# Using MDA to integrate services in component platforms

**Olivier Nano, Mireille Blay-Fornarino**

*I3S, Université de Nice-Sophia Antipolis*
*{nano,blay}@essi.fr*

**Abstract:** Each component platform (Jonas, JBoss, OpenCCM, .NET, Julia) proposes its own software infrastructure. Objects like containers, controllers, interceptors compose the architecture of these platforms. These objects play a different role in the treatment of the requests depending on the platform. To integrate a new service in different component platforms, the developer has to understand the architecture of each targeted platforms and to interlace calls to the new service with the calls of the services already provided by the platforms.
We propose to express the services integration on a behavioural meta-model independently of any component platforms. These services integrations descriptions can then be merged to deduce the interlacing of calls or to find conflicts, implying to coordinate services integrations descriptions. These abstract definitions of the services integration can then be projected into the targeted component platforms.

## 1. Introduction

Component programming allows a better code modularisation. This modularisation is mainly achieved by two aspects. Firstly by the use of interfaces and communication protocols that isolate components from one an other. And secondly by the separation of the components code and the services code (authentication, transactions, persistency). The programmer concentrates on the component code and declaratively specifies which services will be added during the deployment of the component.

Component platforms provide services to components they host. Usually we can find life cycling, activation/passivation, distribution and the ones already mentioned. To provide these services, the component platforms define a software infrastructure that will coat the execution of components. Objects like containers intercept messages intended to components and apply on them some pre and post treatments. The calls to the services are located there.

Each platform (EJB[2], CCM[1], .NET[7], Fractal[11]) proposes its own software infrastructure. Even in the case of the implementation of the same component model the infrastructures differ (cf. the two EJB implementations: Jboss[6] and Jonas[5]). These differences are more important for different component models. For example let's compare the architecture of Jonas (an EJB implementation (fig. 1)) and OpenCCM[8] (a Corba Component Model implementation (fig. 2)). These figures are simplification of the real infrastructure of these two platforms, but still expose the differences about how the services are handled. In the Jonas infrastructure the services are linked sequentially, whereas in the OpenCCM infrastructure the sequencing is imposed by the coordinator.

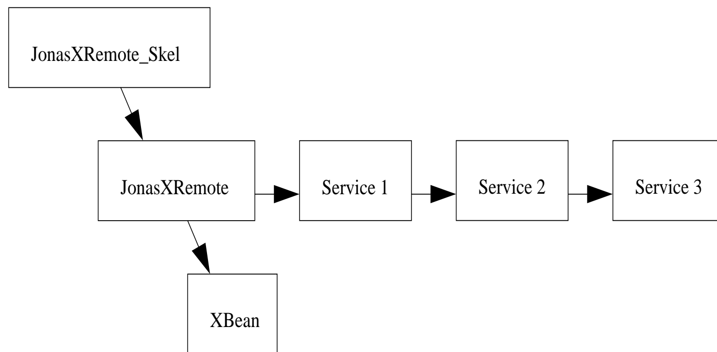


Figure 1

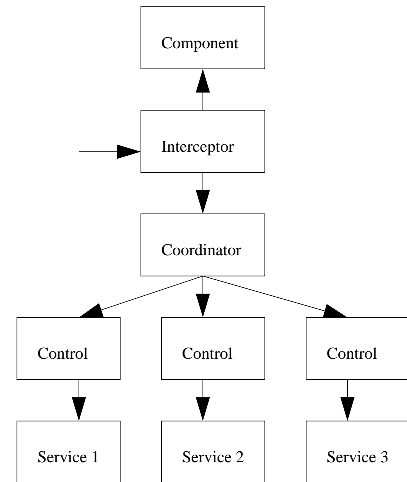


Figure 2

This heterogeneity of infrastructures makes the development of new services harder. Indeed to add a new service, we have to know what are the details of the infrastructure, where to insert the code to call the new service. We have to understand the interlacement of the existing services to keep the correct composition semantic. The code of the interposing objects is often generated (following the contract of the component). In this case we have to modify the generator in order to generate the correct interposition code that includes the calls to the new service.

In our approach we consider a service as a set of components and rules specifying how and when to call this service. To use a service, a component has to call the correct methods (in the correct order) on the service. For example to be protected by a security service, a component has to ask the security manager (which is part of the service) if a call is authorized or not and take a decision on what to do next upon the answer of the security manager. We don't need to know the platform to specify this scenario. So we consider in our approach that the introduction of code to call a service is independent of the platform.

We call 'integrator' a service integration description. An integrator is a set of operations to do on the behaviour and on the structure of a component to integrate the calls to a service.

In this paper we propose to integrate services in component platforms by basing our integrator on an execution meta-model. This meta-model allows us to take a description of a service integration and to project it (generate the appropriate code) in different component platforms. In the first part we detail the meta-model and the set of operations we allow on this meta-model to introduce calls to a service. Then we show some projections and compositions properties and we conclude.

## 2. Meta-model description

To answer the problem of integrating services in different component platforms we propose the use of integrators that are based on an execution meta-model and a set of rewriting rules. The choice of an execution meta-model that describes the execution of a message by a component is due to the heterogeneity of the infrastructures. A structural meta-model would

have been hard to exhibit because of the too many different objects in the infrastructure of the component platforms.

Our approach is an MDA[9] (Model Driven Architecture) approach to services integration. MDA claims that the development process must be thought at a higher level. So a model is developed in a particular meta-model. Then we can transfer this model onto other meta-model (by transformation) or project this model into “real” platforms (code realisation). In our context we use a meta-model as a base for the description of services integration. In the MDA vocabulary our meta-model is referred as a PIM (Platform Independent Model). Then we project integration descriptions into component platforms, referred as PSM (Platform Specific Model).

An integrator (a service integration description) is formed by a set of operations. These operations represent the transformations to apply to a component (infrastructure included) to introduce calls to the service. To abstract our self from a particular platform, operations take place upon the execution meta-model that derivate from the work of CODA[10]. The meta-model provides a basic plan (that can be extended). The basic plan is composed of the following meta-objects: Send, Accept, Receive, Execute, SendBack, Return.

Those meta-objects are linked together with the following set of Prolog rules. We use Prolog rules to describe the link between meta-objects in order to detect inconsistency between the extension of the basic plan and the integrators that use the basic plan as reference.

- Send(m) :- _send(m,m’) ; Accept(m’)
- Accept(m) :- _accept(m,m’) ; Receive(m’)
- Receive(m) :- _receive(m,m’) ; Execute(m’)
- Execute(m) :- _execute(m,m’) ; _exec(m’,m’’) ; SendBack(m’’)
- SendBack(m) :- _sendback(m,m’) ; Return(m’)
- Return(m) :- _return(m,m’)

The parameters of the Prolog predicates are from the type ‘Message’ which contains the signature of the methods, parameters value and the return value. Predicates preceded by an underscore express the modifications operated on the message. For example, the “_execute” will modify the message to affect the return value, whereas “_receive” in a majority of platforms will not modify the message.

The following figure shows the sequencing between the meta-objects and localises meta-objects on the transmitter of a message and the receiver.

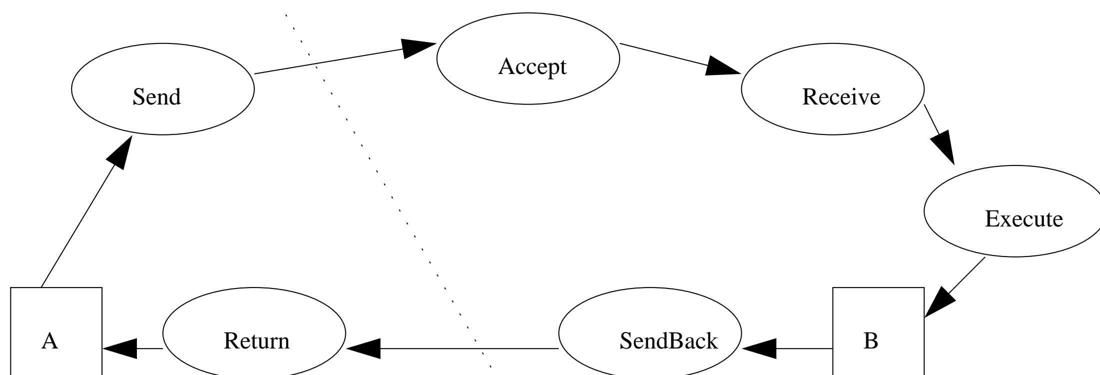


Figure 3

Based on this basic plan, we are now able to define how to integrate services, without knowing the targeted platform, automatically merge them, and report conflicts when needed. However, we will also show how this basic plan can be modified and which problems can occur.

# 3. Integrators description and projection

Integrators specify how to integrate a service with a set of rewriting rules and operations occurring on meta-objects.
A rewriting rule is composed of two parts. The left part is the message to rewrite and the right part is the transformation of the message. We can have delegation, sequencing, concurrency, and conditionals. We will not give much more details here; we will describe the rules as necessary when used. More on rewriting rules can be found in [3]. These rules have some interesting properties like composition (with explicit failure case) and associativity (the composition result does not depend of the order of the rules).
The structural operations allow the modification of variables and methods as a set (adding, removing).

## *3.1. Using the basic plan to add and compose data and behaviour for services integration*

As examples we show how to integrate call counters. The semantic of the call counter integrator is to count every call to the component even the ones that will be rejected. Another counter integrator aims to count every call received by the component (not the rejected ones).

```
Integrator CallCounter (Component c1){
  [Accept] int nbCall = 0;
  c1.* [Accept(m, m')] :- nbCall++ ; _accept(m, m').
}

Integrator ReceivedCallCounter (Component c1){
  [Receive] int nbCall = 0;
  c1.* [Receive(m, m')] :- receive(m, m'); nbCall++ .
}
```

Figure 4

**Composition:** The integration of these two counters on the same component is easy because they don't occur on the same meta-object.
**Projection:** At projection time, according to the targeted platform, the advice on meta-objects will be splitted in different entities and in different portion of code. In Jonas, according to the deployment files, the integrators are composed and the corresponding services calls are generated in the expected order in the interposition object. In Open-CCM, they will be projected in different entities (interceptor and controller). In consequence, the variables must be renamed when projected in the same object as in the case of Jonas.

Let's take another example: the integration of a notification service. We have chosen to notify after the reception of a message for the "Notify" integrator and after the acceptation of a message for the "NotifyCall" integrator. Other choices (before accept allowing to notify the arrival of a message (not yet unmarshalled), before its execution, and so on) are possible.

```
Integrator Notify (Component c1; Channel c){
  c1.* [Receive(m, m')] :- _receive(m, m') ; c.notify(m').
}

Integrator NotifyCall (Component c1; Channel c){
  c1.* [Accept(m, m')] :- _accept(m, m') ; c.notify(m').
}
```

Figure 5

**Composition:** If the "ReceivedCallCounter" and "Notify" integrators are plugged on the same component a merging has to be realized because the rewriting rules occurs on the same meta-objects. Due to formal rules of merging [3], there are no conflicts. It will result in the following rule:

c1.* [Receive(m, m')] :- _receive(m, m') ; c.notify(m') // nbCall++ .

**Projection:** For referencing the channel at projection time, the user will have to complete properties and deployment files, specifying which channel to use. Integration of such a service could result in the extension of components interfaces, for example to allow inscription of listeners, modification of the channel. As we want to focus in this short paper on composition and projection we don't discuss it anymore here. However, we can notice that, such paradigm implies to modify the contract published by the component in order to integrate such a service (as for transaction and life cycle services).

Until now, we have defined simple integrators that don't need to modify the meta-model. We will now focus on integrators that need such feature. We can distinguish two kinds of modification on the basic plan: one that modify the arguments of the meta-objects and one that modify the sequencing between meta-objects.

## *3.2. Modification of the arguments in the meta-behaviour*

Let's take an example for integrating encryption in our application. The figure 6 defines the corresponding integrator.

```
Integrator Crypting (Component c1, CryptingComponent cc){
  c1.* [Send(m,m')] :- m"=cc.cript(m); _send(m",m').
  c1.* [Return(m,m')] :- _return(m,m"); m'= cc.decrypt(m").
  c1.* [Accept(m,m')] :- _accept(m,m"); m'= cc.decrypt(m").
  c1.* [SendBack(m,m')] :- m"=cc.cript(m); _sendBack(m",m').
}
```

Figure 6

In this example, the value returned by the different meta-objects is modified. So this integrator modifies the basic plan.

**Composition:** These modifications are compatible with the other integrators because they don't share any information, but with the notifyCall integrator.
Indeed the value returned by the accept meta-object is modified by the crypting service. Do we have to notify the crypted message or not? We have chosen to not automatically consider modification of the basic plan as delegation and then to notify the scripted message. However a warning will be generated at merging time. The service provider can then express modification of the basic plan as delegation, writing:

c1.* [Accept(m,m')] :- _delegate(_accept(m,m"); m'= cc.decrypt (m")).

The composition of "Crypting" and "NotifyCall" integrator will then generate the following rule:

c1.* [Accept(m,m')] :- _delegate(_accept(m,m"); m'= cc.decrypt (m")); c.notify(m').

**Projection:** Such modification of the basic plan can result in modifying the 'Message' type. In a first time, we limit such changing in subclassing. According to the projection, the crypting component could be shared or copied on different site. Such information are given in deployment file.

## *3.3. Modification of sequencing between meta-objects*

Some services integration needs to modify the sequencing between meta-objects. For example, integration of a security service should be defined as forwarding request to the SendBack objects when a message has to be rejected for a security reason.

Figure 7 shows such an integrator.

```
Integrator Security (Component c1; SecurityManager s){
  c1.* [Accept(m, m')] :- _accept(m, m') ;
                          if (s.checkSecurity(m'))
                              _next(m').
                          else
                              setException(m',m","SecurityException");
                              SendBack(m'').
}
```

Figure 7

**Composition:** As no other integrators have specified modification of message arguments before end of call (assuming crypting as been defined using delegation), this service integration doesn't conflict with any other integrator.
If the integrator "NotifyCall" is merged with "Security", the call will be notify even in case of security reject, whereas with "Notify" integrator, there will not be notification in case of reject.

However, we have simplified in this example, the integration of security service. Another integration of security service would consist in adding security information in the message before sending it. In such a case, a conflict with crypting integrator is detected because the same arguments are modified in a non deterministic way. Must the crypting of a message be executed before or after adding security properties? The service provider will then have to explicit an order defining a new integrator modifying in a consistent way the basic plan.

Modifying sequencing can consist in adding new meta-objects such as management of messages queues. We consider the need to introduce a new meta-object only when operations of integrators are expressed on this new meta-object.

## 4. Conclusion and perspective

We have shown very quickly how our approach let us describe services integration based on a meta-model. The meta-objects of the meta-model are linked using Prolog rules that allow extension of the meta-model. Based on this meta-model we use integrators that are set of rewriting rules and operations. Those rules describe what behaviour needs to be change in order to make the calls to the services or to extend the meta-model to fit their need (for example changing the order of the meta-objects).
We also have shown how composition mechanism allows us to compute composition or to detect conflict. Some elements and problems occurring when projecting integrators have also been discussed.
At present time, we are using a structural meta-model of each platform to validate the projection of integrators on these different platforms. Next step is generating real code.

## Bibliography


[1] OMG *CORBA Component Model (CCM) Technical White Paper* http://www.omg.org/
[2] Sun Microsystem *Entreprise Java Bean Specification* http://java.sun.com/products/ejb/
[3] Laurent Berger *Phd Thesis* Rainbow 2001, University of Nice Sophia Antipolis
[4] OMG *Meta-Object Facility Specification 1.3* http://www.omg.org/
[5] ObjectWeb *Jonas* http://www.objectweb.org/jonas/
[6] Jboss http://www.jboss.org
[7] Microsoft *.net* http://www.microsoft.com/net/
[8] ObjectWeb *openccm* http://www.objectweb.org/openccm/
[9] MDA. *White paper, Draft 3.2* http://www.omg.org/mda/papers.htm, November 2000
[10] J. Mc Affer. *Meta-level programming with CodA* In ECOOP'95. SpringerVerlag, August 1995