Interactions in Aspect-Oriented Middleware


Abstract — Middleware platforms typically provide a series of services that must be composed and configured according to the target run-time environment. Often, interactions occur between these services that can cause run-time errors when these services conflict or dependencies are not fulfilled. This paper aims to manage these interaction issues in an Aspect-Oriented middleware platform by allowing interaction contracts between services to be specified which are enforced at run-time. Our approach was validated by applying it to a series of interaction issues that were discovered when implementing middleware services in our prototype Aspect-Oriented middleware platform - CustAOMWare.

Index Terms — Middleware, aspect components, interactions, contracts.

I. INTRODUCTION

When composing middleware services (e.g. transaction, security, persistence etc. services), interactions often arise between these services, causing a variety of issues that need resolving. It is important that the services are correctly configured for the target run-time environment and that all services are compatible (i.e. no conflicts occur and all dependencies are fulfilled). Similar constraints also occur when composing aspects, in that aspects can conflict and also have dependencies that need to be met [1]. The purpose of this paper is to investigate and propose a solution to manage such interaction issues in the context of an Aspect-Oriented (AO) middleware platform.

AO techniques have been applied to a variety of middleware platforms to aid the composition of middleware services and resolve some of the issues raised as the result of interactions [2-5]. However, this body of work does not go far enough in explicitly specifying the conflicts and dependencies that occur in AO middleware and AO in general [1]. Allowing the explicit specification of conflicts and dependencies is the first step towards managing and dealing with such interaction issues.

In the context of this work, middleware services are implemented using a series of aspects and components, with the services composed together using aspects. As such, interactions between services can be resolved by manipulating the aspects’ composition.

Initially, this work will focus on two broad categories of aspect interactions [6]: conflicts and dependencies. A conflict (a negative interaction) between two aspects occurs when the data/behavior added by one aspect is incompatible with another and should be prevented. Conversely, a dependency (a requires relationship) between two aspects occurs when one aspect provides data or functionality that is needed by a second aspect to operate correctly. Further sub-sets of these relationship types are detailed in Section 2.

Conflicts and dependencies can occur at a variety of points within the target system. Typically, interactions occur on some shared element such as a joinpoint, component instance, component type or meta-data item (we assume that aspects provide and consume shared metadata). However, a set of aspects could interact without sharing a common element (apart from the base-application). For example, a security aspect could increase resource usage resulting in an aspect with real-time constraints to miss deadlines. It is vital that any conflicts and dependencies are specified with the correct scope (i.e. in terms of shared elements if they exist) to ensure that only valid conflicts and dependencies are enforced.

The solution presented in this paper includes a component model that supports a well-defined interaction model. This interaction model supports a variety of relationships including: conflicting elements, required elements, precedence between elements and resolution elements. These relationships are specified using interaction contracts which are evaluated at run-time to ensure conflicts do not occur and dependencies are fulfilled. This solution is validated using a prototype AO middleware platform. Explicitly specifying these contracts improves the management and control of such interactions.

The remainder of this paper is structured as follows: Section 2 details the core concepts used in our approach. Section 3 provides an overview of the run-time elements that support the concepts in Section 2. Section 4 then applies the approach to a case-study. Finally, Sections 5 and 6 conclude this paper by discussing related work and summarizing its findings.

II. APPROACH: CORE CONCEPTS

This section describes the core concepts used to manage the issues outlined in Section 1. First, we give a brief overview of the AO-component model on which the middleware services
are built.

A. AO Component Model

Middleware platforms often employ component-based solutions as components can be independently deployed and composed without modification, according to a particular composition style [7]. This enables middleware platforms to be easily configured and customized [8]. Furthermore, an AO composition style can be applied and so add further benefits to traditional component models and allow crosscutting concerns to be encapsulated. In our approach, this introduces the notion of Aspect-Components (ACs), which are regular components playing the role of aspects; as such, both components and ACs must conform to the structure illustrated in Figure 1.

![Diagram of AO Component Model]

A capsule is used to encapsulate all other elements including component types that can be dynamically instantiated to create component instances (henceforth components). Components interact using interfaces and bindings. These bindings (which themselves are components) bind a required interface to a provided interface.

AO functionality is built on top of this component model by providing AO extensions. A specialized AOBinding can be used to implement AO functionality such as managing and executing advice-chains (i.e. a sequence of advice attached to a joinpoint). ACs can then be added to these bindings to weave advice to the correct joinpoints dynamically.

In addition to this, an AOLoader component is used to process AO-composition XML specifications. These AO-compositions ensure that ACs are correctly composed with the base application, and separate the binding between the ACs and components to promote reuse. Each AO-composition consists of one pointcut to define where the ACs should be applied, and one (or more) advice definitions. These definitions include the component type, interface and operation to identify the advice. The AOBinder then instantiates the AOBindings according to this specification.

B. Aspect Component Kernel

The underlying kernel that supports the component model described in section 2.A provides an API that implements a series of operations (Load, CreateInstance, Bind, etc.) to instantiate the component model at run-time. Importantly for this paper, the kernel also provides two operations that are used to retrieve and associate meta-data (in the form of name-value pairs) from all elements that contribute to the component model (see Figure 1). These operations are called GetMetaData and PutMetaData respectively.

C. Interaction Model

As outlined in Section 1 aspects can interact at a variety of locations. Our interaction model is based on shared elements (i.e. aspects share a common joinpoint, component instance/type or base-application), or shared meta-data attached to component model elements. This model is appropriate to resolve AO conflicts due to the way aspects are composed and interact: aspects are composed by location, and interact via shared meta-data. In terms of location- or aspect-based interactions, four different categories can occur:

- **Conflict (negative interaction)** – one aspect requires the absence of another aspect to function correctly.
- **Requires (dependency)** – one aspect requires the presence of another aspect to function correctly.
- **Precedence (ordered dependency)** – the aspects applied to a common element must be executed in a certain order to function correctly.
- **Resolution (conditionally required)** – a set of incompatible aspects may require the presence of a resolution aspect to enable them to co-exist.

Work implementing a reusable transaction framework [1] has also identified comparable aspect relationships that need resolving and so validates these interaction categories. Similar relationships also exist in terms of the meta-data an aspect requires to be present/absent. The interaction model requires the aspect to explicitly specify the following:

- **Provides Meta-Data** – the meta-data the aspect adds to the meta-data infrastructure; note that this relationship is not applicable in terms of aspects as the AO-composition inherently defines the provided aspects.
- **Conflicting Meta-Data** – the meta-data that cannot be added together on a common element.
- **Requires Meta-Data** – the meta-data the aspect pulls/requires from the meta-data infrastructure.

The proposed contract model (see Section 3) enables each of these conflicts and dependencies to be explicitly specified. To offer a consistent mechanism for specifying both aspect and meta-data based contracts, aspect composition is mapped to a meta-data infrastructure. This involves adding meta-data to each of the advised elements. From this the necessary contracts can be specified in terms of the meta-data representing the presence of ACs or the meta-data that is required. For example, if an aspect invokes GetPutMetaData, it must explicitly define the meta-data it pulls or adds to enable the contracts to be assessed (Section 3.B). As the aspect composition is also represented using meta-data, it too requires meta-data to be added to the advised elements to announce the presence of these ACs.

III. FLAox Aspects Component Kernel

This section provides an overview of the kernel run-time prototype that has been implemented (called FLAox) and how it supports the contract model detailed in Section 2.
A. Kernel Prototype

A prototype of the kernel to support the component model described in Section 2 has been created using C# on .NET 2.0. At the core of this prototype is the IComponentInstanceImple abstract class which implements the IComponentInstance interface. All components (including the kernel) must extend this abstract class. ACs must extend the IAdviceInstanceImple abstract class (which implements the IAdviceInstance interface) and provides the proceed() operation. Both extend IComponentInstance and IComponentInstanceImple respectively allowing ACs to be treated as regular components.

The implementation of AOBindings is based on .NET dynamic proxies: which extends RealProxy, and implements IAdviceInstance to comply with the component model.

flAOx provides dynamic composition that allows both regular components and ACs to be composed at run-time. This has consequences on the contract assessment procedures as the structure of the composed components can be altered at run-time and so contracts can also be broken at run-time. As a result of this, it is necessary for any contract enforcement procedure to be executed at run-time.

B. Basic Aspect Interaction Contracts

Two different views of the contract specifications are provided in flAOx: the contracts can be listed separately in the AO-composition specification mentioned in Section 2.A (as XML) or can be specified as annotations (in the form of .NET attributes) within the actual AC implementation they relate to. However, the underlying behavior of the contract enforcement of these two views is identical. For the remainder of the paper, the annotation format is used.

Requires Contracts. This type of contract is used to specify either the meta-data or other ACs that must be present for a particular AC to operate correctly. This contract is used to ensure that all dependencies of the AC are met prior to its execution. The scope – i.e. the shared element where the AC expects the required entity (i.e. meta-data or AC) to be found – must also be specified. Scopes include: capsules, component types, instances, invocations (joinpoints), threads and global.

The following code illustrates how this contract should be used:

```csharp
ComponentInstance ins = Kernel.getInstance("advised id");
String name = ins.getName("User");
```

This extracts the User meta-data from the instance this AC is advising. This requires the following contract to be specified to explicitly state that the User meta-data is expected to be found on the advised instance:

```csharp
requires(instance["User"])```

The same contract construct and scoping types are also used to specify the other ACs that are required:

```csharp
requires(type["Aspect-Component B"])```

This contract defines that Aspect-Component B must be woven to the same type as this AC. By externally specifying this contract, dependencies between ACs are removed and so lower their coupling which in turn increases reuse. Without these contracts, the ACs would have to check the current configuration to ensure the necessary meta-data or aspects were present.

Provides Contracts. In order for the requires contracts to be successfully evaluated, the meta-data and ACs provided to the component infrastructure also need to be specified. Each AC must specify the meta-data it adds and the scope to identify the element which the meta-data is added to. For example, to satisfy the earlier required contract, an AC must add meta-data to a component instance using the following code:

```csharp
ComponentInstance ins = Kernel.getInstance("advised id");
ins.addMetaData("User");
```

This operation has to be externally specified in a provides contract to fulfill the requires contract; this will ensure that when the contracts are verified no violations occur:

```csharp
provides(instance["User"])```

As ACs could also be required to fulfill a certain dependency, they also need provides contracts to be added. However, this information is already implicit in the AO-composition. Instead of explicitly specifying these provides contracts for ACs and so introducing duplicate information, the provides contracts are added to the component elements automatically. Contracts are added to the advised elements to identify each AC composed with them.

The scoping mechanism used above does not explicitly specify the instance which the meta-data is added to. This is to allow generic contracts to be specified where the ACs must share the scoping element (i.e. the ACs must be applied to the same capsule, component type, instance, joinpoint or thread). Future work is planned to allow named scoping elements to be defined (e.g. name a component instance where an AC is required). This will allow interactions between disjoint ACs with no element in common to be specified.

C. Advanced Aspect Interaction Contracts

In addition to these basic requires and provides contracts other more complex contracts must also be supported to implement the interaction model described in Section 2.B.

Conflict Contracts. Both aspect- and meta-data-based interactions require contracts to be specified which define sets of elements that conflict. This necessitates a contract construct that allows all the ACs or meta-data that should be absent from the specified scope to be identified. This is achieved via the conflict contract type. For example, if the developer wishes to specify that Aspect-Component B should not be applied within the same thread as Aspect-Component A, the following contract can be specified in the context of Aspect-Component A (i.e. within its AO-composition or implementation):

```csharp
conflict (thread["Aspect-Component B"])```

In this case, Aspect-Component A is only executed when it is woven in different threads to Aspect-Component B. As Aspect-Component A specifies the contract, it is this AC that is not executed. However, if both ACs specify equivalent contracts then neither will be executed.

Precedence Contracts. When requires contracts are specified, there is an implicit temporal ordering associated with these contracts. Required aspects/meta-data should be
executed/added before they are required by other aspects. However, in some cases an aspect may not explicitly require the presence of another aspect/meta-data to operate correctly; instead interaction issues could arise through the incorrect ordering of other aspects. As a result, it is necessary to allow the desired aspect execution order to be defined separately which does not place a dependency on the presence of an aspect.

precedence(type["Aspect-Component A, Aspect-Component B, Aspect-Component C"])  
The ordering is specified using a comma-separated list with the scoping used to define the elements where the ordering is applicable. For example, if the scope is specified as type, the set of ACs specified (or a sub-set of the ACs if only a partial set of them are woven) are ordered when they are applied to a common type.

Resolution Contracts. Finally, the resolution or conditionally required contracts have to be specified in order to allow aspects to be added based on the current configuration of the base application. Note that resolution contracts only define the conditions under which the resolution aspect(s) are required. When these conditions occur the relevant AC is activated based on its AO-composition specification. The following contract specifies such conditions for Aspect-Component A:

resolve(instance["Aspect-Component B"] AND instance["Aspect-Component C"])  
The above contract specifies that Aspect-Component A should only be applied when Aspect-Component B and Aspect-Component C are applied to the same component instance. When these conditions occur, Aspect-Component A will be executed based on its AO-composition specification.

The resolution contract listed above also demonstrates another feature of the contract specification, logical operators can be used to combine conditions and also list alternatives. Logical operators can be used with any contract type to combine contracts. For example, to state that Aspect-Component A conflicts with either Aspect-Component B or Aspect-Component C, the following contract can be specified:

conflict(thread["Aspect-Component B"] OR thread["Aspect-Component C"])  

D. Contract Enforcement  
Before the contracts are enforced, provides contracts must be added to the advised elements to announce the presence of each AC. This is achieved using the meta-data operations provided by the kernel. Rather than relying on each AC to perform this, the ALOader is responsible for attaching the relevant meta-data to the advised elements.

To validate the contract at run-time, some system-wide element is needed. This element is clearly a crosscutting concern and so is implemented as an AC. The ContractEnforcement AC is added to all advice-chains and is given a high priority so it is executed first in the chain ensuring that it assesses each contract before any other AC is executed. The ContractEnforcement AC is then responsible for querying each contract specification of all subsequent ACs in the advice-chain. If any contract violations are detected, the ContractEnforcement AC is responsible for either manipulating the advice-chain accordingly or issuing error messages notifying the developer of the contract violation and preventing the execution of the affected ACs. To avoid reassessing the contracts each time an advice-chain is encountered, the outcome of the contract assessments are cached, and only when changes to the advice-chain configuration are made is the cache entry expunged.

Under certain circumstances it may be undesirable to assess the contracts at run-time due to the performance penalty imposed. However, in systems where dynamic weaving of aspects is supported it is also necessary to perform the contract enforcement dynamically. This is to ensure that all context information is taken into account due to the possibility of configuration changes occurring after the system has started.

IV. VALIDATION IN CUSTAOMWARE  
As outlined in Section 1, interaction issues can occur when composing and configuring middleware services. This section details some of these issues and describes how the interaction contract model outlined in Section 3 can be used to resolve/prevent these issues.

This work has involved the creation of an AO middleware platform called custAOMWare which is based on the flAOx aspect component model. A number of common middleware services were implemented for this middleware platform including: distribution; persistence; and security (authentication and authorization). When implementing these services, limitations with regard to the flexibility and customizability were encountered due to certain relationships having to be hard-coded within the services to ensure other necessary services were correctly composed and also to prevent undesirable interactions. The following sections detail how the contracts were applied to the issues encountered.

A. Requires (Dependency)  
The Security service implemented for custAOMWare involved two sub-services: authentication and authorization. For the authorization service to operate correctly it requires meta-data to be attached to the advised invocation: the identity of the current subject; and the subject’s access credentials. This meta-data is provided by the authentication service. However, the requires contract should not be defined in terms of the authentication AC but instead in terms of the actual required meta-data as in future configurations the meta-data could be provided by a different service. The necessary contract for this service:

requires(invocation["ID"] AND invocation["credentials"])  

Similarly, the authentication service needs to explicitly define the meta-data it provides:

provides(invocation["ID"] AND invocation["credentials"])

This ensures that the required meta-data is provided to the authentication service so that it can make the necessary checks to determine whether the user has the required access credentials.
With both of these contracts specified, the ContractEnforcement AC will be executed prior to the authorization service and will ensure that the necessary meta-data will be attached to the invocation. Previously, the check would have had to be made by the authorization service itself, with the dependency towards the identity and access credentials meta-data not being explicit.

B. Conflict (Negative Interaction)

In some composition scenarios, it was discovered that the distribution and persistence services conflict. The distribution service enables component instances to be accessed remotely, whereas the persistence service assumes that the component instances are local. This causes conflicts due to the possibility of concurrent accesses to the persistent repository. As a result, the distribution and persistence services cannot be both applied to the same component instance simultaneously.

Unlike the previous dependency example, the conflict stems from the actual behavior of the two services rather than some meta-data. As such, the conflict contract must be specified in terms of the ACs. The necessary persistence service contract is as follows:

```
conflict(instance["distribution"])
```

This contract prevents the distribution and persistence services being executed on a common component instance. The resolution of this contract violation cannot be achieved automatically resulting in an error message being displayed and the persistence AC not being executed. Unlike, the dependency example, the distribution service does not have to explicitly specify a provides contract related to this conflict as the AOLoader will automatically add the necessary meta-data to announce the presence of this AC.

C. Precedence (Ordered Dependency)

Unlike the persistence service, the security service can be applied to the same component instances as the distribution service (to offer secure access to remote components). However, these two services must be executed in the correct order (security before distribution) to ensure subjects are authenticated prior to the remote service being accessed. The nature of this dependency means it is an application-wide contract, so it should be applied to all elements where these two services interact and not be limited to a certain sub-set of elements. The contract specified to enforce this relationship does not have an associated scope (to signify a global scope) and is defined as follows:

```
precedence(ClientAuthentication ,Distribution)
```

In this example, the ContractEnforcement AC is able to take proactive steps to resolve any cases where this contract is violated. The actions necessary to resolve (i.e. the necessary reordering) this violation are explicit from the contract specification. The ContractEnforcement AC is able to re-order the advice in the advice-chain and so fulfill the contract.

D. Resolution (Conditional Collaboration)

As described in Section 4.B, the distribution and persistence services can conflict when executed on the same component instance. This is due to the possibility of concurrent updates. This issue can be resolved by the introduction of a transaction service to ensure the ACID properties [9] (Atomicity, Consistency, Isolation and Durability) are enforced. When the persistence service is deployed in isolation, this transaction service is unnecessary as the transaction mechanisms of the repository can handle local updates.

The transaction service is only needed when the distribution and persistence services are applied to the same component instances. This resolution contract is specified as follows:

```
resolve(instance["distribution"] AND instance["persistence"])
```

As this contract relates to the conditions when the transaction service has to be applied it must be defined in the context of its AO-composition or AC implementation. Initially, the transaction service will not be executed. However, if the ContractEnforcement AC detects that the distribution and persistence have been applied to the same component instance, the transaction service will be enabled according to its AO-composition specification. This approach requires that the transaction service AO-composition defines the correct deployment information (i.e. pointcut etc.) to resolve the conflict between the distribution and persistence services.

E. Summary

This section has covered a series of examples which illustrate the various conflicts and dependencies that can occur in middleware. Our contract model was then applied to these examples to demonstrate how contracts can be specified to explicitly define these relationships to resolve any undesirable interactions and ensure all dependencies are fulfilled. In cases where an automatic resolution is not possible the developer is notified of such cases to allow him/her to prevent future problems occurring while ensuring the affected aspect-components are not executed.

V. RELATED WORK

Identifying and resolving aspect interaction issues are active research topics in the AOSD community. A number of approaches have been proposed to resolve such issues [2-5]. The majority of these relate to providing language extensions that describe certain pre and post conditions that must be fulfilled for the aspect behavior to be correctly applied. For example, CompAr [3] provides language constructs to define execution constraints with regard to advice. However, the limitation of such approaches is that constraints are specified on a per-joinpoint basis meaning that aspect interactions must occur at a common point. Our approach goes beyond this, while still allowing conflicts to be specified on a per-joinpoint basis, the conflicting aspects can intersect on a broader scope of elements or even have no elements in common. Additionally, the majority of existing approaches are limited to resolving issues related to aspect ordering such as ensuring the before and after parts of around advice are executed correctly due to issues regarding the continuation of advice-chains using the proceed operation. The contracts that are able
to be specified in flAOx allow precedence issues to be specified but also allow complex relationships to be specified regarding the presence and absence of aspects.

The relationship types used in our approach are comparable to those found in feature models [10] which are often used in software product lines [11]. Feature models allow inclusive and exclusive relationships to be specified between features to ensure compatibility. However, feature models are used exclusively at build-time (i.e. to build a product family member to meet a certain specification) whereas our approach focuses on the dynamic composition issues that can occur when run-time changes occur in middleware platforms.

Despite these limitations the approaches listed above do have a role to play when the contracts in flAOx need to be specified. For example, the results provided by the CompAr compiler can be used as a basis for constructing the correct contracts. Another relevant approach to achieve this is described in [2]. This approach involves using an expert system implemented using Prolog and OWL whereby domain expertise is collected regarding the interactions in middleware. The configuration of a particular middleware configuration can then be used to query the expert system to identify any potential issues. The outcome of these queries can be used to construct the contracts used in flAOx.

The notion of a WrappingController, introduced in [12], is also relevant here. The WrappingController used in JAC is similar in functionality to our ContractEnforcement AC and is responsible for manipulating the subsequent advice-chain. The limitation of the WrappingController approach is that the logic necessary to manage the advice-chain has to be specified programmatically and so does not make explicit the relationships between aspects.

The approach described in this paper is an extension of earlier work detailed in [13]. This earlier approach involves specifying policies to ensure aspect dependencies are met and conflicts are not introduced from dynamically weaving aspects. However, the structure of these policies is lost at run-time so it becomes difficult to query and manipulate the policies at run-time. Furthermore, it was not possible to scope the relationships to a particular shared element. Instead, all relationships had to be specified as a global context. The contracts used in flAOx do not have this limitation.

Finally, JM (Just in Time Middleware) [14] is a relevant approach worthy of comparison. JM requires three specifications to generate a minimal executable middleware whereby a number of constraints are specified: i) feature dependency: implementation of certain functionality is composed from others. (e.g. Aspect1 depends on aspect2 together with either aspect3 or aspect4); ii) composition constraints: dictate inclusions and exclusions of functionalities reflecting certain conditions about the environment (e.g. J2ME vs J2SE); and iii) convolution descriptions: propositions describing the interactions among features: which aspects should occur together and which not. Although similar conflicts and dependencies can be specified, JM only provides a build-model and does not detect runtime issues.

VI. CONCLUSION

This paper has outlined extensions to an AO component model and run-time platform that allows specifying contracts that explicitly define various interactions, dependencies and conflicts between aspect components. This approach has been validated by applying the approach to a series of interaction issues that occur when implementing services for a flexible and customizable AO middleware platform. By explicitly specifying these relationships in an external contract definition, the coupling between services has been reduced by eliminating the need for the services to check for interaction violations. A run-time aspect is then used to check for contract violations to take into account dynamic adaptations that commonly take place in middleware.

REFERENCES