Multi-Stage Aspect-Oriented Composition of Component-Based Applications

Bert Lagaisse, Eddy Truyen & Wouter Joosen

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium,
{bertl, eddy, wouter}@cs.kuleuven.be

Abstract. The creation of distributed applications requires sophisticated compositions, as various components — supporting application logic or non-functional requirements — must be assembled and configured in an operational application. Aspect-oriented middleware has contributed to improving the modularization of such complex applications, by supporting a component model that offers aspect-oriented composition alongside the traditional composition of provided and required interfaces. One of the recent advances in AO middleware is the ability to express dynamic compositions that depend on the evaluation of available context information — some of this information may only be available at deployment time.
The search for high level composition mechanisms is an ongoing track in the research community, yet the composition logic of a real world application remains complex and it would greatly pay off if composition logic — traditionally encoded in monolithic deployment descriptors — could be reused over ranges of applications and even be gradually refined for specific applications.
This paper presents M-Stage, an AO component and composition model that supports the reuse and adaptation of compositions in distributed applications that are built on AO middleware. We illustrate the power of M-Stage by applying the model in a realistic distributed application where we analyze the reuse and adaptation potential of the M-Stage model.

1 Introduction

Distributed applications are typically built on middleware that offers a component model and execution environment for these applications. Practical middleware platforms nowadays have to support complex composition of application components and have to support a broad range of services that deal with the non-functional concerns in a distributed application.

Aspect-Oriented middleware (AO middleware, AOM) has contributed to improving the modularization of such complex applications, by supporting an aspect-component model that offers aspect-oriented composition (AO composition) alongside traditional composition of provided and required interfaces [7, 8, 14, 17, 18]. The core concept in AO composition is the aspect[3]: a coherent abstraction that encapsulates one specific (often crosscutting) concern in a separate software module. An aspect defines behavior that can be executed and defines composition logic to describe where and when

this behavior should be executed. AO middleware typically separates aspect behavior and composition logic from each other. The composition logic is specified externally in a deployment descriptor while the aspect behavior is represented within traditional components as methods. Composition logic in AOM is thus specified in the form of *Whenever event X in the application occurs, execute method behavior Y of component Z.* An example of such composition logic can be: *whenever a component operation is executed, execute the enforcement method of the Authorization component*. One of the recent advances in AO middleware is the ability to express dynamic compositions that depend on the evaluation of available context information about the distributed infrastructure. This context information may include for example the component names of involved components (e.g. caller and callee of an operation), their container, the application they belong too, the hosts they are deployed on, etc. As a result complex composition with remote events can be expressed more concisely and at a higher-level of abstraction. An example of a such powerful composition logic can be *Whenever a client in the ATM-network calls a component on the application server, the Kerberos authentication scheme must be applied*.

The search for high level composition mechanisms is an ongoing track in the research community, yet the composition logic of a real world application remains complex and it would greatly pay off if composition logic — traditionally encoded in monolithic deployment descriptors — could be reused over ranges of applications and even be gradually refined for specific applications. Before elaborating on this problem we will first define the concept of AO composition more rigorously.

Basic concepts of AO Composition. A key element in the specification of the composition logic is the concept of a *pointcut* which is a description of a set of join points where aspects should execute. *Join points* represent dynamic, runtime conditions that arise during program execution. The occurrence of such a condition is an event that can trigger the execution of aspect behavior. *Advice* specifies what aspect behavior should be executed and when the aspect behavior should be executed (typically before, after or around the event) [4]. Pointcuts select join points by declaratively specifying the kind and context of join points. The *kind* of a join point refers to the type of instruction being executed. For example, two different kinds of join points are method call and field access. The *context* of joint points refers to additional information that can be made available to constrain the pointcut such as the method signature, type and identity of the caller or callee of a method, and various distributed infrastructure properties as mentioned above. Which kind of context and which available context that are supported by an AOM, are defined by the AOM's *join point model*. In general, the richer the join point model, the more complex compositions can be supported.

1.1 Problem statement

In this section, we set up the scene for the rest of the paper. The goal is to identify shortcomings of current AO middleware with respect to supporting the development of reusable composition logic for late integration in various applications and deployment environments. In the following subsections we will illustrate these shortcomings using a pedagogical example in the context of the AO middleware DyMAC [9]. DyMAC has

one of the richest join point models and therefore serves as a good example to illustrate these shortcomings. After this illustration we will shortly summarize the general problem statement of this paper.

The thread of the examples in this paper is the construction of a family of banking applications. To illustrate the shortcomings in this section and to illustrate the basic solution in the next section, we use a pedagogical, more concise example in that application domain. For the validation of the solution in section 3 we define a more elaborate case study, based on the example defined in this section.

Problem 1: Reuse of composition logic across applications. Consider the construction of a family of banking applications from a set of components using AO composition and traditional composition. One of these components is the *Authorization* component which verifies the application-level rights of authenticated users when a banking transaction is executed. Another component is the *Account* component which is a generic software module that can be reused for different banking products such as a basic checking account service, for custody accounts in an investment application, or to keep track of loan balances in a loan application. Consider in particular two specific banking products, a *basic banking service* and an *investment application* that use the Account and the Authorization component. Each application has a facade component that uses the Account entity: *BasicBanking* and *Investment*. The interfaces of Account, BasicBanking and Investment are as follows:

```
interface IBasicBanking {
  void CreateAccount(string id, string owner);
  void Deposit(string id, double amount);
  void Withdraw(string id, double amount);
  void Transfer(string from, string to, double amount);
  AccountInfo GetInfo(string id);}
interface IInvestment {
  void CreateAccount(string id, string owner);
  void Deposit(string id, double amount);
  void Withdraw(string id, double amount);
  void BuyStock(string account, string StockId, int amount, double maxprice);
  AccountInfo GetInfo(string id);}
interface IAccount {
  string GetId();
  double GetBalance();
  void Deposit(double amount);
  void Withdraw(double amount);
  List<Transaction> GetTransactions();}
```

As authorization is a crosscutting concern it is composed using AO composition descriptors of DyMAC, which is illustrated in Figure 1. The AO composition of authorization with the basic banking application specifies for example that the authorize method of the Authorization component should be executed around the execution of the Deposit and WithDraw operation of Account, but also around the execution of the Transfer operation of BasicBanking. Figure 1 also shows the AO composition descriptor that composes the Authorization component in the investment application. Notice that large part of this composition logic is the same as in the basic banking application. Yet currently, the developer has no option then to duplicate this common composition logic in two separate descriptors. This is obviously problematic in the presence of maintenance and evolution. A preferable solution is to modularize this common part in a separate AO

```
ao-composition{
 pointcut{
  kind: execution;
  signature: void Deposit(..) OR void WithDraw(..)
     || void Transfer(..);
  Callee{
   Component: Account OR BasicBanking;}
  Caller{
     Host: NOT *.intranet.bank.com}}
 advice{
     advice-component: Authorization;
     advice-method: authorize;}}
```

```
ao-composition{
 pointcut{
  kind: execution;
  signature: void Deposit(..) OR void WithDraw(..)
     || void BuyStock(..);
  Callee{
   Component: Account OR Investment;}
  Caller{}
 advice{
     advice-component: Authorization;
     advice-method: authorize;
 }}
```

Fig. 1. AO composition of Authorization with the basic banking application and the investment application respectively. The underlined specifications are the same in both AO composition descriptors. Deployment-specific composition logic is tangled into the composition logic of basic banking. The specification in bold indicates the evaluation of deployment-specific context info.

composition descriptor, reuse it for a particular application, and non-invasively refine it to needs of the specific application.

Problem 2: Reuse across deployment environments. Similarly, AO composition logic cannot easily be reused across multiple deployment environments. The composition of authorization may also depend on deployment-specific information in order to enforce a deployment-specific policy. For example in Figure 1, a security policy is superimposed that authorization should not be performed when the call originates from the trusted intranet zone. However, this distributed context information is specific to the concrete deployment environment of basic banking. Encoding this deployment-specific composition logic at the same level of abstraction will cause difficulties when trying to reuse the common composition logic for another deployment environment. To support reusability of the common composition logic, it should be possible to non-invasively extend pointcuts with evaluation of deployment-specific context information. Furthermore, if specifying reusable composition logic is the goal, the aspect-component model should enforce the developer to abstract from context information that is specific to a particular application or deployment environment. For example, a component builder who has to create a reusable secure account component (by composing Account and Authorization) should not be permitted to specify pointcuts that evaluate the host name of calling components. This context information should not be made inaccessible for the developer; rather it should be made irrelevant at a certain level of abstraction.

Summary of the problem. One of the recent advances in AO middleware is the ability to express complex AO compositions with remote events. This strength, however, is also a weakness: when common composition logic in a family of applications must be duplicated in separate deployment descriptors, issues of reuse arise. All AO middleware that use a monolithic representation of pointcuts have difficulty defining reusable composition logic because they do not provide modularization and gradual refinement of composition logic. Furthermore, the ability to abstract from context information when possible, but provide access where necessary, is particularly important for AO middle-

ware with a rich join point model. Although a rich join point model allows to express complex composition logic with remote events, the monolithic representation of pointcuts causes the tangling of the evaluation of various context information. Some of this information may only be available during application construction or at deployment time. This causes additional difficulties with reuse of composition logic.

1.2 The M-Stage solution

This paper presents M-Stage, an aspect-component model that supports the reuse and gradual refinement of AO composition logic on top of AO middleware. An essential element of our solution is that M-Stage integrates AO composition with the hierarchical and phased approach of Component-Based Software Development (CBSD). State-of-the art component models in middleware such as J2EE [23] and CCM [22] typically organize software deployment in multiple subsequent *stages*. These stages include hierarchical component aggregation first, then application assembly and finally application deployment. M-Stage supports specification and gradual refinement of AO composition logic across these multiple stages. We call this *multi-stage composition*. The major contributions of M-Stage are:

- Support for hierarchical modularization of AO composition logic in separate aggregation, assembly and deployment descriptors.
- Support for gradual refinement of AO composition logic across these multiple stages.
- A join point model that abstracts from certain context types during the early stages of aggregation and assembly. These details are thus made irrelevant for the developer at a certain stage to ensure reusability of common composition logic across multiple applications and deployment environments. We call this a multi-stage join point model, which is graphically depicted in Figure 2.

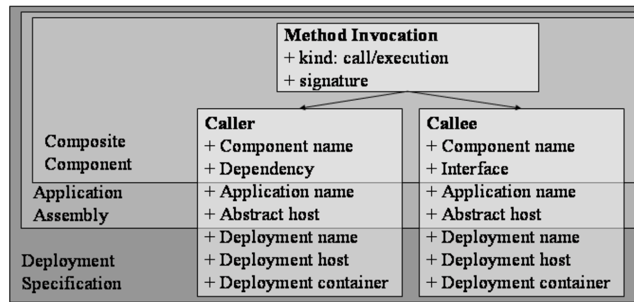

Fig. 2. Multi-stage joinpoint model

The rest of the paper is structured as follows. In section 2 we describe the M-Stage aspect-component model. In section 3 we illustrate M-Stage by means of a case study and evaluate the overall approach. Section 4 summarizes related work. We conclude in section 5.

2 M-Stage

The development of a distributed component-based application involves multiple composition and configuration activities. These activities can be classified in four *stages*:

1. The specification of elementary components, which specifies the provided and required interfaces of elementary components and their implementation.
2. The aggregation of composite components, which consists of repetitive, hierarchic composition of elementary and composite components.
3. Assembling the application, which includes composing the different (aggregate) elements of the application and defining an abstract architecture for it.
4. The deployment specification of the application, which maps the software components to their concrete deployment location.

Each stage leads to specific artifacts, such as elementary components, composite components, application assemblies and deployment specifications. These artifacts are produced by different stakeholders : component providers, application assemblers and application deployers. Composition of components occurs in three stages: hierarchical component aggregation first, then application assembly and finally application deployment. M-Stage supports modular specification and gradual refinement of AO composition logic across these multiple stages. Refinement of an AO composition can occur (repeatedly) within one stage, and also across stages. The M-Stage AO composition model offers a multi-stage joinpoint model, in such way that it supports contextual properties specific to each stage. The model also restricts the visibility of those properties to the appropriate stages.

First, we define the basic component model of M-Stage. We define what kind of artifacts are created in each of the different stages. Second, we define the multi-stage joinpoint model, multi-stage AO composition and the support for composition refinement.

2.1 The M-Stage Component Model

We define the basic component model with the four different stages and we explain which information is specified in the different kinds of artifacts in each stage.

Elementary Components. The elementary components are object-based entities with well-defined interfaces. An elementary component can have multiple interfaces. It can also have a set of dependencies that specify the required interfaces of the component. The specification (or descriptor) of an elementary component defines (1) the component name, (2) the provided interfaces, (3)the implementation file and (4) the dependencies of the component. A dependency is defined by a dependency name and an interface that is expected to be bound to the dependency.

As an example we describe the Account component in the next listing. It provides the IAccount interface as defined in section 2. It has a dependency *tx* for the required ITransaction interface. The implementation is defined in AccountImpl. The numbered labels in comment refer to the enumeration of the elements as introduced in the previous paragraph

Composite components. A composite component is a hierarchical composition of a set of components. The components that are part of the composite component can be elementary or composite components. The specification of a composite component defines (1) a name for the composite, (2) the set of components it contains, and (3) composition specifications. These compositions can be normal dependency - component mappings or AO compositions. AO compositions are further discussed in detail in the next subsection.

An example of a composite component is illustrated in the next listing. We define the bank's generic account component: *SecureAccount*, an aggregation of *Account*, *Transaction* and *Authorization*. We define the dependency-component mapping of Account with Transaction. The AO composition of Authorization in this composite is discussed later.

```
component Account { //1
  provides: IAccount; //2
  implementation: AccountImpl; //3
  dependency tx: ITransaction; //4
}
```

```
composite SecureAccount {
  contains: Account, Transaction,
    Authorization;
  composition{
    dependency: Account.tx;
    component: Transaction;}
  ao-composition {...}}
```

Application assemblies. Application assemblies define distributed component-based applications that are deployable to multiple environments. They consist of a set of components (elementary or composite), similar to composite components. The assembly specification contains reusable architectural information about the application by specifying an abstract deployment topology. This is a set of abstract hosts on which the components are allocated. A possible architecture that could be included is a multi-tier architecture or a simple client-server architecture. Concretely, the specification of an application assembly contains (1) a name for the application, (2) the set of components it contains, (3) definitions of abstracts hosts, (4) mappings of components to abstract hosts and (5) composition specifications specific to the application.

The example in Figure 3 illustrates an abstract architecture for a banking application: SecureAccount and BasicBanking are located on the abstract appserver, EmployeeClient on the workstations and ATMClient on the ATM machines.

Deployment specification. A deployment specification of an application specifies (1) the application it deploys, (2) unique deployment names for the public accessible components, (3) mappings of abstract hosts to concrete hosts, (4) mappings of a component to a container on a concrete host, (5) a set of already deployed components it uses and (6) compositions specific to the deployment environment.

The deployment example in Figure 3 deploys the banking application defined above. It maps the abstract host *appserver* to the concrete host *appserver1.mybank.net*, and deploys the BasicBanking component with the unique name MyBasicBanking on a concrete container on the application server. It also declares that it uses an already deployed component in the environment : runtime monitor, which will monitor and log the distributed execution trace of all sessions on the application server's containers. We illustrate the deployment-specific compositions further on.

```
application BasicbankApp { //(1)
 contains: SecureAccount, BasicBanking,
        EmployeeClient, ATMClient; //(2)
 abstracthost: atm, workstation,
        appserver; //(3)
 locate{ //(4)
  component: BasicBanking, SecureAccount;
  abstracthost: appserver;}
 locate{
  component: EmployeeClient;
  abstracthost: workstation;}
  ...
  //(5): compositions
}
```

```
deployment MyBank{
 contains: BasicbankApp; //(1)
 map{ //(3)
  abstracthost: BasicbankApp.appserver;
  host: appserver1.mybank.net;}
 deploy{ //(4)
  component:BasicbankApp.BasicBanking;
  deploymentname: MyBasicBanking; //2
  host: appserver1.mybank.net; //4
  container: container1;}
 usedeployed: RuntimeMonitor; //(5)
 //(6) compositions
}
```

Fig. 3. Application assembly and deployment specification

2.2 AO Composition Model

Composition of components is supported in three stages: (a) composite component aggregation, (b) application assembly and (c) deployment. We continue the description of the composition model by focusing on AO composition. We first explain the multi-stage joinpoint-model. Then we elaborate on the multi-stage AO compositions with support for composition refinement.

Multi-stage joinpoint model. The joinpoint model defines the kind and context of joinpoints. The kind of the joinpoint can be either a call or an execution of a method. The contextual information available about the calling and called component depends on the stage in which the AO composition is defined. For each stage we define a set of contextual properties in the joinpoint model that can be evaluated in that specific stage. The joinpoint model enforces that a stakeholder specifying a certain AO composition in a certain stage only has to reason over the contextual information that is relevant at that stage. A definition of the contextual properties in each stage is explained next, and is also depicted in Figure 2.

In the composite aggregation stage, the contextual properties about the calling component are the component name and the dependency that is used. Contextual properties about the called component are the component name and the interface on which the method invocation is done. In the application assembly stage, the application name and abstract host of caller and callee become available, next to the already available properties of the composite aggregation station. The contextual properties that become available in the deployment stage are the component's deployment name, host, and container.

Multi-stage AO composition. AO compositions can be specified in each stage that supports composition of components. Such an AO composition consists of three parts: (c1) a name for the AO composition, (c2) a pointcut expression and (c3) a set of advices. Figure 4 shows the high-level grammar, which is the same for all stages. We discuss the details of a pointcut expression and an advice next.

A pointcut expression evaluates over the kind and context of the joinpoints. The kind of the joinpoint can be either a call or an execution of a method invocation. Pointcuts can further evaluate over the contextual properties of the joinpoint. As defined in

```
ao−composition <name>{ //(c1)
  Pointcut <name>{ //(c2),(p1)
    Kind: [call|execution]; //(p2)
    Signature: <method−pattern>; //(p3)
    Caller{
      [<property>: <string−pattern>;]*} //(p4)
    Callee{
      [<property>: <string−pattern>;]*}} //(p5)
  [Advice <name>{ //(c3)
    Advice−Component: <component−name>;
    Advice−Type: [before|after|around];
    Advice−Method: <interface>.<advice−method>;
  }]*
}
```

Fig. 4. Grammar for AO compositions.

the grammar, a pointcut expression consists of (p1) a name for the pointcut and evaluates over (p2) the kind of joinpoint, (p3) the method signature, (p4) caller properties (contextual properties about calling component) and (p5) callee properties (contextual properties about the called component). The available contextual properties about caller and callee depend on the stage in which the pointcut is specified. Furthermore, if pointcuts do not specify a value for a certain property, it has a default value. This default value is the least restricting value for that contextual property. Furthermore, an advice is described by (1) the component that provides the aspect behavior, (2) an advice method of the advising component and (3) an advice type (before, after or around).

A first example is SecureAccount's AO composition, specified in the aggregation stage. It illustrates the use of joinpoint properties that are only visible in the aggregation stage. A second example is RuntimeMonitor's AO composition, defined in the deployment stage. It illustrates the modularization of deployment specific composition logic: the pointcut as well as the advice are deployment-specific.

```
composite SecureAccount{
 contains: Account, Transaction,
   Authorization;
 ...
 ao−composition aoc1{
  Pointcut sensitive{
   Kind: execution
   Signature: void Deposit(..)
     OR void Withdraw(..);
   Callee{
    Component: Account;}}
  //advices of Authorization to call.
  Advice checkrole{
    Advice−Component: Authorization;
    Advice−Method:
      IAuthorization.VerifyRole;
    ...
 }}}
```

```
deployment MyBank{
 ...
 ao−composition monitor{
  Pointcut monitortrace{
   Kind: execution
   Signature: * *(..);
   Callee{
    Host: appserver1.mybank.net;}}
  Advice logtrace{
   Advice−Component: RuntimeMonitor;
        ...
  }}
```

Refining AO compositions. When a composition artifact is used in an other artifact, of possibly another stage, it might be necessary to further refine the pointcuts. This refinement can be a different evaluation of an existing contextual property or the use

of a newly available property in the actual stage. The *with* directive allows to refine the pointcut in an AO composition or in parts of it. The *old* keyword allows to refer to the previous pointcut definition. The refined pointcut is expressed by referring to *the pointcut name within its naming path*. First we define the structure of the naming path, then we define the specification of a refinement.

```
<namingpath> =
<artifact>[.<containingartifact>]*.<ao-composition>
```

```
<artifacttype> <name> {
   ...
   with <namingpath>.<pointcut> {
     Signature = <new value>;
     with Caller {
       //refinement of properties
      }
     with Callee{
      //refinement of properties
     }}}
```

As an illustration of reuse and refinement we reconsider the examples from the introduction. SecureAccount's AO composition of Account and Authorization has already been defined earlier. The stepwise refinement of the AO composition for the basic banking application is defined in the Figure 5. The AO composition of authorization is refined in the definition of the basic banking application assembly, so that it is applied to BasicBanking too, and not applied to Account when the calls come from BasicBanking. In the deployment specification of the basic banking application, the authorization is refined so that it is never enforced when the calls originate from the trusted intranet zone.

```
Application BasicbankingApp{
   contains: SecureAccount;
   contains: BasicBanking;
   with SecureAccount.aoc1.sensitive{
     Signature: old.Signature || void Transfer(..)
     with Callee{
       Component: old.Callee.Component || BasicBanking;}
     with Caller{
       Component: !BasicBanking;}}}}
```

```
deployment MyBank{
  ...
  with BasicBankingApp.SecureAccount.aoc1.sensitive{
    with Caller{
      Host: !*.intranet.bank.com;}}}}
```

Fig. 5. Refinement of Authorization

3 Validation

In this section, we validate the power of M-Stage by applying the model in a family of e-finance applications. Figure 6 depicts the core assets of this family, which are (1) a set of reusable elementary components, which can be reused for different e-finance applications, (2) a set of reusable composite components encoding reusable composition logic, and (3) reusable application assemblies encoding reusable architectures that are deployable to multiple deployment environments.

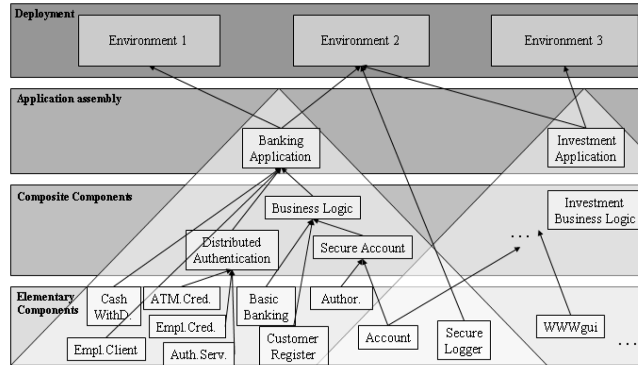

Fig. 6. Multi-stage AO composition

We illustrate how M-Stage[1] supports this and how a concrete family instance is built, using multi-stage composition and gradual refinement. We evaluate against the reuse potential of DyMAC's own component model.

3.1 The elementary components.

Three elementary components are typical business components: *CustomerRegister*, *BasicBanking* and *Account*. The CustomerRegister and BasicBanking components are remotely accessible services that offer the core business operations to manage customers, create new accounts and execute transactions. The BasicBanking component uses the Account component, which is an entity that contains the basic information about accounts: a unique account id, a balance, and a record of transactions executed on the account. The account component is a generic account that offers two operations: deposit and withdraw. The interfaces of the three core business components are defined in Figure 7.

The employees at the branch offices of the bank use the *EmployeeClient* component at their workstations to manage the customers' accounts and to handle customer requests. The customers can also use an ATM to withdraw money from their accounts.

[1] We use the M-Stage component model implementation on top of the DyMAC runtime environment.

```
interface IBasicBanking {
  void CreateAccount(string id, string owner);
  void Deposit(string id, double amount);
  void Withdraw(string id, double amount);
  void Transfer(string from, string to, double amount);
  AccountInfo GetInfo(string id);}
interface ICustomerRegister {
  void CreateCustomer(string id, string name,...);}
interface IAccount {
  string GetId();
  double GetBalance();
  void Deposit(double amount);
  void Withdraw(double amount);
  List<Transaction> GetTransactions();}
```

Fig. 7. Interfaces of BasicBanking, CustomerRegister and Account

The *CashWithdrawal* component on the ATM terminals as well as the EmployeeClient component uses the BasicBanking component to execute the transactions. The different components in the application are depicted in Figure 8. Figure 8 also describes the set of elementary aspect-components.

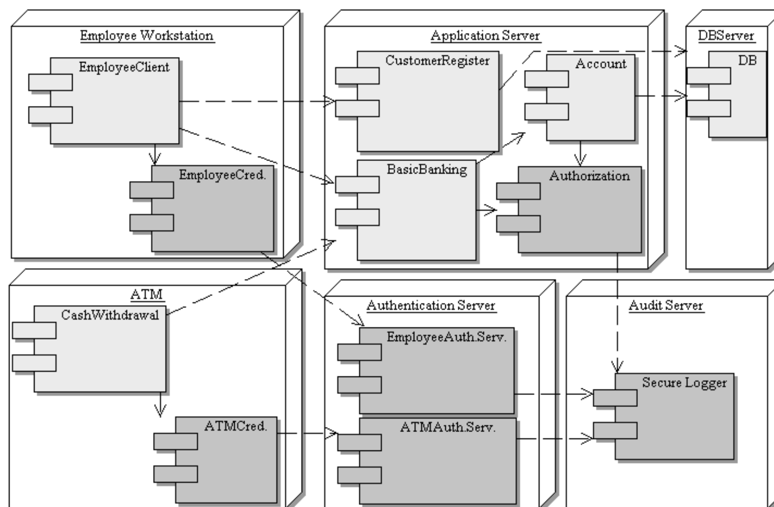

Fig. 8. Application overview

There are four collaborating aspect-components for authentication in the application. First, a client-side component, *EmployeeCredentials*, asks the employees for credentials, before the EmployeeClient component starts up. A second component, which is a local component on the ATM, *ATMCredentials*, asks the ATM-users for credentials when they want to withdraw cash. Third, a server-side component *EmployeeAuthenticationService* authenticates the credentials of the employees and generates an employee

authentication token. Forth, the *ATMAuthenticationService* authenticates the credentials of atm users and generates an atm authentication token. Both authentication services are located at the central authentication server and are called after the client has provided his credentials. The returned authentication token is stored in the client side credential components. Each time a call is made from the clients to the application server, an advice in the client side credential components will push the stored authentication token along with the call. A fifth aspect-component is the *Authorization* component that verifies the application-level rights associated with the authenticated user when a banking transaction is executed. Employees can only do transactions during office hours. ATM-users can only withdraw from their own account, with a maximum of 5000 Euro. The sixth aspect-component is the *SecureLogger* component at the central audit server. It keeps track of all authentication and authorization attempts and of the results.

The interfaces of the advising components in the example are defined as follows. EmployeeCredentials and CustomerCredentials provide the interface ICredentials. It defines an operation to get the credentials of the user, an operation to store the authentication token after it is returned by the authentication service and an operation to push the authentication token with every call that is made to the application server.

```
interface ICredentials{
  void GetCredentials(RuntimeJoinPoint rjp);
  void StoreToken(RuntimeJoinPoint rjp);
  void PushToken(RuntimeJoinPoint rjp);}
interface IAtmAuthenticationService{
  void VerifyAtmCredentials(RuntimeJoinPoint rjp);}
interface IEmployeeAuthenticationService{
  void VerifyEmployeeCredentials(RuntimeJoinPoint rjp);}
interface IAuthorization{
  void VerifyRole(RuntimeJoinPoint rjp);
  void VerifyTime(RuntimeJoinPoint rjp);
  void VerifyOwner(RuntimeJoinPoint rjp);
  void VerifyAmount(RuntimeJoinPoint rjp);
interface ISecureLogger{
  void Log(RuntimeJoinPoint rjp);}
```

3.2 Composition and deployment of the basic banking application

The composition of the basic banking application, as depicted in Figure 6, defines 11 AO compositions (9 definitions + 2 refinements) across the composite components (5+1), the application assembly (2+0) as well as the deployment specification (2+1).

Composite components. The composite components in the systems are SecureAccount, BusinessLogic and DistributedAuthentication. The composite component *SecureAccount* contains Authorization and Account. It specifies one AO composition enforcing the authorization rules on the transaction operations of Account.

```
composite SecureAccount{
 contains: Account, Authorization;
 ao-composition aoc1{
  Pointcut sensitive{
   Kind: execution
   Signature: void Deposit(..) || void Withdraw(..);
   Callee{
    Component: Account;}}
  //advices of Authorization to call.
```

```
  Advice checkrole{
    Advice−Component: Authorization;
    Advice−Type: before;
    Advice−Method: IAuthorization.VerifyRole;}
  ... }}
```

BusinessLogic is a composite component containing SecureAccount and BasicBanking. When the SecureAccount composite is reused in the BusinessLogic composite, the AO composition is refined to also enforce authorization on the basic banking service. Security checks need to be verified as early as possible in the call-chain. Calls coming from BasicBanking to SecureAccount then need no authorization. Concretely, the Signature property will be broadened with the Transfer method, and the component name of the callee will be broadened with the BasicBanking component name. The component name of the caller will be restricted from its default value (*) to !BasicBanking because authorization would not occur twice.

```
composite BusinessLogic{
  contains: SecureAccount;
  contains: BasicBanking;
  with SecureAccount.aoc1.sensitive{
    Signature: old.Signature || void Transfer(..)
    with Callee{
      Component: old.Callee.Component || BasicBanking;}
    with Caller{
      Component: !BasicBanking;}}}}
```

DistributedAuthentication is a composite component containing the authentication components in the application. This component encapsulates four AO compositions specifying the following internal interactions between the authentication components. First, after gathering the credentials of the employee, the employee authentication service is called to verify the credentials and return an employee authentication token. Second, after the authentication token is returned it is stored in the employee credential component at the client side. In the next listing, we define these two AO compositions of EmployeeCredentials and EmployeeAuthentication in DistributedAuthentication. The other two AO compositions for ATM authentication are similar.

```
composite DistributedAuthentication{
 contains: EmployeeCredentials, EmployeeAuthenticationService,...;
 ao−composition checkEmployee{
  Pointcut credentials{
   Kind: execution
   Signature: * GetCredentials(..);
   Callee{
    Component: EmployeeCredentials;}}
  Advice {
    Advice−Component: EmployeeAuthenticationService;
    Advice−Type: after;
    Advice−Method: IEmployeeAuthenticationService.VerifyCredentials;}}
 ao−composition storeEmployeeToken{
  Pointcut verify{
   Kind: call;
   Signature: * VerifyCredentials(..);}
  Advice storeToken{
   Advice−Component: EmployeeCredentials
   Advice−Type: after;
   Advice−Method: IEmployeeCredentials.StoreToken;}}
 ao−composition checkAtmUser {...}
 ao−composition storeAtmToken {...}}
```

Application Assembly. The application *BankingApplication* assembles 4 components: BusinessLogic, CashWithdrawal, EmployeeClient and DistributedAuthentication. The two AO compositions between DistributedAuthentication and the other application components are defined in the application assembly. Concretely, the two advices that get the credentials, defined in ATMCredentials and EmployeeCredentials, are composed as before advice on the entry-methods of the client components (e.g. the main method of EmployeeClient). We define the composition in BankingApplication of DistributedAuthentication with the employee client and BusinessLogic: the employee authentication token is pushed along with all calls leaving a workstation towards the appserver. The other AO composition to push the atm token is similar.

```
application BankingApplication{
 contains: DistributedAuthentication ,...
        EmployeeClient , ATMClient;
 abstracthost: atm, workstation ,
        appserver;
 locate{
  component: BusinessLogic;
  abstracthost: appserver;}
 locate{
  component: EmployeeClient;
  abstracthost: workstation;}
 ao-composition pushEmployeeToken{
  Pointcut employeeCalls{
   Kind: call
   Signature: void *(..);
   Caller{
    abstracthost: workstation;}
   Callee{
    abstracthost: appserver;}}
  //Advice push employee token ...
 }
 ao-composition pushATMToken{...}}
```

```
deployment MyBank{
 ...
 with BankingApplication.BusinessLogic
      .SecureAccount.aoc1.sensitive{
   with Caller{
     Host: !applicationserver;}}
 ao-composition authentication_audit{
  Pointcut operationstoaudit{
   Kind: execution
   Signature: * *(..);
   Callee{
    Host: authenticationserver;}}
  //Advice: log after exception
 }
 ao-composition authorization_audit{
}
```

Deployment specification. In the deployment specification above, the secure logger at the audit server of a specific deployment environment is composed with the authentication services and with the authorization component. A failed credential verification or authorization will be recorded in the audit trail by the secure logger. The host name of the authentication server is used in the pointcut in stead of enumerating all authentication services. Failures of any authentication service on the host will then be logged.

The refinement of the authorization pointcut in the deployment specification is a refinement based on new contextual information: all components in BusinessLogic are deployed on the same container on the applicationserver. Therefor, if an invocation comes from within the applicationserver, authorization should not be applied. The host property of the caller will therefore be restricted from its default value * to !applicationserver.

3.3 Evaluation

The goal of this section is to quantify to which extent M-Stage improves reusability of AO composition logic. To achieve this goal we have measured how many AO compositions in the above banking application can be reused in another hypothetical usage

context (for example, the investment application) and we have compared this with composition descriptors in DyMAC. We have used the following metrics in particular:

A The total number of AO compositions defined.
B The number of lines of code defined.
C The number of AO compositions that can be reused across multiple applications.
D The number of AO compositions that can be reused across multiple deployment environments.

The table below gives an overview of our results of this comparison. For this relatively small configuration (83 LOC), M-Stage allows to reuse 55% of the AO composition logic across applications, and 72% across deployment environments. This gain of reuse comes at the cost of 22% more AO compositions that must be specified due to the gradual refinement, and an increase of 18% in total lines of code. We have achieved the increase of reuse because of four main points:

1. Pointcuts in the application assembly per definition do not contain deployment host information and therefore don't tie the application to a fixed deployment environment.
2. Deployment-time AO compositions do not need to be defined in the application assembly.
3. Reusable AO compositions that should already be defined in a reusable composite component can be localized in a separate descriptor.
4. It is possible to refine existing AO compositions.

with refinement necessary	**A**	**B**	**C**	**D**
DyMAC	9	83	n/a	n/a
M-Stage	11 (9+2)	98	6 (5+1) out of 11	8 (7+1) out of 11
Relative increase	+22%	+18%	n/a	n/a

4 Related Work

Three categories of related work are considered: AO middleware, stepwise refinement models and model-driven middleware.

AO middleware (AOM). The general relation between M-Stage and AOM platforms has already been discussed in the motivation of this paper. Summarized, M-Stage augments existing AOMs with *multi-stage composition*, which modularizes AO compositions across multiple stages, and with a *multi-stage join point model*, which enables abstraction of certain context information during the early stages. Multi-stage composition pays off for all AOMs, whereas the multi-stage join point model only pays off for AOMs with a rich join point model. For example, JBoss AOP [8], AspectWerkz [15], Spring AOP [14], Prose [17], CAM/DAOP [10], DADO [18], FAC [28] and GlueQoS [19] only benefit from multi-stage composition because these platforms do not support the evaluation of context properties concerning the application architecture or deployment. On the other hand platforms such as JAC [7], DJCutter [20], AWED [11] and DyMAC [9] also benefit from a multi-stage joinpoint model because they do support the evaluation of distributed context and application architecture.

Stepwise refinement. Batory et al. presents a software composition model and associated tool set, called AHEAD [16], that supports large-scale refinement of aspect-like modules in a product family. Atkinson and Kühne present the concept of stratified architectures [21] for gradually refining and introducing aspect behavior across multiple levels of abstraction in the design of a distributed application. There are two important differences between these works and M-Stage. First M-Stage has a more focused goal: it supports stepwise refinement of aspect composition logic, not aspect behavior. Furthermore, M-Stage supports stepwise refinement across multiple development stages and not across multiple levels of abstraction in the design of a software system. Finally, the AHEAD tool set does not target AO middleware. Having said this, it should be noted that AHEAD supports a uniform *compose* operation that also targets *non-code artifacts*. As such it is possible that AHEAD can also be used to express stepwise refinement of composition descriptors. Exploring this interesting idea is however subject to further research.

ADL-driven and Model-driven middleware. CAM/DAOP [10] is an AO middleware that specifies the composition of components and aspects using an architectural description language (ADL) [29], named DAOP-ADL [27]. This ADL-based approach provides an interesting complement to M-Stage. After all, using an ADL, application deployers are able to comprehend the overall aspect-component composition, facilitating a better understanding and easier verification of the application. This is of course an important software engineering quality that improves the safety and robustness when deploying aspects. M-Stage could also be nicely complemented by model-driven middleware (e.g. [25, 26, 24] In model-driven middleware, multiple design models of aspects and applications can be specified, composed and possibly verified. Once composed, these models can be automatically synthesized to deployment descriptors for a specific (non-AO) middleware platform of choice [25] or to middleware implementations itself [26]. An approach with similar goals to M-Stage in this context is the CoSMIC and the DAnCE toolsets [25]. These toolsets specifically address crosscutting deployment and configuration concerns of distributed real-time and embedded systems. The difference between M-Stage and CoSMIC/DAnCE is that the latter targets system life-cycle challenges in standard middleware, while M-Stage addresses reuse problems in AO middleware.

5 Conclusion

M-Stage is an aspect-component model that offers reuse and refinement of compositions in distributed applications that are built on AO middleware. Key elements in M-Stage are its support for multi-stage composition and its multi-stage join point model. Multi-stage composition supports modularization and refinement of AO composition logic across multiple composition and deployment stages. The multi-stage join point model enables abstraction of certain context information during the early composition stages. We have illustrated the power of M-Stage by applying the model in a realistic distributed application where we analyzed the reuse and adaptation potential of the M-Stage model.

References

1. C. Szyperski. Component software: beyond object-oriented programming. Second Edition. ACM Press/Addison-Wesley.
2. G. Heineman & W. Councill. Component-based Software Engineering. Addison-Wesley.
3. G. Kiczales. Aspect-Oriented Programming. In Proc. ECOOP'97.
4. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold. An Overview of AspectJ. In Proc. ECOOP'01.
5. E. Hilsdale & J. Hugunin. Advice Weaving in AspectJ. In proc. AOSD'04.
6. R. Filman, T. Elrad, S. Clarke & M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2004.
7. R. Pawlak, L. Seinturier, L. Duchien & G. Florin. JAC: A Flexible Solution for Aspect-oriented Programming in Java. In Proc. Reflection'01.
8. M. Fleury & F. Reverbel. The JBoss extensible server. In Proc. Middleware 2003.
9. B. Lagaisse & W. Joosen. True and Transparent Distributed Composition of Aspect-Components. In Proc. Middleware'06.
10. M. Pinto, L. Fuentes & J.M. Troya. A Dynamic Component and Aspect-Oriented Platform. The Computer Journal,2005.
11. L.D.B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine & D. Suve. Explicitly distributed AOP using AWED. In Proc. AOSD'06.
12. T. Cohen & J.Y. Gil. AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework. In Proc. ECOOP'04.
13. JBoss AOP homepage, http://labs.jboss.com/jbossaop
14. Spring website, http://www.springframework.org/
15. AspectWerkz homepage, http://aspectwerkz.codehaus.org/
16. D. S. Batory, J. N. Sarvela and A. Rauschmayer. Scaling Step-Wise Refinement. In Proc. ICSE 2003, pp. 187-197
17. A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In Proc. of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005), June 2005.
18. E. Wohlstadter and P. T. Devanbu. Aspect-Oriented Development of Crosscutting Features in Distributed, Heterogeneous Systems. in Transactions of Aspect-Oriented Software Development II, 2006, pp. 69-10.
19. E. Wohlstadter, S. Tai, T. A. Mikalsen, I. Rouvellou and P. Devanbu. GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. ICSE 2004, pp. 189-199/
20. M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. AOSD 2004, pp. 7-15.
21. C. Atkinson, T. Kühne. Aspect-Oriented Development with Stratified Frameworks. IEEE Software 20(1), 2003, pp. 81-89.
22. N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the CORBA Component Model. In Component-based software engineering: putting the pieces together, 2001, pp. 557-571.
23. R. Monson-Haefel. *Enterprise JavaBeans, 3rd Edition.* O'Reilly, September 2001.
24. J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modeling. In Proc. GPCE 2003, pp 151-168.
25. G. Deng, D. C. Schmidt, A. S. Gokhale: Addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems via aspect-oriented and model-driven software development. In Proc. ICSE 2006, pp. 811-814.
26. C. Zhang, D. Gao and H. Jacobsen. Generic Middleware Substrate Through Modelware. in Proc. Middleware 2005, pp. 314-333.
27. M. Pinto, L. Fuentes, and J.M. Troya. DAOP-ADL: An architecture description language for dynamic component and aspect-based development. In Proc. GPCE 2003, pp. 118-137.
28. N. Pessemier, L. Seinturier, T. Coupaye and L. Duchien. A Model for Developing Component-based and Aspect-oriented Systems. In proc. Software Composition 2006.
29. M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, 1996.