *AcceptOnExiting* service implemented since the current version does not support many Virtual ambients. In addition, the *AcceptOnEntering* service of a Virtual ambient not only modifies attachments for the proper communication of the mobile architectural element but it also adds the architectural element in its *tempFolder* list (see section 10.3.9.7). This is due to the fact that the architectural element is staying in the Virtual ambient temporally. As a result, the mobile architectural element does not physically move and it stays in the same host. In this way, the attachment that connects the DCapPort port of the mobile architectural element and the InCapabilitiesPort port is a distributed attachment (see Figure 153).

10.4 Conclusions

This chapter has presented the Ambient-PRISMA Case Tool prototype. The current version of the Ambient-PRISMA Case Tool includes the Type Definition Graphical Modelling Tool, and the Ambient-PRISMANET. The PRISMA Types Graphical Modelling Tool gives support to the specification of Ambient-PRISMA software architectures following the MDE approach and using the Ambient-PRISMA AOADL in a graphical way. As a result, distributed and mobile aspect-oriented software architectures are specified in a more intuitive and friendly way and by providing mechanisms to verify their models.

The Ambient-PRISMANET middleware supports the execution of Ambient-PRISMA specifications of distributed and mobile applications in .NET technology. The Ambient-PRISMANET maps Ambient-PRISMA ambients into C# classes. In addition, it gives support for Ambient-PRISMA applications to execute in a distributed runtime environment in a transparent way. Thus, Ambient-PRISMANET provides a distributed DNS, support for distributed transactions and support for code mobility. In addition, the Ambient-PRISMANET implements the needed services for automatically configuring a distributed software architecture in a transparent way to the user and reconfigure the mobile software architectures with ambients.

Ambient-PRISMANET middleware has a console which allows the user to manage and consult the state of the distributed software architecture environment. In addition, a generic GUI which allows the user to interact with the executing distributed software architecture is provided.

The Ambient-PRISMANET has been used to manually implement the Mobile Agents Auction, a bank system case study, and a tele-operated robot case studies. As a result, the code generation patterns have been identified to automatically generate distributed applications from Ambient-PRISMA. The code generation patterns describe how classes of the Ambient-PRISMANET have to be extended in order to generate distributed and mobile applications. In future works, the detected

patterns are going to be incorporated to the Ambient-PRISMA Case Tool in order to automatically generate the code.

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

- **Nour Ali**, Carlos Millán, Isidro Ramos, "Developing Mobile Ambients using an Aspect-Oriented Software Architectural Model", 8th International Symposium on Distributed Objects and Applications (DOA 2006), **LNCS** Springer Verlang, Montpellier, France, October, 2006.
- Cristobal Costa, **Nour Ali,** Carlos Millan, Jose A. Carsi**,** "Transparent Mobility of Distributed Objects using .NET", 4th International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2006.
- Jennifer Pérez**, Nour Ali**, Cristobal Costa, José Á. Carsí, Isidro Ramos, "Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*"*, 3rd International Conference on .NET Technologies, pp. 97-108, Pilsen, Czech Republic, May-June 2005.
- **Nour Ali**, Jennifer Pérez, Cristobal Costa, Jose A. Carsí, Isidro Ramos, "Implementation of the PRISMA Model in the .Net Platform", II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- Carlos Millán, **Nour Ali**, Isidro Ramos, "Developing Distributed and Mobile Applications from an Aspect-Oriented Architectural Model", Technical Report DSIC, Polytechnic University of Valencia, October, 2006 (In Spanish).
- **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "A Compiler for the Automatic Generation of the Distribution Aspect to Distributed Applications", Technical Report DSIC-II/15/04, Polytechnic University of Valencia, October 2004. – 2004

CHAPTER 11

AMBIENT-PRISMA METHODOLOGY

"Though this be madness, yet there is method in 't. Will you walk out of the air, my lord?"

Shakespeare Hamlet, *1600*

Ambient-PRISMA provides a methodology for modelling distributed and mobile aspect-oriented software architectures. This methodology offers guidelines to the analyst usinf Ambient-PRISMA for allowing him/her to correctly develop distribution and mobility in an easy and simple way. The methodology presented in this chapter takes into account that the user is modelling software architectures using the methodology presented in section 6.3. The PRISMA methodology includes three stages: detection of architectural elements and aspects, software architecture modelling, and code generation. The modelling methodology of Ambient-PRISMA extends each stage in order to develop distributed and mobile software systems.

In the following the different stages of the methodology are explained.

11.1 Detection of architectural elements and aspects

The first stage of the methodology is to detect which architectural elements and aspects are relevant. In Ambient-PRISMA architectural elements consist of ambients, components, connectors, and systems. The main aspects that have to be detected for Ambient-PRISMA are the distribution aspects and the replication aspects.

This stage consists of the following steps:

11.1.1 Identification of Virtual and Site ambients

A distributed system consists of physical locations that can be located on different domains. The kinds of site ambients and virtual ambients depend on the physical locations and the domains, respectively. As a result, the analyst has to know the kind of network to be modelled. For example, in the Mobile Agents case study the network is a WAN that consists of hosts located on different LANs. However, a wireless sensor actuator network consists of regions and a host.

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

11.1.2 Identification of Components, Connectors and Systems

This step consists of identifying which components compose the software architecture, which connectors coordinate their functionalities, and which systems are composed of components and connectors (see [Per08]).

Once components, connectors, and systems are identified, the analyst has to answer the following questions for each architectural element:

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

Once an analyst answers this kind of questions, the services related to mobility and replication can be detected. As a result, interfaces needed for requesting and providing objective moves or replicas are detected. If there are no objective moves or

replicas, then there may be no interfaces for distribution or mobility. In addition, ports and connections between ports are identified.

11.1.3 Identification of Group ambients

Group ambients are identified after components, connectors, and systems due to the fact that group ambients represent logical domains, i.e, a group of architectural elements that share a set of characteristics. As a result, an analyst can detect group ambients by detecting whether there are a set of architectural elements that need to move together or not, share some resources, security characteristics, etc.

Once group ambients are detected, the analyst has to answer to the questions presented in section 11.1.2. For example, if a group ambient is mobile or not, if it is mobile through objective moves, etc. Interfaces, ports, and connections that a group ambient needs can be detected by answering these questions.

11.1.4 Identification of distribution and replication aspects

All architectural elements in Ambient-PRISMA need a distribution aspect. Each kind of ambient, needs a different distribution aspect as explained in CHAPTER 8. In addition, an analyst has to answer the following questions in order to detect the different distribution and replication aspects of architectural elements:

- What causes an architectural element to move or replicate?
- Does an architectural element have any constraints on the locations it can move?
- Does a mobile architectural element need to perform specific tasks after or before moving or replicating?

When these questions are answered, an analyst can detect the different kinds of behaviours that are needed. As a result, the sections of an aspect such as attributes, services, preconditions, constraints, triggers, transactions, etc can be also detected.

11.2 Software Architecture Modelling

Once architectural elements and aspects are identified, the analyst can use either the Ambient-PRISMA CASE Tool for modelling graphically or by textually specifying a software architecture. The modelling process is flexible enough for allowing the user to choose between two options for including distribution and mobility characteristics in the modelling process:

- 1st Option: To first model the PRISMA aspect oriented software architecture and then complete it with the distribution and mobility characteristics such as distribution aspects, replication aspects, and ambients.
- 2nd Option: To model the distribution and mobility characteristics in parallel with the PRISMA aspect-oriented software architecture.

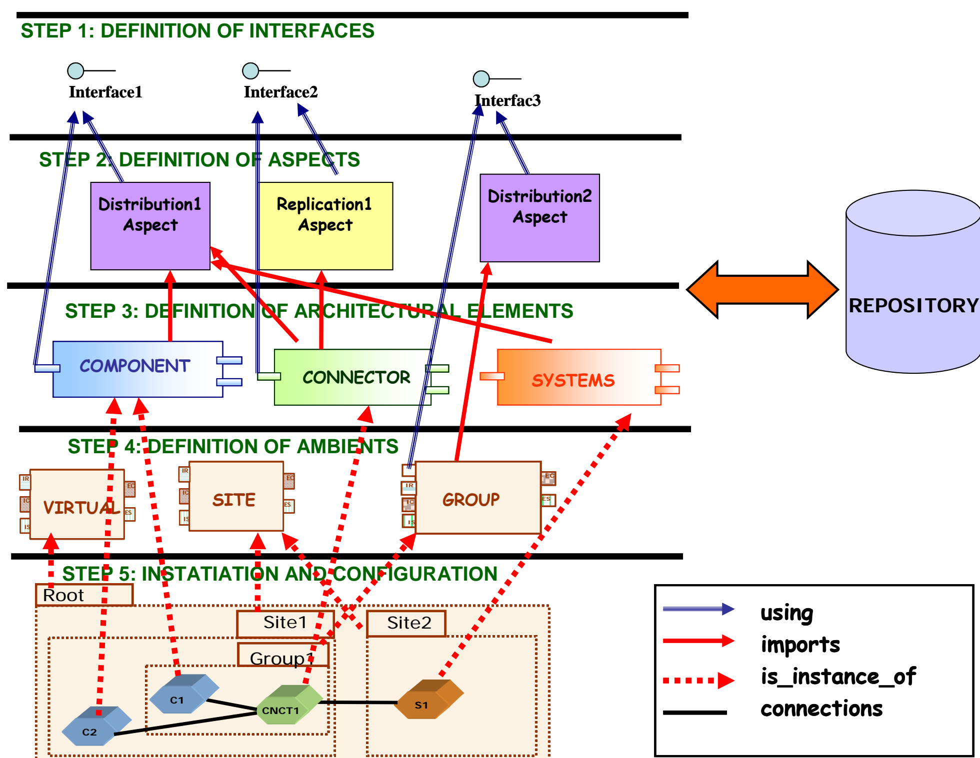


Figure 154. The modelling stage of the Ambient-PRISMA methodology

The methodology includes five modelling steps: 1st interfaces, 2nd aspects, 3rd components, connectors, and systems, 4th ambients, and 5th instantiation and configuration (see Figure 154). The enumeration of these steps is not restrictive. Similarly to PRISMA, this enumeration only indicates the dependencies between the concepts. For example, the modelling of ambients can be done before, during or after the components and connectors, and systems modelling step. Thus, the modelling process is incremental.

In the following, the different steps for modelling distribution and mobility characteristics are explained:

11.2.1 Interfaces

The interfaces identified in step 11.1.2 and 11.1.3, are specified by modelling the services and their signatures. The interfaces specified are stored in a repository for being reused (see step 1, Figure 154). Interfaces defined for distribution and mobility can be defined in parallel with interfaces for functional or other concerns of the software architecture.

The specifications of the interfaces of the Auction System Case Study are presented in APPENDIX B.

11.2.2 Aspects

The distribution and the replication aspects identified in section 11.1.4 are modelled after the specification of interfaces because many aspects have to use interfaces if they need to offer and provide public services.

Aspects are reusable entities that define a specific behaviour of a crosscutting concern (see step 2, Figure 154). As a result, aspects can be reused not only in a specification of a software architecture but in many software architectures. Distribution and replication aspects can be defined in parallel with aspects that describe other crosscutting concerns of PRISMA software architecture. The

specifications of the aspects of the Auction System Case Study are presented in APPENDIX B.

11.2.3 Components, Connectors, and Systems

Components, connectors, and systems are defined in step 3 because they import aspects, and use interfaces for offering and requesting services through their ports (see step 3, Figure 154). They are defined by importing aspects, defining ports, and defining weavings. Systems are also defined by importing other components and connectors. In Ambient-PRISMA, the specification of components, connectors and systems has to import a distribution aspect. As a result, in this step each one of the architectural elements identified in step 11.1.2 has to indicate which distribution aspect it imports. These architectural elements can also import a replication aspect.

In addition, architectural elements that have been detected to be mobile have to define the port called DCapPort for requesting capabilities from their parent ambients, an architectural element that has been detected to be replicable has to define the port called RCapPort, and an architectural element that needs to request routes of other architectural elements has to define the port called DRoutePort (see section 7.2.5). Other ports can be defined depending on whether there are objective moves or replicas.

An architectural element can define weavings that trigger services for distribution and mobility or weavings that trigger services of other crosscutting concerns for distribution and mobility. Weavings for mobility and replication are defined when mobility or replication are caused due to the fact of a behaviour specified in another aspect. For example, mobility or replication of an architectural element can be caused due to a change in behaviour of a functional aspect.

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

The specification of components, connectors and systems of the Auction System Case study are presented in APPENDIX B.

11.2.4 Ambients

Ambients that are identified in steps 11.1.1 and 11.1.3 are defined in a similar way to the rest of architectural elements. The difference is that ambients have predefined constructs. As a result, some ambients need to import extra interfaces and aspects (extra behaviours) and others do not (see step 4, Figure 154). Ambients that define extra behaviours can specify extra ports, aspects and weavings.

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

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

The specifications of the ambients types of the Auction System Case study are presented in APPENDIX B.

11.2.5 Instantiation and Configuration

Finally, a specific configuration of a distributed software architecture is defined. Types of ambients, components, connectors, and systems are instantiated by specifying where the instances are located. Then, attachments and bindings are instantiated for connecting a set of components, systems, and connector instances

with each other. If an ambient instance has some extra ports also attachments are defined to connect the ambient with other architectural elements.

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

- Physical locations of the software architecture are instantiated by the *loc* data type.
- Ambients are instantiated. First, virtual ambients are instantiated by indicating their parent ambient with the exception of the Root ambient which does not have a parent ambient. Second, each site ambient is instantiated by indicating both its parent ambient and an instance of the *loc* data type. Finally, group ambients are instantiated by indicating their parent ambient.
- Constraints for specifying which ambients cannot be connected.
- Components, connectors, and systems are instantiated by indicating the name of their parent ambients.
- Attachments that connect architectural elements are instantiated.

The specification of an initial configuration of the Auction Site Case study is presented in APPENDIX B.

11.3 Code Generation

Once a specific configuration is defined, the automatic C# code generation can be performed. However, the current version of the Case Tool does not support this step.

In future versions the code generation templates are going to be integrated in the Tool. Once the code is generated, the Ambient-PRISMANET middleware will automatically deploy the architectural elements in their corresponding hosts and execute the application. The user can interact with the application using the generic console that the CASE Tool provides.

11.4 Conclusions

This chapter presents a methodology that follows MDE in order to develop distributed and mobile software systems from Ambient-PRISMA models. The methodology gives guidelines to the analyst in order to facilitate its task when modelling distribution and mobility characteristics of its software architectures. It also takes into account that the analyst needs flexibility while he/she is modelling the structural and functional behaviour of the software architecture using PRISMA.

The guidelines that the methodology includes, shows how an analyst can model different distribution, mobility and replication behaviours using Ambient-PRISMA. Specifically, the methodology shows how objective and subjective moves or replicas can be specified in a platform independent way.

Applying the methodology of Ambient-PRISMA is simple thanks to the predefined constructs of ambients presented in CHAPTER 8. In addition, Ambient-PRISMA interfaces, distribution aspects, replication aspects, and ambients are stored in a repository which allows their reusability not only in the same software architecture but also in different ones.

Finally, the Code Generation step is proposed but it is still not available since the current version of the Case Tool does not generate automatically the code.

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

- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose A. Carsí, "Distributed Replication in Aspect-Oriented Software Architectures Using Ambients", Revista IEEE America Latina, Special Edition - JISBD'2006, Volume 5, Issue 4, July, pp. 231- 237, 2007. (Invited: JISBD)(In Spanish)
- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, "Distributed Replication in Aspect-Oriented Software Architectures Using

Ambients", XI Conference on Software Engineering and Databases (JISBD) (JISBD), Sitges, Barcelona, Octubre 3-6, 2006. (In Spanish)

PART VI. CONCLUSIONS AND FURTHER WORKS

CHAPTER 12

CONCLUSIONS AND FURTHER WORKS

"Wherever life takes us, there are always moments of wonder."

Jimmy Carter

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

12.1 Conclusions

This thesis presents a framework called Ambient-PRISMA for developing distribution and mobility software systems in a platform independent way. Ambient-PRISMA integrates different software engineering techniques with formal methods. Specifically, Ambient-PRISMA enriches an aspect-oriented software architecture framework called PRISMA in order to allow its software architectures to be distributed and mobile. Concretely, it extends its metamodel, language, and Case Tool in order to incorporate the ambient concept of Ambient Calculus and primitives for mobility. In addition, Ambient-PRISMA concepts have been formalized using ChAC in order to provide unambiguous specifications of distributed and mobile behaviour. A case study of an Auction System with mobile agents has been used throughout the thesis to illustrate Ambient-PRISMA.

One of the contributions of this thesis is the introduction of the ambient concept in order to provide an explicit notion of location at the software architecture phase of software development. An ambient in Ambient-PRISMA has been introduced as a new type of connector that controls or coordinates a boundary where other architectural elements are located. The ambient connector uses aspects in order to provide architectural elements located in its boundary with mobility services and distributed communication. Three kinds of ambient have been defined in order to allow the modelling of a hierarchy of a distributed network.

Since an ambient has been introduced as an architectural element, mobility of architectural elements is supported by reconfiguring the software architecture. Since mobile architectural elements have to change their connections adapting them to the new location. Another characteristic of Ambient-PRISMA is that it allows all architectural elements to be mobile entities and does not restrict mobility to a particular type of architectural elements. Ambient-PRISMA provides architectural elements to be aware of their location. As a result, architectural elements can be autonomous in taking decisions about their location. Also, the migration decision in Ambient-PRISMA can be modelled as both objective and passive.

Another contribution of the thesis is that Ambient-PRISMA follows MDE in order to develop distributed and mobile software systems by using models. As a result, Ambient-PRISMA concepts have been included in the PRISMA metamodel in order to enable the construction of correct models. The definition of a metamodel has facilitated the task in using tools such as DSL for providing a graphical notation to the Ambient-PRISMA language and in future works will facilitate the automatic code generation.

In addition, the Ambient-PRISMA AOADL has extended the PRISMA AOADL with ambients at the type definition level and at the configuration level. At the type definition level, analysts can specify ambients and architectural elements with distributed and mobile behaviours using aspect-oriented techniques. This improves the maintainability and reusability of the specifications of distributed and mobile

software systems. At the configuration level, an analyst can reuse the different ambients and architectural elements in order to configure the topology of distributed software architecture.

Ambient-PRISMA also provides a Case Tool that currently provides a graphical Modelling Tool and the Ambient-PRISMANET middleware. The graphical modeling tool provides Ambient-PRISMA software architectural models to be specified in a more intuitive and friendly way as well as by providing mechanisms to verify the models. The Ambient-PRISMANET middleware implements the mappings between Ambient-PRISMA concepts and .NET, provides Ambient-PRISMA applications to be executed on .NET platform, provides them with a distributed runtime environment for executing distributed transactions, automatic deployment of distributed architectural elements on different hosts, management of failures of hosts, weak mobility, etc.

In addition, Ambient-PRISMA proposes a methodology which gives guidelines to the analyst in order to facilitate its task for modelling distribution and mobility characteristics of its software architectures. Specifically, the methodology shows how objective and subjective moves or replicas can be specified in a platform independent way.

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

INTERNATIONAL JOURNALS

- Jennifer Perez, **Nour Ali**, Jose A. Carsi, I. Ramos, Barbara Alvarez, Pedro Sanchez, Juan A. Pastor, "Integrating Aspects in Software Architectures: PRISMA Applied to Robotic Tele-operated Systems", Information and Software Technology, 2007. (accepted)

NATIONAL JOURNALS

- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose A. Carsí, "Distributed Replication in Aspect-Oriented Software Architectures Using Ambients", Revista IEEE America Latina, Special Edition - JISBD'2006, Volume 5, Issue 4, July, pp. 231- 237, 2007. (Invited: JISBD)(In Spanish)

INTERNATIONAL CONFERENCES

- Cristobal Costa, **Nour Ali**, Jennifer Perez, Jose Angel Carsi, Isidro Ramos, "Dynamic Reconfiguration of Software Architectures Through Aspects", First International Conference on Software Architecture (ECSA 2007), **LNCS** 4758, Springer Verlang, Madrid, September, 2007 (poster).
- **Nour Ali**, Carlos Millán, Isidro Ramos, Developing Mobile Ambients using an Aspect-Oriented Software Architectural Model, 8th International Symposium on Distributed Objects and Applications (DOA 2006), **LNCS** 4276, Springer Verlang, pp. 1633Montpellier, France, October, 2006.
- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, "Mobile Ambients in Aspect-Oriented Software Architectures", IFIP Working Conference on Software Engineering Techniques (SET 2006), **Springer Series in Computer Science,** Warsaw, Polond, October 17-20, 2006.
- Cristobal Costa, **Nour Ali,** Carlos Millan, Jose A. Carsi**,** "Transparent Mobility of Distributed Objects using .NET, 4th International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2006.
- Jennifer Pérez, **Nour Ali**, Jose Ángel Carsí, Isidro Ramos, "Designing Software Architectures with an Aspect-Oriented Architecture Description Language", 9th Symposium on the Component Based Software Engineering (CBSE), Springer Verlang **LNCS** 4063, pp. 123-138, ISSN: 0302-9743, ISBN: 3-540-35628-2, Vasteras, Sweden, June 29th-July 1st, 2006.
- **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí, " Introducing Ambient Calculus in Mobile Aspect-Oriented Software Architectures", Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA

2005), **IEEE Computer Society**, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper).

- **Nour Ali**, Isidro Ramos, Jose A. Carsí, “A Conceptual Model for Distributed Aspect-Oriented Software Architectures”, International Conference on Information Technology Coding and Computing (ITCC 2005), **IEEE Computer Society**, Las Vegas, NV, USA, 2005.
- Jennifer Pérez, **Nour Ali**, Cristobal Costa ,Isidro Ramos, Jose A. Carsí, “Executing Aspect-Oriented Software Architectures in .NET”, 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005.
- **Nour Ali**, Jennifer Pérez Isidro Ramos, Jose A. Carsí , “Aspect Reusability in Software Architectures”, 8th International conference of Software Reuse (ICSR), July, 2004 (poster)
- **Nour Ali**, Jennifer Perez, Isidro Ramos, “High Level Specification of Distributed and Mobile Information Systems”, Second International Symposium on Innovation in Information & Communication Technology (ISSICT 2004), April, Amman, Jordan,2004.
- **Nour Ali**, Josep Silva, Javier Jaén, Isidro Ramos, José Ángel Carsí, Jennifer Pérez, “Mobility and Replication Patterns in Aspect-oriented component Based Software Architectures”, 15th IASTED International Conference, PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS~PDCS 2003~, November 3-5, Marina del Rey, CA, USA, 2003.

INTERNATIONAL WORKSHOPS

- Jennifer Pérez, **Nour Ali**, Jose Ángel Carsí, Isidro Ramos, “Dynamic Evolution in Aspect-Oriented Architectural Models”, European Workshop on Software Architecture, (EWSA), **LNCS** 3527, Springer Verlang , Pisa, Italy, June 2005.

NATIONAL CONFERENCES

- **Nour Ali**, Jennifer Pérez, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, "Distributed Replication in Aspect-Oriented Software Architectures Using Ambients", XI Conference on Software Engineering and Databases (JISBD) (JISBD), Sitges, Barcelona, Octubre 3-6, 2006. (In Spanish)
- Cristóbal Costa, Jennifer Pérez, **Nour Ali**, Jose Angel Carsí, Isidro Ramos, "PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures"**,** X Conference on Software Engineering and Databases (JISBD), pp. 27-34, ISBN: 84-9732-434-X, Granada, September, 2005. (In Spanish)
- **Nour Ali**, Jose Angel Carsi, Isidro Ramos,"Analysis of a Distribution Dimension for PRISMA", IX Conference on Software Engineering and Databases (JISBD 2004), Málaga, 10-12 November 2004
- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, "Development of a Tele-Operation System using the PRISMA Approach", VIII Conference on Software Engineering and Databases (JISBD), pp. 411-420, ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)
- Josep Silva, **Nour Ali**, Jose Angel Carsi, Isidro Ramos, "The Distribution Aspect of PRISMA", VIII Conference on Software Engineering and Databases (JISBD), ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)

NATIONAL WORKSHOPS

- **Nour Ali**, Isidro Ramos, "Ambient-PRISMA: Ambients in Distributed and Mobile Aspect-Oriented Software Architectures", V Workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, 2006.
- **Nour Ali,** Jennifer Perez, Jose Angel Carsi, Isidro Ramos, "Mobility of Objects in PRISMA", III Worhshop DYNAMICA – DYNamic and Aspect-

Oriented Modeling for Integrated Component-based Architectures,, AlMagro, Ciudad Real, 2005.

- Mª Eugenia, **Nour Ali**, Jennifer Pérez, Isidro Ramos, Jose A. Carsí**, "**DIAGMED: An Architectural model for a Medical Diagnosis", IV workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 1-7, Archena, Murcia, November, 2005. (In Spanish)
- Rafael Cabedo, Jennifer Pérez, **Nour Ali**, Isidro Ramos, Jose A. Carsí, Aspect-Oriented C# Implementation of a Tele-Operated Robotic System, III Workshop on Aspect-Oriented Software Development (DSOA), X Conference on Software Engineering and Databases (JISBD), pp. 53-59, ISBN: 84-7723-670-4, Granada, September, 2005. (In Spanish)
- **Nour Ali**, Jennifer Pérez, Cristobal Costa, Jose A. Carsí, Isidro Ramos, "Implementation of the PRISMA Model in the .Net Platform", II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- **Nour Ali**, Jose Cercos, Isidro Ramos, Patricio Letelier, Jose Angel Carsi, "Distribution in PRISMA", workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- **Nour H. Ali**, Josep F. Silva, Javier Jaen, Isidro Ramos, Jose Á. Carsí, Jennifer Pérez, "Distribution Patterns in Aspect-Oriented Component-Based Software Architectures", IV Workshop Distributed Objects, Languages, Methods and Environments (DOLMEN), pp.74-80, Alicante, November, 2003.
- Jennifer Pérez, **Nour H. Ali**, Isidro Ramos, Jose A. Carsí, "PRISMA: Aspect-Oriented and Component-Based Software Architectures", Workshop on Aspect-Oriented Software Development (DSOA), Conference on Software Engineering and Databases (JISBD), Technical Report TR-20/2003 of the

Polytechnic School of the University of Extremadura, pp. 27-36, Alicante, November, 2003. (In Spanish)

TECHNICAL REPORTS

- Carlos Millán, **Nour Ali**, Isidro Ramos, "Development of Distributed and Mobile Applications from an Aspect-Oriented Architectural Model", Technical Report DSIC, Polytechnic University of Valencia, October, 2006. (In Spanish)
- Jennifer Pérez, **Nour Ali** , Jose A. Carsí, Isidro Ramos, "PRISMA Architecture of the Robot 4U4 Case Study", Technical Report DSIC-II/13/04, pp. 72, Polytechnic University of Valencia, 2004. (In Spanish)
- **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "A Compiler for the Automatic Generation of the Distribution Aspect to Distributed Applications", Technical Report DSIC-II/15/04, Polytechnic University of Valencia, October 2004. - 2004
- **Nour Ali**, Isidro Ramos, Jose Angel Carsi, "Distribution in an Aspect-Oriented Component Based Software Architecture through PRISMA", Technical Report DSIC-II/14/04, Polytechnic University of Valencia, October 2004.

12.2 Further Works

Ambient-PRISMA opens further works to be explored. These further works include improvements to the current work and new research areas.

Ambient-PRISMA has been applied to different case studies such as the Auction System (presented in this thesis), bank system, tele-operated robots, and the GSM handover protocol. Currently, we are working in a coordinated project with the Polytechnic University of Cartagena, Spain, in order to apply Ambient-PRISMA to a case study of the area of Wireless Sensor Actuator Networks (WSAN) for controlling

irrigation of agricultural cultivations. This case study will permit us to validate Ambient-PRISMA for other kinds of networks which are not LANs.

In the near future, the Case Tool proposed in this thesis has to be extended. New facilities have to be introduced in the Modelling tool such as including the specifications of predefined ambients, distribution aspects, ports, and replication aspects. This would decrease the effort and time of an analyst in specifying them. In addition, the Modelling Tool has to be extended in order to allow an analyst to graphically configure a software architecture with ambients. The code generation patterns have to be implemented in order to automatically generate the code from the Ambient-PRISMA graphical specifications.

The Case Tool can also be extended in order to incorporate model checking and simulation tools that use the formalization performed in Ambient-PRISMA. This will enable an analyst to verify specific properties such as liveness and deadlocks before a model is automatically generated. The Channel Ambient Machine can be used in order to perform this task.

Another open research area is to use Ambient-PRISMA for Mobile Computing where Sites can be mobile. A study has to be made in order to analyze whether the current Ambient-PRISMA metamodel and language can support Mobile Computing. In the current version of the Ambient-PRISMA metamodel and language there is no restriction for not allowing mobile sites. However, the Ambient-PRISMANET middleware does not support this kind of mobility.

In addition, mobile software systems strongly need non-functional requirements to be specified such as limitations for resources, network bandwidths and latencies. An interesting area is to extend Ambient-PRISMA with needed non-functional properties fir being specified along with the distribution and mobility aspects. These non-functional properties constrain mobility of architectural elements or their remote communication.

Finally and not least, an open research area is security of mobile applications. A study has to be made in order to provide an analyst for specifying security protocols using ambients. As a result, the Ambient-PRISMA metamodel, language, and Case Tool have to be extended in order to provide such primitives.

BIBLIOGRAPHY

[Aba99] Abadi, M., Gordon, A.D., "A calculus for cryptographic protocols: The spi calculus", Information Computation, Vol. 148, N° 1, Academic Press, pp. 1-70, 1999.

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

[Alb03] 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, pp. 121-126, Orlando, USA, 2003.

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

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

[Ama00] Amadio, R., "On Modelling Mobility", Theoretical Computer Science 240, pp. 147-176, 2000.

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

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

[Bau00] Baude, F., Caromel, D., Huet, F., Vayssière J., "Communicating Mobile Active Objects in Java", In Proc. of the 8th International Conference - High

Performance Computing Networking'2000 (HPCN Europe 2000), LNCS vol. 1823, pp. 633--643, 2000.

[Bau00a]Baumeister, H., Koch, N., Kosiuczenko, P., Wirsing, M., "Extending Activity Diagrams to Model Mobile Systems" In Proceedings of International Conference NetObjectDay (NODe) '02, LNCS 2591 Springer, pp. 278-293, Germany, Oct 2002.

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

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

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

[Ber96] Bernstein, P.A., "Middleware: a model for distributed system services", Communications of the ACM, Volume 39, Issue 2, pp. 86-98, February, 1996.

[Bet02] Bettini, L., Loreti, M., Pugleise, R.," An Infrastructure Language for Open Nets », Proc. Of the 2000 ACM Symposium on Applied Computing (SAC`02), Special Track on Coordination Models, Languages and Applications, ACM Press, pp. 373-377, 2002.

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

[Bou02] Boudol, G., Castellani, I., Germain, F., Lacoste, M., "Models of distribution and mobility : state of the art ", MIKADO Global Computing Project IST-2001-32222, RR/WP1/1, August, 2002.

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

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

[Bug01] Bugliesi, M., Castagna, G., Crafa, S., "Boxed ambients ", In TACS 2001, 4th International Symposium on Theoretical Aspects of Computer Science, LNCS 2215, pp. 38-63, Sendia, 2001.

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

[Car01] Cardelli, L., Gordon, A., "Principles of Wide Area Programming", Presentation at the 13th International School for Computer Science Researchers Lipari Island. Available at: lucacardelli.name/Slides/2001-07%20%20Lipari%20School%20-%2002%20Ambient%20Calculus.pdf

[Car97] Carzaniga, A., Picco, G.P., Vigna, G., "Designing Distributed Applications with Mobile Code Paradigms", Proc. of the 19th Conference on Software Engineering (ICSE'97), R. Taylor, ed., ACM Press, pp. 22–32, 1997.

[Car98a]Cardelli, L., Gordon, A. D. "Mobile Ambients", Foundations of Software Science and Computational Structures: First International Conference, FOSSACS '98, LNCS 1378, Springer, 1998, pp. 140-155.

[Car98b]Cardelli, L. "Abstractions for Mobile Computation", In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, volume 1603 of LNCS, Springer Verlag, pp. 51-94, 1998.

[Cas01] Castellani, I. "Process algebras with localities", Handbook of Process Algebra, J.A. Bergstra, A. Ponse, S.A. Smolka editors, Elsevier, 2001.

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

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

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

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

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

[Col05] Coulouris, G., Dollimore, J., and Kindberg, T., "Distributed Systems: Concepts and Design", Addison-Wesley, 4th Edition, June, 2005.

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

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

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

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

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

[Dur00] Duran, F.,Eker, S., Lincoln, P., Meseguer, J., "Principles of Mobile Maude", In: Kotz, D., Mattern, F.(eds.) .Agent Systems, Mobile Agents, and Applications 2000, LNCS 1882, Springer, pp. 73-85, Berlin, 2000.

[Emm00] Emmerich, W.,"Engineering Distributed Objects", John Wiley & sons, 2000.

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

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

[Fil00] Filipe, J. K., "Foundations of a Module Concept for Distributed Object Systems", PhD thesis, Technical University Braunschweig, 2000.

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

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

[Fou96] Fournet, C., Gonthier, G., "The reflective CHAM, and the join calculus", POPL 96, pp. 372-385, 1996.

[Fug98] Fuggetta, A., Picco, G. P., Vigna, G., "Understanding Code Mobility", IEEE Transactions on Software Engineering, Volume 24, Nº 5, pp. 342-361, 1998.

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

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

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

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

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

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

[Gra04] Grassi, V., Mirandola, R., Sabetta, A., "UML based Modeling and Performance Analysis of Mobile Systems", In Proceedings of the 7th International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems, ACM, pp. 95-104, Venice, Italy, October, 2004.

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

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

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

[Har02] Harrison, W.H., Ossher, H.L., Tarr, P.L., "Asymmetrically Vs. Symmetrically Organized Paradigms for Software Composition", IBM Research Report,

RC22685 (W0212-147) Thomas J. Watson Research Center, IBM, December 2002.

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

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

[Hen98] Hennessy, M., Riely, J., "Resource Access Control in Systems of Mobile Agents", In Proceedings HLCL 98, Electronic Notes in computer Science, Vol. 16, 1998.

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

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

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

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

[Kal99] Kalbarcyk Z. T., Iyer R. K., Bagchi S., Whisnant K., Chameleon, ":A software infrastructure for adaptive fault tolerance", IEEE Transactions on Parallel and Distributed Systems, vol. 10, pp. 560-579, June, 1999.

[Kav03] Kaveh, N., Emmerich, W., "Validating Distributed Object and Component Designs", Third International School on Formal Methods 2003, LNCS 2804, pp. 63-9, Bertinoro, Italy, September 2003.

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

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

[Kle01] Klein, C., Rausch, A., Sihling, M., Wen, Z., "Extension of the Unified Modeling Language for mobile agents", In Keng Siau and Terry Halpin, editors, Unifed Modeling Language: Systems Analysis, Design and Development Issues, chapter8, pp. 116-128, .Idea Publishing Group, 2001.

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

[Kos03] Kosiuczenko, P., "Sequence diagrams for mobility", Advanced Conceptual Modeling Techniques: ER 2002 Workshops, LNCS 2784, Tampere, Finland, October, 2003.

[Kot97] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S., Cybenko, G., "Agent TCL: Targeting the Needs of Mobile Computers", IEEE Internet Computing, Vol. 1, Nº 4, pp. 58-67, 1997.

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

[Kus05] Kusek , M., Jezic, G., "Modeling Agent Mobility with UML Sequence Diagram", Agentlink III – AOSE TFG2, Ljubljana, Slovenia, 2005.

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

[Lan98] Lange, D., B., Oshima, M., "Programming and Deploying Java Mobile Agents with Aglets", Addison-Wesley, 1998.

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

[Lev00] Levi, F., Sangiorgi, D., "Controlling interference in ambients", In Proceedings POPL 2000, ACM Press, pp. 352-364, 2000.

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

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

[Lop02a] Lopes, C. V., "Aspect-Oriented Programming: An historical perspective (what's in a name?) ", Technical Report, Institute for Software Research, University of California, Irvine, December 2002.

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

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

[Luc95] Luckham, D.C., Kenney, J.J., Augustin, L. M., Vera, J., Bryan, D., Mann, "Specification and Analysis of System Architecture Using Rapide", IEEE Transactions on Software Engineering, Vol. 21, Nº 4, pp. 336-355, April, 1995.

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

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

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

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

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

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

[Mcl03] McLean S., Naftel J., Williams K., “Microsoft .NET Remoting”, Microsoft Press, 2003.

[MDA07] Object Management Group. Model Driven Architecture Guide, 2003

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

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

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

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

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

[Mil89] Milner, R., Communication and Concurrency, Prentice Hall 1989.

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

[Mil99] Milner, R. "Communicating and Mobile Systems: the π-calculus", Cambridge University Press, 1999.

[Mil99a] Milojičić, D., Douglis, F., Wheeler, R., "Mobility: processes, computers, and agents", Addison-Wesley Professional, 1st Edition, April, 1999.

[MOC07] Calculi for Mobile Processes: http://lampwww.epfl.ch/mobility/

[Mor95] Moricon, M., Qian, X., Riemenschneider, R., " Correct architecture refinement", IEEE Transactions on Software Engineering, Special Issue on Software Architecture, Vol. 21, Nº 4, pp. 356-372, April, 1995.

[Mos87] Moss, J. E. B. "Log-based recovery for nested transactions", In Proceedings of the International Conference on Very Large Data Bases VLDB, pp. 427-432, 1987.

[Nic98] Nicola, R. D., Ferrari, G., Pugliese, R., "KLAIM: A Kernel Language for Agents Interaction and Mobility", IEEE Transactions on Software Engineering, Vol. 24. Nº 5, pp. 315-330, 1998.

[Nis04] Nishizawa, M., Shiba, S. and Tatsubori, M., "Remote pointcut - a language construct for distributed AOP", Proceedings of the 3rd international

conference on Aspect-oriented software development, ACM Press, pp. 7-15, 2004.

[Oli05] Oliveira, C., Wermelinger, M., Fiadeiro, J. L., Lopes, A., "Modelling the GSM handover protocol in CommUnity", Electronic Notes in Theoretical Computer Science, ISSN 1571-066, Vol. 141 Nº3, pp. 3-25, 2005.

[Oqu04] Oquendo, F., "π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures", ACM SIGSOFT Software Engineering Notes, Vol. 29 Nº3, May 2004.

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

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

[Pei97] Peine, H., Stolpmann, T., "The Architecture of the Ara Platform for Mobile Agents", In Proc. Of 1st International Workshop on Mobile Agents, LNCS 1219, pp. 50-61, 1997.

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

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

[Per06a]Pérez, J., Ali, N., Carsí, J.A., Ramos, I., "Designing Software Architectures with an Aspect-Oriented Software Architectures", The 9th International Symposium on Component-Based Software Engineering (CBSE), Lecture

Notes Computer Science, Springer Verlang, LNCS 4063, pp. 123-138, Västeras, Sweden, June-July, 2006.

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

[Per08] Perez, J., Ali, N., Carsi, J.A., Ramos, I., Alvarez, B., Sanchez, P., Pastor, J. A., "Integrating Aspects in Software Architectures: PRISMA Applied to Robotic Tele-operated Systems", Information and Software Technology, 2007. (accepted)

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

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

[Phi05] Philips, A. "Specifying and Implementing Secure Mobile Applications in the Channel Ambient System", PhD thesis, Imperial College London, October 2005.

[Pin02] Pinto, M., Fuentes, L., Fayad, M.E., Troya, "Separation of coordination in a dynamic aspect oriented framework", Proceedings of the 1st International Conference on Aspect-oriented Software Development, pp. 134-140, Enschede, The Netherlands, 2002.

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

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

[Pul99] Pulvermueller, E., Klaeren, H., Speck, A., "Aspects in Distributed Environments", Proceedings of the First International Symposium on Generative and Component-Based Software Engineering, (GCSE'99), Lecture Notes In Computer Science; Vol. 1799, p. 37 - 48 , Erfurt, Germany, September, 1999.

[Ram02] Rammer, I., "Advanced .NET Remoting", Apress, 2002.

[Ras02] Rashid, A.., Sawyer, P., Moreira, A., Araujo, J., "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", IEEE Joint Conference on Requirements Engineering, Essen, IEEE Computer Society, p. 199-202, Germany, September 2002.

[RMI07] Remote Method Invocation (RMI) Website of Sun Developer Network: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[Rom00] Roman, G., Picco, G., Murphy, A., "Software Engineering for Mobility: A Roadmap", International Conference on Software Engineering - Future of SE Track, pp. 241-258, Limerick, Ireland, 2000.

[Rom02] Roman, G. C., McCann, P.J.,"A Notation and Logic for Mobile Computing", Formal Methods in System Design, Volume 20, Issue 1, pp. 47-68, 2002.

[Rum04] Rumbaugh, J., Jacobson, I., Booch, G., "Unified Modeling Language Reference Manual", 2nd Edition, Addison Wesley Professional, July, 2004.

[Sal04] Saleh, K., El-Morr, C., "M-UML:an extension to UML for the modelling of mobile agent-based software systems", Journal of Information and Software Technology, Vol. 46, Nº 4, pp. 219-227, 2004.

[San00] Sandholm, T., Huai, Q., "Nomad: mobile agent system for an Internet-based auction house", Internet Computing, IEEE, Vol.4, Nº 2, pp.80-86, Mar/Apr, 2000.

[San00a]Sanchez, F., Heranandez, J., Murillo, J.M., Rodríguez, R., "Adaptability of object distribution protocols using the disguises model approach", In Proceedings of International Symposium on Distributed Objects and

Applications (DOA 00), IEEE Computer Society, pp. 315-322, Antwerp, Belgium, 2000.

[San93] D. Sangiorgi, D., “From π-calculus to Higher-Order π-calculus and back”, In Proc. TAPSOFT '93, LNCS 668, Springer Verlag, pp. 151-166, 1993.

[San98] Sanchez, F., Heranandez, J., Murillo, J.M., Pedraza, E., “Runtime adaptability of synchronization constraints in COOLs”, II ECOOP Workshop of Aspect Oriented Programming, 1998.

[Sar97] Sartipi, K., “A Survey on Software Architecture Domain”, PhD. Thesis, University of Waterloo, Ontario, Canada, February, 1997.

[Sch06] Schafer, C., “Modeling and Analyzing Mobile Software Architectures”, In Proceedings of the Third European Workshop on Software Architecture, (EWSA2006), Springer-Verlag, 2006.

[Sch06] Schmidt, D. C., “Model-Driven Engineering”, Computer, Vol. 39, Nº 2, IEEE Computer Society, February, 2006.

[Sco03] McLean S., Naftel J., Williams K., “Microsoft .NET Remoting”, Microsoft Press, 2003.

[Sha94a] Shaw, M., “Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status”, Proceedings of Workshop on Studies of Software Design, January, 1994.

[Sha94b] Shaw, M., Garlan, D., “Characteristics of Higher-Level Languages for Software Architecture”, Technical Report CMU-CS-94-210, SEI-94-TR-23, ESC-TR-94-023, Software Engineering Institute, Carnegie Mellon University, December, 1994.

[Sha96] Shaw, M. and Garlan, R., “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, April, 1996.

[Smi04] Smith, G., “A Formal Framework for Modelling and Analysing Mobile Systems”, Proceedings of the 27th Australasian conference on Computer science - Volume 26, pp. 193-202, Dunedin, New Zealand, 2004.

[Soa02a] Soares, S. and Borba, P., “PaDA: A pattern for distribution aspects”, In Second Latin American Conference on Pattern Languages Programming - SugarLoafPLoP 2002, Itaipava, RJ, Brazil. ICMC - University of São Paulo Magazine, p. 87-99, August, 2002.

[Soa02b] Soares, S., Laureano, E. and Borba, P. , “Implementing Distribution and Persistence Aspects with AspectJ”, In Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02, ACM Press, Seattle, WA, USA, p. 174-190, November, 2002.

[Soa04] Soares, S., “An Aspect-Oriented Implementation Method”, PhD thesis, Informatics Centre, Federal Universisty of Pernambuco, Recife, Brazil, October 2004.

[Sta04] Staneva, D., Dobreva, D., “MAPNET: a .NET-Based mobile agent platform”, In Proc. of the 5th International Conference on Computer systems and technologies, 2004.

[Sub05] Subotic, S. and Bishop, J., “Emergent Behaviour of Aspects in High Performance and Distributed Computing”, Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, ACM International Conference Proceeding Series; Vol. 150, White River, South Africa, p. 111-119, 2005.

[Suv03] Suvée, D., Vanderperren, W, Jonckers, V., “JAsCo.: An aspect-oriented approach tailored for component based software development”, In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, ACM Press, p. 21-29, 2003.

[Suz99] Suzuki, J. and Yamamoto, Y., "Extending UML with aspects: Aspect support in the design phase", In Int'l Workshop on Aspect-Oriented Programming (ECOOP), Lisbon, 1999.

[Szy00] Szyperski C., Booch G., Meyer B., "Beyond Objects", Software Development Magazine, March, 2000 (originally BrucePowel Douglass).

[Szy02] Szyperski, C., "Component Software: Beyond Object Oriented Programming", ACM Press and Addison Wesley, New York, USA, 2002.

[Tan07] Tanenbaum, A. S., Steen, M., "Distributed Systems: Principles and Paradigms", Prentice Hall, 2nd Edition, 2007.

[Tay95] Taylor, R., N., Medvidovic, N., Anderson, K. M., Whitehead, Jr., E., J., Robbins, J. E., "A Component-and Message-Based Architectural Style for GUI Software", Proceedings of the 17th international conference on Software Engineering, pp. 295-304, Seattle, Washington, United States, 1995.

[Til03] Tilevich, E., Urbanski, S., Smaragdakis, Y., Fleury, M., "Aspectizing Server-Side Distribution", In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03), IEEE Computer Society, p. 130-141, 2003.

[Tre05] Treaster M, "A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems", ACM Computing Research Repository (CoRR), (cs.DC/ 0501002), January, 2005

[UML07] The Unified Modeling Language Website, Object Management Group (OMG), http://www.uml.org/

[Vit99] Vitek, J., Castagna, G., "Seal : A Framework for Secure Mobile Computations ", In Workshop on Internet Programming Languages, 1999.

[War03] Warmer, J., Kleppe, A., "Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition", Addison Wesley Professional Pub, August 27, 2003

[Woj00] Wojciechowski, P. T., "Nomadic Pict: Language and infrastructure Design for Mobile Computation", PhD thesis, University of Cambridge, June 2000.

[Xu03] Xu, D., Yin, J., Deng, Y., Ding, J., "A Formal Architectural Model for Logical Agent Mobility", IEEE Transactions on Software Engineering, Vol. 29, N° 1, January 2003.

[Zac02] Zachriadis, S. Mascolo, C. Emmerich, W., "Exploiting logical mobility in mobile computing middleware", 22nd International Conference on Distributed Computing Systems, pp. 385 – 386, 2002.

APPENDIX A

AMBIENT-PRISMA AOADL SYNTAX

This appendix presents the syntax of the AOADL that has to be used for specifying Ambient-PRISMA software architectures.

A.1 ARCHITECTURAL MODEL

<architectural_model> ::= **Architectural_Model** <model_name>

<interface_block>

<aspect_block>

<component_block>

<connector_block>

[<system_block>]

[<virtual_ambient_block>]

<site_ambient_block>

[<group_ambient_block>]

<attachment_block>

End_Architectural_Model <model_name>**‘;**

A.2 INTERFACES

<iservice> ::= <service_name> **‘(‘** [<param_service_list>] **’)’ ‘;’**

<interface> ::= **Interface** <interface_name>

<iservice_list>

End_ Interface <interface_name> **;**

A.3 ASPECTS

<aspect> ::= <aspect_type> **Aspect** <aspect_name>

[**using** <interface_name_list>]

[<constant_attributes>]

[<variable_attributes>]

[<derived_attributes>]

<services>

[<preconditions>]

[<transactions>]

[<constraints>]

[<triggers>]

[<played_roles>]

<protocol>

End_Aspect <aspect_name>**';'**

<aspect_type> ::= **functional** | **coordination** | **distribution** | **replication** | **mobility**

A.4 Attributes

<constant_attributes> ::= **Constant** <var_cons_attribute_seq>

<variable_attributes> ::= **Variable** <var_cons_attribute_seq>

<var_cons_attribute> ::= <attribute_name>**':'** <data_type>[**','** **NOT NULL**]

A.5 Services

<services> ::= **Services**

begin **'('** [<param_service_list>]**')'**';'** [<valuations>]

< service_section_seq> [<valuations>]

end '(' ')'';'

<service_section> ::= <service> [**as** <service_name>]

<service> ::= <service_type> <service_name>'**('** [<param_service_list>]**')'**

<service_type> ::= **in** | **out** | **in/out**

<valuations> ::= **Valuations** <valuation_seq>

<valuation> ::= [**'{'** <condition> **'}'**] **'['** <action> **']'** <assignation_list>

<action> ::= <service_type> <service_name>**'('** [<parameter_name_list>]**')'**

<assignation> ::= <property> **':='** <formulae>

<property> ::= <attribute_name> | <parameter_name>

A.6 Preconditions

<preconditions> ::= **Preconditions** <precondition_seq>

<precondition> ::= <invocation> **if** **‘{‘** <condition> **‘}’**

<invocation> ::= <service_name>**‘(‘** [<parameter_name_list>]**‘)’**

A.7 Transactions

<transactions> ::= **Transactions** <transaction_seq>

<transaction> ::= <transaction_name>**‘(‘** [<param_service_list>] **‘)’‘:’**
<initial_transaction_process> <transaction_process_seq>

<initial_transaction_process> ::= <transaction_name>**‘::=’** <process> **‘→’**
<process_name>**‘;’**

<transaction_service> ::= [**‘{’** <condition> **‘}’**] <transaction_service_type>
<channel_kind> **‘(‘** [<parameter_name_list>]**’)’**

<transaction_service_type> ::= <trans_public_service> | <private_service>

<trans_public_service> ::= <interface_name>**’.’**<service_name>

<channel_kind> ::= <input_channel> | <output_channel>

<input_channel> ::= **‘?’**

<output_channel> ::= **‘!’**

<private_service> ::= <service_name>

A.8 Constraints

<constraints> ::= **Constraints** <constraint_seq>

<constraint> ::= **always** **‘{‘** <condition> **‘}’** | **sometimes** **‘{‘** <condition> **‘}’** |

[**‘{‘** <condition> **‘}’**] **next** **‘{‘** <condition> **‘}’** |

<condition_before> **since** <condition_after> |

<condition_before> **until** <condition_after> |

always <condition_before> **since** <condition_after> |

sometimes <condition_before> **since** <condition_after>

<condition_before> ::= <condition>

<condition_after> ::= <condition>

A.9 Triggers

<triggers> ::= **Triggers**

<trigger_def_block >

<trigger_def_block > ::= <service_name> <kind_channel>

‘(‘ [<parameter_name_list>]**’)’**

when **‘{’** formula **‘}’** **‘;’**

<kind_channel> ::= <input_channel> | <output_channel>

<input_channel> ::= **‘?’**

<output_channel> ::= **‘!’**

A.10 Played_Roles

<played_roles> ::= **Played_Roles** <played_roles_seq>

<played_role> ::= <played_role_name> **for** <interface_name> **'::='**

<process>**';'**

<played_role_service> ::= <service_name> <channel_kind>

'(' [<parameter_name_list>]')'

A.11 Protocol

<protocol> ::= **Protocol** <initial_process> <protocol_process_seq>

<initial_process> ::= <aspect_name> **'::='** <process> **'→'** < process_name>**';'**

<protocol_process> ::= <process_name> **'::='** <process>

[**'→'** <process_name>]

<protocol_service> ::= [**'{'** <condition> **'}'**] <protocol_service_type>

<channel_kind> **'('** [<parameter_name_list>]')'**':'**

<priority>

<process_service_type> ::= <public_service> | <private_service>

<public_service> ::= <played_role_name>**'.'**<service_name>

<priority> ::= <priority_value>

A.12 PORTS

<ports> ::= **Ports** <port_seq> **End_Ports';'**

<port> ::= <port_name>**':'** <interface_name>',' **Played_Role**
<aspect_name>**'.'**<played_role_name>

A.13 WEAVINGS

<weavings> ::= **Weavings** <weaving_seq> **End_Weavings';'**

<weaving>::= <aspect_name>**'.'**<service_name>**'('** [<parameter_name_list>]**')'**
<weaving_operator> <aspect_name>**'.'**<service_name>
'(' [<parameter_name_list>]**')'**

<weaving_operator> ::= **after** | **before** | **instead** | **afterif'('** <condition> **')'**
| **beforeif'('** <condition> **')'** | **insteadif'('** <condition> **')'**

A.14 VIRTUAL AMBIENTS

<Virtual> ::= **Virtual** <virtaul_name>
<aspects_importation_seq>
[<weavings>]
<ports>
<creation>
<destruction>
End_Virtual <virtual_name>**';'**

A.15 Site Ambients

<Site> ::= **Site** <site_name>
<aspects_importation_seq>
[<weavings>]
<ports>
<creation>
<destruction>
End_Site <site_name>**';'**

A.16 Group Ambients

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

<aspects_importation> ::= <concern> **Aspect Import** <aspect_name>
<creation> ::= **new**'(' [<param_service_list>]')' '**{**' <start_aspects_seq> '**}**'
<destruction> ::= destroy'(' ')' '**{**' <stop_aspects_seq> '**}**'
<start_aspects> ::= <aspect_name>'**.**'**begin**'(' [<parameter_name_list>]')'
<stop_aspects> ::= <aspect_name>**'.'end'(' ')'**

A.17 COMPONENT

<Component> ::= **Component_type** <component_name>

<aspects_importation_seq>

<weavings>

<ports>

<creation>

<destruction>

End_Component _type<component_name>**';'**

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

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

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

<start_aspects> ::= <aspect_name>**'.'begin'('** [<param_service_list>]**')'**

<stop_aspects> ::= <aspect_name>**'.'end'('** **')'**

A.18 CONNECTORS

<connector> ::= **Connector_type** <connector_name>

<aspects_importation_seq>

<weavings>

<ports>

<creation>

<destruction>

End_Connector _type <connector_name>**';'**

A.19 ATTACHMENTS

<attachments> ::= **Attachments** <attachment_seq> **End_Attachments';'**

<attachment> ::= <attachment_name> <component_name>**'.'**<port_name>

'(' <card_min_name> **'..'** <card_max_name **>')'** **'←→'**

'(' <card_min_name> **'..'** <card_max_name>**')'**

<connector_name>**'.'**<port_name>

|

<attachment_name> <ambient_name>**'.'**<port_name>

'(' <card_min_name> **'..'** <card_max_name **>')'** **'←→'**

'(' <card_min_name> **'..'** <card_max_name>**')'**

<connector_name>**'.'**<port_name>

<card_min> := **string** [2]

<card_max> ::= **string** [2]

A.20 CONFIGURATION

---

[2] They are usually natural numbers, but the letter 'n' is also used to specify "many instances" without specifying a specific number of instances

<architectural_model_configuration> ::=
Architectural_Model_Configuration <configuration_name> ‘=’
new <model_name> ‘**{**‘<loc_instantiation_seq>
[<virtualAmbient_instantiation_seq>]
<siteAmbient_instantiation_seq>
[<groupAmbient_instantiation_seq>]
<components_instantiation_seq>
[<systems_instantiation_seq>]
<connectors_instantiation_seq>
<attachments_instantiation_seq> ‘**}**’

<LOC_instantiation> ::= <loc_name> **‘=’ new loc‘(‘** [<param_value _list>] **‘)**

<virtualAmbient_instantiation> ::= <virtualAmbient_instance_name> **‘=’ new**
<virtualAmbient_name>**‘(‘**
[<param_value _list>] **‘)**

<siteAmbient_instantiation> ::= <siteAmbient_instance_name> **‘=’ new**
<siteAmbient_name>**‘(‘**
[<param_value _list>] **‘)**

<groupAmbient_instantiation> ::= <groupAmbient_instance_name> **‘=’ new**
<groupAmbient_name>**‘(‘**
[<param_value _list>] **‘)**

<ambient_Constraints> ::= <ambient_instance_name> **‘.’** <port_name>
‘<←X→’
<ambient_instance_name> **‘.’** <port_name>

<components_instantiation> ::= <component_instance_name> **‘=’ new**
<component_name>**‘(‘** [<param_value _list>] **‘)**

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

<attachments_instantiation> ::= <attachment_instance_name> **‘=’ new**

<attachment_name>**‘(‘**
<param_attachment_value>**‘)’**

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

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

< bindings_instantiation> ::= <binding_instance_name> **‘=’ new**
<binding_name> ‘(‘ <param_binding_value> ‘)’

APPENDIX B

AMBIENT-PRISMA SOFTWARE ARCHITECTURE OF THE AUCTION SYSTEM CASE STUDY

This appendix presents the specification of the *Auction System* using the Ambient-PRISMA AOADL.

B.1 Interfaces

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

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

End_Interface ICustProc
```

```
Interface ICustBidder

  biddingInf(input LotId:string, input SaleRoom: string,
             input SaleNum:string,input DateofAuction:date,
             input MaximumBid: double);
  changeMaximumBid(input NewMaximumBid: double);

  biddingStatus(input Quantity: double, input Situation: string);

End_Interface ICustBidder;
```

```
Interface IProcurAuction
in/out searchforLot(input Keywords:string, output LotId: string,
                    output Saleroom:string, output SaleNum:string,
                    output DateOfAuction:date, output Lotdescrip: string);
End_Interface IProcurAuction
```

```
Interface IBidderAuct
 bid(input LotId: string, input BidAmount: double);
 updateNextBid(output WinngBid: double, output NextBid: double);
 finishedAuction(output BidWon);
End_Interface IBidderAuct
```

```
Interface IAuctHseLot
getLotDescrip(output LotId: string, output Saleroom: string,
              output SaleNum: string, output DateOfAuction: date,
              output Lotdescrip: string);
End_Interface IAuctHseLot
```

B.2 Aspects

B.2.1 Distribution Aspect of the Customer component

```
Distribution Aspect CustDist using IMobility

 Attributes
  Constant
  location : Ambient NOT NULL;
 Services
  begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)] location := ParentAmbient;

  out move (input NewAmbient: Ambient );
  end;

 Played_Roles
  //There are two played roles with the same interface. This is because one
role is for the movement of the Procurement and the other for the Bidder.
  MOVEProc for IMobility ::= move!(input NewAmbient);
  MOVEBidder for IMobility::=move!(input NewAmbient);
 Protocol
 // The protocol specifies that the customer moves the procurement agent and
then it either moves the Bidder or ends.
  CUSTDIST:= begin(ParentAmbient)→ CUSTDIST1;
  CUSTDIST1:= MOVEProc.move!(NewAmbient)→CUSTDIST2;
  CUSTDIST2:= MOVEBidder.move!(NewAmbient)+end;

End_Distribution Aspect CustDist
```

B.2.2 Distribution Aspect of the Procurement component

```
Distribution Aspect ProcurDist using IMobility, ICapability

 Attributes
  Variable
  location: Ambient NOT NULL;
  nextAuctionSiteLoc: Ambient NOT NULL;
  counter: integer(0);
  Constant
   originLocation: Ambient NOT NULL;
 Services
  begin(input ParentAmbient: Ambient, input OriginLocation: Ambient,
        input NextAuctionSiteLoc: Ambient);
     Valuations
```

```
      [begin (ParentAmbient, OriginLocation, NextAuctionSiteLoc)]
          location := ParentAmbient, originLocation:= OriginLocation,
          nextAuctionSiteLoc := NextAuctionSiteLoc;
  initializeCounter()
     Valuations
      [initializeCounter ()]
          counter := 0;
  setCounter2()
     Valuations
      [setCounter2 ()]
          counter := 2;
  addCounter()
     Valuations
      [initializeCounter ()]
          counter := counter +1;


  //These services are concerned with mobility in order to interact with the
parent ambient.
  in changeLocation(input Name: ArchitecturalElement,
                    input NewLocation: Ambient)
        Valuations
          [changeLocation()] location:=NewLocation;
  out startMovement(input Name:ArchitecturalElement);
  out exit (Name: ArchitecturalElement);

  out finishMovement(input Name:ArchitecturalElement);

  out enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
  end();

//


 Preconditions
 //This precondition is concerned with the interaction with the ambient
      in  changeLocation(Name, NewLocation)
                                 if {self.Name==Name};

 Triggers
       initializeCounter?()  when {originLocation==location};
       move?(nextAuctionSiteLoc)   when {counter==1};
       move?(originLocation)   when {counter==2};

 Played_Roles
  CapParent for ICapability ::= startMovement!(Name)→
                                exit!(Name)→
                                enter!(Name, NewAmbient)→
                                finishMovement!(Name) →
                                changeLocation?(Name, NewLocation);
  CUSTMOVESPROC for IMobility::= move?(NewAmbient);

TRANSACTIONS in MOVE (NewAmbient: Ambient)
      move =  CapParent_startMovement!(self.Name)→
              MOVE1;
      MOVE1 = CapParent_exit!(self.Name)→ MOVE2;
      MOVE2 = CapParent_enter!(self.Name, NewAmbient)→MOVE3;
      MOVE3 = CapParent_finishMovement!(self.Name);
      MOVE4 = CapParent_changeLocation?(Name, NewLocation);
```

```
 Protocol
 // The mobility should be first executed by the customer and then it can
decide.
  PROCURDIST:= begin→ PROCURDIST1;
  PROCURDIST1:= MOVEProc.move!(NewAmbient)→ PROCURDIST2;
  PRDIST2:= move(NewAmbient)+end;

End_Distribution Aspect ProcurDist
```

B.2.3 Distribution Aspect of the Bidder component

```
Distribution Aspect BidderDist using IMobility, ICapability

 Attributes
  Variable
   location: Ambient NOT NULL;

  Constant
   originLocation: Ambient NOT NULL;
 Services

  begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)]
          location := ParentAmbient, originLocation:= ParentAmbient;


  //These services are concerned with mobility in order to interact with the
parent ambient.
  in changeLocation(input Name: ArchitecturalElement,
                    input NewLocation: Ambient)
        Valuations
          [changeLocation()] location:=NewLocation;
  out startMovement(input Name:ArchitecturalElement);
  out exit (Name: ArchitecturalElement);

  out finishMovement(input Name:ArchitecturalElement);

  out enter (input Name: ArchitecturalElement, input NewAmbient: Ambient);
  end();
 Preconditions
 //This precondition is concerned with the interaction with the ambient
     in  changeLocation(Name, NewLocation)
                               if {self.Name==Name};


 Played_Roles
  CapParent for ICapability ::= startMovement!(Name)→
                                exit!(Name)→
                                enter!(Name, NewAmbient)→
                                finishMovement!(Name) →
                                changeLocation?(Name, NewLocation);
```

```
  CUSTMOVESBIDDER for IMobility ::= move?(NewAmbient);

//
 TRANSACTIONS in MOVE (NewAmbient: Ambient)
      move =  CapParent_startMovement!(self.Name)→
              MOVE1;
      MOVE1 = CapParent_exit!(self.Name)→ MOVE2;
      MOVE2 = CapParent_enter!(self.Name, NewAmbient)→MOVE3;
      MOVE3 = CapParent_finishMovement!(self.Name);
      MOVE4 = CapParent_changeLocation?(Name, NewLocation);


 Protocol
 // The mobility should be first executed by the customer and then it can
moveback to the customer location by its own.
  BIDDERDIST:= begin→ BIDDERDIST1;
  BIDDERDIST1:=(CUSTMOVESBIDDER_MOVE?(NewAmbient)+ MOVE?(originLocation)
                + end)
                →
                BIDDERDIST1;

End_Distribution Aspect BidderDist
```

B.2.4 Distribution Aspect of the HostSite ambient

```
Distribution Aspect ADist using IGetLocation, IGetRoute
 Attributes
  Constant
  location : Ambient NOT NULL;
  physicalLocation: loc NOT NULL;
 Services
  begin(input ParentAmbient: Ambient, input PhysicalLocation: loc)
     Valuations
      [begin (ParentAmbient, PhysicalLocation)]
                          location := ParentAmbient,
                          physicalLocation:=PhysicalLocation;
  in/out getLocation(output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;
  in/out getRoute(input Name: ArchitecturalElement, output Route[]: Ambient)
    Valuations
         [in getRoute(input Name, output Route)] Route:= deriveRoute(Name);
end;

 Played_Roles
   INTROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);

 Protocol
  DIST:= begin(ParentAmbient, PhysicalLocation) → DIST1;
  DIST1:=(getLocation?(Location)→ getLocation!(Location))+
         (INTROUTE.getRoute?(Name, Route)→ INTROUTE.getRoute!(Name, Route))+
          end;

End_Distribution Aspect ADist
```

B.2.5 Distribution Aspect of the Inmobile elements

```
Distribution Aspect Dist
 Attributes
  Constant
  location : Ambient NOT NULL;
 Services
  begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)] location := ParentAmbient;
  end;

 Protocol

  DIST:= begin→ end;


End_Distribution Aspect Dist
```

B.2.6 Distribution Aspect of the Root ambient

```
Distribution Aspect RootDist using IGetLocation, IGetRoute

Attributes
  Constant
  location : Ambient (NULL);

Services
  begin()
     Valuations
      [begin ()] location := NULL;
   in/out getLocation( output Location: Ambient)
     Valuations
         [getLocation(output Location)] Location := location;
  in/out getRoute(input Name:ArchitecturalElement, output Route[]: Ambient)
    Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);
   end;

Played_Roles
   INTROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);

Protocol
  DIST:= begin→ DIST1;
  DIST1:=(getLocation?(Location)→ getLocation!(Location))+
         (INTROUTE.getRoute?(Name, Route)→ INTROUTEgetRoute!(Name, Route))+
          end;


End_Distribution Aspect RootDist
```

B.2.7 Distribution Aspect of MobileAmb

```
Distribution Aspect MobileAmbDist using IMobility, ICapability, IGetLocation,
                                  IGetRoute
 Attributes
  location : Ambient NOT NULL;


Services


   begin(input ParentAmbient: Ambient)
     Valuations
      [begin (ParentAmbient)]
                          location := ParentAmbient ;

   … …
   in/out getLocation( output Location: Ambient)
     Valuations
         [in getLocation(output Location)] Location := location;
   in/out getRoute(input Name:ArchitecturalElement, output Route[]: Ambient)
    Valuations
         [in getRoute(input Name, output Route)] Route := deriveRoute(Name);
    end;

Played_Roles
   INTROUTE for IGetRoute ::= getRoute?(Name, Route)→ getRoute!(Name, Route);

  … …
  TRANSACTIONS in MOVE (NewAmbient: Ambient)
      move =  CapParent_startMovement!(self.Name)→ MOVE1;
      MOVE1 = CapParent_exit!(self.Name,location)→ MOVE2;
      MOVE2 = CapParent_enter!(self.Name, NewAmbient)→MOVE3;
      MOVE3 = CapParent_finishMovement!(self.Name);
      MOVE4 = CapParent_changeLocation?(Name, NewLocation);

  Preconditions
  ….

End_Distribution Aspect MobileAmbDist
```

B.2.8 Functional Aspect of the Customer component

```
Functional Aspect CustFunct using ICustProc, ICustBidder

 Attributes
  Variables
    iLotId: string;
    iSaleroom:string,
    iSaleNum:string;
    iDateAuction: date;
    iLotNumber:string;
    interested: boolean;
    maximumBid: double;
    currentQuantity: double;
```

```
    currentsituation: string;

 Services
   begin();
   end();
   setInterested(input CustomerInterest:boolean)
      Valuations
        [setInterested(input CustomerInterest)]
                          interested:=CustomerInterested;
   in NewMaximumBid()
      NewMaximumBid::= setMaximumBid(input Maximumbid)→NewMaximumBid1;
      NewMaximumBid1::= out changeMaximumBid(input maximumBid);


   saveItem(input LotId:string, input SaleRoom:string,
            input SaleNum:string, input DateOfAuction:date,
            input Lotdescrip:string)
       Valuations
         {interested==true}[saveItem(LotId, SaleRoom, SaleNum, DateOfAuction,
                                    LotNumber)]
                     iLotId:=LotId,iSaleRoom:=Saleroom, iSaleNum:=SaleNum,
                     iDateOfAuction:=DateAuction, iLotdescrip:=Lotdescrip;

   setMaximumBid(input MaximumBid:double)
       Valuations
         [setMaximimBid(MaximumBid)] maximumBid:=MaximumBid;



 //this service is used to create the attachments of the Bidder and the
Procurement at runtime. So the customer connects the agents at runtime with
the auctions before he sends them. He creates the attachments between the
Procurement and the two auctions and then he creates the attachments between
the bidder and the auction it has been decided to bid in.
  out createAttachment( InstanceName:string, Ports: Port: string,
                        InstanceName:string, Port: string);

  //Customer and bidder
  out biddingInf(input LotId:string, input SaleRoom: string,
                 input SaleNum:string,input DateofAuction:date,
                 input MaximumBid: double);

  out changeMaximumBid(input maximumBid);
  in/out biddingStatus( input Quantity: double, input Situation: string)
     Valuations
         [in biddingStatus( input Quantity: double, input Situation: string)]
         currentQuantity:=Quantity, currentSituation: Situation;

//Customer receives from proc the lots available and it answers if it is
//interested
  Transactions
   in/out NOTIFYPRODINTEREST(input LotId: string, input Saleroom:string,
                             input SaleNum:string, input DateOfAuction:date,
                             input Lotdescrip: string,
                             output Interested: boolean);

     notifyProdInterest = setInterested?(CustomerInterested)→ SAVEITEM;
     SAVEITEM = saveItem?(SaleRoom, SaleNum, DateOfAuction,LotNumber);
         Valuations
```

```
          [in NOTIFYPRODINTEREST(LotId, SaleRoom, SaleNum, DateOfAuction,
                                 Lotdescrip, Interested)]
            Interested =interested;



  triggers
       setMaximumBid(input MaximumBid)  when interested==true;

  Played_Roles
   CUSTPROC for ICustProc ::= notifyProdInterest?(LotId, Saleroom, SaleNum,
                                                  DateOfAuction, Lotdescrip,
                                                  Interested)
                             →
                             notifyProdInterest!(LotId, Saleroom, SaleNum,
                                                  DateOfAuction, Lotdescrip,
                                                  Interested);

   CUSTBIDDER for ICustBidder ::= biddingInf!(input iSaleRoom,
                                               input iSaleNum,
                                               input iDateofAuction,
                                               input iLotNumber,
                                               input  maximumBid)→
                                  (changeMaximumBid!(input MaximumBid)+
                                    biddingStatus!(input Quantity,
                                                   input Situation)+
                                    biddingStatus?(input Quantity,
                                                   input Situation));

Protocol
       ……

End_Functional Aspect CustFunct
```

B.2.9 Functional Aspect of the Procurement component

```
Functional Aspect ProcurFunct using IProcurAuction, ICustProc

 Attributes
  Variables
    keywords: string NOT NULL;
    limitDate: date NOT NULL;
    lotId: string;
    saleroom: string;
    saleNum:string;
    dateOfAuction: date;
    lotdescrip: string;
    keepSearching: boolean;
    finishedSearching:boolean;

 Services
  begin(input Keywords: string, LimitDate: date)
      Valuations
       [begin(input Keywords, input LimitDate)]
                                 keywords:= Keywords, limitDate:= LimitDate;
```

```
  setKeepSearchingToTrue()
      Valuations
       [setKeepSearchingToTrue()] keepSearching:=true;
  finishedSearchingWithoutResults()
      Valuations
       [finishedSearchingWithoutResults()] finishedSearching:=true;

//Procurement and the Auction
  in/out searchforLot(input Keywords:string, output LotId: string,
                      output Saleroom:string, output SaleNum:string,
                      output DateOfAuction:date, output Lotdescrip: string)

      Valuations
       {DateOfAuction<= limitDate}[in searchforlot(Keywords, LotId
                                                   SaleRoom, SaleNum,
                                                   DateOfAuction,Lotdescrip)]
          lotId:= LotId, saleroom := Saleroom, saleNum:= SaleNum,
          dateOfAuction:= DateOfauction, lotdescrip:=Lotdescrip;

//Procurement and the Customer
     in/out notifyProdInterest(input LotId, input Saleroom:string,
                               input SaleNum:string,
                               input DateOfAuction:date,
                               input Lotdescrip:string,
                               output Interested: boolean);
     Valuations
       {Interested== false}[in notifyProdInterest(LotId, Saleroom, SaleNum,
                                                  DateOfAuction, Lotdescrip,
                                                  Interested)]
                                          keepSearching:=true;
       {Interested== true}[in notifyProdInterest(LotId, Saleroom, SaleNum,
                                                  DateOfAuction, Lotdescrip,
                                                  Interested)]
                                          keepSearching:=false;

   end();
  triggers
     notifyProdInterest!(LotId, Saleroom,SaleNum, DateOfAuction, Lotdescrip,
                         Interested)
                      when {DateOfAuction<=limitDate};

     searchforlot!(input Keywords, output Saleroom, output SaleNum,
                  output DateOfAuction, output Lotdescrip)
                      when  keepSearching==true;


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

```
                                                Lotdescrip);

Protocol

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

End_Functional Aspect ProcurFunct
```

B.2.10 Functional Aspect of the Bidder component

```
Functional Aspect BidderFunct using ICustBidder, IBidderAuct

 Attributes
  Variable
    lotId: string;
    saleRoom: string;
    saleNum:string
    lotDateofAuction: date;
    lotMaximumBid: double;
    currentBiddingAmount: double;
    biddingSituation: string {"bid with you", "bid against you ", "LOST",
                              "LOT SOLD TO YOU"};
    currentDate: date;

 Services
  begin()
    Valuations
        [begin()] biddingSituation:= "bid against you";
  //Customer and bidder
  in biddingInf(input LotId:string, input SaleRoom: string,
                input SaleNum:string,input DateofAuction:date,
                input MaximumBid: double)
     Valuations
         [in biddingInf(LotId, SaleRoom, SaleNum, DateofAuction,
                        MaximumBid)]
         lotId:= LotId, saleRoom:=SaleRoom, saleNum:=SaleNum,
```

```
        lotDateofAuction:=DateofAuction, lotMaximumBid:= MaximumBid,
        biddingSituation:= "bid against you";

  in changeMaximumBid(input NewMaximumBid:double)
     Valuations
        [in changeMaximumBid(NewMaximumBid)]
          lotMaximumBid:= NewMaximumBid;
 //the bidder informs the customer the current bid and if it with it, against
it, lot sold to it, finished.
  out biddingStatus(input currentBiddingAmount, input biddingSituation);


// bidder and auction

   in finishedAuction(output BidWon)
     Valuations
       {BidWon==currentBiddingAmount} [in finishedAuction(BidWon)]
             biddingSituation:= "WON";
       {BidWon <currentBiddingAmount} [in finishedAuction(BidWon)]
             biddingSituation:= "LOST";


   in/out bid(input LotId: string, input BidAmount: double)
      Valuations
             [in bid(LotId, BidAmount)]
                 LotId:= lotId, BidAmount := currentBiddingAmount;


    in updateNextBid(input WinngBid:double, output NextBid:double)
      Valuations
                 [in updateNextBid(WinngBid, NextBid)]
                                         NextBid:= nextBid;

                 [in incrementNextBid(NextBid)]
                                         NextBid:= nextBid;

 Triggers
      bid!(input currentBiddingAmount, output Situation) when
        { biddingSituation=="bid against you"
          and currentBiddingAmount<=lotMaximumBid};


      biddingStatus!(currentBiddingAmount, biddingSituation) when
        {biddingSituation=="LOST" or biddingSituation=="LOT SOLD TO YOU"};

 Played_Roles

CUSTBIDDER for ICustBidder ::= biddingInf?(input iSaleRoom, input iSaleNum,
                                           input iDateofAuction,
                                           input iLotNumber,
                                           input maximumBid)→
                               (changeMaximumBid?(input MaximumBid)+
                                biddingStatus!(input Quantity,
                                               input Situation));
BIDDERAUCT for IBidderAuct ::= bid!(input currentBiddingAmount, output
Situation )→
                               (changeStatus?(input NewBid)
```

```
                                + finishedAuction(input WonBid, input
Situation));


End_Functional Aspect BidderFunct
```

B.2.11 Functional Aspect of the AuctionHouse system

```
Functional Aspect AuctHseFunct using IProcurAuction, IAuctHseLot

 Attributes
  Variables
    lotId: string;
    saleroom: string;
    saleNum: string;
    dateOfAuction: date;
    lotdescrip: string;


 Services
  begin();
  end();

 in/out getLotDescrip(output LotId: string, output Saleroom: string,
                     output SaleNum: string, output DateOfAuction: date,
                     output Lotdescrip: string)
      Valuations
          [in getLotDescrip(LotId, Saleroom, SaleNum, DateOfAuction,
                            Lotdescrip)]
               lotId:= LotId, saleroom:= Saleroom, saleNum:=SaleNum,
               dateOfAuction:= DateOfAuction, lotdescrip:= Lotdescrip;

//Procurement and the Auction. The auction returns a lot with which are
compatible with the keywords in the lot description. It checks the keywords
and returns a product or another

 in/out searchforLot(input Keywords:string, output LotId: string,
                     output Saleroom:string, output SaleNum:string,
                     output DateOfAuction:date, output Lotdescrip: string)

      Valuations
        {(Keywords==lotDescrip) and (Saleroom!=NULL)}
                       [in searchforlot(Keywords, LotId
                                        SaleRoom, SaleNum,
                                        DateOfAuction, Lotdescrip)]
          LotId:= lotId, Saleroom := saleroom, SaleNum:= saleNum,
          DateOfAuction:= dateOfAuction, Lotdescrip:=lotdescrip;


 Played_Roles

  PROCURAUCT for IProcurAuction ::= searchforlot?(Keywords, LotId,
                                                  SaleRoom, SaleNum,
                                                  DateOfAuction, Lotdescrip)
```

```
                                    →
                                   searchforlot! (Keywords, LotId,
                                                 SaleRoom, SaleNum,
                                                 DateOfAuction, Lotdescrip);

  LOTAUCT for IAuctHseLot::= getLotDescrip!(LotId, Saleroom, SaleNum,
                                           DateOfAuction, Lotdescrip)
                              →
                             getLotDescrip?(LotId, Saleroom, SaleNum,
                                           DateOfAuction, Lotdescrip);

 Protocol

  AUCTHSEFUNCT::= begin()→ AUCTTHSEFUNCT1;
  AUCTTHSEFUNCT1::= (PROCURACUCT_ searchforlot?(Keywords, LotId, SaleRoom,
                                              SaleNum, DateOfAuction,
                                              Lotdescrip)→ AUCTTHSEFUNCT2)
                     + end();
  AUCTTHSEFUNCT2::= LOTAUCT_ getLotDescrip!(LotId, Saleroom, SaleNum,
                                            DateOfAuction, Lotdescrip)
                     →
                     LOTAUCT_getLotDescrip?(LotId, Saleroom, SaleNum,
                                           DateOfAuction, Lotdescrip)
                     →
                     PROCURACUCT_ searchforlot?(Keywords, LotId, SaleRoom,
                                              SaleNum, DateOfAuction,
                                              Lotdescrip)
                     → AUCTTHSEFUNCT1;

End_Functional Aspect AuctHseFunct
```

B.2.12 Functional Aspect of the LotOnAuction component

```
Functional Aspect lotFunct using IAuctHseLot

 Attributes
  Variables
    lotId: string NOT NULL;
    saleroom: string NOT NULL;
    saleNum: string NOT NULL;
    dateOfAuction: date NOT NULL;
    currentDate: date;
    lotdescrip: string NOT NULL;
    biddingAmount: double NOT NULL;
    endDate:date NOT NULL;
    nextBid: double;
    status: string {"sold", "onAuction", "pending"}

 Services
  begin(input LotId: string, input Saleroom: string, input SaleNum: string,
        input DateOfAuction: date, input Lotdescrip: string,
        input StbiddingAmount: double, input EndDate: date)
      Valuations
       [begin(LotId, Saleroom, SaleNum, DateOfAuction, Lotdescrip,
              StbiddingAmount, EndDate)]
```

```
        lotId:= LotId, saleroom:= Saleroom, saleNum:=SaleNum,
        dateOfAuction:=DateOfAuction, lotdescrip:=Lotdescrip,
        StbiddingAmount:= biddingAmount, status:="pending",endDate:=EndDate ;

  beginAuction()
     Valuations
      [startAuction()]status:= "onAuction";
  endAuction()
     Valuations
      [endAuction()]status:= "sold";


// AuctionHouse and the lot.

   in/out getLotDescrip(output LotId: string, output Saleroom: string,
                        output SaleNum: string, output DateOfAuction: date,
                        output Lotdescrip: string)
      Valuations
      {status!= "sold"}
          [in getLotDescrip(LotId, Saleroom, SaleNum, DateOfAuction,
                            Lotdescrip)]
               LotId:= lotId, Saleroom:= saleroom, SaleNum:=saleNum,
               DateOfAuction:= dateOfAuction, Lotdescrip:=lotdescrip;

//Bidder and the lot.

   in/out finishedAuction(output BidWon)
     Valuations

               BidWon:= biddingAmount ;

    in bid(input LotId: string, input BidAmount: double)
      Valuations
       {(LotId== lotId) and (BidAmount==nextBid) and status=="onAuction"}
             [in bid(LotId, BidAmount, Situation)]
                 biddingAmount:= BidAmount;

    in/out updateNextBid(output WinngBid: double, output NextBid: double)
     Valuations
      {(biddingAmount>=100) and (biddingAmount<200)}
                    [in updateNextBid(WinngBid, NextBid)]
                                 nextBid:= biddingAmount+10,
                                 NextBid:= nextBid, WinngBid:= biddingAmount;
      {(biddingAmount>=200) and (biddingAmount<300)}
                    [in updateNextBid(WinngBid, NextBid)]
                                 nextBid:= biddingAmount+20,
                                 NextBid:= nextBid, WinngBid:= biddingAmount;
 Transactions
  in ENDAUCTION()
   endAuction = finishedAuction?(BidWon)→ finishedAuction!(BidWon);
     Valuations
      [in endAuction()]status:= "sold";

 trigger
   updateNextBid?(NextBid) when biddingAmount==nextBid and
                                 status="onAuction";
   beginAuction?() when currentDate== dateOfAuction;
   endAuction?() when currentDate== endDate;
```

```
 Played_Roles

  BIDDERAUCT for IBidderAuct ::=(bid?(LotId, BidAmount)→
                                bid!(LotId, BidAmount)) →
                                updateNextBid(WinngBid, NextBid) +
                                finishedAuction!(BidWon);

  AUCTLOT for IAuctHseLot::=getLotDescrip?(LotId, Saleroom, SaleNum,
                                       DateOfAuction, Lotdescrip)
                               →
                              getLotDescrip!(LotId, Saleroom, SaleNum,
                                       DateOfAuction, Lotdescrip);


 Protocol
 //

End_Functional Aspect lotFunct
```

```
Coordination Aspect AuctionCnctrCoor using IAuct

  Services
    begin;
        in/out bid(input ProductID: string, input Quantity: real);
       in/out lookforProduct(input Description: string,
                             output ProductID: string ,
                             output CurrentBid:  real);
       in/out notifyBuy(Buy : bool);
       in/out registerForPayment( input UserName: string,
                                  input Password: string,
                                  input ProductID: string,
                                  output Register: bool);
    end;

  Played_Role
    AUCTCUST for IAuct =
    (
      (bid?( ProductID,Quantity )
       →
       (bid!( ProductID,Quantity ) )
      +
      (lookforProduct?( Description, ProductID, CurrentBid )
       →
       (lookforProduct!( Description, ProductID, CurrentBid) )
      +
      (notifyBuy?( Buy )
       →
       (notifyBuy!( Buy ) )
      +
      (registerForPayment?( UserName, Password, ProductID, Register)
       →
       (registerForPayment!( UserName, Password, ProductID, Register) )
    );

        CUSTAUCT for IAuct =
    (
      (bid?( ProductID,Quantity )
```

```
       →
       (bid!( ProductID,Quantity ) )
      +
      (notifyBuy?( Buy )
       →
       (notifyBuy!( Buy ) )
    );


  Protocol
    AUCTIONCNCTRCOOR = begin.COORD;
    COORD =
    (
      (AUCTCUST.bid?( ProductID,Quantity ) →
       CUSTAUCT.bid!( ProductID,Quantity ) ).COOR
    +
      (AUCTCUST.lookforProduct?( Description, ProductID, CurrentBid ) →
       CUSTAUCT.lookforProduct!( Description, ProductID, CurrentBid)).COOR
     +
      (AUCTCUST.notifyBuy?( Buy ) )→
       CUSTAUCT.notifyBuy!( Buy )).COOR
     +
      (AUCTCUST.registerForPayment?( UserName, Password, ProductID, Register)
→
       CUSTAUCT.registerForPayment!( UserName, Password, ProductID, Register)
).COOR
     +
      (CUSTAUCT.bid?( ProductID,Quantity ) )→
       AUCTCUST.bid!( ProductID,Quantity )  ).COOR
     +
      (CUSTAUCT.notifyBuy?( Buy ) )→
       AUCTCUST.notifyBuy!( Buy ) ).COOR
    +
        end
    );

End_Coordination Aspect AuctionCnctrCoor;
```

```
Coordination Aspect AgentCustCnctrCoor using ICustAgent, IMobility

  Services
    begin;
     in/out purchase(changeProductDescription(input ProductDescr: string);
     in/out changeMaxBidQuantity(input NewMaxBidQuantity: real);
     in/out move(NewAmbient: string);
    end;

  Played_Role
    CUSTAGENT for ICustAgent =
    (
      (changeProductDescription?( ProductDescr)
       →
       changeProductDescription!( ProductDescr) )
      +
      (changeMaxBidQuantity?( NewMaxBidQuantity)
       →
       changeMaxBidQuantity!( NewMaxBidQuantity) )
```

```
    );
    AGENTCUST for ICustAgent =
    (
      (changeProductDescription?( ProductDescr)
       →
       changeProductDescription!( ProductDescr) )
      +
      (changeMaxBidQuantity?( NewMaxBidQuantity)
       →
       changeMaxBidQuantity!( NewMaxBidQuantity) )
    );
        AGENTCUSTMOB for IMobility =
    (
      move?( NewAmbient)
       →
       move!( NewAmbient)
    );
    CUSTAGENTMOB for IMobility =
    (
      move?( NewAmbient)
       →
       move!( NewAmbient)
    );

  Protocol
    AGENTCUSTCNCTRCOOR = begin.COORD;
    COORD =
    (
      (CUSTAGENT.changeProductDescription?( ProductDescr ) →
        AGENTCUST.changeProductDescription!( ProductDescr ) ).COOR
    +
      (AGENTCUST.changeProductDescription?( ProductDescr ) →
        CUSTAGENT.changeProductDescription!( ProductDescr ) ).COOR
    +
      (AGENTCUSTMOB.move?( NewAmbient) →
        CUSTAGENTMOB.move!( NewAmbient) ).COOR
    +
       (CUSTAGENTMOB.move?( NewAmbient) →
        AGENTCUSTMOB.move!( NewAmbient) ).COOR
    +
        end
    );

End_Coordination Aspect AgentCustCnctrCoor;
```

B.3 Ambients

```
Ambient_Site type HostSite
 Import Mobility Aspect MobilityAspect;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;
 Weavings
     ADist.getLocation(Location)instead
                                MobilityAspect.getParent(Parent);
 End_Weavings
```

```
 Ports
     InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
     ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
     EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
     InServicesPort: ICall Played_Role ACoordination.INTERIOR;
     InRoutePort: IGetRoute Played_Role ADist.INTROUTE;
 End_Ports

End Ambient_Site type HostSite;
```

```
Ambient_Virtual type Root

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

 Weavings
     RootDist.getLocation(Location)instead
                                MobilityAspect.getParent(Parent);
 End_Weavings


 Ports
     InCapabilitiesPort: ICapability Played_Role MobilityAspect.INTERIOR;
     ECapabilitiesPort: ICapability Played_Role MobilityAspect.EXTERIOR;
     EServicesPort: ICall Played_Role ACoordination.EXTERIOR;
     InServicesPort: ICall Played_Role ACoordination.INTERIOR;
     InRoutePort: IGetRoute Played_Role RootDist.INTROUTE;

 End_Ports
End Ambient_Virtual type Root;
```

B.4 Components

B.4.1 Customer

```
Component_type Customer

 Import Distribution Aspect CustDist;
 Import Functional Aspect CustFunct;
 Weavings
  CustDist.move(NewAmbient) after
                         CustFunct.biddingInf(LotId, SaleRoom, SaleNum,
                                              DateofAuction,MaximumBid);
 End_Weavings

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

```
    CUSTPROCPort: ICustProc Played_Role CustFunct.CUSTPROC;
    CUSTBIDDERPORT: ICustBidder Played_Role CustFunct.CUSTBIDDER

 End_Ports
   new(input ParentAmbient: string)
       {
        CustDist. begin(ParentAmbient);
        CustFunct.begin();
        }
    destroy()
        {
             CustDist.end();
             CustFunct.end();
         }

End Component Customer;
```

B.4.2 Procurement

```
Component_type Procurement
 Import Distribution Aspect ProcurDist;
 Import Functional Aspect ProcurFunct;
 Weavings
   //This weaving is for moving to the customer location
   ProcurDist.setCounter2()
        afterif(Interested==true)
                       ProcurFunct.in notifyProdInterest(input Saleroom,
                                                          input SaleNum,
                                                          input DateOfAuction,
                                                          input Lotdescrip,
                                                          output Interested);

   ProcurDist.addcounter() after
                       ProcurFunct.finishedSearchingWithoutResults();

   ProcurFunct. setKeepSearchingToTrue() after
                       ProcurDist.move(nextAuctionSiteLoc);
 End_Weavings

 Ports
    DCapPort: ICapability Played_Role ProcurDist. CapParent;
    DMovingPort: IMobility Played_Role ProcDist.CUSTMOVESPROC;
    CUSTPROCPort: ICustProc Played_Role ProcurFunct. CUSTPROC;
    PROCURAUCTPort: IProcurAuction Played_Role ProcurFunct. PROCURAUCT;
 End_Ports


End Component_type Procurement;
```

B.4.3 Bidder

```
Component_type Bidder
  Import Distribution Aspect BidderDist;
  Import Functional Aspect BidderFunct;

  Weavings
   //This weaving is for moving to the customer location after the auction
     has finished.

   BidderDist.move(CustLocation) after BidderFunct. finishedAuction(BidWon)
  End_Weavings

  Ports
     DCapPort: ICapability Played_Role BidderDist. CapParent;
     DMovingPort: IMobility Played_Role BidderDist.CUSTMOVESBIDDER;
     CustBidderPort: ICustBidder Played_Role BidderFunct.CUSTBIDDER;
     BidderAuctPort: IBidderAuct Played_Role BidderFunct. BIDDERAUCT;
  End_Ports


  new(input ParentAmbient: string)
       {
        BidderFunct. begin();
        BidderDist.begin(ParentAmbient);
        }
  destroy()
        {
             BidderDist.end();
             BidderFunct.end();
         }
End Component_type Bidder;
```

B.4.4 LotOnAuction

```
Component_type LotOnAuction
 Import Distribution Aspect Dist;
 Import Functional Aspect lotFunct;
 Ports
     LotAuctPort: IAuctHseLot Played_Role lotFunct.AUCTLOT;
     BidderAuctPort: IBidderAuct Played_Role lotFunct.BIDDERAUCT;
 End_Ports
 new(input LotId: string, input Saleroom: string, input SaleNum: string,
     input DateOfAuction: date, input Lotdescrip: string,
     input StbiddingAmount: double, input EndDate: date)


       {

     lotFunct. begin(input LotId: string, input Saleroom: string,
                    input SaleNum: string, input DateOfAuction: date,
```

```
                    input Lotdescrip: string, input StbiddingAmount: double,
                    input EndDate: date);

      CustFunct.begin();
        }
        {
             lotFunct.end();
             CustFunct.end();
         }

End Component_type LotOnAuction;
```

B.4.5 WrappAspSystem

```
Component_type WrappAspSystem
 Import Distribution Aspect Dist;
 Import Functional Aspect AuctHseFunct;
 Ports
     LotAuctPort: IAuctHseLot Played_Role AuctHseFunct.LOTAUCT;
     BidderAuctPort: IProcurAuction Played_Role AuctHseFunct.PROCURAUCT;
 End_Ports
 new()
       {
        AuctHseFunct. begin();
        }
     destroy{
             AuctHseFunct.end();
         }

End Component_type WrappAspSystem;
```

B.5 Systems

B.5.1 AuctionHouse

```
System_type AuctionHouse
 Import Distribution Aspect Dist;
 Import Functional Aspect AuctHseFunct;
 Ports
     ProcurAuctPort: IProcurAuction Played_Role AuctHseFunct.PROCURAUCT;
     SysBidderPort: IBidderAuct Played_Role lotFunct.BIDDERAUCT;
```

```
 End_Ports

Import Architectural Elements WrappAspSystem, LotOnAuction, CnctLotAuction;


Attachments
 AttchAWrapCnct:
          CnctLotAuction.CoorWPort(1,1) <-> WrappAspSystem.LotAuctPort(1,1);
 AttchCnctLot:
          CnctLotAuction.CoorLPort (1,n) <-> LotOnAuction.LotAuctPort (1,1);
End_Attachements;

Bindings
  BndWrap: ProcurAuctPort(1,1)<--> WrappAspSystem.ProcurAuctPort(1,1);
  BndL: SysBidderPort(1,n)<--> LotOnAuction.LotAuctPort(1,1)
End_Bindings;


 new(input num_LotOnAuction: integer,input num_CnctLotAuction: integer,
      input num_AttchAWrapCnct: integer, input num_AttchCnctLot: integer,
      input num_ BndWrap: integer, input num_ BndL: integer)
  {
    new WrappAspSys();
    new LotOnAuction(input LotId: string, input Saleroom: string,
                     input SaleNum: string, input DateOfAuction: date,
                     input Lotdescrip: string,
                     input StbiddingAmount: double, input EndDate: date);

     new CnctLotAuction();
     new AttchAWrapCnct(input ArgCnctName: string, input ArgCnctPort: string,
                      input ArgCompName: string, input ArgCompPort: string);
     new AttchCnctLot (input ArgCnctName: string, input ArgCnctPort: string,
                      input ArgCompName: string, input ArgCompPort: string);
     new BndWrap (input ArgSysPort: integer, input ArgAEName: integer,
                 input ArgAEPort: integer);
    new BndL  (input ArgSysPort: integer, input ArgAEName: integer,
                 input ArgAEPort: integer);
 }
End System_type AuctionHouse;
```

B.6 Connectors

B.6.1 AgentCustCnct

```
Connector_type AgentCustCnct
 Import Distribution Aspect Dist;
 Import Coordination Aspect CustCnctrCoor;
 Ports
    CustPortProcur: ICustProc;
    CustPortProcurMo: IMobility;
    CustPortBidder: ICustBidder
    CustPortBidderMo: IMobility;´
    ProcurPortCust: ICustProc;
    ProcurPortCustMo: IMobility;
    BidderPortCust: ICustBidder
    BidderPortCustMo: IMobility;
 End_Ports

End Connector_type AgentCustCnct;
```

B.6.2 AuctionHouseCnct

```
Connector_type AuctionHouseCnct
 Import Distribution Aspect Dist;
 Import Coordination Aspect AuctCnctrCoor;
 Ports
    AuctPortCust: ICustAuct;
    CustPortAuct: ICustAuct;
    CnctAuctPortBidder: IBidderAuct;
    BidderPortAuct: IBidderAuct;
    ProcurPortAuct: IProcurAuction;
    AuctPortProcur: IProcurAuction;
 End_Ports

End Connector_type AuctionHouseCnct;
```

B.6.3 CnctLotAuction

B.7 Attachments

```
Attachments
 AttchAuctCnct:
     AuctionHouseCnct.CnctAuctPortBidder(1,1)<-> AuctionHouse.BidderAuctPort(1,1);
 AttchCustAuc:
     Customer.CUSTAUCTPort(1,n)<-> AuctionHouseCnct.CustPortAuct(1,n);
 ……
End_Attachements
```

B.8 Initial Configuration

```
Architectural_Model_Configuration MobileAgentsAuctionConf =

New MobileAgentsAuction
{
       IP1 = new loc(ip.of.host.1);
       IP2 = new loc(ip.of.host.2);
       IP3 = new loc(ip.of.host.3);

       ROOT = new Root() ;

       ClientSite = new HostSite(ROOT, IP1);
       AuctionSite = new HostSite(ROOT, IP2);
       LondonSaleRoomSite = new HostSite(Root, IP3);

       ClientSite.* ←X→ LondonSaleRoomSite.*;

       AuctionHouse1 = new AuctionHouse (1,1,1,1,1,1,1, "AuctionSite")
                       { WrapAspSys = new WrapAspSys();
                         LotOnAuction1 =new (1235, London, 1678, 17/10/2008,
                                             "Painting", 500, 18/10/2008);
                         CnctLotAuction = new CnctLotAuction();
                         …..
                        }
       Customer1 = new Customer("ClientSite");
       Procurement1 = new Procurement ("ClientSite");
       Bidder1 = new Bidder("painting", "3 Jul", "ClientSite");
       AgentCustCnct1 = new AgentCustCnct("ClientSite");
       AuctionHouseCnct1 = new AuctionHouseCnct("AuctionSite");
       AttchAuct1Cnct = new AttchAuctCnct (AuctionHouseCnct1,
                                           CnctAuctPortBidder,
                                           AuctionHouse1, BidderAuctPort);


       AttchCust1Auc1 = new AttchCustAuc (Customer1, CUSTAUCTPort,
                                     AuctionHouseCnct1, CustPortAuct);
       AttchProc1Auc1 = new AttchCustAuc (Procurement1, PROCURAUCTPort,
                                     AuctionHouseCnct1, ProcurPortAuct);


        …….
       ATT1 = new CustCnctrAtt(CUST, AgentsPort,
                               CustAgentPort, ACNCTR);
       ATT2 = new CustCnctrAtt(CUST, MovePort,
                               AccountOp_port, ACNCTR);
       ATT3 = new CustAuctCnctrAtt(CUST, AuctionPort,
                               CustAgentMobPort, ACNCTR);
```

```
      ATT4 = new CnctrAgntsAtt(ACCNCTR, AgentCustPort,
                              EDistServicesPort, AAP);
      ATT5 = new CnctrAgntsAtt(ACCNCTR, AgentCustMobPort,
                              EMobilityPort, AAP);
      ATT6 = new AgntsHostAtt(AAP, DCapabilitiesPort,
                              ICapabilitiesPort, CLIENTSITE);
      ATT8 = new PurAucAtt(PUR, PURAUCPort,
                              AuctionPortCUST, AUCNCTR);
      ATT9 = new PurCnctrAtt(PUR, COLLPURPort,
                              PurchaserPort, ACNCTR);
      ATT10 = new ColCnctrAtt(COL, CollPurPort,
                              CollectorPort, ACNCTR);
      ATT11 = new ColAuctAtt(COL, COLLAUCTPort,
                              AuctionPortCUST, AUCNCTR);
      ATT12 = new ColCustAtt(COL, CustAgentPort,
                              CustAgentPort, ACCNCTR);
      ATT13 = new AuctCnctrAtt(AUCT, AuctionPort,
                              AuctionPortAUCT, ACNCTR);

};
```

ACRONYMS

AC	Ambient Calculus
ACID	Atomicity, Consistency, Isolation, and Durability
ACT	Abstract Communication Type
ADL	Architecture Description Language
AOADL	Aspect-Oriented Architecture Description Language
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
CASE	Computer Aides Software Engineering
CBSD	Component-Based Software Development
CCS	Calculus of Communicating Systems
ChAC	Channel Ambient Calculus
COOL	Coordination Aspect Language
CORBA	Common Object Request Broker Architecture
DAOF	Dynamic Aspect-Oriented middleware Framework
DNS	Domain Name Server
DSL	Domain-Specific Language
EJBs	Enterprise Java Beans
GOTECH	General Object to EJB Conversion Helper
HO π-calculus	Higher-Order π-calculus
ID	Identifier
JVM	Java Virtual Machine
JVMDI	Java Virtual Machine Debugger Interface
KLAIM	Kernel Language for Agents Interaction and Mobility
LAN	Local Area Network
LSN	Log Sequence Number
MAS	Multi-Agent System

MDA	Model Driven Architecture
MDE	Model Driven Engineering
OCL	Object Constraint Language
OMG	Object Management Group
OOSD	Object-Oriented Software Development
PaDA	Pattern for Distribution Aspects
PARC	Palo Alto Research Centre
PC	Personal Computer
PDA	Personal Digital Assistant
prevLSN	Previous Log Sequence Number
PROSE	PROgrammable extenSions of sErvices
RIDL	Remote Interface Aspect Language
RMI	Remote Method Invocation
SoC	Separation of Concerns
TO	Trace Operation
UML	Unified Modelling Langugae
UMLAUT	UML All pUrpose Transformer
URL	Unified Resource Locator
WAN	Wide Area Network
WSAN	Wireless Sensor Actuator Network

INDEX

#

π_1-calculus · 46
π-ADL · 67
π-calculus · 45, 61, 67, 246, 247

A

After · 377
Afterif · 377
ambient · 204, 205, 212, 217, 218, 222, 224, 227, 228, 229, 230, 231, 232, 237, 240, 245, 246, 248, 278, 279, 280, 281, 282, 283, 297, 307, 308, 309, 310, 311, 326, 329, 330, 332, 333, 335, 336, 343, 344
Ambient · 46, 109, 163, 165, 171, 175, 194, 197, 210, 236, 239, 247, 249, 261, 311, 337
  Group Ambient · 191, 200, 312
  Root Ambient · 168, 200, 273, 314
  Site Ambient · 191, 199, 313, 332
  Virtual Ambient · 191, 200, 314, 326, 328
Ambient Calculus · 29, 44, 45, 46, 105, 109, 110, 163, 179, 246, 247, 308, 343, 409
Ambient-PRISMANET
  heartbeat · 287
  heartbeats · 287
  Leader · 287
  Notifier · 287
Architectural Model · 371
architectural style · 57
architectural views · 58
Architecture Description Language (ADL) · 53, 54, 55, 56, 57, 59, 62, 64, 66, 67, 72, 74, 75, 76, 364
Aspect · 372
AspectJ · 79, 81, 85, 86
  advice · 81
  aspects · 80
  Join points · 81
  pointcut · 81
Aspect-Oriented Programming · 78, 79
Aspect-Oriented Software Development · 78, 79
Attachment · 264, 266, 267
attachments · 57, 194
AWED · 100

B

Before · 377
Beforeif · 377
Binding · 269
black box · 55
BNF PRISMA AOADL
  Instance · 240, 381
Boxed Ambients · 47, 247

C

C2Sadel · 62
  C2 · 62
  connectors · 62
  context reflective interfaces · 62
Calculus of Communicating Systems · 45

Ch

Channel Ambient Calculus · 47, 245, 247

C

component · 55, 56, 164
Composition Filters · 81, 95
Con Moto · 68
  components · 68
configuration · 149
Configurations · 54, 57
connections · 57
connector · 165
Connector · 193

Crosscutting Concern · 79

D

D Language · 92
  Aspect Weaver · 93
  D aspects · 92, 93
  DJ · 93
Dπ · 46
Darwin · 60, 61
disguises model · 98
  distribution aspect · 99
  replication aspect · 99
DJCutter · 89
Domain-Specific Languages Tool
  Domain Model tool · 153
  Model Designer Tool · 154
Domain-Specific Languages Tools · 151

G

GOTECH · 89
Group ambient
  Mobility · 297

H

HO π-calculus · 45

I

Instead · 377
Insteadif · 377
Interface · 371

J

JAsCo · 82, 100

K

KLAIM · 47

M

metamodel · 192, 193
middleware · 40, 71, 82, 152, 155, 283
mobility · 63, 65, 66, 69, 70, 88, 92, 95
Mobility · 296, 320, 325
  Group ambient · 297
  physical mobility · 246
  processes · 41
  Strong mobility · 43
  virtual mobility · 246
  Weak mobility · 43
  Weak Mobility · 297
Modal Logic of Actions · 118
Model Driven Architecture · 151
Model-Driven Engineering · 151

N

Nomadic π-calculus · 46, 47, 247

O

OASIS · 118
Object-Oriented Software Development · 78, 410

P

PaDA · 86, 89
Played_Role · 376
polyadic π-calculus · 68
Polyadic Pi-Calculus · 118
Port · 376
  DCapPort · 255
  EServicesPort · 167
  InServicesPort · 183, 327
ports · 55
Ports
  DCapPort · 185, 233, 255, 266, 323, 336
  DRoutePort · 186, 336
  InCapabilities · 266, 322
  RCapPort · 186, 336
Precondition · 374
PRISMA · 113
  AOADL · 113, 118
  configuration · 149

metamodel · 113, 118
PRISMA AOADL · 149
PROSE · 82, 410
Protocol · 376

R

replication · 92, 99
reuse · 54, 56
reused · 116

S

Scattering of Concerns · 79
Separation of Concerns · 56, 77, 410
Software Crises · 77
SPI-calculus · 45

T

tangled code · 82
topologies · 57
Transaction · 218, 374
Transaction Manager · 299
Transactions
Heterogeneous · 298
Homogeneous · 298
Log · 303
nested transaction · 299
transaction manager · 305
transactional context · 298

U

UML · 44, 45

W

Weaving
Dynamic Weaving · 81
Static Weaving · 81
Weaving
Operator
After · 136
Weaving
Operator
Before · 136
Weaving
Operator
Instead · 136
Weaving
Operator
Afterif · 136
Weaving
Operator
Beforeif · 136
Weaving
Operator
Insteadif · 136
Weaving · 377
Weaving
Operator · 377
Weaving
Operator
After · 377
Weaving
Operator
Before · 377
Weaving
Operator
Instead · 377
Weaving
Operator
Afterif · 377
Weaving
Operator
Beforeif · 377
Weaving
Operator
Insteadif · 377