

## **Requirements and Definition of AO Middleware Reference Architecture**

### **ABSTRACT**

The development of distributed systems software is becoming increasingly complex. Aspect-oriented middleware (AOM) platforms aim to ease this complexity by allowing otherwise crosscutting concerns to be modularised, promoting understandability, evolvability and reusability. However, AOM platforms are becoming more widespread, diverse and also more complex. In this report we present a *reference architecture for aspect-oriented middleware*. The reference architecture provides a common reference point for the creation of key concerns which can then be mapped to a particular AOM platform. Additionally, the reference architecture facilitates the creation of new AOM platforms themselves.

Document ID:	AOSD-Europe-ULANC-15
Deliverable	
/Milestone No:	D21
Work-package No:	WP9
Type:	Deliverable
Status:	COMPLETE
Version:	1.0
Date:	29 October 2005
Author(s):	Neil Loughran and Geoff Coulson (ULANC) Lionel Seinturier and Renaud Pawlak (INRIA) Eddy Truyen, Frans Sanen, Maarten Bynens and Wouter Joosen (KUL) Andrew Jackson, Siobhan Clarke and Neil Hatton (TCD) Monica Pinto, Lidia Fuentes and Mercedes Amor (UMA) Tal Cohen (Technion) Adrian Colyer (IBM) Christa Schwanninger (Siemens)

## Table of Contents

1.	Introduction .....	6
1.1	Structure of Document .....	6
2.	Scope and Goals .....	7
2.1	Levels .....	7
3.	Level 1 .....	8
3.1	Motivation and Overview .....	8
3.2	Architectural Elements of Component Microkernel .....	8
3.3	Microkernel .....	10
3.3.1	What is a microkernel? .....	10
3.3.2	Aspect-oriented microkernel .....	10
3.4	Abstract Pointcut Language .....	11
3.4.1	Design goals .....	11
3.4.2	Joinpoint types .....	11
3.4.2.1	Transport level joinpoint types .....	12
3.4.2.2	Application level joinpoint types .....	12
3.4.2.3	Relationship between the two levels .....	13
3.4.3	Extending the joinpoint model .....	14
3.4.4	Domain definitions .....	14
3.4.4.1	Static properties .....	15
3.4.4.2	Dynamic properties .....	15
3.4.5	Pointcut expressions model .....	16
3.4.5.1	Component-based pointcut expressions .....	16
3.4.5.2	Joinpoint selectors .....	17
3.4.5.3	Embedding language level pointcut expressions .....	17
4.	Level 2 .....	19
4.1	Motivation and Overview .....	19
4.2	Philosophy of Building Middleware in terms of Aspects .....	20
4.3	Distribution .....	20
4.3.1	Distribution as an aspect .....	20
4.3.1.1	Deployment concern .....	20
4.3.1.2	Lookup concern .....	21
4.3.1.3	Communication concern .....	22
4.3.2	Mapping to the reference architecture .....	22
4.3.2.1	Mapping to the component model .....	23
4.3.2.2	Mapping to the APL .....	24
4.3.2.3	Mapping to the microkernel .....	25
4.3.2.4	Discussion .....	25
4.4	Example Concern Mappings .....	26
4.4.1	Mobility .....	26
4.4.1.1	Mobility as an aspect .....	26
4.4.1.2	Mapping to the component model .....	27
4.4.1.3	Mapping to the APL .....	31
4.4.1.4	Mapping to the microkernel .....	33
4.4.1.5	Discussion .....	33
4.4.2	Persistence .....	33
4.4.2.1	Persistence as an aspect .....	33
4.4.2.2	Mapping to the component model .....	34
4.4.2.3	Mapping to the APL .....	38

4.4.2.4	Mapping to the microkernel .....	40
4.4.2.5	Discussion.....	40
4.4.3	Security .....	40
4.4.3.1	Security as an aspect.....	40
4.4.3.2	Mapping to the component model.....	41
4.4.3.3	Mapping to the APL .....	42
4.4.3.4	Mapping to the microkernel .....	44
4.4.3.5	Discussion.....	44
4.4.4	Coordination .....	45
4.4.4.1	Coordination as an aspect.....	45
4.4.4.2	Mapping to the component model.....	50
4.4.4.3	Mapping to the APL .....	52
4.4.4.4	Mapping to the microkernel .....	53
4.4.4.5	Discussion.....	53
4.4.5	Transactions.....	54
4.4.5.1	Transaction management as an aspect.....	56
4.4.5.2	Mapping to the component model.....	58
4.4.5.3	Mapping to the APL .....	59
4.4.5.4	Discussion.....	59
4.5	Summary.....	60
4.5.1	Issues w.r.t. the mapping to the component model .....	60
4.5.2	Issues w.r.t. the mapping to the abstract pointcut language .....	61
5.	Conclusion.....	63

## List of Figures

Figure 1: Conceptual view of Level 1 .....	8
Figure 2: The component model for the AO middleware reference architecture.....	9
Figure 3: The transport level joinpoints. ....	12
Figure 4: Two-way invocation. ....	13
Figure 5: One-way invocation. ....	14
Figure 6: Initial state of the client.....	24
Figure 7: Binding to a bank component. ....	24
Figure 8: Mobility concern and its required interfaces. ....	27
Figure 9: Mobility provided interfaces.....	28
Figure 10: Mobility concern aspects. ....	28
Figure 11: Network roaming. ....	29
Figure 12: Network roaming & QOS assurances concerns composed.....	31
Figure 13: High level component model for persistence.....	35
Figure 14: Core persistence functionality component.....	35
Figure 15: SQL translation component. ....	36
Figure 16: Object-relational mapping component.....	37
Figure 17: Complete persistence component framework.....	37
Figure 18: Design of the PIM system.....	41
Figure 19: Access control model.....	41
Figure 20: Access control components.....	42
Figure 21: Bindings in our PIM system. ....	43
Figure 22 : Part of the design of an auction system. ....	46
Figure 23: Buyer – seller interaction protocol.....	47
Figure 24: Compound bind in RM-ODP. ....	49
Figure 25: Example 1 of the provided interface of the auction coordination aspect.....	50
Figure 26: Component model for the coordination concerns as an Adapter.....	50
Figure 27: Component model for Adapter-Binding Coordination.....	51
Figure 28: Example 2 of the provided interface of the publish and subscribe coordination aspect.....	51
Figure 29: Component model for the publish and subscribe coordination model.....	51
Figure 30. Component model for transaction manager .....	59

## List of Tables

Table 1: Transaction support advice.....	58
--	----

# 1. Introduction

Distributed software systems are increasing in their complexity, making their development and evolution problematic. To aid in the construction of large scale distributed systems, many software developers have adopted *middleware approaches*. Middleware facilitates the development of distributed software systems by accommodating heterogeneity, hiding distribution details, and providing a set of common and domain specific services.

However, in [27] we, the applications lab part of AOSD-Europe [9], surveyed a range of traditional and aspect-oriented middleware (AOM) platforms and found that traditional middleware itself was becoming increasingly complex. We concluded that AOM could alleviate much of this complexity by allowing crosscutting concerns such as persistence, security, etc. to be modularised hence facilitating evolution and reducing the risk of architectural erosion [28]. We also suggested that AOM was in danger of becoming overly complex due to the number of AOM platforms and, therefore, the next natural step was to work towards a *reference architecture for AOM*.

The reference architecture is layered into three levels and serves to provide a common list of proto-ontological concepts (Level 1) which facilitate the development of new AOM platforms as well as provide a high level abstraction for the mapping of key concerns (provided by Level 2). A future document will detail mapping these concerns to specific AOM implementations (Level 3).

This document describes the work done by the applications lab of AOSD-Europe towards this goal.

## 1.1 Structure of Document

The next section describes the scope and goals of the reference architecture. Section 3 introduces Level 1, i.e. the component model and the abstract pointcut language (APL). Section 4 introduces Level 2, the key concerns mapped to the APL and component model. Finally, Section 5 concludes the document.

## **2. Scope and Goals**

A key objective of the applications lab, in regard to the AOM, is the integration of each network partner's research in the development of a prototypical AOM platform to support embedded, pervasive and ambient systems. To achieve middleware support for such domains, the AOM platform is intended to facilitate aspect composition in heterogeneous and dynamic environments.

A primary aim of AOM research integration is to provide a conceptual vocabulary for AO middleware systems. Such a vocabulary is intended to allow a standardised means for analysing, comparing and discussing AOM platforms.

AOM research integration is intended to establish, embody and encourage best practice and patterns in the design of AOM systems. Through integration, it is expected that the best practice in AOM will emerge. The AOM reference architecture derived from this integration could then act as a guideline for following best practice in AOM.

A goal of AOM research integration is to form the basis of middleware implementation toolkits. The generality and commonality expected in the reference architecture may support the development of toolkits for AOM implementations.

A final goal in the applications lab is the identification and specification of concerns that need to be considered in the development of AOM. Thus far, we have identified security, persistence, mobility, context-awareness, transactions, coordination and distribution as key concerns to be supported in an AOM platform [16].

### **2.1 Levels**

Three levels have been identified in the description of the AOM reference architecture. Level 1 defines general AOM concepts, principles and patterns. Level 2 defines mappings of key concerns taken from [16] to Level 1 constructs. Level 3 is defined as a configuration of Level 2 with distributed concerns underlying in a specific application or set of applications and this will be detailed in a future document.

### 3. Level 1

#### 3.1 Motivation and Overview

The primary aim of Level 1 is to provide a list of proto-ontological concepts that will form the basis for our generic aspect-oriented (AO) middleware reference architecture. The reference architecture facilitates the building of AO middleware platforms and additionally provides a generic abstraction between distribution specific (e.g. CORBA [1], .NET [2], etc.) and platform specific AO models (e.g. JBoss AOP [3], JAC [4], DAOP [5], etc.).

To this end, Level 1 defines the *component microkernel* (i.e. our architectural component model with a set of operating system primitives), the extensible *abstract pointcut language* (APL) and binding patterns that are utilised.

The component microkernel approach is utilised as the concepts are well established within industry and are therefore the natural choice. The generic pointcut language provides a way for aspects to be written in a language independent manner so that concerns can be mapped to a specific AO platform of choice.

Figure 1 illustrates a view of Level 1 and the key features it provides. ‘Trap doors’ provide mappings from the APL to distribution and AO specific implementations, as well as extensions to the APL itself.

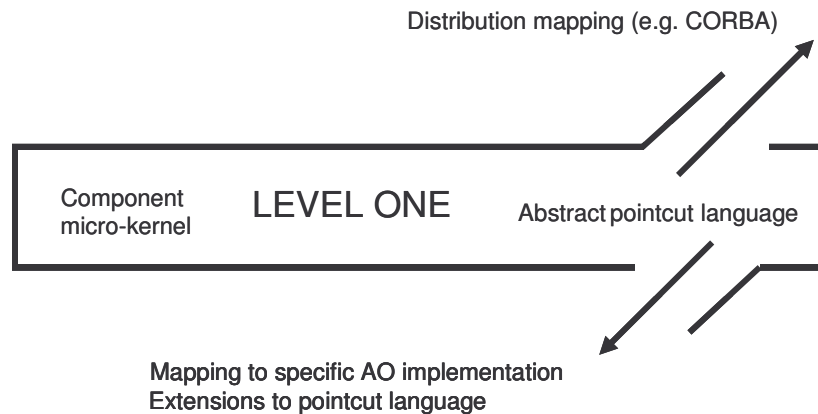


Figure 1: Conceptual view of Level 1

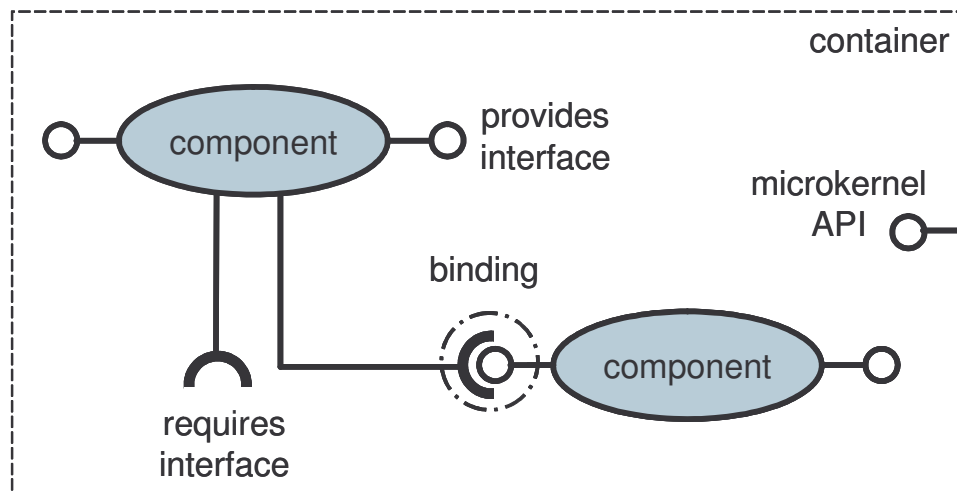
#### 3.2 Architectural Elements of Component Microkernel

The architectural elements in our reference architecture are based upon the standard notion of a *component-based* programming model which is now widely established [1] [2]. The key characteristics of the component-based approach to building software are:

- high degree of abstraction in the design, implementation, deployment and management of software;
- facilitation of configuration and run-time reconfiguration;
- software reuse.

Szyperski [6] states that software components are “*units of composition with contractually-specified interfaces and explicit context dependencies only... that can be deployed independently and subject to composition by third parties.*” Component models may restrict composition to *build-time* or *load-time* but many such as CORBA and .NET for example, now support component composition at *run-time*.

The primitive architectural elements in our reference architecture will therefore be *containers*, *components*, *interfaces* and *bindings* shown in Figure 2. Aspects and advice are roles taken by components and interface operations respectively.



**Figure 2: The component model for the AO middleware reference architecture**

*Containers* represent containing entities for components, interfaces, bindings and aspects.

*Components* are encapsulated units of functionality that interact with other components via interfaces and their connectors. Components may be deployed into a container at run-time and their microkernel functions can be requested from any component within the container. Components are purely black-box and may have multiple interfaces which can be useful in embodying separations of concern.

*Interfaces* are used to bind components together and are expressed in terms of sets of operation signatures and associated data-types. *Required* interfaces make explicit the dependencies of a particular component on another component. *Provided* interfaces state the provided functionality of a particular component.

*Bindings* are explicit associations between components via their respective ‘required’ and ‘provided’ interfaces.

The Reference Model for Open Distributed Processing (RM-ODP) [7] interaction styles are adopted on component interfaces. That is, request/reply, streaming and signals (i.e. one way message interactions). To this we will then add a set of canonical joinpoints in order to realise the aspect capable component model.

## 3.3 Microkernel

### 3.3.1 What is a microkernel?

The term microkernel is most frequently used with respect to operating systems [8]. It describes a form of operating system design in which the amount of code that must execute in privileged kernel mode is kept to an absolute minimum (the “micro” kernel), and other system services are built on top of the kernel and run in user mode. Besides modularity improvements, a key advantage of this design is that (user mode) modules can be much more easily replaced without having to recompile or reinstall the entire system. The role of the microkernel can be described in general as a layer between the hardware and the major system components.

The design of large middleware systems has much in common with operating systems and also benefits from a kernel-based approach. Middleware designed along these lines is typically written as a set of services defined as modules that plug-in to a small core. Such designs are also known as microkernel designs [10].

The JBoss Application Server [3] is an example of a middleware system with a microkernel architecture. JBoss uses a JMX server [11] as the microkernel itself, and configures middleware services as MBeans [12] that are registered with the kernel.

### 3.3.2 Aspect-oriented microkernel

The aspect-oriented microkernel is designed to act as a core on top of which highly modular systems can be built. The design goals for the microkernel are as follows:

- small in size - we are interested in building systems that scale down as well as up;
- support loading and unloading of component implementations at runtime;
- support reconfiguration of loaded components without having to shut down and restart the system;
- provide a very simple plug-in development model for writing components that run in the microkernel;
- maintain very loose coupling between components and between components and the kernel. Amongst other reasons, this makes it easy to develop and test components in isolation;
- support aspect-based modularity for feature implementation. In particular, the kernel must support:
  1. the opt-in model of service development, whereby one component extends one or more abstract aspects to tailor a service to its own
  2. the co-opted model of service development, whereby one component dictates the binding of functionality to other components;
- support the development of autonomic systems, for which I use the definition that an autonomic system has self-configuring, self-healing, self-optimising, and self-protecting properties [13];
- provide a common mapping between multiple AO middleware implementations.

The main operations in our microkernel are therefore:

- LOAD – Loads a specified component into the container;
- UNLOAD – Unloads a specified component from the container;
- START – Services the requests send to the component;
- STOP – Stops servicing the requests send to the component;
- BIND – Creates a binding between components via specified ‘provides’ and ‘requires’ interfaces;
- UNBIND – Unbinds specified bound components;
- INSTANTIATE – Instantiates a specified component;
- DESTRUCT – Destroys the component instance, freeing up memory.

### 3.4 Abstract Pointcut Language

Within the context of the AO middleware reference architecture, the abstract pointcut language (APL) is designed to be a framework for defining pointcuts. Following the standard definitions of AOP [32], a pointcut picks out a set of joinpoints, and a joinpoint is a point in a program execution flow.

#### 3.4.1 Design goals

The main design goal for the APL is to provide a generic framework for expressing pointcuts. The syntax of the APL is abstract in the sense that we want the APL to be language neutral. We want to design a language that is generic enough and can be mapped onto several different concrete pointcut languages.

The second design goal for the APL is to be a pointcut language for component-based applications. The scope of the APL is restricted to the architectural elements defined in our component model (see Section 3.2). In particular, we do not make any assumptions on the way components are implemented (a class, a set of functions, etc.). Therefore, the joinpoints related to this implementation level are out of the scope of the APL.

A third design goal for the APL is to be able to define *distributed pointcuts*, i.e. pointcuts that span across several distributed hosts or containers.

#### 3.4.2 Joinpoint types

The main purpose of the APL is to express **quantifications** over a set of component related joinpoints. We consider a model of middleware technology with two levels: a transport level and an application level. Two sets of joinpoints are defined, one in each level. These sets are presented below and the relationship between the two is discussed in Section 3.4.2.3. Within this context, a pointcut may be defined by picking out joinpoints in one or the two levels, or in both. The advice code associated with the pointcut will then define treatments to be executed before and/or after these joinpoints.

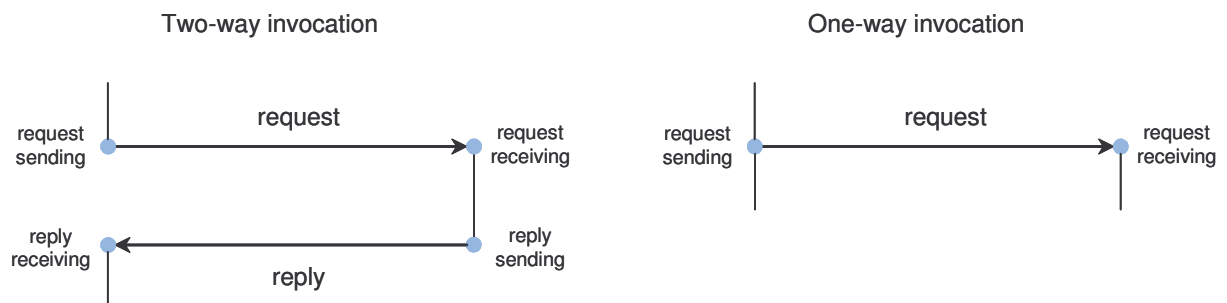
### 3.4.2.1 Transport level joinpoint types

The transport level is the middleware layer which is closest to the network. This level is concerned with the sending and the receiving of messages. In many existing middleware technologies (e.g. CORBA [1], RMI [14], .Net Remoting [2]), this level relies on the use of TCP sockets. However, no special hypothesis is made here on the chosen network transport technology. We only assume that the middleware is open enough to expose the event related to the sending and receiving of messages.

We then define four different types of transport level joinpoints:

- request sending (client side)
- request receiving (server side)
- reply sending (server side)
- reply receiving (client side).

Figure 3 summarises these joinpoints. The left side of the figure illustrates two-way request/reply invocations. The right side illustrates one-way (streaming and signals) invocation. In the latter case, the request sending and the request receiving are the only two relevant joinpoints.



**Figure 3: The transport level joinpoints.**

### 3.4.2.2 Application level joinpoint types

This level is the top-most level in the middleware stack. This is the level where the middleware meets the application. This level deals with the artefacts which are provided by the middleware to issue remote calls. In most cases the artefact is a method call on local stub which can be a component too. In other cases, this can be the call of an API primitive.

We define two different types of application level joinpoints:

- incoming calls on provided interface operations;
- outgoing calls on required interface operations.

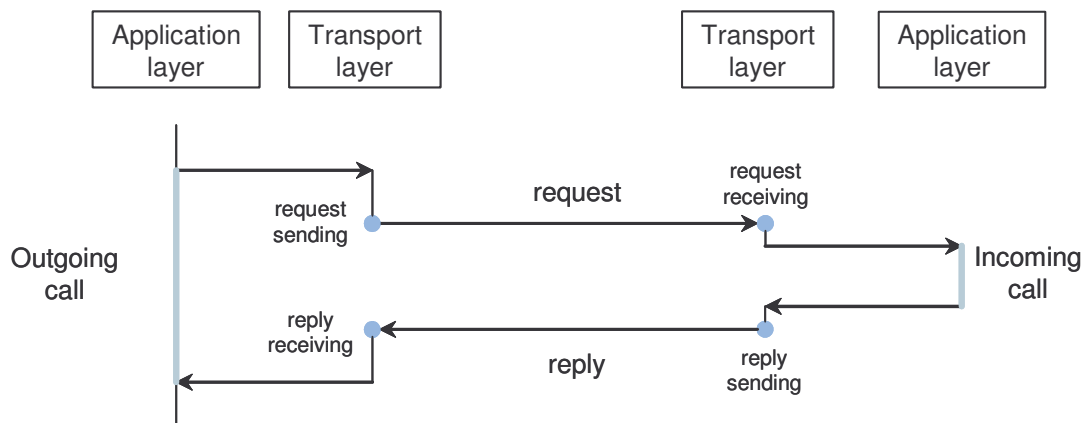
In AspectJ [15] terms, incoming calls are similar to execution joinpoints, and outgoing calls are similar to call joinpoints. However, the joinpoints defined here are *component based*, and coarser grained than the language based joinpoints of AspectJ therefore access to non-interface methods and members is not possible.

### 3.4.2.3 Relationship between the two levels

Depending on the application, programmers may need to define pointcuts picking out transport level joinpoints, or application level joinpoints, or both. When application and transport level joinpoints coexist in the same pointcut, their order depends on the invocation semantics.

For two-way invocations (Figure 4), the ordering of advice code is as follows:

1. before the outgoing call
2. before the client side request sending
3. after the client side request sending
  4. before the server side request receiving
  5. after the server side request receiving
    6. before the incoming call
    7. after the incoming call
  8. before the server side reply sending
  9. after the server side reply sending
10. before the client side reply receiving
11. after the client side reply receiving
12. after the outgoing call



**Figure 4: Two-way invocation.**

For one-way invocations (Figure 5), the situation slightly differs. Just after issuing the call, the client goes on executing while the message traverses the layers and the network. The outgoing call after code is then unrelated to the other pieces of advice.

The ordering of advice code is as follows:

1. before the outgoing call
2. before the client side request sending
3. after the client side request sending
  4. before the server side request receiving
  5. after the server side request receiving
    6. before the incoming call
    7. after the incoming call
- 1.1 The “after the outgoing call” event occurs after event #1 but is unrelated to the other 6 events.

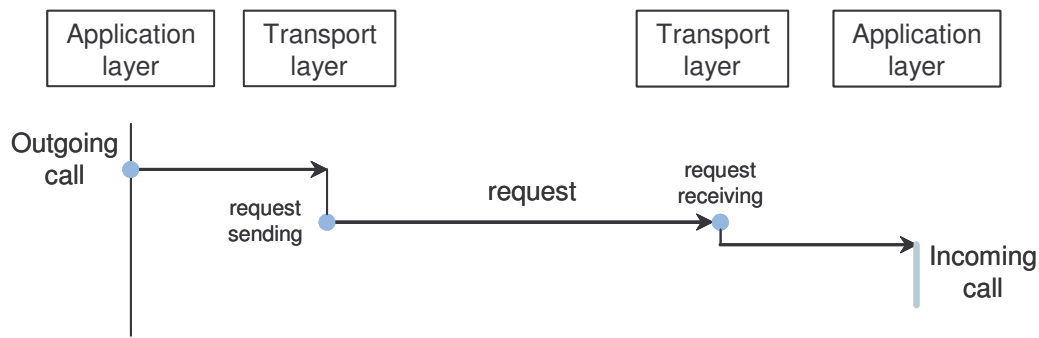


Figure 5: One-way invocation.

### 3.4.3 Extending the joinpoint model

The two previous levels may not fit all the requirements encountered when developing aspect-oriented middleware. For instance, developers may need to reify other events which occur in the middleware, network or system stacks, such as data marshalling or unmarshalling.

In order to accommodate these unforeseen needs, the joinpoint model provides an extension mechanism, where new types of joinpoints can be embedded into APL expressions. Each new type should be associated with a selector function which returns the set of corresponding joinpoints.

### 3.4.4 Domain definitions

In conformance with the model of Section 3.2, a component-based application will be defined by the following tuple consisting of the sets  $\langle CT, C, I, O \rangle$  where:

- CT is the set of containers on which the application is deployed;
- C is the set of components of the application;
- I is the set of component interfaces;
- O is the set of operations.

Notice that we do not assume a particular weaving model. Hence, the concrete definition of these sets may vary depending on the model. For instance, these sets may represent types when compile-time weaving is involved or instances when runtime weaving is involved. Notice also that in the case of runtime weaving, these sets are mutable, i.e. their content may evolve depending on the dynamicity of the application: new components may be created, or the application may be redeployed on different containers (hosts).

Two categories of properties are assigned to elements from these sets: *static* and *dynamic* properties.

#### 3.4.4.1 Static properties

Static properties are the minimal set of properties which we assume are available whatever the chosen middleware technology is. Given the four sets CT, C, I and O, we assume that:

- each  $ct \in CT$  owns a URI which can be referenced with  $ct.uri$ ; the concrete form of the URI depends on the chosen middleware technology: for instance, COSNaming names or IOR (Interoperable Object Reference) in the case of CORBA, RMI URL in the case of Java RMI, URL in the case of Web Services, etc.
- each  $c \in C$  owns:
  - a name which can be referenced with  $c.name$ ; when considering compile-time component weaving, the name may be the name of the component type; when considering runtime component weaving, the name may be that of the component instance (if relevant) or the name of the component type;
  - a set of interfaces which can be referenced with  $c.interfaces$ ; these are the interfaces implemented by the component.
- each  $i \in I$  owns:
  - a name, which can be referenced as  $i.name$ ;
  - a category,  $i.category$ , which is one of {provided, required};
  - a style,  $i.style$ , which is one of {request, stream, signal};
  - a set of operations,  $i.operations$ , which are the operations defined in the interface.
- each  $o \in O$  owns a signature which can be referenced with  $o.signature$ ; the concrete form of the signature depends on the chosen technology: for instance, UML signature strings, AspectJ-like signature strings, Java bytecode signature strings, etc.

All names ( $ct.name$ ,  $c.name$  and  $i.name$ ) are assumed to be unique.

#### 3.4.4.2 Dynamic properties

In order to accommodate for the use of any middleware technology, or to extend the expressive power of the pointcut language, the APL allows for the use of dynamic properties. Dynamic properties are  $\langle key, value \rangle$  pairs which can be attached to any architectural elements of the component model.

- Let  $x$  be an element of CT, C, I or O
  - $x.dynprop(\langle key \rangle, \langle value \rangle)$  attaches the  $(\langle key \rangle, \langle value \rangle)$  to the architectural element  $x$ ;
  - $x.dynprop(\langle key \rangle)$  retrieves the value attached to  $x$  with the given key, or *null* if the key does not exist

### 3.4.5 Pointcut expressions model

#### 3.4.5.1 Component-based pointcut expressions

A pointcut expression in APL is a 4-term expression  $\langle ctExpr, cExpr, iExpr, oExpr \rangle$  where each sub-expression is a first-order logic predicate selecting elements respectively from CT, C, I and O. Each sub-expression may use:

- universal quantifiers and quantified variables; for instance  $\forall o \in O$  designates all the operations available in the application;
- comparison expression on property values:
  - The list of legal properties is defined in Section 3.4.4;
  - The comparison operators are set theory operators ( $\in$  and  $\notin$ ) for set-like properties (the properties *category* and *style* for interfaces); for instance, the following expression:  
$$\forall i \in I, i.style \in \{request, stream\}$$
designates all the request/response and stream interfaces (thus excluding signal ones);
  - Pattern matching operations for string-like properties; the concrete syntax of the pattern language is implementation dependent; for instance, for method signatures, AspectJ pattern language or GNU regular expressions or etc.; for instance, the following expression:  
$$\forall o \in O, o.signature = \text{"public * org..C*.find(.."}$$
designates with the AspectJ pattern language, all public *find* method in sub-packages of *org* and in classes with a name which starts with *C*;
- boolean operators  $\wedge$  (logical-AND)  $\vee$  (logical-OR) and  $!$  (logical-NOT).
- meta-variables; meta-variables act as a mechanism for “passing” information from an APL expression to an aspect; meta-variable identifiers are prefixed with a question mark; for instance, the following expression:  
$$\forall o \in O, o.signature = \text{"Bank.*(parameters):?result"}$$
defines two meta-variables *?parameters* and *?result*. For each joinpoint picked out by an APL expression, meta-variables are bound to the values defined in the corresponding joinpoint.

Some examples of pointcut definitions in APL and the corresponding picked out joinpoints follow.

$$\forall ct \in CT, \forall c \in C, \forall i \in I, \forall o \in O$$

picks out all the joinpoints for all the containers, components, interfaces and operations.

$$\forall ct \in CT, \forall c \in C, \forall i \in c.interfaces, \forall o \in i.operations$$
$$c.name = \text{"Bank"} \wedge i.category \in \{provided\} \wedge o.signature = \text{"find*"}$$

picks out all the joinpoints for *find* operations located on the provided interfaces of the *Bank* component.

$$\forall ct \in CT, \forall c \in C$$
$$c.name = \text{"Bank*" } \wedge c.dynprop(\text{"bean"}) = \text{"entity"}$$

picks out all the joinpoints for the components named *Bank* with an attached property *bean* which values to *entity*.

### 3.4.5.2 Joinpoint selectors

Two types of joinpoint types have been defined previously: application level joinpoints and transport level joinpoints.

Application level joinpoints can be picked out by writing an APL expression which selects operations:

- incoming calls can be picked out by selecting an operation on a provided interface,
- outgoing calls can be picked out by selecting an operation on a required interface.

Transport level joinpoints can be picked out by using the following four functions which are defined on operations:

- reqsend(o): request sending for the *o* operation
- reqrec(o): request receiving for the *o* operation
- repsend(o): reply sending for the *o* operation
- reprec(o): reply receiving for the *o* operation

The example below picks out all the sending of the reply for the *find* operations of the *Bank* component.

```
∀ ct ∈ CT, ∀ c ∈ C, ∀ i ∈ c.interfaces, ∀ o ∈ i.operations  
  c.name="Bank" ∧ o.signature="find*" ∧ repsend(o)
```

### 3.4.5.3 Embedding language level pointcut expressions

Let us remind ourselves that the APL is a component based pointcut language and that we do not make any assumptions on the way components are implemented (class, functions, etc.) or on the language chosen for this implementation.

The pointcut model introduced so far allows expressing component level pointcuts. In some cases, we may need to “cross the boundary” of the component to express pointcuts relying on language elements. We will then end up with a mix of component and language level elements. Although the language level elements will not be reusable across different component technologies, this feature enhances the expressive power of APL.

Each first-order logic predicate selecting elements from CT, C, I and O, may use, in addition to the elements defined in Section 3.4.4.1:

- The function *embed()* allows embedding a language level pointcut expression into an APL expression.

An example of a pointcut definition in APL with embedded language level elements follows:

```
∀ c ∈ C  
  c.name="Bank" ∧ (let x=c in embed("set(x point)"))
```

picks out all the component level joinpoints located in the *Bank* component and all the language level joinpoints which are write operations on the *point* field defined in the implementation of the *Bank* component.

## 4. Level 2

### 4.1 Motivation and Overview

Level 2 conceptually consists of sets of possible distributed aspects using the basic microkernel tools that were discussed in Level 1.

The main goal of this second level is to map the reference architecture concepts and ideas of Level 1 to a number of key concerns in order to discover open issues that need further investigation. Roughly speaking, we chose to do a mapping for the key concerns which we elaborated on in [16]. Concerns mapped to our reference architecture are distribution, mobility, persistence, security, coordination and transactions.

We decided to adopt a particularly different approach for the distribution mapping with regard to the other concerns, because some bootstrapping infrastructure for deployment and *lookup* [31] has to be introduced to realise the distribution mapping. Therefore we treated the study and mapping of distribution as a special section that is a kind of bootstrap for getting remote deployment and lookup in place.

Each section which discusses one of our concern mappings is structured according to the same template. Each of them starts with an introduction, titled “<concern> as an aspect”, which summarises the most important findings of the key concern studies in [16]. The second and third parts present the mapping to the component model and the APL (see Section 3.4) respectively. The mapping to the component model can describe:

- potential new architectural elements/components needed for expressing a specific concern;
- potential deployment-specific extensions, a concrete example of how an aspect can be represented in the component model;
- a validation whether the recursive model is strong enough or a validation of the synchronous nature of the reference architecture.

The mapping to the APL can describe potential extensions to the joinpoint model. These may be needed for expressing the binding of a specific concern or attempt to generalise canonical pointcut expressions that frequently occur. A fourth section containing the mapping to the microkernel is optional, because some concerns don't have any particular important dependencies on the implementation strategies of the micro-kernel operations. The mapping to the microkernel can describe binding behaviour where the interactions between the binding and the microkernel are specified or some desired implementation strategies for a number of microkernel operations that are required for a specific concern. Finally, we end each concern mapping subsection with a discussion on problems and/or open issues for that specific concern that need further investigation.

Hence, the rest of this section is structured as follows. Before we start describing the mappings for each key concern, we sketch our philosophy behind the idea of viewing middleware as a set of aspects in Section 4.2. Section 4.3 provides the distribution mapping, while Section 4.4 discusses the other concern mappings, respectively for

mobility, persistence, security, coordination and transactions. We summarise our Level 2 description in Section 4.5.

## **4.2 Philosophy of Building Middleware in terms of Aspects**

We have not yet explicitly stated why Level 2 of the reference architecture is structured as a set of key concerns. The reason for this is that the philosophy behind the reference architecture is that middleware is no longer defined as a large black box, but rather as flexible composition of different middleware services. Each of these middleware services typically deals with one or more of the key concerns that are outlined in the mappings that follow.

We strongly believe in this design philosophy because this enables us to invest in a product line approach for building families of distributed applications. Here, each application, and especially the underlying middleware, can be considered as a multi-concern composition.

The component model, microkernel and APL defined at Level 1 of the reference architecture are crucial to support this philosophy effectively. The component model is a strong enabler for smooth cooperation between the different middleware services, while the microkernel enables dynamic composition of those services.

Finally, we want to point out the important relationship between this reference architecture and the study of key concerns [16]. One of the goals of that study is also to better understand and in the long run solve aspect interaction between key concerns. Being able to manage wanted or unwanted interactions is a prerequisite for making the above design philosophy work in practice.

## **4.3 Distribution**

This section firstly introduces the core concerns that appear in distributed environments when building distributed architectures. The second part of this section will map the introduced core concerns to the reference architecture.

### **4.3.1 Distribution as an aspect**

In this section, we provide a high-level presentation and analysis of the core concerns that appear in distributed environments, namely: deployment, lookup and remote communication. This analysis will particularly focus on point-to-point communications.

#### *4.3.1.1 Deployment concern*

When creating a distributed architecture, a major concern is the deployment concern. Deployment consists of defining where the different available components (e.g. services, resources, etc.) will be located on the topology. The way the deployment is done has direct impacts on many “ities” of the application (e.g. availability, security, extensibility, scalability, and manageability), which, most of the time, are of primary

importance. Moreover, since the topology of the environment might change often, it is sometimes important to control and tune the deployment in an efficient way (i.e. sometimes dynamically, and sometimes automatically).

The deployment is tightly linked to the application's architecture, since the architecture describes what components should be defined, and how these components should be related to each other. In component-based development, the architecture is often defined in ADLs (Architecture Definition Languages), which are domain specific languages that allow for the definition of the main interfaces and services of the application and also the possible bindings that can be made between the components. ADLs allow for an abstract specification of the application's architecture, which is used as an input of deployment tools that validate the architecture and drive the deployment. ADLs can be seen as aspect DSLs for the deployment concern, which is already a research domain of its own. As such, it does not make sense to apply AOSD here, since ADLs are already aspects for deployment.

ADLs can be seen as domain specific languages (DSLs) for the deployment concern, which is already a research domain of its own. As such, it does not make sense to apply AOSD here, since ADLs capture already aspects for deployment.

#### 4.3.1.2 *Lookup concern*

Once the application has been deployed, the components need to access other components of the distributed application. This is the *lookup* concern [31]. Lookup is done at the component's runtime when a component needs another component, since static binding is too restrictive in distributed environments. Once the lookup is done, the component receives a remote reference on the requested component. The protocol to access this reference and whether it remains valid over time depends on the underlying middleware infrastructure, which implies that all the components' implementations may depend on the distribution layer for the lookup concern.

Most of the distributed middleware layers provide the same kind of infrastructure and protocols for lookup. It consists of a well-known (i.e. each container can retrieve a reference to it) naming service where newly deployed components need to be registered in order to be accessible by other components. Therefore, the naming service basically defines a distributed protocol, which consists in registering and resolving component references. During the registering, the component must inform the service of its public name and its location, while the resolving will provide the client with a reference, which is basically a communication stub that will allow the client to communicate with the resolved component – classically through remote method invocations. We will study the communication concern in the next section.

The lookup concern can become increasingly complex depending on the features supported by the distribution layer. For instance, when the environment supports dynamic migration of components or automatic replication (for instance, for load-balancing and fault-tolerance), the access protocol can be hard to implement for the clients. Additionally, concerns such as security and scalability can interact with the lookup concern. Because of this, higher-level concepts such as namespaces, groups (and group references), access control services can be involved in lookup, which can make it more complex to manage and less transparent in the application's design.

#### 4.3.1.3 *Communication concern*

Once a specific distributed service (i.e. component) has been resolved, the client component can communicate with it through the distribution layer. This is the communication concern. Most of the distributed layers try to make the communication concern as transparent as possible. For instance, Java-RMI or CORBA generate stubs that can be directly accessed by the client components code as regular objects in the target language. The reference to a remote component can then be seen as a proxy, which implements a remote interface, and that can be used “almost” as a regular local interface.

However, several communication-related concerns often need to be handled by the client, in order to ensure extra properties on the communication channel. These properties are required because of a given architectural choice, or possibly because the communication layer provides elementary communication primitives, which would need to be composed or tuned in order to get more sophisticated mechanisms.

For instance, multicast protocols can be built on the top of point-to-point communication, which can be either synchronous or asynchronous. Multicasts have numerous uses, including facilitating the building of more efficient (get the fastest answer) or reliable (active replication) architectures. They can have various properties such as reliability (all the destination components will be reached), atomicity (all the invoked services will be executed without being interrupted by other multicast requests), transactional (the multicast implements the *ACID* properties which we detail in Section 4.4.5), which implies for example the possibility to rollback a multicast if one of the destination components cannot handle the request).

Using point-to-point communication, several basic policies/mechanisms can be implemented to reach all these properties. These policies/mechanisms include retrying (replay the invocation if an error occurred), timeout (give up after a given time) and timestamp (for ensuring real-time properties or causal, total, partial ordering).

In practice, it is not possible to offer all the possible properties at once. So, typical communication layers implement:

- simple mechanisms (such as simple point-to-point remote method invocation) in a synchronous or asynchronous way;
- very secure protocols (such as transactional protocols) which ensure the distributed system to stay in a consistent state.

However, using distributed transactional protocols can be inefficient, and distributed applications can, most of the time, implement more light-weight and specific protocols by using the simple point-to-point method invocation as a base element.

In that case, several concerns can be dealt with at the same time for building these protocols and aspects are of a great help for separating them, as we will show in the next section.

### **4.3.2 Mapping to the reference architecture**

This section shows how to map the previously identified concerns to the reference architecture.

#### 4.3.2.1 Mapping to the component model

Deployment and lookup are tightly linked concerns. Most of the deployment occurs at pre-runtime and is parameterised by a deployment configuration that is defined through ADLs. Each distributed application then defines its own deployment configurations, which allows names and locations (container URIs) to be given to the deployed components. In general, an application defines its own namespace, so that it is easier for the components of the application to resolve other components of the same application. We call an application-specific component name an *AID* (Application-level ID). The AID is usually valid in the default namespace associated to the application. However, an application can define several namespaces or access to other namespaces (defined by other applications or by the system). As a consequence, an AID can be local (to the default namespace), or global (it then includes a namespace name).

For instance, a simple deployment configuration could be the following:

```
namespace: name=bank (default)
bank: URI=containerURI_1, component=Bank, protocols=<supported_protocol_defs>;
simple: URI=containerURI_2, component=Simple;
binding: client=simple, server=bank, protocol=<a_communication_protocol>;
```

When the deployment occurs, the deployment engine registers the deployed components in a centralised component (accessible by everybody), which we call a *deployment repository*. This repository component defines:

- a *lookup* operation that allows for the lookup of an actual component's location depending on its AID, and
- a *deploy* operation, which deploys a new component to a given location (a container).

These two operations return a unique system-wide lookup ID, a *SID* (System-level ID), which contains all the information to access a component deployed in the middleware layer. In our architecture, the SID can be defined as a container URI and a component name.

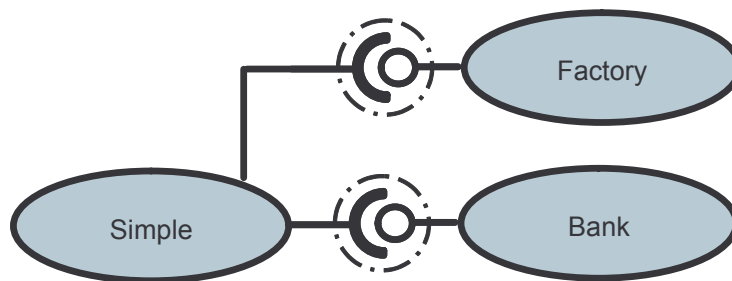
From a client's perspective, a component that needs to localise another deployed server component will use a local factory component, which shall be instantiated by a bootstrapping engine. This factory shall provide two operations (which can be merged into one unique operation):

- the *resolve* operation allows the client to get a proxy component to the actual server component. Note that the actual proxy component is dynamically instantiated by the factory depending on the deployment configuration. For instance, the proxy should support the communication protocol declared in the deployment configuration shown before.
- the *create* operation allows the client to create a new component, which will be deployed accordingly to the deployment configuration and to get a proxy for the required interface. This can be seen as a dynamic deployment feature that is useful in many cases (for instance, when the Bank component needs to create a new account for a bank customer).



**Figure 6: Initial state of the client.**

As a consequence, the initial state of the client is given in Figure 6. The resolving process will then create the appropriate bindings. In our simple example, when the client’s implementation invokes the resolve operation for the component named “Bank”, it will be bounded to a bank component as depicted in Figure 7, which is actually a proxy for this component.



**Figure 7: Binding to a bank component.**

#### 4.3.2.2 Mapping to the APL

In order to modularise the lookup concern, we can define an aspect that adds around advice on the factory. Instead of hard-coding the resolving protocol in factories, using an aspect here allows us to implement the resolving concern in a more flexible way. In particular, the pointcuts can be defined with regard to the deployment configuration, which allows for more specific implementation of the middleware infrastructure.

With the APL, a definition of the lookup concern would be:

```

 $\forall i \in I, \forall o1 \in O, \forall o2 \in O$ 
  i.category  $\in$  {required}  $\wedge$ 
  ( o1.signature="Factory.resolve(Aid ?aid):Interface"  $\vee$ 
    o2.signature="Factory.create(Aid ?aid):Interface" )

```

We use a “logic-variable-like” notation to map the arguments of the invocation to the advice. In our opinion, this should be seen as a (simple) syntactic proposal to pass data from the pointcut to the advice.

Implementing the communication concern with aspects consists of applying around advice to the stub (proxy) calls. This allows for the implementation of specific protocols like retrying, time-stamping, broadcasting, etc.

A possible definition for the bank operations would then be:

```
∀ i ∈ I, ∀ o1 ∈ O
  i.category ∈ {required} ∧
  o1.signature="Bank.*(?parameters):?result"
```

A specific aspect can then be written for each communication concern.

#### 4.3.2.3 Mapping to the microkernel

Given the design of the microkernel, two operations are directly impacted by the distribution concern: the BIND and UNBIND operations, which respectively, create and remove a binding between two components. When distribution comes into play, the establishing of a binding is modified as follows.

- Source and target references must be remote references (not mere memory pointers). The concrete form of those references may vary depending on the chosen middleware technology. For generality sake, one may assume that these references are URIs.
- With regard to a communication path, when establishing a binding between components located in different memory spaces, a simple “direct” binding is no longer adequate. Most of the time, a chain of middleware components will be interposed between the source and the target components. The concrete form taken by these components varies depending on the chosen middleware technology. Most of the time, these components will provide services for data marshalling and unmarshalling, message sending and receiving. The microkernel BIND operation is then responsible for setting up these middleware components and establishing the communication path between the source and target components.

The other microkernel operations (LOAD, UNLOAD, START, STOP, INSTANTIATE, DESTRUCT) may also be impacted by the distribution concern. One may assume that these operations are provided by the microkernel (seen as a component) through one or several interfaces. Nothing prevents then these interfaces from (1) being remotely accessible and (2) being seen as regular component interfaces. Invoking these operations is then a matter of establishing a distributed binding between the requesting component and the microkernel. The previously described mechanisms can be used in this case too.

#### 4.3.2.4 Discussion

In the case we want to optimise data transfer between the client and the server, approaches such as *event based aspect-oriented programming* (EAOP) [26] can be used in order to detect specific protocols and transform the interactions into more efficient ones. In particular, we have studied the transfer object technique, which requires the creation of a specific data structure (e.g. `CustomerAndAccountInfos`) in order to perform cache in upload or download. It also requires the introduction of a new operation in the server component, which takes as an argument the data structure.

With the APL, we would first need to introduce the new operation in the interface with the following pointcut:

```

∀ i ∈ I
    i.category ∈ {provided} ∧
    i.name ="Bank"

```

The introduction of the “`upload(CustomerAndAccountInfos):`” should then be specified.

From the client’s perspective, implementing this kind of data-transfer optimisation natively would definitely necessitate some other APL enhancements or the use of the *embed* primitive in order to apply EAOP pointcut expression languages. However, we would like to study here how this kind of optimisation could be natively supported by the APL.

A first solution would consist of detecting a sequence of messages in an advice. Once the sequence is detected, the aspect can then call the newly introduced *upload* operation. Here, the pointcut is simple:

```

∀ i ∈ I, ∀ o1 ∈ O
    i.category ∈ {required} ∧
    o1.signature="Bank.*(?parameters):?result"

```

For detecting a sequence of invoked operations, the advice must implement a state machine, which is a classically used solution when implementing EAOP.

However, it would be nice to be able to specify this in a more abstract way with the APL. A simple solution would be to add a set of causal/temporal composition operators to the set of already existing logical ones. In our example, we just need to add a sequence operator (+). The pointcut operator can then be expressed as:

```

∀ i ∈ I, ∀ o1 ∈ O, ∀ o2 ∈ O
    i.category ∈ {required} ∧
    ( o1.signature="Bank.createCustomer(?parameters):?result" +
      o2.signature="Bank.createAccount(?parameters):?result" )

```

This pointcut triggers the advice only when an invocation sequence of the two specified operations is detected.

## 4.4 Example Concern Mappings

In this section, the five example concern mappings are described: mobility, persistence, security, coordination and transactions. As mentioned previously in Section 4.1, each subsection is structured according to the same template.

### 4.4.1 Mobility

#### 4.4.1.1 Mobility as an aspect

In [16], we introduced the mobility concern through domain analysis. In the domain analysis, we identified a number of application independent crosscutting concerns that needed to be addressed when we want to support mobility in middleware.

The following are a list of middleware aspects which address these mobility concerns:

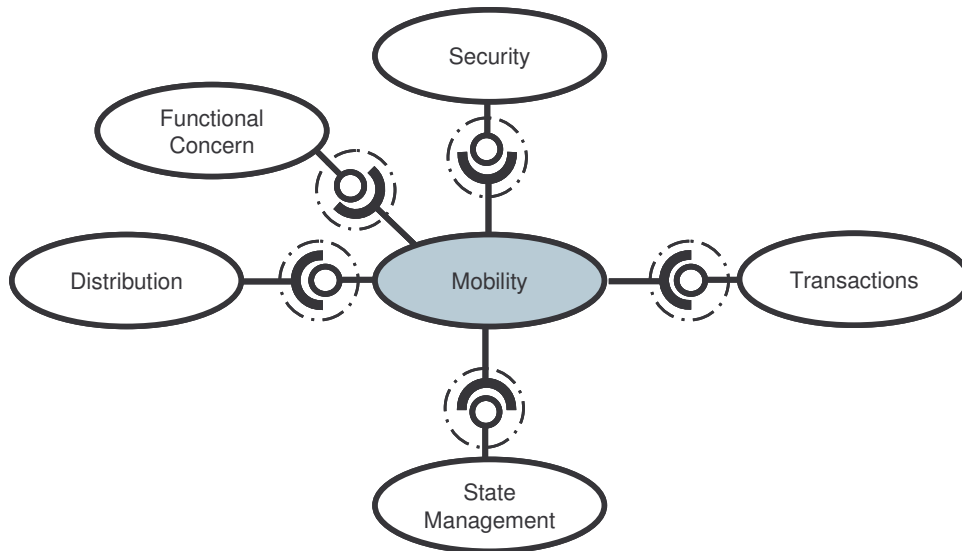
- Network Roaming
- Software Roaming
- Ad-hoc Networking
- Limited Connectivity
- Location and Proximity
- Service Discovery
- Quality of Service Assurance
- Limited Device Resources

Our goal in this section is to map the mobility concern specification to the component model, microkernel operations and APL which were described in Section 3.

#### 4.4.1.2 Mapping to the component model

Figure 8 is a representation of the mobility concern which is described in terms of the architectural elements of the component microkernel. The mobility concern is deployed in a container which manages the concerns, concern interfaces and bindings.

The mobility concern component is encapsulated functionality which interacts with other concern components through their interfaces.



**Figure 8: Mobility concern and its required interfaces.**

The mobility concern exposes both required and provided interfaces. The mobility concern requires behaviour encapsulated in the security, transaction (which we cover explicitly in Section 4.4.5), distribution and state management concerns.

Figure 9 illustrates the provided interfaces of the mobility concern. These interfaces are intended to address the issues relevant to the mobility concern.

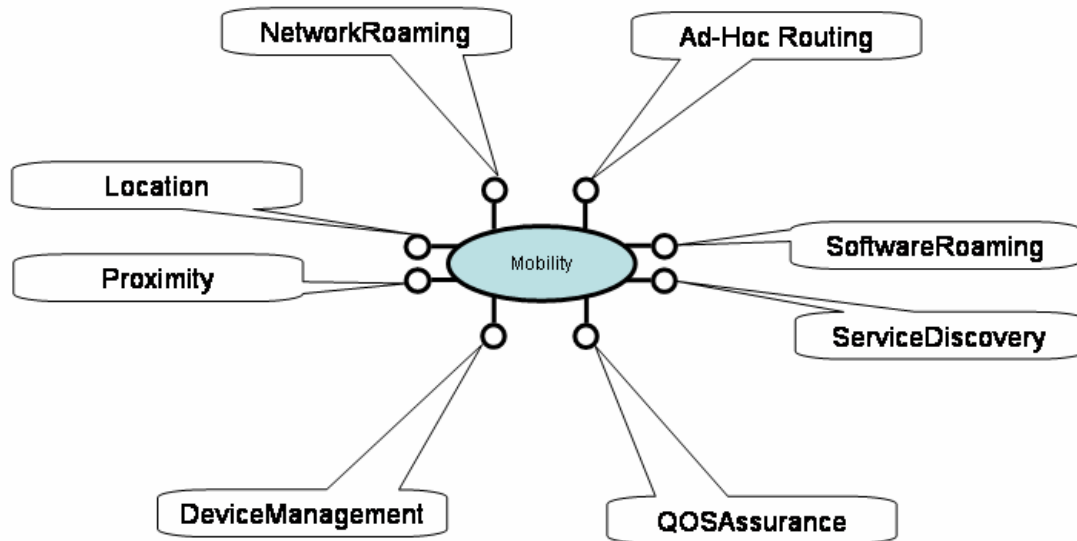


Figure 9: Mobility provided interfaces.

As illustrated in Figure 8, we anticipate that there will be bindings between other middleware and application components with the mobility concern. In the mapping of the architectural elements of the component microkernel to the mobility concern, we have not identified any new kinds of architectural elements/components needed for expressing the mobility concern. As described in [16], with regard to mobility, and illustrated in Figure 8, Figure 9 and Figure 10, the aspects that have been specified in the mobility concern are symmetric in nature. Mobility aspects crosscut one another as well as affect the application components.

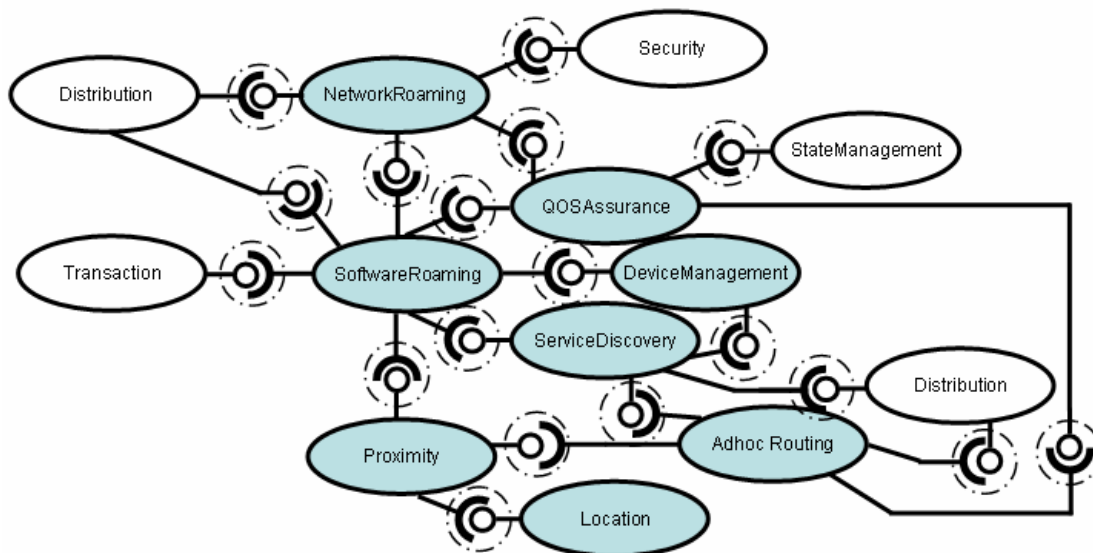


Figure 10: Mobility concern aspects.

In Figure 10, we show that the mobility aspects are interdependent, in that there is a dependency between mobility aspects. There is also a dependency between the mobility aspects and other concerns, such as distribution.

To explore the mapping between the mobility concern and Level 1 of the reference architecture we will focus on one aspect of the mobility concern – Network Roaming. As illustrated in Figure 11, the network roaming aspect requires services from the security aspect in cases where various networks may need clients to be authorised or where encryption protocols such as WEP are required. The network roaming aspect may also require support from the quality of service aspect to allow the network roaming aspect to choose the best network based upon a network’s properties/services. The network roaming aspect also provides services to the software roaming aspect allowing the software roaming aspect to switch networks as and when the mobile agents move. It provides interfaces to the ad-hoc routing aspect to allow it to change network if it needs to. Additionally, the network roaming aspect provides an interface to the distribution aspect. Distributed communications require availability of a network and in some cases a network that provides specific features, e.g. performance.

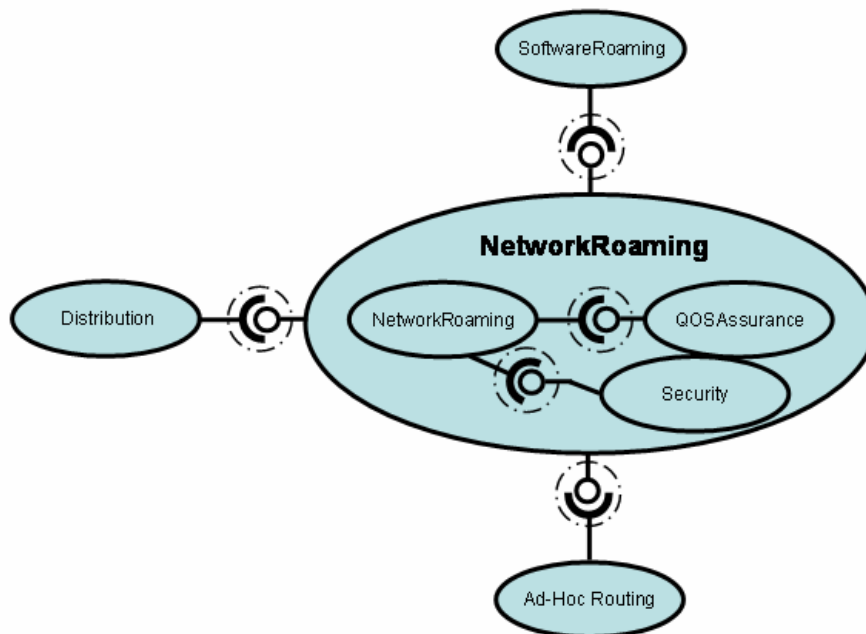


Figure 11: Network roaming.

The `NetworkRoaming` aspect addresses the situation where a mobile device loses network connections as the device moves outside of the network range. A mobile device can potentially have multiple networks available to it at any given point in time. As the mobile device changes location the device may lose connections to some networks and gain connections to others. To ensure that an appropriate network is always available to the application, the `NetworkRoaming` aspect provides a facility for selecting networks that the application uses.

We assume that the distribution concern will fully encapsulate network access. For this reason, it is our intuition that the network roaming aspect will need to be composed with the distribution aspect. Such a composition may be required when the supported application executes on a mobile device.

The component model facilitates composite components. To fully support `NetworkRoaming` based on application requirements, the quality of service (QOS) aspect needs to be composed with the Network roaming aspect. In the following paragraph we illustrate a composite aspect.

In order for the network roaming aspect to select the best network for any particular component operation which is distributed in nature, the QOS assurance aspect must be composed with the network roaming aspect to provide a means of selecting the optimal network. Without this composition, the network roaming aspect would select the next available network, without determining the suitability of the network.

This is illustrated in Figure 11. The composition of the required QOS and network roaming is presented in Figure 12. Theme/UML<sup>1</sup> is used to describe the composite aspect. The individual QOS and network roaming themes/aspects are described in [16].

The `NetworkRoaming` aspect monitors the state of all networks that are available to the distribution concern components. Where network activity is required, the network roaming crosscutting concern is triggered. The `NetworkRoaming` aspect searches for new networks that may have become available to the component since the last time the aspect was triggered. This occurs prior to revalidating each of the network properties that the application may be interested in, before selecting a network. The QOS assurance aspect acquires the required assurances from the assurances database for the operation that is crosscut, and then establishes a test scenario. The test scenario compares the assurance values of the properties that the aspect is watching against the requirements of the crosscut method. When composed with the QOS aspect, the `NetworkRoaming` aspect facilitates the selection of a network that best suits the network requirements of the crosscut application method. The composition of the two aspects can be seen in Figure 12.

---

<sup>1</sup> Please see D11 Section 5.4.2 for an overview of Theme/UML.

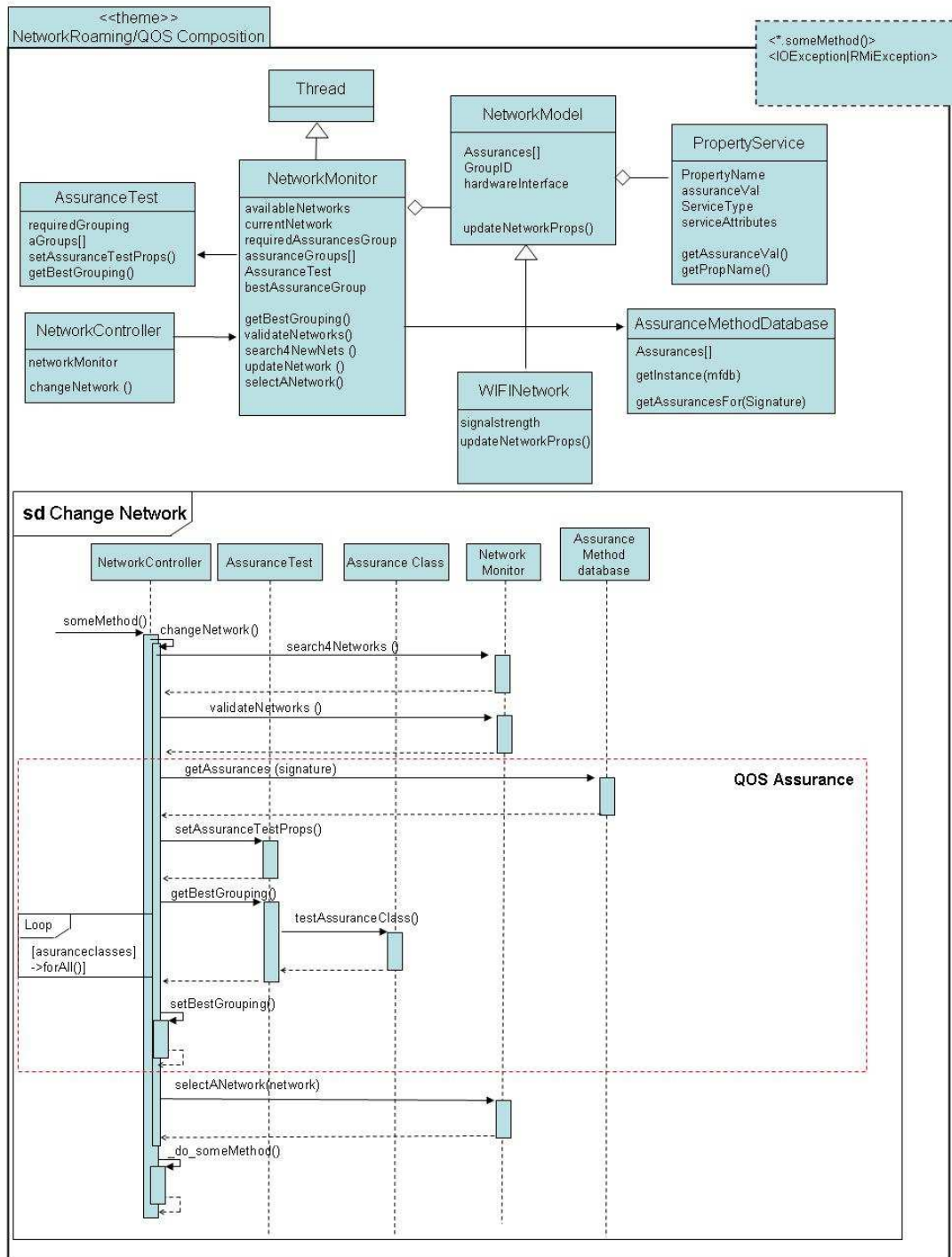


Figure 12: Network roaming & QOS assurances concerns composed.

#### 4.4.1.3 Mapping to the APL

The network roaming aspect affects the joinpoints where an operation that requires network access is invoked. The APL specification to capture the joinpoints at which the `NetworkRoaming` aspect is activated is provided next in `NetworkRoaming-Distribution-1`. We assume that where an operation throws a `NetworkException`, calling this operation will involve network access.

### ***NetworkRoaming-Distribution-1***

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {provided} ∧
  o.signature = "*.*(?parameters):?result throws NetworkException"
  ∧ reqsend(o)
```

Another case for the `NetworkRoaming` aspect activation is when there are network problems, such as the loss of a network connection due to mobility. The APL specification to capture the joinpoints, at which the `NetworkRoaming` aspect is activated in this case, is provided below in `NetworkRoaming-Distribution-2`.

### ***NetworkRoaming-Distribution-2***

```
∀ o ∈ O
  o.signature = "NetworkException.new()" ∧
  reqsend(o)
```

The QOS aspect crosscuts the network roaming aspect and introduces the ability to select the best available network, given the requirements of the application operation that causes network access. The `QOS-NetworkRoaming` specifies the `selectANetwork(Network)` operation on the `AssuranceTest` component within the `NetworkRoaming` component.

### ***QOS-NetworkRoaming***

```
∀ c ∈ C, ∀ o ∈ O
  c.name = "NetworkRoaming" ∧
  o.signature = AssuranceTest.selectANetwork(Network)
```

The software roaming aspect is crosscut by the network roaming aspect. The software roaming aspect enables components to become mobile between devices. The network roaming aspect allows for new network connections to be established by the component once it has migrated.

### ***NetworkRoaming-SoftwareRoaming***

```
∀ c ∈ C, ∀ o ∈ O
  c.name = "SoftwareRoaming" ∧
  o.signature = RemoteAgentManagerInterface.receiveAgent(Agent)
```

The ad-hoc routing aspect is crosscut by the network roaming aspect. The network roaming aspect provides interfaces to the ad-hoc routing aspect to allow it to change network if required.

### ***NetworkRoaming-Ad-Hoc-Routing***

```
∀ c ∈ C, ∀ o ∈ O
  c.name = "SoftwareRoaming" ∧
  o.signature = RouteTable.findRouteToDestination(destination)
```

#### 4.4.1.4 Mapping to the microkernel

In the following list, each of the microkernel operations is described in relation to the `NetworkRoaming` aspect of the mobility concern.

- **LOAD** – The `NetworkRoaming` aspect is loaded when deployed on a device that is mobile.
- **UNLOAD** – The `NetworkRoaming` aspect is unloaded when the application migrates to a device that is stationary.
- **START** – The `NetworkRoaming` aspect starts when network features do not meet the requirements of the application or when networks connections are lost.
- **STOP** – The `NetworkRoaming` aspect stops when network features meet the requirements of the application or when networks connections are stable.
- **BIND** – Bindings between the `NetworkRoaming` aspect and QOS aspects occur when the application requires networks that fit the requirements of the application.
- **UNBIND** – The `NetworkRoaming` aspect and QOS aspects are unbound when the application just wants to move to the next available network when mobile, and the application does not have particular requirements on the network which it uses.
- **INSTANTIATE** – The `NetworkRoaming` aspect is instantiated when the distribution concern is loaded, the location of the device is changing or when networks connections are being lost and need to be replaced. The `NetworkRoaming` aspect is a singleton instance.
- **DESTRUCT** – The `NetworkRoaming` aspect is destroyed when the distribution concern is destroyed, the location of the device is not expected to change or when networks connections are expected to remain stable.

#### 4.4.1.5 Discussion

The mobility aspect framework encapsulates issues which relate to mobility. The encapsulation of these concerns ensures that the mobility concern is isolated and reusable across applications. We have demonstrated that the aspects which reify the mobility concern can fit the component model and that the APL can be used to describe the interactions between the mobility aspects as well as the other concerns. We provide a description of how the concerns fit the operations defined for the microkernel. It is our intuition that further work may be required to fully specify the semantics of the component model, APL and microkernel operations. At the level of abstraction that is being applied here, semantics are not as well defined as those in existing middleware platforms.

### 4.4.2 Persistence

#### 4.4.2.1 Persistence as an aspect

Providing persistence support (i.e. the storage and retrieval of application data from secondary storage media) is vitally important for many software applications.

However, using traditional object-oriented approaches to develop persistence is problematic due to the inherent crosscutting and tangling that are exhibited upon integration with an application. Crosscutting and tangling impacts heavily on comprehensibility making the concern hard to modify, maintain, reuse and evolve. Additionally, persistence is a concern that tends to be highly coupled to a specific application and/or data schema which further reduces the possibility for reuse in other contexts.

It has been shown that persistence can be modularised in a more effective manner using AOP [29] or *computational reflection* [17]. [16] provides a detailed comparison of both approaches using AspectJ [15] and nitro [18]. In this section we describe persistence using the Level 1 component model to facilitate mappings to different AO platforms.

There are two important points which must be taken into account when designing a persistence framework.

1. There needs to be a means to separately identify persistent and transient data.
2. The framework should be left as abstract as possible in order to instantiate it with different parameters to increase reusability.

In order to address the first point, we can use the notion of *persistent root classes* from object-oriented database systems [30]. With this notion, all classes that are required to be persistent, extend a common base class which contains persistence related functionality.

With regard to the second point, the use of a component framework allows the binding together of different components thereby allowing the framework to be used in different contexts and applications.

#### 4.4.2.2 Mapping to the component model

The mapping we chose is based upon our prior experience of implementing a persistence framework using AspectJ [29]. By mapping each class and aspect module to a component, and their respective method and advice implementations to a *provided* interface, we aim to raise the level of abstraction to one which can then be mapped to other AO platforms. *Required* interfaces are used on *joinpoints* or when a component requires an implementation that is provided by another component. In Figure 13, we first present the high level component model for providing persistence based upon the architectural elements previously described in Section 3.2. It should be noted that each component shown may also consist of subcomponents and we will detail these where required. The persistence component consists of four different sub-components, namely, `CorePersistenceFunctionality`, `RDBMSSettings`, `SQLTranslation` and `Object-RelationalMapping`, which we will now discuss in more detail.

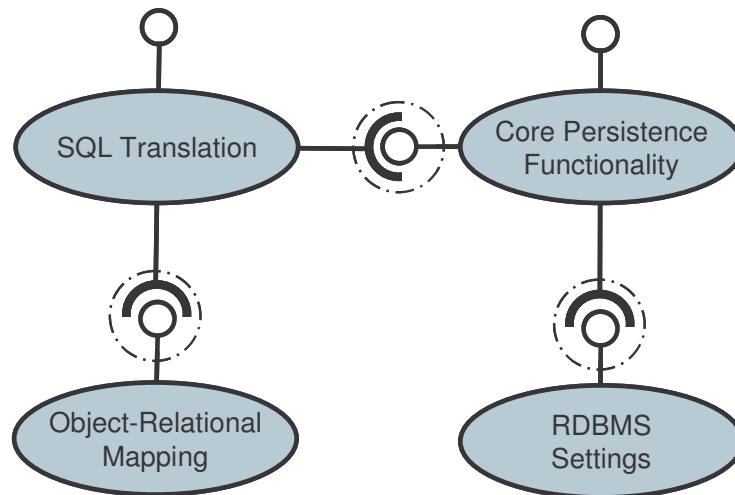


Figure 13: High level component model for persistence.

### *Core Persistence Functionality and RDBMS Settings*

The `CorePersistenceFunctionality` component shown in Figure 14 is the principle component of the persistence aspect and it consists of two sub components, namely `DatabaseAccess` and `PersistentDataImplementation`.

The main functions of the `DatabaseAccess` component are to:

- open and close connections to the database management system;
- trap instantiations, retrievals, deletions and updates of data objects in the main application code that are to be persisted.

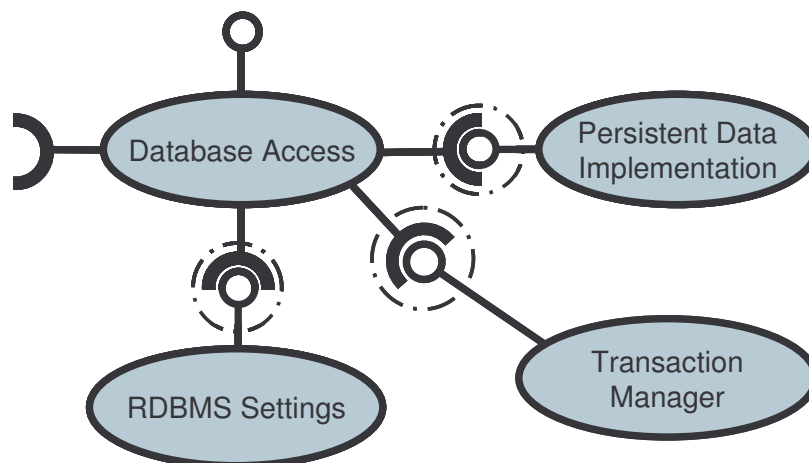


Figure 14: Core persistence functionality component.

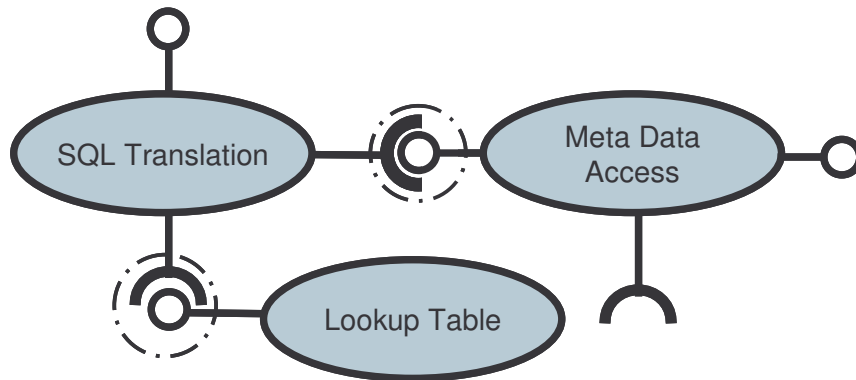
The `PersistentDataImplementation` component provides a procedural querying interface over the persisting classes.

The `RDBMSSettings` component provides context specific information to the `DatabaseAccess` component such as the database driver (e.g. Oracle, MySQL, etc.), location of the database (i.e. the URI) and also application specific information about how connections to the database should be opened and closed.

The `TransactionManager` component provides transaction management support (i.e. ‘commit’ and ‘rollback’). We detail transaction management explicitly in Section 4.4.5.

### *SQL Translation*

This component, shown in Figure 15, supports the object-relational mapping. For the component to be reusable in other contexts, it must be independent of application specific mappings.

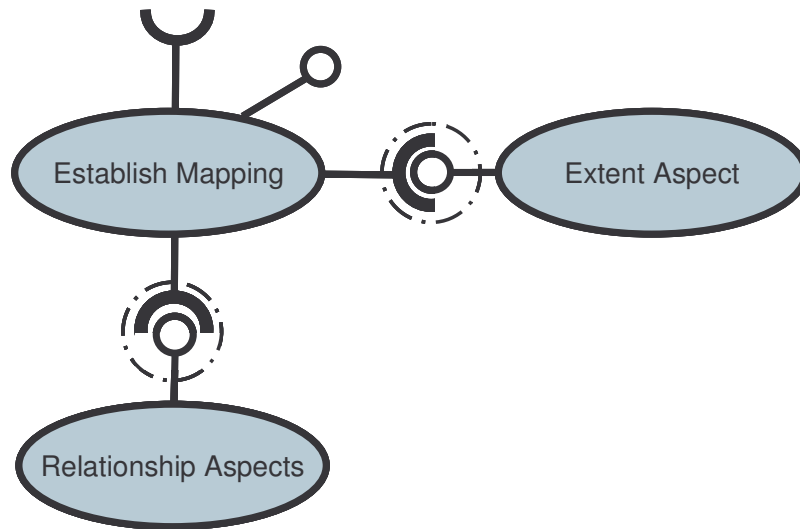


**Figure 15: SQL translation component.**

A key reason behind this is to allow the framework to be used with OO databases, where such SQL translations are not needed. In such a situation, it should be possible to decouple the `SQLTranslation` component; therefore it is designed as a separate component. The main function of the component is to trap SQL updates. It uses reflection to trap various ‘get’ methods and a mapping in the `LookupTable` component to map objects, their updates and deletions to the database. On retrieval the objects are then recreated. The `MetaDataAccess` component provides access to database meta-data such as column names in a table or its foreign key links.

### *Object-Relational Mapping*

The components shown in Figure 16 map objects and tables using application specific data. In other words, the component contains explicit mappings between classes in the application and these will obviously change for different applications. Mappings must be established before a connection can be made.

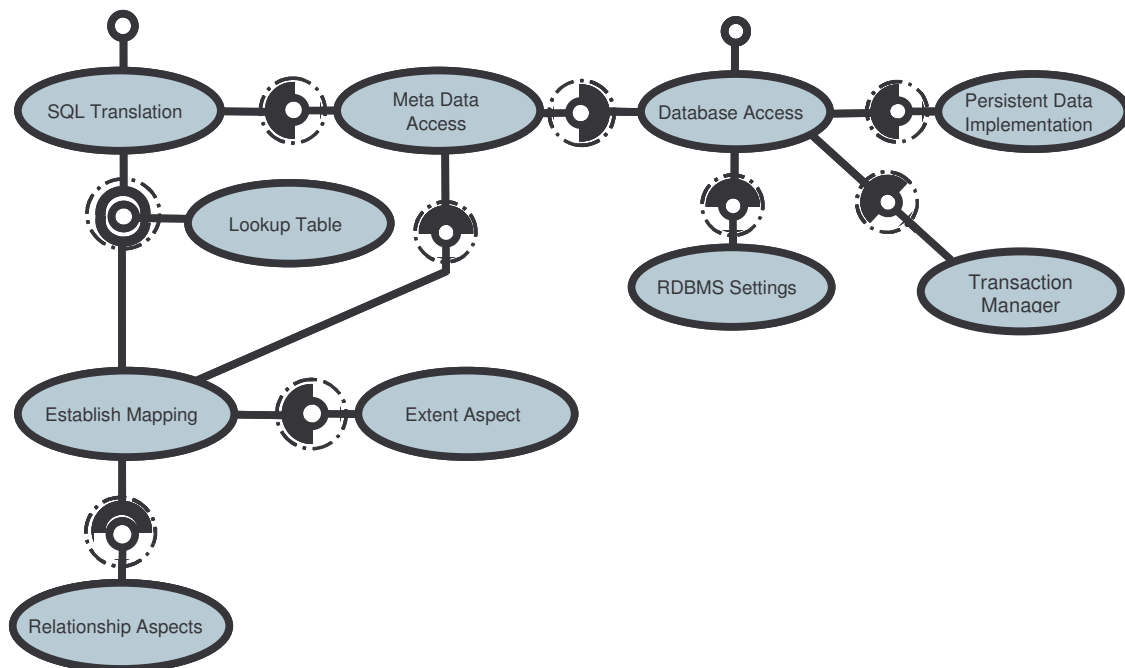


**Figure 16: Object-relational mapping component.**

The `EstablishMapping` component sets up the class to table mapping via the `LookupTable` component and also obtains the foreign keys from the `MetaDataAccess` component. `RelationshipAspects` are a number of aspects which introduce relationships between the persistent classes. The `ExtentAspect` is the component which explicitly states which classes will be persisted.

### *Complete Persistence Component Framework*

Figure 17 illustrates the complete persistence framework and how the elements are bound together. The two unbound provided interfaces, on the `SQLTranslation` and `DatabaseAccess` components, are then bound to the underlying application as previously described via pointcut declarations.



**Figure 17: Complete persistence component framework.**

### 4.4.2.3 Mapping to the APL

In this section we illustrate how the pointcuts are mapped to our APL described in Section 3.4. We describe each component separately and the functionalities along with the appropriate APL abstractions.

#### *Database Access Component*

The `DatabaseAccess` component consists of five pointcut definitions which all rely on the classes to be persisted being subclasses of `PersistentRoot`.

#### *Trap instantiations*

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {required} ∧ o ∈ i.operations ∧
  o.signature="PersistentRoot+.new(..)"
```

#### *Trap retrievals*

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {required} ∧ o ∈ i.operations ∧
  o.signature="Vector PersistentData.getExtent(String):?className" ∧
  embed( let className=?className in "args(className)" )
```

#### *Trap deletes*

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {provided} ∧ o ∈ i.operations ∧
  o.signature="public void PersistentRoot+.delete():?obj " ∧
  embed( let obj=?obj in "this(obj)" )
```

#### *Detect deleted objects*

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {provided} ∧ o ∈ i.operations ∧
  (o.signature="public * PersistentRoot+.get*(..)" ∨
  o.signature="public * PersistentRoot+.set*(..)" ∨
  o.signature="public * PersistentRoot+.toString()" ) ∧
  embed( let obj=?obj in "this(obj)" )
```

#### *Trap updates*

```
∀ i ∈ I, ∀ o ∈ O
  (i.category ∈ {required} ∧ o ∈ i.operations ∧
  o.signature="Vector getObjects (ResultSet, String) ") ∧

  (i.category ∈ {provided} ∧ o ∈ i.operations ∧
  o.signature="PersistentRoot+.set*(..):?obj " ∧
  embed( let obj=?obj in "this(obj)" )
```

With respect to the ‘trap updates’ APL description, it should be noted that the APL as yet does not provide the abstractions necessary for capturing control flow (e.g. cflow in AspectJ) constructs. This is a key requirement which should be fulfilled in a future update to the APL.

### ***SQL Translation Component***

The `SQLTranslation` component contains a single advice which intercepts SQL updates.

#### ***SQL execution***

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {required} ∧ o ∈ i.operations ∧
  o.signature="int Statement.executeUpdate(String):?sqlStatement
?statement " ∧
  embed( let sqlStatement=?sqlStatement in "args(sqlStatement)") ∧
  embed( let statement=?statement in "target(statement)")
```

### ***RDBMS Settings Component***

The `RDBMSSettings` component contains two pointcut definitions for establishing connection on starting the application and closing the connection on system shut down.

#### ***Establish connection***

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {provided} ∧ o ∈ i.operations ∧
  o.signature="public static void Main.main(String[])"
```

#### ***Close connection***

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {required} ∧ o ∈ i.operations ∧
  o.signature="System.exit(int)"
```

### ***Establish Mapping Component***

The `EstablishMapping` component has a pointcut definition for recovering foreign key links from the `MetaDataAccess` component.

#### ***Obtain Foreign Keys***

```
∀ i ∈ I, ∀ o ∈ O
  i.category ∈ {provided} ∧ o ∈ i.operations ∧
  o.signature="Enumeration MetaDataAccess.getForeignKeyLinks(String):
?className" ∧
  embed( let className=?className in "args(className)")
```

#### 4.4.2.4 Mapping to the microkernel

We believe that persistence does not place any special demands upon the microkernel. The microkernel operations should allow the persistence component to be dynamically woven/unwoven into the system at run time as the developer sees fit.

#### 4.4.2.5 Discussion

The persistence framework encapsulates persistence functionality without causing crosscutting of the application and hence promotes reusability via the use of components. The framework has been designed for use with relational databases employing SQL-92. In theory, for object-oriented databases we can remove components that are RDBMS specific and change them to their ODBMS counterparts. However, it should be noted that application developers can only remain partially oblivious to persistent nature of the data, as persistence has to be accounted for as an *architectural decision* during design of the components.

The mappings between different tables will usually be done via hand, although this can be improved by adopting a *code-generation* approach [19]. For example, the creation of SQL queries (as opposed to using reflection which may not be always possible on certain platforms) could be simplified by the use of code templates or a GUI driven design wizard which maps a schema to the underlying persistence application. An approach such as *framed aspects* [20] can help in this regard by removing a lot of the repetitive and error prone tasks involved.

With respect to the APL there appears to be no abstraction for capturing control flow constructs. This is an important point which we will address in the future by adding a new function to the APL.

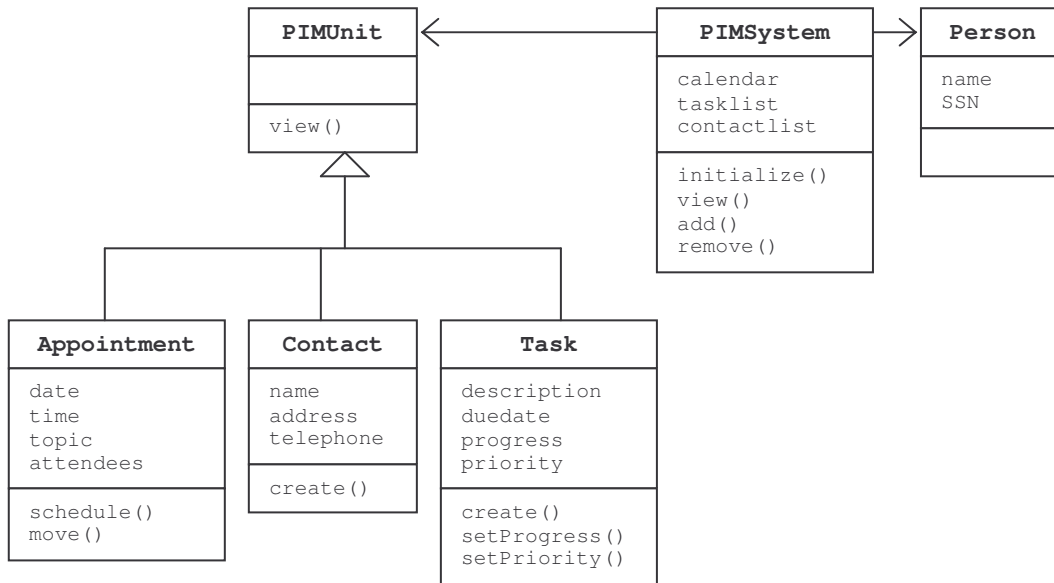
### 4.4.3 Security

In [16], we studied the use of AOSD for security. In this section, we generalise the work on the security concern by mapping the concern to the reference architecture. The APL is used to specify the crosscutting nature of security related aspects. By doing this, we validate the reference architecture and identify where the reference architecture and APL should be extended (or even improved) in order to enable effective modelling of the key concerns.

#### 4.4.3.1 Security as an aspect

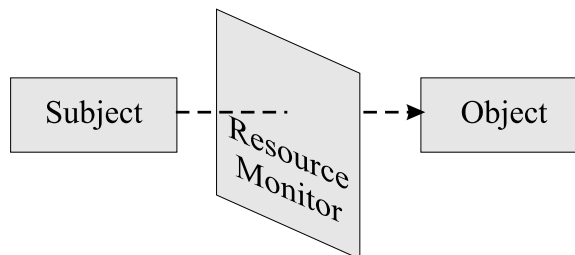
Because security concerns inherently crosscut an application, we have investigated, in [16], how these security concerns can be better modularised as an aspect. We discussed the modularisation of access control using AspectJ in the running example of a Personal Information Management (PIM) system. We also elaborated on opportunities and challenges when applying AOSD for other application-level specific requirements, such as confidentiality and integrity, non-repudiation, auditing and anonymity and privacy. In the rest of this section, we will focus on the mapping of the access control concern to our reference architecture. We illustrate our mapping using the example PIM system, and Figure 18 shows its design. The implemented access control model is owner-based and we chose to guarantee the following rules.

- The owner (i.e. creator) of a `PIMUnit` can invoke all operations on that unit.
- Contacts are only accessible to their owner.
- All other accesses to a `PIMUnit` are restricted to viewing.



**Figure 18: Design of the PIM system.**

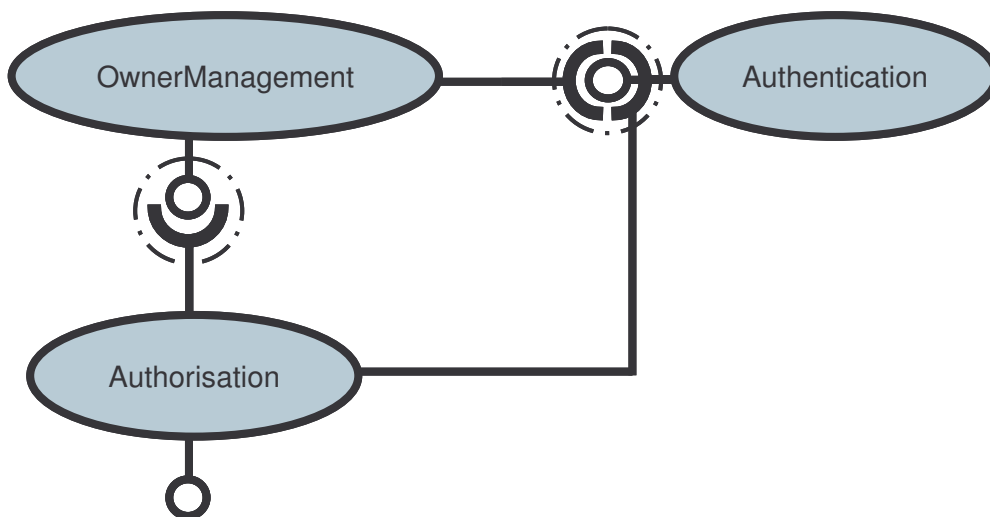
We start from the conceptual access control model involving three entities, depicted in Figure 19. These entities are an *object* (i.e. the entity to which access is requested), a *subject* (i.e. the entity requesting access) and a *reference monitor* (i.e. the entity checking every access attempt from a subject to an object). Both the subject and the object can have a number of security or non-security related attributes that can influence the access control decision. This decision possibly is also influenced by other information not directly related to the subject or object.



**Figure 19: Access control model.**

#### 4.4.3.2 Mapping to the component model

In order to modularise the access control concern, we can define three components. All three components along with their interfaces and bindings are depicted respectively in Figure 20. The unbound provided interface on the `Authorisation` component is bound to the underlying application via a `pointcut` declaration.



**Figure 20: Access control components.**

### ***Owner Management***

`OwnerManagement` is responsible for storage and initialisation of object owners. To this end, an `OwnerManagement` component instance is associated with every object. Every object will be decorated with an owner attribute that is initialised by the `OwnerManagement` component when that object is created.

### ***Authentication***

`Authentication` is used to authenticate subjects and serves as a kind of login functionality. The component includes an attribute that represents the current subject trying to access an object and an operation that initialises this attribute. In the context of the PIM system, the `currentUser` attribute represents the current subject and gets initialised by the `getUser()` operation.

### ***Authorisation***

`Authorisation` implements the actual access control. The component contains an around advice that verifies the equality of the owner of the object that is currently being accessed and the current subject trying to access it as identified through authentication and acts accordingly. In terms of our PIM example, the equality of the PIM unit owner and the current user is verified and will be acted upon.

#### ***4.4.3.3 Mapping to the APL***

With the APL, pointcuts to be defined with regard to access control are now described. To bind the `OwnerManagement` component with the other components, two pointcuts are needed: one enabling the introduction of an owner attribute in each relevant component (1) and one enabling the initialisation of these owner attributes (2). Both example pointcuts below are defined within our PIM system context. The first pointcut can be used to introduce the owner attribute in the `PIMUnit` interface. The second one captures the operations after which the owner attribute should be initialised.

- (1)  $\forall i \in I$   
 $i.category \in \{provided\} \wedge$   
 $i.name = "PIMUnit+"$
- (2)  $\forall i \in I, \forall o \in O$   
 $i.category \in \{required\} \wedge$   
 $o \in i.operations \wedge$   
 $( o.signature="Appointment.schedule(*):*" \vee$   
 $o.signature="Contact.create(*):*" \vee$   
 $o.signature="Task.create(*):*" )$

The pointcut defining the places for enforcing the actual access control, again within the context of our example PIM system, is given next (3). The example pointcut captures all the operations that are relevant for access control policy within a PIM unit.

- (3)  $\forall i \in I, \forall o \in O$   
 $i.category \in \{required\} \wedge$   
 $o \in i.operations \wedge$   
 $( o.signature="Appointment.move.*(*):*" \vee$   
 $o.signature="Contact.view.*(*):*" \vee$   
 $o.signature="Task.setProgress.*(*):*" \vee$   
 $o.signature="Task.setPriority.*(*):*" )$

These pointcuts constitute a number of concrete bindings in the context of our example. The components and their bindings are depicted using UML in Figure 21. A binding pattern consists of the set of bindings that start from a specific aspect component.

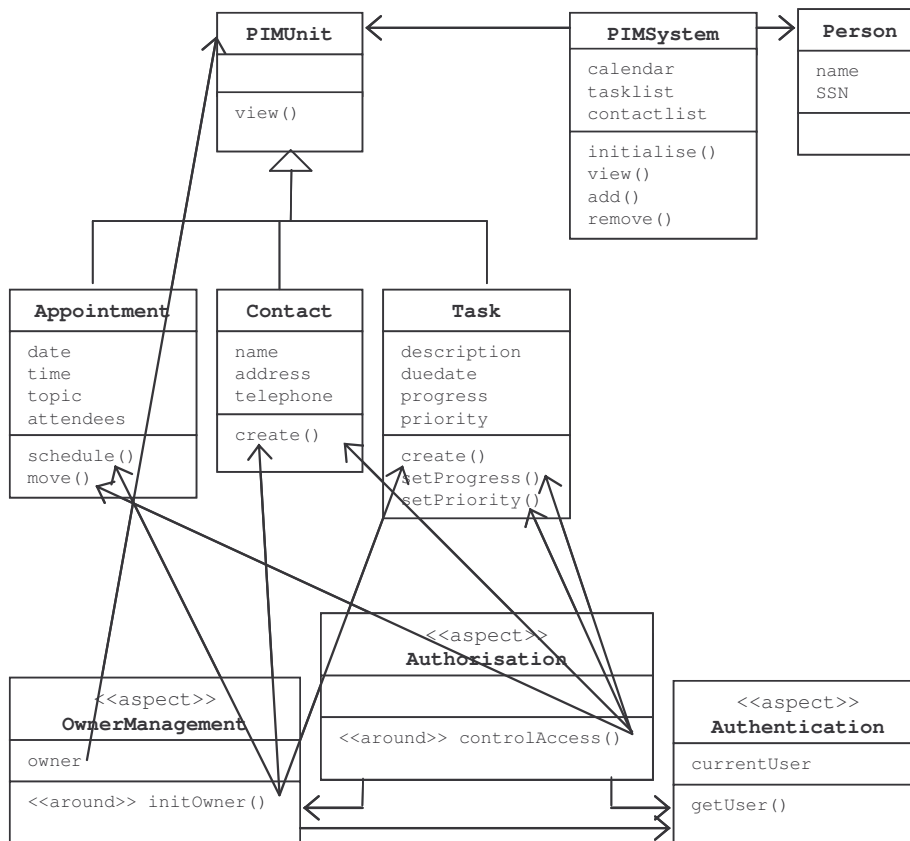


Figure 21: Bindings in our PIM system.

#### 4.4.3.4 Mapping to the microkernel

Recognising the design goals of our microkernel, we don't feel that security depends on specific implementation strategies of the microkernel operations listed before (load, unload, start, stop, bind, unbind, instantiate and destruct). However, we do want to stress that no unauthorised access to these main operations must be allowed. For instance, it should not be possible for an arbitrary malicious person to load unsafe components or unload security-critical components.

#### 4.4.3.5 Discussion

There are some important problems that inhibit us to map security concerns to the reference architecture in a fully satisfactory way.

- First of all, there is the need for context passing. For instance, when we are advising a kind of login method, we possibly need to have the credentials available. How do we map such a parameter passing mechanism to the existing reference architecture? Is it sufficient to define an aspect component instance per control flow of that login method or do we need an extension of the architecture? Another possibility can be using the `embed()` operator enclosing a complete APL pointcut.
- Secondly, as stated before, there are different kinds of information on which an access control decision can be based. Possibly, part of this information belongs to the internal state of one or more components. Hence, we must be able to access this internal state under certain circumstances. The only option here is the use of lower level pointcuts enabling us to capture points that are not exposed through public interfaces. This is where the `embed()` operator comes into play, by which language specific low level pointcuts can be introduced in an APL pointcut definition.
- Third, there exist a number of aspect component instantiation methods. For example, one component instance can serve for an entire program, while in a different setting you possibly are required to create an aspect component instance for each component executing at any of the joinpoints within a pointcut. How do we express these different kinds of component instantiation?
- Fourth, the order in which components have to execute at the same joinpoint can be of importance. How do we express this kind of “dominates” relationship?
- Fifth, we agreed to allow a recursive component model. But then the issue of being able to access the interfaces of these internal components arises.
- Finally, we only can write quantifications with the APL over the sets of components, interfaces, containers and operations. When quantifying over several sets in the same expression, we may need to express that the quantified variables are related. In the example pointcuts in this section, we opted for explicitly requiring these relationships.

#### 4.4.4 Coordination

This concern has been proposed as a new key concern candidate and the use of AOSD for coordination is studied in [16]. In this section, we will specify the coordination concern by mapping the concern to the reference architecture. The APL will be used to specify the crosscutting nature of different types of coordination aspects. By doing this, we will validate the reference architecture and identify where the reference architecture and APL should be extended (or even improved) in order to enable effective modelling of the key concerns.

In this part of the document we refer to *operations* and *signals* as defined in RM-ODP, and as adopted in the definition of Level 1 of the reference architecture. Notice that these concepts are equivalent to the concept of *messages* and *events* respectively in other component models, such as the CORBA Component Model.

##### 4.4.4.1 Coordination as an aspect

The main goal of a coordination model is to separate the coordination concern from the computational units. In the context of CBSD (Component-Based Software Development), the coordination concern can be defined as the interaction pattern that governs the communication and invocation of operations among two or more software components. Software components are not isolated entities, they communicate with different types of components following a coordination protocol. A coordination protocol can be defined as the list of operations and/or signals that a component is able to send and receive, along with a set of coordination rules that state the order in which operations and signals must be interchanged by the participant components of the interaction. A coordination protocol can be specified by a STD (State Transition Diagram) or using any other formalism.

In SOA (Service Oriented Architecture) the coordination concern corresponds to a business process or business protocol among the different components involved in a service orchestration. This information can be usually described in Business Process Execution Language (BPEL) or using XML schemas. BPEL4WS [33] provides a language for the formal specification of business processes and business interaction protocols. In this case, the description of the coordination protocol is already separated from the web service functionality.

A first step towards the separation of the coordination concern is to specify it as part of the public interface of components. To carry out an effective (re)use of COTS components, the developer should know not only the list of provided or required services, but the coordination protocol that must be followed, to avoid performing an inaccurate use of the component services. If the coordination protocol is unknown, an incorrect usage of the component services may lead the component to an error state as a consequence of receiving unexpected operations in the current state.

Some approaches propose to extend component interfaces with the specification of the coordination protocols using some formalism, which is the main goal of the semantic interfaces community. In [21] and [22] it is shown how to add protocol information to CORBA IDLs, and the sort of benefits that can be obtained from it. This work extends traditional IDLs with protocol descriptions (i.e. partial ordering of operations and blocking conditions) described using  $\pi$ -calculus. As major benefits, the information needed for object reuse is now available as part of their interfaces and more precise interoperability checks can be achieved when building applications.

Moreover, it is more important than knowing the coordination protocols that describe the behaviour of a component, to separate this information from the source code of components [23]. Usually the sending and receiving of operations is encoded as part of the implementation of component services. The information about which code will be executed when an operation or a signal arrives to a component is part of the component implementation. Choosing the suitable communication primitive in order to send a piece of information to the environment is a task for the programmer. The information about which target component(s) will receive the data contained inside an operation or a signal, it is also part of the communication primitive. Therefore, all the information related to the coordination protocol is hard coded inside components, making it impossible to perform any change regarding the coordination protocol without breaking the component code.

To illustrate how coordination and computation are usually intermingled we will use the example of an auction system. Concretely, we will focus on the interaction between the seller and the buyer in an active auction (to simplify the example we suppose that the seller has already initiated an auction by informing the system).

Usually, at the design phase, the functionality and the coordination of the system components are described in separate diagrams. In our example the UML class diagram in Figure 22 shows the provided and the required interface of the components participating in an auction, while the UML sequence diagram in Figure 23 describes the interaction protocol among them. However, when the system is finally implemented, the coordination information, shown in Figure 23, is lost.

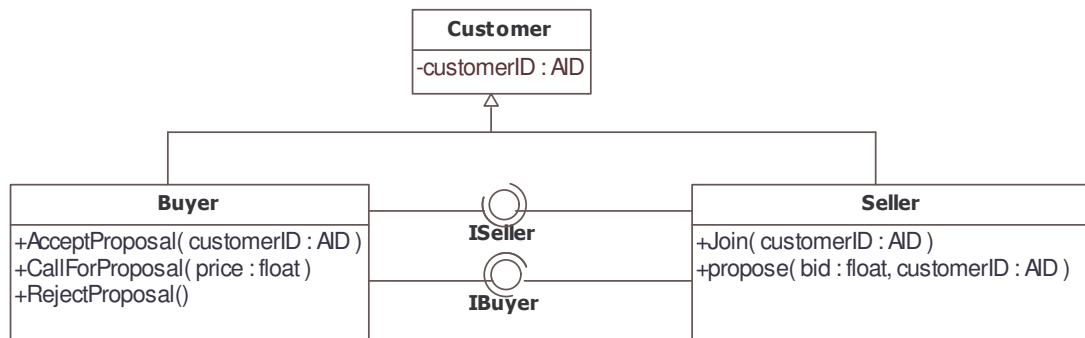
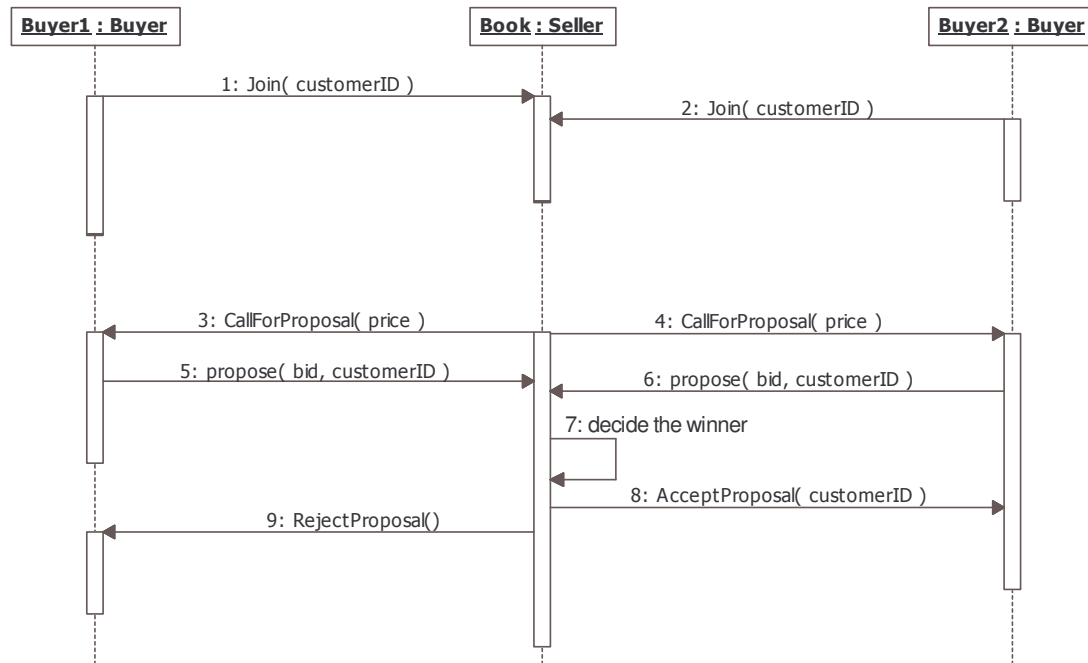


Figure 22 : Part of the design of an auction system.



**Figure 23: Buyer – seller interaction protocol (English auction protocol).**

This means that there isn't a unique component in the system where this information is located, since, as mentioned before, it is directly hard coded as part of the implementation of the `Seller` and the `Buyer` components. That is, coordination is not separated in an independent concern and crosscuts component's functionality.

As a consequence, as we will show with the following examples, there are situations that will require changing the component implementations (the `Buyer` and/or `Seller` components in our example) due to changes in their interaction protocol.

### *When an auction finalises*

First of all, in the above example (see Figure 23), when an auction finalises:

- the `Seller` component has to decide the winner bid (step 1),
- notify one of the `Buyer` components that his/her bid was the winner using the `AcceptProposal(customerID)` operation (step 2), and
- notify the rest of the `Buyer` components that his/her bid did not win using the `RejectProposal(customerID)` operation (step 3).

The issue is that taking the decision of which bid is the winner is part of the functionality of the `Seller` component (step 1), but deciding what to do after this decision is part of the coordination protocol in which the component is participating (steps 2 and 3). If the coordination protocol changes, and now only the winning buyer needs to be informed, or if all the buyers have to be notified about which buyer is the winner. With the current implementation of the application the `Seller` component would have to be modified.

However, separating coordination as a concern, the required interface of the `Seller` component would contain the `AcceptedProposal(customerID)` signal which would indicate that the buyer with ID `customerID` is the winner. The component will send this signal when an auction finalises. Then, this signal would be managed by a coordination

concern, which will encapsulate a particular coordination protocol and which will send, for instance:

- an *AcceptProposal(customerID)* operation to the winning component and a *RejectProposal(customerID)* message to the rest of them, or
- only an *AcceptProposal(customerID)* operation to the winning component, or
- an *AcceptProposal(customerID)* operation to all the buyers, where *customerID* is the ID of the winner component.

The main advantage is that the implementation of the `Seller` component does not need any modification.

### ***Public auction***

Secondly, let us suppose now that we want to develop a public auction (one in which all bids are shared among the participants) and we desire to (re)use the `Buyer` and `Seller` components in the example below (see Figure 22 and Figure 23). In this case the `Seller` component should have to notify the `Buyer` components that a new bid is proposed, and this requires modifying its implementation. Concretely, after receiving the *propose(bid,customerID)* operation invoked by *customerID* buyer, the `Seller` has to notify the rest of the buyers of the bid using the *CallForProposal(bid)* operation. Once again this is part of the interaction protocol codified crosscutting the functionality of the `Seller` component.

However, by encapsulating the interaction protocol in the coordination component, the notification would be done by the coordination concern, after intercepting the *propose(bid,customerID)* message sent by a buyer. The implementation of the `Seller` component can be reused with no change.

### ***Complex interactions***

Thirdly, another advantage of separating the coordination concern is that it is easier to control the different states of a complex interaction. Let us suppose that we want to assure that after initiating the auction (the `Seller` component sends the first *CallForProposal()* operation), any other buyer is allowed to join it. Looking at the provided interface of the `Seller` component, there is no information that helps us to know if this restriction is considered or not. In any case, the implementation of the `Seller` component may need to be modified in order to model different states during the auction. In this case, the `Seller` component has to store information about the state of the auction to consult it when a `Buyer` tries to join the auction.

However, if the coordination concern is separated in an independent component, this component manages the different states of an interaction modelling them, for instance, using a STD.

### ***Uses of the coordination concern***

The coordination concern can have different uses in a distributed component application.

**Adapter Concern** When (re)using components we can find that required and provided interfaces may not always match. Although the semantics of the operations included in the provided interface respond to the expectation of the required interface of another component, some syntactical discordance can appear. In these cases we can use the coordination concern as an adapter of the required and/or provided interfaces.

**Binding Concern** The use of the coordination concern in our reference architecture can be seen as the concept of a *binding object* in the computational view of RM-ODP. The coordination component is responsible for binding different components among them, according to a particular interaction protocol. In this case, components will interact by means of signals. This is called *compound bind* in RM-ODP (see Figure 24).

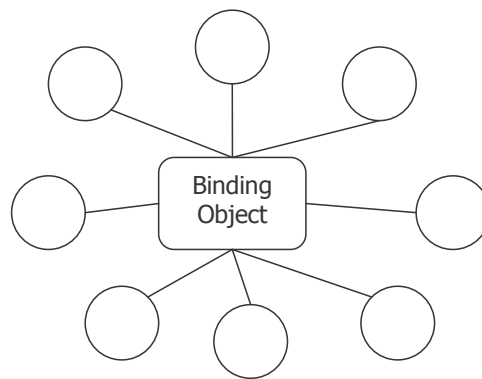


Figure 24: Compound bind in RM-ODP (extracted from [7]).

The interaction protocol encapsulated in the coordination aspect describes how and when the components interact. Its description can be very simple (being just a set of simple rules) or can be highly complex (being necessarily some formalism for its specification, for instance a STD). With this approach, components are considered entities characterised by the complete ignorance of how output operations (their required interface) influence the application execution. Thus, components can be reused in different contexts and engaged in different interactions.

**Coordination Model Concern** Components in distributed applications can interact by using different kinds of coordination models. Each coordination model has different types of properties and characteristics, which are suited to different types of tasks and environments. The main advantage of separating the coordination model is that components are loosely coupled. Some of the best known are described in [24].

- *Publish-and-Subscribe Model*. Following this coordination model, a component can send a signal meant for any component having some kind of specific service, instead of sending it to a particular component (point-to-point model). In this interaction, publishers post signals with specific “subjects” or “topics”, rather than sending operations to specific components. The coordination aspect then broadcasts the posted operations or signals to all interested components (subscribers).
- *Shared Data Space*. In a shared data space, all components read and write data, usually tuples like in Linda [25], from and to a shared space. The tuples contain data, together with some conditions. Any component satisfying these conditions can read a tuple, since tuples are not explicitly targeted.

- *Channel*. A channel is a one-to-one connection that offers two ends, its source and its sink, to components. A component can write by inserting values to the source-end, and read by removing values from the sink-end of a channel; the data-flow is locally one way: from a component into a channel or from a channel into a component. The communication is anonymous: the components do not know each other, just the channel-ends they have access to. Channels can be synchronous or asynchronous, mobile, with conditions, etc. and allow several different types of connections among components, e.g., synchronous, FIFO, etc.

#### 4.4.4.2 Mapping to the component model

The coordination concern is mapped to a component in our component model, with the following features:

**Provided Interface.** There are different possibilities to describe the provided interface of the coordination aspect, depending on the different kinds of the concern that were identified in previous section.

- The concern is going to be implemented as an adapter or a binding concern to coordinate a known set of components and for a known set of operations/signals. If this is the case, the provided interface would have the definition of a method for each one of the operations/signals the aspect is able to intercept. The parameters of these elements would be (generally) the same as the intercepted operations/signals. For instance, in order to decouple the interaction protocol of the auction example from the `Buyer` and `Seller` components the interface of the coordination aspect should be the one as illustrated in Figure 25. Figure 26 and Figure 27 show the component model for the adapter component and the binding component respectively.

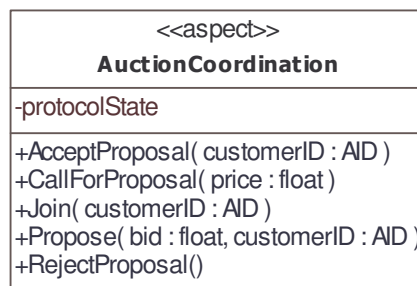


Figure 25: Example 1 of the provided interface of the auction coordination aspect.

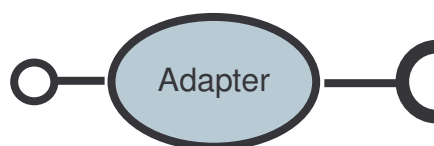


Figure 26: Component model for the coordination concerns as an Adapter.

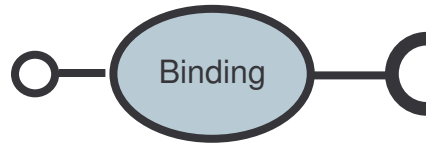


Figure 27: Component model for Adapter-Binding Coordination.

- The coordination concern is used for implementing a particular coordination model, for instance, the publish-and-subscribe coordination model. If this is the case, the provided interface of the coordination aspect would include the usual primitives used by components when they interact following such coordination model as shown in Figure 28.
- Figure 29 shows the component model for the publish and subscribe coordination model. This figure depicts not only the aspect, but also the rest of the components that participate in the coordination model. In this case, the provided interface of the coordination concern defines a *subscribe* operation that allows components registering for the events in which they are interested. And a *publish* operation that allows components notifying when a new event has been produced. Publishers and subscribers components will preferably use signals to communicate.

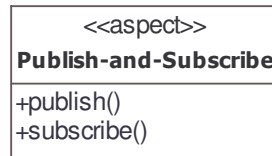


Figure 28: Example 2 of the provided interface of the publish and subscribe coordination aspect.

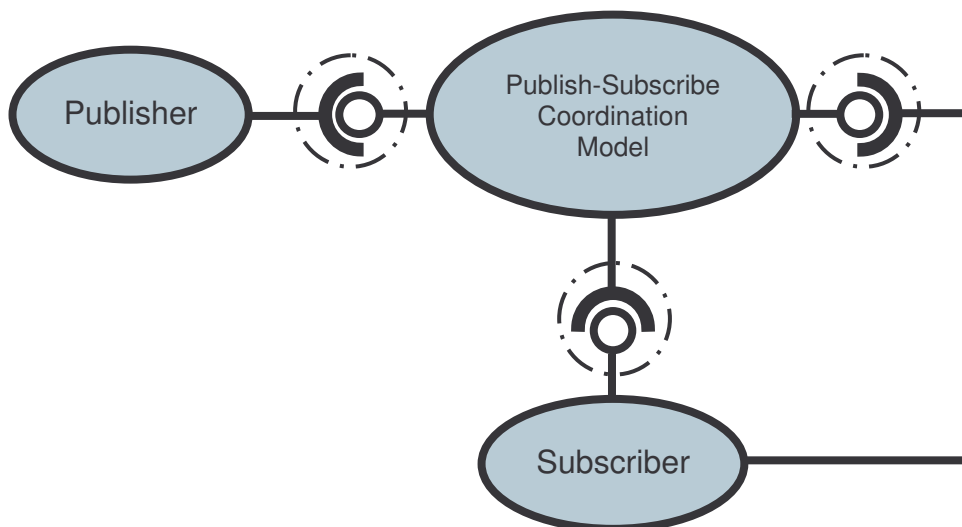


Figure 29: Component model for the publish and subscribe coordination model.

**Required Interface.** The required interface of the coordination aspect would be the same than for other components in the model. No special considerations or necessities have been identified yet.

#### 4.4.4.3 Mapping to the APL

In this section, we use the APL to specify the crosscutting nature of our coordination aspects.

##### *Adapter/binding concerns definition with the APL*

A possible definition for the pointcuts for the coordination aspect in the Auction example would be:

$$\forall c \in C, \forall i \in c.\text{interfaces}, \forall o \in i.\text{operations}$$

$$(c.\text{name} = \text{"Buyer"} \wedge i.\text{category} \in \{\text{required}\} \wedge$$

$$o.\text{signature} = \text{"Seller.*(?parameters):?result"})$$

$$\forall c \in C, \forall i \in c.\text{interfaces}, \forall o \in i.\text{operations}$$

$$(c.\text{name} = \text{"Seller"} \wedge i.\text{category} \in \{\text{required}\} \wedge$$

$$o.\text{signature} = \text{"Buyer.*(?parameters):?result"})$$

These APL expressions pick out the interaction among `Seller` and `Buyer` components. The first pointcut definition picks out the outgoing calls of the operations located on the required interfaces of the `Buyer` when the target component is the `Seller`. Similarly, the second definition picks out the outgoing calls of the operations located on the required interfaces of the `Seller` when the target component is the `Buyer`.

A possible definition for the coordination aspect to intercept the signals would then be:

$$\forall c \in C, \forall i \in c.\text{interfaces}, \forall s \in i.\text{signals}$$

$$(c.\text{name} = \text{"Buyer"} \wedge i.\text{category} \in \{\text{required}\} \wedge$$

$$s.\text{signature} = \text{"*(?parameters)"} )$$

$$\forall c \in C, \forall i \in c.\text{interfaces}, \forall s \in i.\text{signals}$$

$$(c.\text{name} = \text{"Seller"} \wedge i.\text{category} \in \{\text{required}\} \wedge$$

$$s.\text{signature} = \text{"*(?parameters)"} )$$

Notice that the APL would need to be extended for expressing this kind of pointcuts. This modification is discussed in Section 4.4.4.5.

##### *Publish-and-Subscribe Coordination aspect*

A possible definition of the pointcuts for the publish-and-subscribe signals would be:

$$\forall c \in C, \forall i \in c.\text{interfaces}, \forall s \in i.\text{signals}$$

$$(i.\text{category} \in \{\text{required}\} \wedge$$

$$s.\text{signature} = \text{"publish(?parameters)"} \vee$$

$$s.\text{signature} = \text{"subscribe(?parameters)"} )$$

This APL expression pick out the publish and subscribe signals thrown by the components participating in the publish-and-subscribe coordination model.

Notice that we have specified the use of signals in the definition of the pointcut as discussed in previous example and also in our discussion below.

#### 4.4.4.4 Mapping to the microkernel

In the following list, each of the microkernel operations is described in relation to the coordination concern.

- **LOAD** – The coordination aspect is explicitly described as part of the architecture of an application and it is loaded together with the rest of the components in the application.
- **UNLOAD** – The coordination aspect is unloaded when the application finishes.
- **START** – The coordination aspect starts when components begin communicating by sending *signals*.
- **STOP** – The coordination aspect stops when the application finishes.
- **BIND** – Bindings between the coordination aspect and the distribution aspect always occur when a coordination aspect is being used in an application.
- **INstantiate** – The coordination aspect is instantiated when the distribution concern is loaded or a component sends a signal.
- **DESTRUCT** – The coordination aspect is destroyed when the distribution concern is destroyed.

#### 4.4.4.5 Discussion

Regarding the mapping of the coordination concern to the component model, the concept of signal (or event) is highly relevant for the separation of the coordination concern. Therefore, the component model, which distinguishes between the *provided* and *required* interfaces of a component, should allow to specify for each interface if it contains *operations* and/or *signals*. This will align our model with other component models, including RM-ODP, where the definition of the interfaces of a component includes this distinction. Another example is the CORBA Component Model, in which one of the interfaces of a component is the *event interface*.

It should be mandatory to have a coordination aspect when a signal is thrown by a component. The reason is that in our model the coordination aspect is always responsible to manage such signals. It is optional to use this aspect when messages are sent. Additionally, the advice of the coordination aspect should be always invoked before those in the distribution aspect.

We have some considerations and extensions to the APL:

- Defining the APL expressions we have assumed that using “(?parameters)” the pointcut maps all the arguments of the invocation to the advice.
- The APL should distinguish between operations and signals in the definition of the component interfaces. This can be done extending the domain definitions in Section 3.4.4 of the Level 1 definition, including a new set, *S*, that is the set of signals. The signature of a signal can be specified as an operation with no target component and no return result.

- The APL should offer the possibility to express that the source component of an operation/signal (its `c.name`) is passed as an argument of the advice. Although this information may be captured using the `embed()` expression, we consider that it should be included in the definition of the APL.

#### 4.4.5 Transactions

The effective management of transactional processes is an important concern as transactions tend to permeate throughout a software architecture. Notable examples of applications that use transactions are in the domain of multi-tiered distributed systems, such as web based banking and e-commerce applications. However, even simple database-backed systems often require transaction management in order to function correctly.

Conceptually, transactions are units that encapsulate operations on a database as an indivisible block, in order to take the database from one consistent state to another. This is why they are sometimes called LUWs, “Logical Units of Work”. The classical properties of a transaction are the *ACID* properties:

- *Atomicity* implies that a transaction is a unit of operation on the database. Either all the changes made during a transaction are reflected in the database or none at all. A transfer of funds between bank accounts can either succeed or fail, but if it fails, the database is never left in a situation where one account is debited unless the other is credited, and vice versa.
- *Consistency* means that the transaction takes the database from one consistent state to another. Any constraints that hold on the database before the transaction still hold after it is completed. In particular, if a transaction leaves the database in a state that violates a constraint, that transaction will automatically be rolled-back.
- *Isolation* is the property referring to the visibility of a transaction to other transactions executing against the database. Each transaction operates as if it is the only one manipulating the database and is not aware of the existence of other concurrent transactions; one operation can never see the database in an intermediate, possibly inconsistent, state created during the execution of another operation. Perfect isolation implies that all transactions are *serialisable*. Due to performance considerations, the isolation property is often relaxed. Different levels of isolation are discussed below.
- *Durability* means that the changes made by a transaction are committed to the database regardless of any system or media failures. Technically, this is often achieved by means of a log file; a transaction is not considered to be committed before it was fully logged.

A transaction monitor, also known as a *TP Monitor*, guarantees that concurrent transactions do not work on one another’s intermediate results. TP Monitors employ concurrency control and recovery mechanisms to ensure that multiple clients can concurrently access and manipulate the database in a consistent fashion.

Once initiated by a client, a transaction can end either in a *commit* operation, or a *rollback*. The former indicates that the client considers the transaction to be successfully completed, and wishes to make all changes durable, whereas the latter indicates that the client wishes to abort the operation and undo any partial work done so far during this transaction.

The handling of rollbacks is a key part of transaction management. Rollbacks are the process of undoing modifications that were made to the database during the current transaction. This may happen if a transaction is aborted (e.g. due to deadlock), if the system fails (e.g. system crash), or if the transaction's initiator decides to abort the operation prior to its completion.

When a programmer is developing an application which uses transactions they have to write *transaction demarcation code*. Demarcation code explicitly marks the start and end of each transaction in the code and also describes what to do in case of rollback. This demarcation code presents one of the fundamental problems of transaction management in that it cannot be modularised into a single module. Viewing transactions as an aspect, we aim to move the demarcation code into a module of its own, independent of the rest of the application.

Transactions were originally designed to treat short units of work that have no inherent structure, and lock few items of data. As a consequence, transaction management is rarely suitable in applications that are outside of this domain, e.g. applications that process large units of work which take a long time to complete. Such *long-lived transactions*, which may take several seconds, minutes or even days to complete block the possibility for other transactions to be made, thus causing severe bottlenecks in performance. In order to address the issues of performance, cooperation and rollback handling, advanced transaction mechanisms have emerged. Three examples of such mechanisms are *nested transactions*, *Sagas*, and *Relatively Consistent Schedules* (RCS).

With nested transactions, a hierarchical tree structure of transactions is built. A child transaction can access data used by its parent in a recursive manner up towards the root. On committing data, the child transaction sends this data to its parent rather than to the database. If a child aborts then this does not mean that its parent is also aborted, hence providing a fine grained approach.

Sagas relax the atomicity in long term transactions by splitting them into a sequence of atomic sub-transactions that immediately commit when completed. This sequence should be entirely completed or not at all. To rollback the Saga the currently running sub-transaction is rolled back, and the work of committed sub-transactions is undone by executing a compensating transaction for each sub-transaction. Splitting the transaction into sub transactions allows other transactions to be executed decreasing the likelihood of deadlocks and improving performance.

RCS is a flexible mechanism using semantic consistency whereby the programmer instructs the scheduler how a transaction may be interleaved with other transactions, which removes isolation between these transactions. This has two main advantages in that schedules can easily be verified a priori and that an interleaving will only take

place if it is specifically permitted. This allows the programmer to selectively focus on groups of transactions to be optimised.

A variation on RCS which is common in modern database servers is the use of less strict isolation. The ISO SQL standard specifies four isolation levels:

- The *Serializable* level requires perfect isolation; two transactions may be executed serially only if the illusion of serial execution may be maintained.
- With the *Repeatable Read* level, data rows read by a transaction may not be changed (i.e., the transaction maintains locks on any row read), yet *phantom reads* may occur; a phantom read happens when a repeated SELECT statement matches, in its second execution, rows that did not exist in its first execution (this happens because *range locks* are not maintained).
- Less isolation is possible with the *Read Committed* level, where SELECT statements can only load data that was committed by other transactions; however, *non-repeatable reads* are possible. In a non-repeatable read, a repeated SELECT statement for the same row can yield different results, if the said row was updated by a committed transaction between the two SELECT operations (i.e., no row locks are used).
- Finally, the *Read Uncommitted* level practically eliminates isolation, as a transaction may read data that was updated, but not yet committed, by a concurrently executing transaction. Such a *dirty read* could load data that will eventually be rolled-back rather than committed.

Clearly, lower levels of isolation allow for higher levels of parallelism, at the price of lower data integrity. Mission-critical operations should use the *Serializable* level, whereas unimportant queries can use the faster yet less accurate levels, to the extreme of using *Read Uncommitted* for non-critical operations such as calculating statistical information.

#### 4.4.5.1 Transaction management as an aspect

Any reasonable attempt to modularise transaction management into an aspect module will assume transaction boundaries to be method invocations or field access operations; a transaction is initiated when a method begins and commits when the method completes (or conversely, rolls back when the method throws an exception). In other words, transactions are applied *around* certain joinpoints.

An attempt to initiate a transaction at one joinpoint and leave it active, to be committed or rolled-back in some other joinpoint, is certain to cause problems and will remove any pretence for obliviousness on the programmers' side. Any such scheme is equivalent to the manual demarcation of transactions.

A simple approach to applying transactions as advice will be to apply a single *around* advice to all joinpoints that require transactional protection, such joinpoints normally being method executions or field accesses. However, following the J2EE standard, we

recommend a gamut of six transaction-related advices that can be applied to a joinpoint. This is done in order to properly support the *nested transactions* mechanism.

- The *Requires Transaction* advice initiates a transaction before the joinpoint and commits it (rolls it back) after the joinpoint returns (throws an exception). However, if the joinpoint is entered when a transaction is already in session (a nested or recursive call), no new transaction will be initiated. An exception thrown by a joinpoint to which this advice is applied might abort a transaction that it did not initiate, but was in session when it began.
- The *Requires New Transaction* advice always initiates a new transaction, possibly nested within an active one (a child transaction). This advice can never cause an existing transaction to roll back; at most, it will roll back the transaction it initiated itself.
- The *Transaction Mandatory* advice is applied to joinpoints that may only occur when a transactional context is already active. It never creates its own transaction, but it throws an exception (and prevents the joinpoint from executing) if it is reached when no transaction is in session.  
If an exception is thrown during the joinpoint's execution, it will abort the active transaction.
- The *Transactions Not Supported* advice is applied to joinpoints that must execute *outside* of any transactional context. If invoked while a transaction is in session, this advice will *suspend* the transaction, and then *resume* it after the joinpoint finishes executing (regardless of how the joinpoint existed, either normally or by throwing an exception). Any joinpoint to which this advice is applied cannot affect or abort existing exceptions.
- Conversely, the *Transactions Supported* advice is applied to joinpoints that may execute within a transaction context, but they never initiate a new transaction of their own. This is different from not applying any advice to the joinpoint, since should the joinpoint terminate by throwing an exception, this will rollback the existing transaction – even if the exception will later be caught by the surrounding joinpoint, which initiated the transaction.  
For example, assume we apply the *Requires Transaction* advice to method `foo()`, and the *Transactions Supported* advice to method `bar()`. If `foo()` invokes `bar()`, and `bar()` throws an exception, the transaction initiated by `foo()` will be aborted even if `foo()` catches the exception and returns normally. If no advice was applied to `bar()`, then the transaction would have committed successfully.
- Finally, the *Transactions Prohibited* advice is applied before (not around) any joinpoint that may not be invoked while a transaction is in session. Any attempt to execute such a joinpoint while a transaction is active will cause the advice to abort the transaction and throw an exception (the joinpoint itself is not executed, and the transaction is rolled back even if the exception thrown by the advice is caught by the invoking context).

Table 1 summarises the behaviour of the six advices:

**Table 1: Transaction support advice.**

<i>Advice</i>	<i>Executed while a transaction is in session</i>	<i>Executed while no transaction is in session</i>
<i>Requires Transaction</i>	Proceeds normally. May abort the active transaction by throwing an exception.	Initiates a new transaction, and commits it upon completion.
<i>Requires New Transaction</i>	Initiates a child transaction.	Initiates a new transaction.
<i>Transaction Mandatory</i>	Proceeds normally. May abort the active transaction by throwing an exception.	Throws an exception.
<i>Transactions Not Supported</i>	Suspends the transaction, and resumes it upon completion. May not abort the transaction.	Proceeds normally.
<i>Transactions Supported</i>	Proceeds normally. May abort the active transaction by throwing an exception.	Proceeds normally.
<i>Transactions Prohibited</i>	Rolls the transaction back and throws an exception.	Proceeds normally.

The application of the transaction-control advice to joinpoints must be disjoint; no single joinpoint may be advised by more than one of these six advice types.

Some combinations of advice applications that will certainly fail at runtime might be statically detectable during program development (for example, a method to which the *Requires Transaction* advice is applied which invokes a method to which the *Transactions Prohibited* advice is applied).

To enable support for different levels of isolation, a variation of each transaction-initiating advice should be provided. This implies replacing *Requires Transaction* and *Requires New Transaction* with four advices each: *Requires Serialisable Transactions*, *Requires New Serialisable Transaction*, *Requires Repeatable Read Transaction*, and so on. This brings the total number of advice in this aspect to twelve.

#### 4.4.5.2 Mapping to the component model

Presented as an aspect, transactional support is encapsulated in a single aspect with the six (or twelve) advices named above. The transaction functionality is provided by the `TransactionManagement` component. This component includes transaction management support in the form of transaction demarcation methods: methods to initiate, commit, and rollback a transaction. The transaction advices also require that this module provides a method for detecting if a transaction is currently in session. Figure 30 illustrates the mapping of the transaction aspect to our component model.

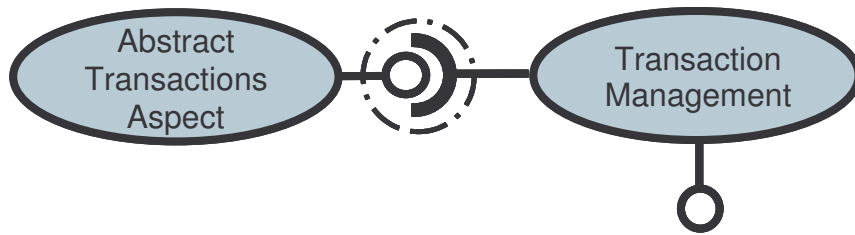


Figure 30. Component model for transaction manager

The `TransactionsAspect` is an *abstract* aspect, and provides an abstract pointcut for each of the advices detailed before. To apply transactional management to any specific component, the transaction aspect should be *subclassed* while providing concrete values to the desired pointcuts, namely the execution of transactional methods or read/write access to transactional fields (which are fields to which the persistence aspect is applied).

#### 4.4.5.3 Mapping to the APL

The base transactions aspect is an abstract aspect, and it defines no pointcuts other than abstract ones. Any application that *uses* this aspect must override these abstract pointcuts. Since not all advice will be used by every application, the APL must include the ability to specify an *empty* pointcut definition.

Other concrete pointcut specifications used by this aspect are simple method-execution and field-access pointcuts.

No control flow (e.g. `cflow`) support is required to implement the advice of the transaction aspect, since the transaction context is not detected using the execution stack; a method of the core persistence module is used to initialise the transaction, but the operations of the transaction do not occur during the lifespan of that method, but rather after it completes execution. Still, control flow expressions can be used to express statically detectable certain-failure scenarios, as described above.

#### 4.4.5.4 Discussion

The transactions aspect is simple to define, but complex to use correctly. Incorrect usage of automated transaction demarcation services, such as the ones offered by this aspect, may lead to performance problems or even to application errors. Errors can occur not only due to *lack* of transactional protection, but also due to *overuse* or *misuse* of such a protection. For example, if two methods are marked as requiring transactional behaviour in perfect (serialisable) isolation, and they both access the same two database tables but in reverse order, the result can be a deadlock, since (depending on the DBMS software in use) the transaction manager might require the table locks in reverse order for each of the two transactions.

Another delicate issue is that to take proper advantage of the transactions aspect, programmers must create sufficiently *coarse-grained* methods; otherwise, a false feeling of security might appear. Consider for example a banking application where an

“Account” component has methods for depositing and withdrawing funds. While each of these two methods will probably be protected by applying the transactions aspect, this by itself is insufficient. In particular, the transfer of funds between two accounts must also be represented as a single method, itself protected by a transaction. Otherwise, part of the transfer operation might succeed while the other fails, resulting in an inconsistent database state.

Therefore, the programmers can never be completely oblivious to the transactional protection applied to their methods, and must be careful in their code-writing to prevent such accidental problems – even through their code never directly relates to transaction-related issues.

Finally, we note that proper interoperability of the transaction and persistence aspects require an aspect precedence mechanism. Without such a mechanism, it is possible that the persistence aspect will operate first, accessing the database before the transactions aspect had a chance to initialise a transaction for this operation.

## 4.5 Summary

In this section, we described Level 2 of our aspect-oriented middleware reference architecture, which conceptually is seen as sets of aspects. By trying to map the reference architecture concepts and ideas of Level 1 to a number of key concerns we discovered some open issues that need further investigation. We have mapped a number of key concerns to our reference architecture, namely distribution, mobility, persistence, security, coordination and transactions.

We adopted a different approach for the distribution mapping than for the other five concerns, because some bootstrapping infrastructure for deployment and lookup had to be introduced to realise the distribution mapping. Each section discussing one of our concern mappings was structured according to the same template. Every concern mapping started off with a sort of introduction, titled “<concern> as an aspect” that summarised the most important findings of the key concern studies in [16]. In a second and third subsection, the mapping to the component model and abstract pointcut language (APL) respectively was presented. An optional fourth subsection contained the mapping to the microkernel. Finally, we ended each concern mapping subsection with a discussion on problems and/or open issues for that specific concern that need further investigation.

In the rest of this summary, we list the issues that were discovered with regard to the mapping to the component model, the mapping to the APL and the mapping to the microkernel.

### 4.5.1 Issues w.r.t. the mapping to the component model

While trying to map the different key concerns to the component model of the reference architecture for aspect-oriented middleware, a number of issues popped up.

- How do we express different kinds of component instantiation? There exist a number of aspect component instantiation methods. For example, one component instance may sometimes have to serve for an entire program, while

at another time an aspect component instance may be needed for each component executing at any of the join points within a pointcut.

- How do we express a kind of aspect precedence mechanism? The order in which components have to execute at the same join point can be of importance.
- We agreed to have a recursive component model. But then, are we able to access the interfaces of these internal components. And if so, how to do this?
- Our component model distinguishes between provides and requires interfaces of a component. With regard to the mapping of the coordination concern, the concept of signal (or event) proved to be highly relevant for the separation of the coordination concern. Should our component model allow specifying for each interface if it contains operations and/or signals?
- Suppose we want to make a certain aspect mandatory when a certain event is thrown by a component. How do we express this kind of constraint?

#### 4.5.2 Issues w.r.t. the mapping to the abstract pointcut language

Issues that surfaced while trying to map the different key concerns to the APL of the reference architecture for aspect-oriented middleware are listed below.

- In order to implement data-transfer optimization natively, some APL enhancements or the use of the `embed()` primitive are inevitable if you want to apply EAOP pointcut expression languages. Can we natively support this kind of optimization?
- What is the right mechanism for parameter passing and how can we best represent control flows? How do we map such a parameter passing mechanism to the existing reference architecture? For instance, when we are advising a kind of login method, we possibly need to have the given credentials available. Is it sufficient to define an aspect component instance per control flow of that login method or do we need an extension of the architecture? Another possibility can be using the `embed()` operator enclosing a complete APL pointcut.
- With regard to the security concern, we must be able to access internal state under certain circumstances. The only option here is the use of lower level pointcuts enabling us to capture points that are not exposed through public interfaces. The `embed()` operator should enable this.
- We only can write quantifications with the APL over the sets of components, interfaces, containers and operations. When quantifying over several sets in the same expression, we may need to express that the quantified variables are related. How do we handle these relationships?
- The APL should distinguish between operations and signals in the definition of the component interfaces. This can be done extending the domain definitions of the Level 1 definition, including a new set, `S`, that is the set of signals. The signature of a signal can be specified as an operation with no target component and no return result.
- The APL should offer the possibility to express that the source component of an operation/signal (its `c.name`) is passed as an argument of the advice.

Although this information may be captured using the `embed()` expression, we consider that it should be included in the definition of the APL.

## 5. Conclusion

In this report we have presented a reference architecture for aspect-oriented middleware. The reference architecture provides a common frame of reference from the perspective of a product-line approach that supports building multi-aspect applications in a distributed and heterogeneous environment. We believe the perspective of a product-line approach is the right angle from which to look at AO middleware because the survey of AO middleware [27] has clearly shown that AO middleware increases the flexibility of composition. Additionally, both functional concerns and extra-functional concerns are simply represented as separate aspects that can be dynamically deployed and composed into a distributed software application on demand. Thus, it is the nature of aspect-oriented middleware that enables the applications lab of AOSD-Europe to invest in this vision of a product-line approach by means of which families of applications can be cost-effectively and qualitatively developed and maintained.

The initial goals of the reference architecture are to provide a conceptual vocabulary for AO middleware systems, to facilitate the analysis and comparison of AO middleware platforms, to embody and encourage best practices and patterns in the design of AO middleware systems and to form the basis of a middleware implementation toolkit. An important property of the reference architecture is that it is as general and non-prescriptive as possible to maximise applicability and future-proofness. As such it is designed to be independent from any particular platform, language, or domain.

The reference architecture has been designed as a structure of three levels. Level 1 defines the general concepts and mechanisms behind the reference architecture. In essence, there are three ingredients:

- a uniform component model that is based on a few, simple concepts and that is recursive of nature, thereby supporting the construction of composite components,
- a micro-kernel that represents the primitive operations for loading, instantiation, binding and activation of components,
- and an APL that is a language-independent framework for expressing patterns of binding of aspectual components to an application.

At Level 2, some of the key concerns and new candidate concerns that need AOSD (i.e. distribution, mobility, security, persistence, transactions, coordination) are mapped to the reference architecture. For each key concern, the component model is used to design a composite component of aspects that supports the key concern. The APL has been used to specify the crosscutting nature of the different types of aspects. Finally, some (but not all) key concerns have dependencies on particular implementation strategies of the microkernel operations. By having this mapping for a representative set of concerns, we have evaluated the reference architecture and identified where the reference architecture should be extended (or even improved) in order to enable effective modelling of the key concerns. Section 4.5 gives a comprehensive overview of all the issues that have been identified.

At Level 3 existing middleware systems are analysed and compared in terms of Level 1 and Level 2 concepts. In terms of Level 1, a middleware system's component model,

pointcut language and interactions with the micro-kernel operations are studied. In terms of Level 2, a middleware system is studied as a composition of key concerns. This report only discussed Level 1 and Level 2. Level 3 is out of the scope of this report because Level 3 already touches upon validation of the reference architecture.

One of the main strengths of the reference architecture is its strong focus on language- and platform independence, genericity and heterogeneity. A weakness is that one of the goals initially set out has not yet been fully achieved. Specifically, the reference architecture does not yet embody a complete catalogue of best practices and patterns in the design of middleware. Although, at Level 2 one or more *patterns of binding* for key concerns have already been expressed using the APL, these patterns are neither complete nor representative for the whole concern domain.

Therefore as future work, we need to continuously refine and validate the reference architecture. The task on study of key concerns will elaborate the designs of *frameworks of aspects*, thereby not only validating the reference architecture, but also discovering and consolidating common patterns of binding. A second point of attention is that all the open issues, summarised in Section 4.5, need to be resolved and consolidated in a next version of the reference architecture. Thirdly, Level 3 must be instantiated with a set of existing AO middleware systems and it must be investigated into how well the reference architecture enables comparison between different AO middleware systems. Finally, through the task on incremental research concept evaluation, the reference architecture will itself be subject to evaluation for acceptance to industry. A vital issue, that has already been raised, is that the reference architecture should not conflict with existing practices that are of common use in the industry.

We believe that the reference architecture offers a foundation for the creation of frameworks of aspects that capture the common concepts and patterns of binding of a particular key concern domain. As such it eases complexity and facilitates reuse. The reference architecture also provides a foundation for representing the composition of these frameworks. As such, it provides a cornerstone for achieving the above mentioned product-line approach where a distributed application is constructed by configuring and assembling a few large-scale, composite components: the ‘application’ component and one or more aspect components that are instantiations of the framework of aspects.

## References

- [1] Object Management Group, CORBA/IIOP v3.0.3, OMG Document formal/2004-03-01.
- [2] Microsoft, .Net Home Page, <http://www.microsoft.com/net>
- [3] JBOSS AOP home page, <http://www.jboss.org/products/aop>
- [4] R. Pawlak, L. Seinturier, L. Duchien and G. Florin, "JAC: A Flexible Framework for AOP in Java", Reflection'01, pp 1-24, September 2001, Kyoto, Japan. LNCS 2192, Springer Verlag.
- [5] M. Pinto, L. Fuentes, J.M. Troya, "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development", GPCE 2003.
- [6] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.
- [7] ISO/IEC and ITU-T 10746-3, "Open Distributed Processing – Reference Model: Architecture", 1996.
- [8] A. Tanenbaum, "Modern Operating Systems", Prentice Hall, 2001.
- [9] AOSD-Europe home page, <http://www.aosd-europe.net>
- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns", Wiley 1996.
- [11] JMX home page <http://java-source.net/open-source/jmx>
- [12] MBeans home page  
<http://tomcat.apache.org/tomcat-5.0-doc/mbeans-descriptor-howto.html>
- [13] A. G. Ganek and T. A. Corbi "The dawning of the autonomic computing era", IBM Systems Journal Vol 42, No. 1, 2003.
- [14] Java RMI home page <http://java.sun.com/products/jdk/rmi/>
- [15] AspectJ home page, <http://eclipse.org/aspectj>
- [16] N. Loughran, A. Rashid, F. Sanen, B. De Win, E. Truyen, W. Joosen, A. Jackson, S. Clarke, et al, "Study of Key Concerns that need AOSD", to be made public February 2006.
- [17] P. Maes, "Concepts and experiments in computation reflection.", ACM SIGPLAN Notices, 22(12): 147-155, December 1987.
- [18] F. Ortin and J. M. Cueva, "The nitrO Reflective Platform", International Conference on Software Engineering Research and Practice (SERP), Las Vegas 2002.
- [19] J. Herrington, Code Generation in Action, Manning, 2003.
- [20] N. Loughran, A. Rashid, "Framed Aspects: Supporting Variability and Configurability for AOP". International Conference on Software Reuse, Madrid, Spain 2004.
- [21] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, A. Vallecillo. "Adding Roles to CORBA Objects". IEEE Transactions on Software Engineering 29(3):242-260, Mar. 2003.
- [22] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, A. Vallecillo. "Extending CORBA Interfaces with Protocols". The Computer Journal 44(5):448-462, Oct. 2001.
- [23] L. Fuentes and J. M. Troya. "A Java Framework for Multimedia and Collaborative Applications over a Web-based Platform". IEEE Internet Computing 3(2):55-64, March-April 1999.
- [24] G. Andrews. "Paradigms for Process Interaction in Distributed Programs". ACM Computing Surveys, 23(1):49-90, March 1991.
- [25] D. Gelernter. "Generative communication in Linda". ACM Trans Program. Lang Syst. 7(1):80-112, 1985.
- [26] R. Douence, O. Motelet, M. Südholt. "A Formal Definition of Crosscut. Proceedings of Reflection 2001". LNCS 2192, pp. 170-186, Kyoto, Japan, September 2001.
- [27] N. Loughran, N. Parlavantzias, M. Pinto, L. Fuentes, P. Sánchez, M. Webster, A. Colyer, "Survey of Aspect-Oriented Middleware". AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-10. Editor(s): N. Loughran, M. Pinto., 2005.
- [28] J. van Gurp and J. Bosch, "Design Erosion: Problems & Causes", Journal of Systems & Software, vol 61, issue 2, 2002.
- [29] A. Rashid and R. Chitchyan, "Persistence as an Aspect". In AOSD '03, ed by Aksit, M., 2nd International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, USA, March 2003 (ACM 2003), pp. 120-129.
- [30] M. Atkinson and R. Morrison, "Orthogonally persistent object systems." The VLDB Journal 4, 3 (Jul. 1995), 319-402.
- [31] M. Kircher and P. Jain, "Pattern-Oriented Software Architecture, Patterns for Resource Management", Wiley, 2004.
- [32] K. van den Berg, J. M. Conejero and R. Chitchyan "AOSD Ontology 1.0 - Public Ontology of Aspect-Oriented", AOSD-Europe Project Deliverable No: AOSD-Europe-UT-01, May 2005.

- [33] Business Process Execution Language for Web Services version 1.1,  
<http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>