

# Interactions in AO Middleware

P. Greenwood, B. Lagaisse, F. Sanen, G. Coulson, A. Rashid, E. Truyen and W. Joosen

**Abstract**— Middleware platforms often provide a series of services that must be composed and configured according to their deployment and run-time context. Often, interactions occur between these services that can cause run-time errors when these services conflict or dependencies between them are not fulfilled. This paper aims to solve these interaction issues in AO middleware 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 AO middleware platform - custAOMWare.

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

## I. INTRODUCTION

WHEN composing middleware services, interactions often arise between these services causing a variety of issues that need to be resolved. It is important that the middleware services are correctly configured for the current context 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. The purpose of this paper is to investigate and propose a solution to these interaction issues in the context of an 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 [1-4]. However, this body of work does not go far enough in explicitly specifying the conflicts and dependencies that occur in AO middleware. Allowing the explicit specification of conflicts and dependencies is the first steps towards managing and dealing with such interaction issues. In the context of this work, we assume that middleware services are implemented using a series of aspects and components, with the services composed together using aspects. As such, interactions between services solely occur between aspects.

Manuscript received May 7, 2007. This work was supported in part by AOSD-Europe.

P. Greenwood, G. Coulson and A. Rashid are with Computing Department, Lancaster University, Lancaster, LA1 4WA UK (e-mail: greenwop@comp.lancs.ac.uk, geoff@comp.lancs.ac.uk, awais@comp.lancs.ac.uk).

B. Lagaisse, F. Sanen, E. Truyen and W. Joosen, are with Department of Computer Science, K. U. Leuven, Heverlee, Belgium (e-mail: bertl@cs.kuleuven.be, frans.sanen@cs.kuleuven.be, eddy.truyen@cs.kuleuven.be, wouter@cs.kuleuven.be).

Although the relationships between middleware services are acting as our driver to study aspect-interactions, the various interactions are still valid in other domains.

Initially, this work will focus on two broad categories of aspect interactions [5]: *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 required relationship) between two aspects occurs when one aspect provides data or functionality that is needed by a second aspect to operate correctly. In other words, a dependency defines a required relationship from one aspect to another. Further sub-sets of these relationship types are detailed in Section 2.

These 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 or meta-data item (i.e. aspects provide or consume shared meta-data). However, a set of aspects could easily interact without sharing a concrete common element (the only common element is 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 the shared element if one exists) to ensure only conflicts and dependencies are applied to the relevant element.

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 will improve 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 implemented to 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 resolve the

issues outlined in Section 1. First, a brief overview of the aspect-component model on which the middleware services are built is described.

### 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 [6]. This enables middleware platforms to be easily configured and customized [7]. Furthermore, an AO composition style can be easily applied and so add further benefits to traditional component models. In our approach, this involves 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 same structure as illustrated in Figure 1.

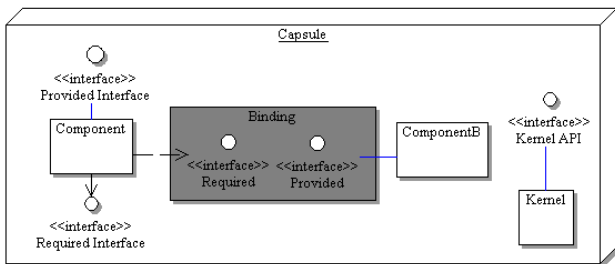


Figure 1 The component model.

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) are created through *required* and *provided* interfaces.

The 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 allow advice to be woven to the correct joinpoints.

In addition to this, an *AOLoader* component is necessary that is responsible for reading and processing *AO-Composition* specifications. These AO-compositions are used to define information to ensure the ACs are correctly applied to the base application. Each AO-composition consists of one pointcut element 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 to be applied. The *AOBinder* is then responsible for instantiating the *AOBindings* according to this specification.

### B. Interaction Model

As outlined in Section 1 aspects can interact at a variety of locations. The interaction model used 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 conflicts within the AO model due to the way aspects are composed and interact. For example, ACs

are composed based on location and aspects 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.
- *Required (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.

A similar set of relationships also exist in terms of meta-data an aspect requires to be present/absent. The interaction model requires the aspect to explicitly specify the following.

- *Provided Meta-Data* – the meta-data the aspect adds to the meta-data infrastructure. Note 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.
- *Required Meta-Data* – the meta-data the aspect pulls/requires from the meta-data infrastructure.

The proposed contract model (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, the 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. The latter is elaborated upon in Sections 3 and 4.

### C. Aspect Component Kernel

The underlying kernel that supports the component model described in section 2A 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 (*Get/PutMetaData*) that can be 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).

If an aspect invokes the *Get/PutMetaData*, it must explicitly define the meta-data it adds or pulls to enable the contracts to be assessed (Section 3B). As the aspect composition is also represented using meta-data which involves adding meta-data and contracts to the relevant elements to announce the presence of these ACs

## III. FLAOX ASPECT 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 *IComponentInstance* interface and *IComponentInstanceImpl* abstract class. All components (including the kernel) must extend this abstract class and in turn extend this interface. ACs must extend the *IAdviceInstanceImpl* abstract class and *IAdviceInstance* interface which extends *IComponentInstance*. *IAdviceInstanceImpl* extends the *IComponentInstanceImpl* class and provides the *proceed()* operation. This hierarchy enables ACs to be treated as regular components.

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

fIAOx 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 fIAOx: the contracts can be listed separately in the AO-composition specification mentioned in Section 2A (as XML) or can be added as annotations (in the form of attributes) to the actual AC implementation they relate to. However, the underlying behavior of the contract enforcement of these two views is identical.

*Required Contracts.* This type of contract is used to specify either the meta-data or other ACs that must be woven 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 or shared element where the AC expects the required entity (i.e. meta-data or AC) to be found must also be specified. Scoping types include: capsules, component types, instances, invocations (joinpoints), threads and global. The following code example illustrates how this contract should be used:

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

The code above 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:

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

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

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

This contract defines that *Aspect-Component B* must be woven to the same type as the AC which specifies the contract. By externally specifying this contract, dependencies between ACs are removed and so lower their coupling which in turn increases reuse. Previously, the ACs would have to

check the current configuration to ensure the necessary meta-data or aspects were present.

*Provided Contracts.* In order for the required 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:

```
ComponentInstance ins= Kernel.getInstance("advised id");
ins.putMetaData("User","Name");
```

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

```
provide(instance["User"])
```

As ACs could also be required to fulfill a certain dependency, they also need provided contracts to be specified accordingly. However, this information is already implicit in the AO-composition mentioned earlier. Instead of explicitly specifying these provided contracts for ACs and so introducing duplicate information, the provided contracts are added to the component elements automatically, whereby a contract is added for each AC that advises that particular element. The contracts are added to the advised elements to ease the process of evaluating the required contracts. Rather than having to query the pointcuts of ACs to verify whether the contracts are met or not, the advised element can be easily queried to verify if the required AC is present on that element.

The scoping mechanism used above does not explicitly specify the instance which the meta-data is provided to. This is to allow generic contracts to be specified, whereby the component instance has joinpoints intercepted by both ACs and so when *instance* is specified this is assumed to mean the *same* component instance. Future work is planned to allow a specific instance (or component type, thread etc.) to be specified when the joinpoints are disjoint.

### C. Advanced Aspect Interaction Contracts

In addition to these basic provided and required contracts other more complex contracts must also be supported to implement the interaction model from Section 2B.

*Conflict Contracts.* Both the aspect and meta-data based interactions require contracts to be specified which define sets of aspects or meta-data that conflict. This requires a contract construct to be provided that is similar to the required contract described in Section 2B but instead lists all the aspects or meta-data that should be absent. This is achieved via the *conflict* contract type. For example, if we wish 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):

```
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 to ensure interaction issues do not occur. As Aspect-Component A

specifies the contract, it is this aspect that is not executed. However, if both aspect components specify the same equivalent contracts then neither will be executed.

*Precedence Contracts.* When provided and required contracts are specified, there is implicit temporal ordering associated with these contracts. In such a contract, provided aspects/meta-data should be added/executed *before* they are required by other aspects. However, the limitation of this is that the collaboration contracts specify a dependency towards the provided elements, in some cases the interactions may only be dependent on the ordering and not on the actual presence of the aspects. As a result, it may be necessary to explicitly specify the ordering of aspects separately.

*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 aspects specified (or a sub-set of the aspects if only a partial set of the aspects 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 applied 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 deployed according to 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, specify 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, provided 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 AOLoader is responsible for attaching the relevant contract (as meta-data) to the advised component.

For the contracts to be validated at run-time, some system-wide element needs to be applied that will check the current

system configuration to ensure no contracts are being violated. This element is clearly a crosscutting concern and so is implemented as an AC. The *ContractEnforcement* aspect is added to all advice-chains which is given a high priority to ensure that it is executed first in the chain and that it assesses the contracts before any other aspect is executed. The *ContractEnforcement* aspect is then responsible for querying each of the contract specifications for all subsequent aspects in the advice-chain. If any contract violations are detected, the *ContractEnforcement* aspect is responsible for either manipulating the advice chain accordingly or issuing error messages notifying the user 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.

In some circumstances it may not be desirable to assess the contracts at run-time due to the performance penalty imposed. However, by performing the contract assessment as late as possible all the available context information can be taken into account. For example, suppose Aspect-Component A requires Aspect-Component B. If Aspect-Component B was woven dynamically at run-time and Aspect-Component A was woven at load-time then any contract analysis performed when Aspect-Component A was woven would raise a contract violation error. However, by performing the analysis immediately prior to the advice chain execution would not raise a contract violation error (assuming that Aspect-Component B has been woven prior to this point).

## 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* based on the *fIAOx* 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 services being composed. The following sections details how the contracts were applied to the issues encountered.

### A. Requires (Dependency)

The initial implementation of the Security service in *custAOMWare* involved the implementation of 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/aspect. However, the specified contract should not be specified in terms of the authentication aspect but instead in terms of the actual required meta-data as in future configurations the meta-data could be provided by a different aspect/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"])
```

With both of these contracts specified, the ContractEnforcement aspect will be executed immediately 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, where as 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 necessary contract must be specified in terms of the applied services/aspects. The necessary persistence service contract is as follows:

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

This contract will ensure the ContractEnforcement aspect prevents both the distribution and persistence services being executed on the same component instance. The resolution of this contract violation cannot be achieved automatically, and so an error message is also presented to the developer. Unlike, the dependency example, the distribution service does not have to explicitly specify a *provided* contract related to this conflict as the AOloader will automatically add the necessary provided contract to announce the presence of this service.

*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 and it should be applied to all elements where these two services interact and not 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 aspect is able to take proactive steps to resolve any cases where this contract is violated. The actions necessary to resolve (i.e. the order necessary) this violation are explicit from the contract specification. The ContractEnforcement aspect is able to re-order the advice in the advice-chain and so fulfill the contract.

*D. Resolution (Conditional Collaboration)*

As described in Section 4B the distribution and persistence services cannot interact on the same component instance. This is due to the possibility of concurrent updates occurring. This issue can be resolved by the introduction of a transaction service to ensure the ACID properties [8] (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 applied, however, if the ContractEnforcement aspect detects that the distribution and persistence have been applied to the same component instance, the transaction service will be deployed 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 illustrates the various conflicts and dependencies that can occur in middleware. The proposed 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 the cases where an automatic resolution is not possible the developer is notified of such cases to allow them to prevent future problems occurring while ensuring the affected aspect-components are not executed.

V. RELATED WORK DISCUSSION

Identifying and resolving aspect interaction issues is an active research topic in the AOSD community. A number of approaches have been proposed to resolve such issues [1-4]. The majority of these approaches 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 [2] provides language constructs to define execution constraints with regard advice code. 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 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.

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 [1]. 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 [9], is also relevant here. The WrappingController used in JAC is similar in functionality to our ContractEnforcement aspect and is responsible for manipulating the subsequent advice-chain. The limitation with the WrappingController approach is that the logic necessary to manage the advice-chain has to be specified programmatically and so does not make the relationships between aspects explicit.

The approach described in this paper is an extension of earlier work detailed in [10]. 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 specified 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 represent a combination of global scoping and joinpoint specific scoping approaches.

Finally, JiM (Just in Time Middleware) is another relevant approach worthy of comparison [11]. JiM requires three specifications to generate a minimal executable middleware whereby a number of constraints are specified: feature dependency - implementation of certain functionality is composed from other functionalities. (e.g: Aspect1 depends on aspect2 together with either aspect3 or aspect4); composition constraints - dictate inclusions and exclusions of

functionalities reflecting certain conditions about the environment (e.g. J2ME vs J2SE); convolution descriptions - propositions describing the interactions among features: which aspects should occur together and which not. Although similar conflicts and dependencies can be specified, JiM only provides a build-model rather than detecting 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 discovered 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 then is used to check for contract violations to take into account dynamic adaptations that commonly take place in middleware.

## REFERENCES

- Sanen, F., E. Truyen, W. Joosen. *Managing Concern Interactions in Middleware*. in *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. 2007. Cyprus: Springer.
- Pawlak, R., L. Duchien, L. Seinturier. *CompAr: Ensuring Safe Around Advice Composition*. in *7th International Conference on Formal Methods for Open Object-Based Distributed Systems*. 2005. Athens.
- Wohlstadtter, E., S. Tai, T. Mikalsen, I. Rouvellou, P. Devanbu. *GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions*. in *26th International Conference of Software Engineering (ICSE)*. 2004: IEEE Computer Society.
- Douence, R., P. Fradet, M. Sudholt. *Composition, Reuse and Interaction Analysis of Stateful Aspects*. in *3rd International Conference on Aspect-Oriented Software Development*. 2004. Lancaster, UK: ACM.
- Sanen, F., N. Loughran, A. Rashid, A. Nedos, A. Jackson, S. Clarke, E. Truyen, W. Joosen. *Classifying and Documenting Aspect Interactions*. in *5th Workshop on Aspects, Components and Patterns for Infrastructure Software (ACP4IS) co-located with AOSD06*. 2006. Bonn, Germany.
- Szyperski, C., *Component Software: Beyond Object-Oriented Programming*. 1998: Addison Wesley.
- Coulson, G., P. Grace, G. S. Blair, W. Cai, C. Cooper, D. Duce, L. Mathy, W. K. Yeung, B. Porter, M. Sagar, J. Li, *A Component-Based Middleware Framework for Configurable and Reconfigurable Grid Computing*. *Concurrency and Computation: Practice and Experience*, 2005. **18**(8): p. pp 865-874.
- Kienzle, J., S. Gelineau. *AO Challenge - Implementing the ACID Properties for Transactional Objects*. in *5th International Conference on Aspect-Oriented Software Development (AOSD)*. 2006. Bonn, Germany: ACM.
- Pawlak, R., L. Seinturier, L. Duchien, G. Florin. *Dynamic Wrappers: Handling the Composition Issues with JAC*. in *TOOLS-USA*. 2001. Santa Monica, USA.
- Greenwood, P., L. Blair, A. *Framework for Policy Driven Auto-Adaptive Systems Using Dynamic Framed Aspects*. *Transactions on Aspect-Oriented Software Development* 2006. **2**: p. 30-65.
- Zhang, C., D. Gao, H. Jacobsen. *Towards just-in-time middleware architectures*. in *5th International Conference on Aspect-Oriented Software Development*. 2005. Chicago, Illinois, USA.