Component-based Open Middleware supporting Aspect-oriented Software Composition

Bert Lagaisse & Wouter Joosen

Dept. of Computer Science, K.U.Leuven, Belgium,
{Bert.Lagaisse, Wouter.Joosen}@cs.kuleuven.ac.be

Abstract. State-of-the-art middleware for component-based distributed applications requires openness to support a broad and varying range of services. It also requires powerful and maintainable composition between application logic and middleware services. In this paper we describe DyMAC (Dynamic Middleware with Aspect-Components), a component and aspect-based middleware framework that supports component-based development of middleware services and offers the power of aspect-oriented composition to connect the application logic to the middleware services. We discuss the issue of a lack of expressive power in the contracts of components and aspects when combining component-based and state-of-the-art aspect-oriented development. We describe how the DyMAC framework offers a component model that solves this problem with aspect integration contracts.

1 Introduction

Software systems nowadays often have a complex distributed architecture. Non-functional requirements like availability or security therefore involve complex support based on distributed algorithms. A typical example is a large-scale distributed application with distributed transactional behavior and a centralized authentication server. The goal of a middleware layer is to isolate this complex support from the functional application logic. We focus in this paper on component-based systems that offer support for designing the application logic of distributed applications. An example of such a component framework is Enterprise Java Beans [11]. The middleware layer we envisage for such a component framework is a set of services that supports the implementation of the non-functional concerns. Our DyMAC framework offers support for two important challenges that state-of-the-art commercial middleware layers still are troubled with.

1. First, the services offered by current middleware layers and platforms are often a closed, limited set. They are not, or only in a limited way adaptable or extensible. Such middleware can be seen as a kind of black box [3]. But the different requirements of software developers towards the middleware layer are often application specific or beyond the provided services of the middleware. This requires that the middleware is extensible with application specific middleware services. But also the different requirements of the

simultaneous end-users of the application require new and potentially concurrent versions of certain middleware services [16]. Also updates of existing services of the middleware layer occur frequently. All these new requirements involve adaptability and extensibility of the middleware layer. It has to evolve from a black box to an open framework where middleware services can be adapted and added.
2. The second problem is situated in the composition of the application logic with the middleware services. The composition logic for services like transactions and security is strongly intermixed with the application logic. The composition problem with such concerns is often referred to as the crosscutting concern problem [4].

Both component-based as well as aspect-based software engineering techniques can contribute to a solution. The first problem can be addressed by integrating properties of component-based software development (CBSD)[1]. The second problem can be tackled by applying concepts from aspect-oriented software development (AOSD)[4]. AOSD is a promising technology for the problem of crosscutting concerns. Middleware and non-functional development concerns are part of the key application domain of AOSD research [9] [10]. In this paper we discuss the DyMAC middleware framework that offers a solution for the challenges mentioned above by combining the advantages of CBSD and AOSD.

The paper is organized as follows: in the second section we summarize how aspect-oriented software design and component-based software design can contribute to a solution for the challenges we mentioned above. In the third section we discuss the problems that are introduced by combining the two software development paradigms. The fourth section illustrates these problems with an example. In the fifth section we describe our DyMAC framework. In the sixth section we compare our solution with the related work in the research domain. Finally, we conclude.

2 The Promise of Integrating Advantages from CBSD and AOSD

In this section we summarize how CBSD and AOSD can contribute to a solution for the challenges we mentioned above. First we explain how component-based techniques can offer extensibility and adaptability of the middleware layer. In the second subsection we explain how aspect-based software composition can contribute to the problem of crosscutting composition logic.

2.1 Component-based Open Middleware

Our first goal is to define a modularly adaptable and extensible architecture for middleware platforms. This includes a definition of the best unit of modularity for a middleware service. It should be possible to make a middleware service deployable and reusable as one software unit. Component-based software development brings a unit of modularity that can achieve our first goal. In [1] a software component is defined as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. A component-based approach to a middleware platform, where middleware services are modularized as software components, meets the requirements of extensibility and adaptability. Middleware services can be developed by third parties as components and can be deployed into a middleware framework. Composition of application logic with the middleware services (which are components) can be realized using the connectors of the provided interface.

However, the connectors of the provided interfaces of components are often methods. This is a consequence of the object-oriented design on which a lot of component frameworks are based. The calls to the provided methods of the middleware service are scattered throughout the application logic, and this causes the problem of crosscutting composition logic.

2.2 Aspect-based Middleware

Aspect-oriented software design is about modularizing crosscutting concerns. Aspects are first class entities that encapsulate a certain behavior (often called advice) and also the instructions on where, when and how to invoke this behavior [4]. Aspect-oriented programming languages offer special programming constructs to specify well localized places in the structure or execution flow of an application. These places are called join points and depend on the main decomposition paradigm of the application. In an object-oriented application, join points can be method definitions, method calls, access to private class members, constructor calls . . . The programming constructs that can define a set of those join points are called pointcut designators. These pointcut designators are the key for enabling the modularization of crosscutting concerns. They provide a way to talk about doing something at many places in a program with a single statement (also called quantification [6]).

An aspect-based approach for the composition of application logic and middleware services offers a solution for the problem of crosscutting composition. It is even possible to manipulate internal application logic and application state, e.g. by accessing and modifying internal class members. From one point of view this could be interesting, because middleware services sometimes need access to the internal state of the application logic, e.g. for persistence, state synchronization in load balancing systems or state transferal in fail-over systems. But from another point of view, access to the whole internal structure of an application breaks encapsulation and can cause a lot of unforeseen problems, which are discussed in the next section.

3 Technical Challenges When CBSD Meets AOSD

A lot of recent research in aspect-oriented software development is situated in the domain of the integration of AOSD and CBSD (Caesar [14], JasCo [13], JAC [12], JBoss/AOP [15]). This integration of AOSD and CBSD is twofold.

1. A first facet of the integration is integrating AOSD into CBSD. This includes offering support for aspect-oriented composition in a component-based system.
2. A second facet of the integration is applying the principles of CBSD onto AOSD software modules. Aspects itself should be handled as components.

In this twofold integration, aspects evolve into a concept of software modules combining the advantages of components and aspects. We shall call these new software modules aspect-components. A typical form of aspect-components is that they encapsulate the advice of an aspect. The interface of an aspect-component provides connectors that make it possible to superimpose the advice on join points in the base components, which are often object-oriented. The kind of advice that the aspect-components can provide is before-advice, after-advice and around-advice [5]. The actual composition of the base components with the aspect-components is specified separately in the composition logic that composes the different components into an application. In this composition logic the base components and aspect-components are connected using pointcut designators that define the set of join points where the aspect-component is superimposed. In the remainder of this section we first discuss the problems that occur when combining the advantages of components and aspects into this new software module.

When composing multiple software components, one of the important issues is managing interference. This means one needs to express and control which modules may use and affect each other. In an object-oriented or component-based software design, each artifact can be equipped with a contract that specifies the provided functionality and the needed (required) functionality that describes the dependencies of a component on other components. In principle, correct behavior can be guaranteed if a component has been designed defensively and if it strictly implements its contract. When aspect-oriented composition is applied, this is no longer guaranteed. The composition of a component with an aspect can cause a component to no longer meet its contractual obligations.

We observe that the state-of-the-art notion of a contract is no longer sufficient in an aspect-oriented programming environment. When a component is composed with an aspect by means of superimposition, there is no expressive power to specify the following:

1. The component must specify what the component provides towards the aspect, i.e. which interference is permitted from certain (types of) aspects. Aspects are often services that are orthogonal with the components functionality, and therefore, the component's contract and provided interfaces are not always suitable for composition with an aspect. Therefore the contract of the component needs to be extended with the required expressive power about composition with aspects.
2. An aspect must specify what the aspect requires from the components it is applied to and which behavior it provides. This also includes in which way it affects those components.

These two facets of the lack of expressive power are explained hereafter. The most important consequences of these shortcomings are also shortly discussed.

The first lack of expressive power : the component contract. The first lack of expressive power is problematic in combination with certain join point models of aspect-oriented technologies. The join point model is the set of possible places in the structure or execution flow of an application that can be localized by the aspect language to apply certain behavior. In current state-of-the-art aspect-component technologies, we can distinguish two approaches to join point models:

1. Some aspect technologies allow complete, uncontrolled access to the whole internal implementation of the (component-based) application logic, overriding all scope modifiers and breaking encapsulation. This approach neglects the provided interfaces of the component because of the orthogonality of the component and the aspect. This can lead to uncontrolled semantic interference. This uncontrolled semantic interference of an aspect with the base component can cause undesirable exposure and modification of data and undesirable exposure and modification of behavior. A more detailed illustration of these problems is elaborated in [17].
   A second problem with providing the whole implementation structure of the component as an interface towards aspects is that it also makes an aspect too strongly tied to the component and therefore reusability of the aspect is compromised. It is clear that the notion of provided interface towards aspects must respect a certain form of encapsulation to achieve reusable aspects and adaptable application components.
2. Other aspect technologies limit the join point model to the interface of a functional component that is provided towards other functional components (e.g. in JasCo [13] only public methods of a Java Bean are a possible join point for aspects). From our point of view, where aspect technology is used for composition with middleware services, this approach is not powerful enough. Because, as mentioned above, middleware services sometimes need access to the internal state of the application. An example of this need is illustrated in the next section.

The second lack of expressive power: the aspect contract. State-of-the-art aspect technologies do not offer the possibility to contractually specify an aspect. There is also no clear notion of what really defines the interface of an aspect. This lack of expressive power is problematic in order to obtain a notion of aspect-component. An aspect should be able to include in its specification what it requires from other components, other aspect-components and the underlying platform. The specification should also include what functionality the aspect provides and in which way it affects the components it is composed with.

The scope of this paper is the composition of a component with an aspect-component and hence we focus on the specification of aspect-components concerning their requirements towards the base components and how they affect those components. In the example in the next section we illustrate these two needs of expressive power.

4 Illustration

To explain the problems above we illustrate them with a rather pedagogical example. Suppose that an entity *person* is the key abstraction in a software system. A person is uniquely defined by his social security number and has a name and a birth date. The software entity *person* also provides an inspector isAdult to check if the person is an adult. Because of privacy reasons a person object should never expose its age or birth date. But it is a necessary property to know whether a person is an adult. The birth date also needs to be stored in a database. If the persistence service is delivered as an aspect, then the persistence aspect needs access to the birth date property, while other software entities should not be able to access this property. The code of the person and the persistence example is shortly illustrated below in Java and pseudo-Aspect/J [5]:

```
public class Person{
 private Date birthDate;
 private String name, ssn;
public Person(String ssn, String name, Date birthDate){
//initialization}
private void setBirthDate(Date bd){}
private Date getBirthDate{...}
public void setName(String name){}
public String getName(){...}
public String getSsn(){...}
public boolean isAdult(){
 // derived from birthDate}}
```

```
Aspect PersonPersistence{
//on constructor execution insert into database
after(Person p): execution (Person.new(..)) && this(p){
        DataBase.insert(p.ssn, p.name, p.birthDate); }
//after mutator execution update database
after(Person p): execution(* set*(..)) && this(p){
        DataBase.update(p.ssn, p.name, p.birthDate);}}
```

The contract of the person class certainly specifies that it provides the *isAdult* functionality. Towards other modules the birth date property remains hidden. However, this property has to be exposed towards the persistence aspect, because that aspect requires *person* to expose encapsulated state that needs to be persistent. Therefore the contract of the persistence aspect must specify that it requires access to the encapsulated (i.e. private) state of a person object. The aspect also needs to specify how it affects the state: will it inspect and/or modify the state.

This section described an example that illustrated the lack of expressive power in the specification of the functional components as well as in the specification of the aspect. In the next section we describe how the component model of DyMAC offers the kind of component types to support aspect-oriented composition. We also describe how the component model offers the expressive power in the specification of components and aspect-components to tackle the problems we discussed.

5 DyMAC: Dynamic Middleware with Aspect-Components

DyMAC is an initial step in our search for a component-based open middleware framework with support for aspect-oriented composition. In this section we first describe the structure of DyMAC applications and the different abstractions into which a DyMAC application can be decomposed. In the second subsection we explain how we applied the principles of component-based software development to those abstractions mentioned in the previous subsection. In this way we achieve component-based building blocks for applications. We also discuss the specification of the components that relates to aspect-oriented composition with other components. Next, we explain how applications can be composed out of those component-based building blocks and how aspect-oriented composition is supported.

5.1 Structure and Overview

The top-level architecture of a DyMAC application can be described as a distributed and layered architecture. As mentioned in the introduction and motivation, the domain of our research is middleware for complex distributed applications. A DyMAC application consists of different subsystems that are running on different nodes in a network. A second property of the top level architecture is its subdivision into two layers: A functional layer on top and a middleware layer underneath. The functional layer contains the core application (or business) logic. The middleware layer offers non-functional services. In [2] a layer is defined as a coherent set of related functionality. In a strictly layered structure, layer n may only use the services of layer n-1. In practice this structural restriction is often lessened; Layers are often designed as abstractions that hide implementation specifics below. This latter approach is also what applies to the middleware layer in DyMAC. Sometimes application specific information is needed from the functional layer towards the middleware layer (recall the person persistence aspect). This can cause up-calls from the middleware layer to the application logic, which breaks the restriction of strictly layered structures.

Each layer in the architecture of a DyMAC application further decomposes into abstractions that are the basic building blocks for the applications. In the remainder of this subsection we describe these abstractions and their main function. In the next subsection we elaborate on how these abstractions can be specified as components to achieve a component-based decomposition.

Functional Layer Decomposition. The functional layer contains two kinds of components: functional components (abbreviated to *funcos*) and client components (shortly called *clients*).

A funco abstracts a key concept of the functional domain. It provides a constructor to instantiate objects that can have a certain state and that provide certain operations. An object of a functional component can send a message to

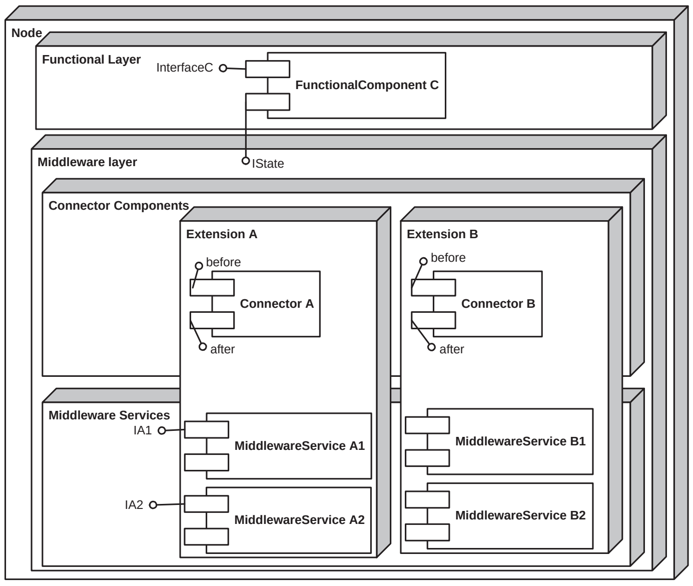

Fig. 1. Different component types in the DyMAC framework

another object of any functional component to invoke an operation. That other object sends a return message with the result of that operation.

Client components are a special kind of funcos. They only provide one operation: an entry point for starting the execution of an application.

Middleware Layer Decomposition. The middleware layer consists of a collection of middleware extensions that offer non-functional services to the functional layer of the application. The middleware layer has a 2-layer architecture. On the lowest layer it has a service layer, providing the different middleware services, on the highest layer it contains connectors, which are used to connect the functional components to the middleware services. Thus, a middleware extension typically consists of a collection of connectors and middleware services.

The service layer is decomposed into *middleware service components*. These components are abstractions of the different non-functional services in the service layer. Alike functional components, middleware services can be instantiated and can have state and behavior. Possible examples of middleware services are an encryption service or an authentication service.

The connector layer is decomposed into *connector components*. A connector component encapsulates the (otherwise crosscutted) calls to the middleware services. The state of the funcos and the runtime arguments of invoked behavior on funcos are possible arguments of the connector's invocations to the middleware services. Therefore a connector component has to be able to inspect and modify the state of a component in the functional layer, but it also has to be able to inspect and modify the messages that are sent between the components in the functional layer. The connectors use the middleware service components to apply the non-functional services to the functional components. The connectors can intercept any message that is sent and add behavior before and after they forward the message to its destination. They can also alter the message or even block it. The technique of interception is a widely used mechanism to achieve aspect-oriented composition. Further is illustrated how these connectors provide a mechanism for quantification.

Decomposition into extensions. The middleware layer is decomposed into middleware extensions. These middleware extensions contain a set of caller-extensions and callee-extensions. Caller and callee refer to the sender and receiver when a message is sent between two functional components.

– Caller extensions itself consist of a collection of connectors that can intercept outgoing messages of funcos and a collection of middleware service components needed at the caller-side.
– Callee extensions itself consist of a collection of connectors that can intercept incoming messages and a collection of middleware service components needed at the callee side.

5.2 Component Types

Each abstraction in the framework has to be modularized in the form of a component, i.e. a unit of composition with contractually specified interfaces and explicit context dependencies only. Each abstraction should be deployable independently and can be subject to composition by third parties. We believe only strict compliance with Szyperski's definition, which contains the basic principles to achieve a true component-based architecture, can eliminate today's problems that are involved with aspect-oriented composition. In the description of the component model we will focus on the specification of the functional components and middleware extensions as units of composition with contractually specified interfaces and explicit context dependencies. Especially we will elaborate on the provided interface of a functional component towards a middleware extension and the dependencies of a middleware extension towards a functional component.

Functional Components. A funco abstracts a key concept of the functional domain. We discuss the interfaces and the contracts of a functional component and especially focus on the interfaces and contracts towards the middleware extensions. Because the description of client components is analogue we will not elaborate on them.

Requirements and provisions towards other functional components. As a component a funco has to specify its provided ans required interfaces. The provided interface specifies the operations that it provides towards other components in the functional layer. This provided interface consists first of a specification how to instantiate the component, and secondly it contains the provided methods on instantiations of the component. The required interfaces are the dependencies of the component. They specify the operations that are required of other functional components in the system.

Requirements and provisions towards middleware extensions. The aspect integration contract of the functional component specifies what its requirements and provisions are towards middleware extensions. This specification contains where the functional component requires, allows or denies interference of middleware extensions.

First a functional component specifies which middleware extensions it requires: e.g. a transaction around some of its method-implementations. So this part of the contract specifies which middleware extensions the component needs to function properly. These required extensions are typically needed by the implementation of the component. To avoid intermixing non-functional development concerns in the implementation of the functional component, these concerns are specified in the requirements part of the aspect integration contract. In other aspect technologies, the concept obliviousness [6] is often used to argument that non-functional development concerns should be completely separated of the functional components. In case of required services to function properly, keeping the whole functional component oblivious to this need would mean an essential deficit in the specification of the component. We believe that the concept of obliviousness of non-functional middleware services only applies to the implementation of the component, and not the specification of it. Of course, non-functional services that are not required to function properly should be kept oblivious of the whole functional component: implementation and specification.

The interface that a functional component provides towards the middleware layer underneath is a little more complex. It contains the incoming and outgoing messages that can be inspected and modified, and also the different members of the state that can be inspected or modified. These two parts of the provided interface need some explanation:

1. The provided interface towards the functional components mentioned above defines the collection of incoming messages. The required interfaces define the collection of outgoing messages.
2. The state of a funco is defined by the properties of the component. These properties are defined by a get and set operation that access the internal representation, which is one or more private class variables. Using properties to decouple the state of a funco from the actual representation allows changes to the representation without affecting the provided state members. These state properties are not directly accessible by the middleware extensions, but all funcos provide an interface towards the middleware layer underneath to inspect or manipulate the state of a funco. This interface is a reflective backdoor/callback interface for the connectors in the middleware layer. It defines operations for inspecting the state and to modify the state. This enables decoupling of the middleware extension from a specific functional component

and makes it reusable for other applications. Providing this generic interface to access the state also restricts the access of the (aspectual) middleware extensions to the internal part of the functional component that the middleware extensions actually need to access. This is a strongly restricted interface in comparison with some aspect technologies that provide the whole implementation structure of the functional component (E.g. The implementation of the operations). When comparing it to the more restrictive aspect technologies, that do not provide a way to access the internal state of a component, this approach certainly offers advantages.

The allowed interference (state inspection and modification and behavior inspection and modification) can be specified in two ways.

1. For each middleware extensions and for each of the funco's members (behavior or state) it can specify if inspection or modification is allowed. This first approach was explained in detail in [17]. It offers the most detailed possibility to control the interference by middleware extensions but it does limit the extensibility of the application. It also makes it impossible to keep certain middleware services oblivious from the functional component. Therefore a more generic way of specifying interference is also possible.
2. The funco specifies a subset of its behavior and its state that is considered *sensitive*. Only middleware extensions that are marked privileged by the deployer can interfere with this sensitive behavior and state.

An example in DyMAC.NET. Recall the example with the person component. We shortly list the code of the functional interface and the implementation of the person component in the .NET implementation of the DyMAC framework. The functional interface of the component consists of a C# interface specifying how to instantiate a person, a second C# interface specifying the methods it provides and a third C# class that implements the specified interfaces. This implementation also specifies the state properties and the internal representation.

This implementation has to provide a constructor with the same arguments as specified in the specification of the instantiation (IPersonCreate). The DyMAC framework uses this constructor to instantiate the component when the DyMAC instantiator is called. The DyMAC instantiator is a static method with a variable numbers of arguments. In this way it can easily be used to instantiate any functional component.

```
public interface IPersonCreate{
 IPerson create(string ssn, string name, Date birthDate);}
public interface IPerson{
 string getSsn();
 void setName(string name);
 string getName();
public class Person : FunCo, IPerson{
 public Person(string ssn, string name, Date birthDate){...}
 private Date BirthDate{
  get{...}
  set(Date value){...}
 }
...}
```

For the specification of components in DyMAC.NET we use XML-files. It contains the name of the component, the provided interfaces, the required interfaces, the implementation, and the *aspect integration contract*. The current form of an aspect integration contract in DyMAC specifies the members of the component that are provided to normal middleware extensions and the sensitive members that are only provided towards privileged middleware extensions. The members of a component can be constructors, methods and state members. Specifying required middleware extensions is still part of our ongoing work.

The following example illustrates the structure of the specification and focuses on the aspect integration contract. All members of the component that are related with the birth date are marked sensitive for inspection and modification.

```
<funco><name>Person</name>
 <provided>...</provided>
 <implementation>...</implementation>
 <required>...<required>
 <aspect-integration>
  <provided> <!-- towards all aspect-components -->
   <method>string getSsn()</method>
   <method>void setName(string name)</method>
   <method>string getName()</method>
   <method>string askName(IPerson p2)</method>
   <method>IPerson clone()</method>
   <state>string name</state>
  <provided>
  <sensitive><inspect/><modify/>
   <constructor>create(string ssn, string name, Date bd)</constructor>
   <method>void setBirthDate(Date bd)</method>
   <method>Date getBirthDate()</method>
   <state>Date birthDate</state>
  </sensitive>
 </aspect-integration>
</funco>
```

The Service Layer Components. Middleware service components are specified in quite the same way as functional components. They specify their provided interfaces towards the connectors and other middleware services. They also specify the interfaces they require from other middleware services they use.

The main difference is they don't have to specify an aspect integration contract. Aspect-oriented composition is only supported between the functional layer and the middleware layer. A hierarchic aspect-oriented composition strategy, where messages between service layer components can be intercepted is out of the scope of this paper, but certainly not out of the scope of our ongoing work.

When we return to the example, the interfaces and implementation of the person persistence service are straightforward.

```
public interface IPersonPersistence{ ...
 void insert(string ssn, string name, Date bd);}
public interface IPersonPersistenceCreate{
 IPersonPersistence create();}
public class PersonPersistence{...
 public void insert(...){
 //insert into persistent storage (database, XML-file ...)
}}
```

```
<service><name>personpersistence</name>
 <provided>
  <method-interface>IPersonPersistence</method-interface>
  <create-interface>IPersonPersistenceCreate</create-interface>
 </provided>
 <implementation><class>PersonPersistence</class></implementation>
</service>
```

Connector Components. All connector components have the same provided interface: a before and after method that contains the calls to the middleware services before and after a message is sent or received.

The connector has to specify the set of middleware services it uses as required interfaces. As a second part of what is required for the connector, the specification contains explicit dependencies towards the functional components. This part of the connector's requirements contains the different members of the functional components that the connector depends on. Next to that, the connector also specifies how it interferes with those members: i.e. inspecting and/or modifying them. This interference can be specified on a per member basis or for all members at once (as in the example below).

In its XML-file the connector also specifies if it applies to the caller or callee side of the message it is superimposed on. Depending on the side that the message is superimposed on, a reference to the functional object is provided. So the connector can inspect or modify the state of that object.

In the example we have to define two kinds of connectors: one for the construction call to insert the person into the persistent storage and one for a mutator call to update the persistent storage. We have illustrated the code of the mutator connector and its specification file.

```
public class MutatorConnector : IConnector{
 public void before(MessageCall mc, FunCo object){}
 public void after(MessageCall mc, ReturnMessage rm, FunCo object){
   IPersonPersistence ipp = DyMAC.createService("service/persistence");
   string ssn = (string)object.getState("Ssn");
   string name = (string)object.getState("Name");
   Date bd = (Date)object.getState("BirthDate");
   ipp.update(ssn, name, bd);}}
```

```
<connector><name>mutator connector</name><callee/>
 <class>MutatorConnector</class>
 <required>
  <service>...</service>
  <funco></inspection></modification>
   <state>String Ssn</state>
   <state>String Name</state>
   <state>Date BirthDate</state>
  </funco>
 </required>
</connector>
```

Middleware Extensions as Components. Middleware extensions consist of a collection of connectors and middleware services. The provided interface of the middleware extension is first defined by the provided interfaces of the middleware services it encapsulates and secondly by the connectors that it contains. The provided interfaces of the middleware services can also be used by the connectors of other middleware extensions. These interfaces define the part of the provided interface of the middleware extension that supports object-oriented composition. The connectors define the part of the provided interface of the middleware extension that supports aspect-oriented composition. As illustrated below, the specification of a middleware extension is a simple list of the components it contains.

```
<extension><name>person persistence extension</name>
 <connector>constructorconnector.xml</connector>
 <connector>mutatorconnector.xml</connector>
 <service>personpersistence.xml</service>
</extension>
```

5.3 Application Assembly

The different components of the application are assembled and composed by means of a declarative specification. First, all components of the application are enumerated by linking to the file with their specification. Secondly, the concrete connections are specified between the functional components and the middleware extensions of the application. In this connection, quantification is realized by using pointcut designators to compose one or more messages of one or more components with one or more connectors. In case multiple connectors are superimposed on a join point, they are invoked with the following precedence rules: first the before advices from connector 1 to n are executed, and then the after advices from connector n to 1.

In the example below, all extensions of the application are marked *privileged.* But it is also possible to specify it more fine grained on a per extension, per connection or per connector basis. The connections in the example are defined in the scope of the persistence extension, therefore the used connectors in a connection should be defined in the persistence extension. But in DyMAC, it is also possible to define connections that are out of the scope of one extension and that superimpose connectors of different extensions.

```
<application><name>PersonApplication</name>
 ... <!-- components in the application -->
 <superimposition><privileged/>
  <extension>persistence extension
  <connection>
   <component>Person</component>
   <constructor>create(string ssn, string name, Date date)</constructor>
   <connector>constructor connector</connector>
  </connection>
  <connection>
   <component>Person</component>
   <method>* set*(..)</method>
   <connector>mutator connector</connector>
  </connection>
  </extension>
 </superimposition>
</application>
```

In the initial problem statement, we defined middleware extensions as application specific. Therefore extensions can only be connected to components within the same application. It is our intention to extend the connections so it is also possible to define system wide middleware extensions that can be connected to funcos of other applications in the system.

How the deployment of the application is specified is beyond the scope of this paper. In this deployment specification, the dependencies of all components are bound to actual components in the system. The deployment specification also allocates the different components of an application on the different nodes in the network.

6 Related Work

Open ORB [3] starts from the same problem: the need for adaptable middleware due to application specific needs. Their solution takes the form of reflective middleware. It uses a reflective API to modify the middleware platform and introspect its implementation.

JBOSS/AOP, JAsCO and JAC offer support for aspect-oriented composition in a Java component-based system. They introduce the concept of aspects that can be used to implement middleware services. But they do not support a true component-based approach to the aspects itself. The base components in the functional layer are not aware of possible interfering aspects, and cannot specify in which way they want to control interference of aspects, e.g. by means of an aspect integration contract as in the DyMAC framework. In these systems any possible join point of each Java component can be superimposed with any aspect.

The join point model of JBOSS/AOP exposes a lot of the internal implementation of components. Possible join points are reads and writes to fields of the class, but also calls of methods and constructors within the implementation of a method or constructor. This exposes details about the implementation of the latter method or constructor. JAsCo limits its join point model to the public methods and events of Java Beans. As mentioned earlier this limits the possibilities for middleware extensions when they need access to the state of the component.

In JAC, the pointcuts that specify where to superimpose an aspect are strings in the code of the aspect, which limits runtime adaptability of the composition logic. Externalizing and modularizing this composition logic in a declarative specification offers better support to change the composition logic without recompiling the application.

Lasagne is a runtime architecture that enables dynamic customization of systems. Based on client-specific needs and context properties it can select and activate the different extensions in the system. These extensions have the form of wrappers that implement the same interface as the components they are superimposed on. Just like in the DyMAC framework, wrappers can add behavior before and after the invocation of a method. Lasagne also lacks the expressive power to specify which kind of extensions a base component allows. The composition logic of Lasagne is also specified in the meta data of the applications, and not hard coded.

7 Conclusion

In this paper we discussed DyMAC (Dynamic Middleware with Aspect-Components), a component and aspect-based middleware framework that offers adaptability and extensibility. It supports component-based development of middleware services and offers the power of aspect-oriented composition to connect the application logic to the middleware services.

DyMAC solves the issue of the lack of expressive power in the contracts of components and aspects and introduces a kind of aspect-component. It also solves the too strong or too weak composition model of existing aspect-component technologies with a more balanced composition model.

References

1. Clemens Szyperski, Component software: beyond object-oriented programming. Second Edition. ACM Press/Addison-Wesley Publishing Co., New York, NY, 2002.
2. Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, Second Edition. Addison-Wesley, 2003.
3. G. S. Blair, et al. The design and implementation of OpenORB version 2. IEEE Distributed Systems Online Journal, 2(6), 2001
4. Kiczales, G. et al. Aspect-Oriented Programming. In Proc. of ECOOP 1997.
5. Kiczales, G. et al. An Overview of AspectJ. In Proc. of ECOOP 2001.
6. R. Filman et al. Aspect-oriented programming is quantification and obliviousness. In OOPSLA Workshop on Advanced Separation of Concerns, 2000.
7. Bertrand Meyer. Design by contract: building bug-free O-O software. In Hotline on Object-Oriented Technology, volume 4, Number 2, December 1992, pages 4-8.
8. Andreas Rausch, Design by Contract + Componentware = Design by Signed Contract. Journal of Object Technology, In Proc. of Tools Usa, 2002.
9. R. Bodkin et al. Applying AOP for Middleware Platform Independence. Practitioner Reports, AOSD 2003.
10. Adrian Colyer et al, Large-scale AOSD for middleware. In Proc. of AOSD 2004.
11. Sun Microsystems, Inc. Enterprise Java-Beans (EJB) Specification v2.0, 2001.
12. R. Pawlak et al. JAC: A Flexible Solution for Aspect-oriented Programming in Java. In 3rd International Conference on Meta-level Architectures and Separation of Concerns (Reflection), volume 2192 of Lecture Notes in Computer Science, pages 1-25. Springer-Verlag, 2001.
13. D. Suvée et al. JAsCo: An aspect-oriented approach tailored for component-based software development. In Proc. of AOSD 2003.
14. Mira Mezini et al, Conquering aspects with Caesar. In proc. of AOSD 2003.
15. JBoss AOP homepage, http://www.jboss.org/developers/projects/jboss/aop.jsp
16. E. Truyen, et al. Dynamic and Selective Combination of Extensions in Component-Based Applications. In Proc. of ICSE'01.
17. B. Lagaisse et al. Managing Semantic Interference with Aspect Integration Contracts. In workshop SPLAT'04, http://www.daimi.au.dk/ eernst/splat04/