

# A Semantic Composition Model to Preserve (Re) configuration Consistency in Aspect Oriented Middleware

Bholanathsingh Surajbali, Paul Grace and Geoff Coulson  
Computing Department,  
Lancaster University  
Lancaster, UK  
{b.surajbali, p.grace, geoff} @comp.lancs.ac.uk

## ABSTRACT

Aspect-Oriented Programming enables the isolation and modularisation of crosscutting concerns that are typically implemented in a tangled fashion within the base system. However, the composition of these aspects is not completely orthogonal; with interactions between aspects involving direct and indirect dependencies, and conflicts that can cause runtime inconsistencies when those interactions are not detected. This is particularly true of the dynamic composition and adaptation of aspects within distributed systems; therefore in this paper we propose a semantic composition model to detect and solve these interaction issues at runtime. Our approach can be employed in dynamic AOP middleware, and we evaluate it here within the AO-OpenCom tool. We measure the overhead incurred by the semantic composition model when performing safe dynamic reconfigurations.

## Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures – Patterns (Reflection).

## General Terms

Orthogonal, crosscutting, modularisation

## Keywords

AO Middleware, reflection, aspects, dynamic reconfiguration.

## 1. INTRODUCTION

Aspect Oriented Programming (AOP) is a method of tackling the problems of tangled code i.e. the basic system implementation becomes tangled with code for features such as security, caching, and monitoring. These concerns are implemented as *aspects* which are made up of individual code elements that implement the concern (*advices*). Advices are deployed at multiple positions in a system (*join points*) which are expressed by *pointcuts*—a particular form of composition language. Dynamic AOP, for example as provided by the JAC [9] and DyReS [15] middleware then allow aspects to be composed and adapted at runtime. Further, [5] advocates the use of reflection to perform fine-grained adaptation of the aspect elements.

Aspects are designed to have orthogonal properties so they can be deployed obliviously from one another; however, it is clear that *semantic interactions* can cause compositional inconsistencies to occur i.e. composing aspects that syntactically match with each other, or with the base system may produce behaviour that is in conflict with the original system operation. For example, the composition of both an authorisation and authentication aspect, where the authentication aspect checks the credentials of the user produces a negative interaction if the authorisation aspect gives

access to users before the credentials are verified. Another example of semantic inconsistency is the weaving of an encryption aspect at the sending end join point without the corresponding decryption aspect at the receiving end join point. While these examples can be detected with knowledge of the aspect types, more complicated semantics may prove more difficult to detect. For example a running system having a cache and a security aspect could result in an increase in resource usage resulting in an aspect with real-time constraints to miss deadlines.

The majority of dynamic aspect middleware approaches focus solely on dealing with *syntactic* inconsistencies; that is detecting if aspects have the same type, version, and interfaces. However, semantic inconsistencies are non-trivial to detect, as the semantics of aspects may interfere with each other without sharing a common element. In this paper, we present a semantic composition model for dynamic aspect-oriented, component-based middleware; this provides the capability to describe the various kinds of built-in and external interactions that affect aspect semantic composition at runtime. This is coupled with a method for *semantic interaction resolution* which detects inconsistencies at run-time and supports methods to resolve semantic interactions.

We evaluate our approach within the AO-OpenCom platform for developing dynamic reconfigurable middleware solutions; this demonstrates the following contributions of our approach:

- *Conflict resolution.* We show that complex semantic inconsistencies can be resolved for a number of case-studies with minimal performance overhead.
- *Transparency.* We apply consistent (re)configuration with minimal programmer effort or change to the underlying component model.
- *Flexibility.* New semantic conflicts can be described dynamically to evolve with the running application or domain context. Moreover, the approach can be applied in different compositions approaches and tools; for example we show how both node-local and distributed (re)configuration semantic consistency can be achieved in this paper.

The remainder of this paper is organised as follows. Section 2 examines the types of aspect interactions that may occur. Then, section 3 describes the design of our semantic composition model, followed by section 4 which validates the proposed semantic composition model. Finally, in section 5 we describe the related research and offer our conclusions in section 6.

## 2. AO-COMPONENT INTERACTION

In this section, we describe the potential sources of composition inconsistencies. To present the different types of semantic interactions we use a use case scenario to describe the different types of (re)configuration interactions. We then describe the

general built-in and external interactions influencing the (re) configuration in the aspect-component model.

## 2.1 Use Case Scenario

To motivate the requirement for semantic interaction consistency we present an online gaming system case scenario. The system allows users to play games via a central server that co-ordinates multiple online players. When a user logs in at the start of a game a list of available users based on their preferences are provided such that the user can contact peers and request them to play. In such an environment, there are various types of application requirements in terms of multi-player, mobile-player, real-time and non-real-time gaming; further, peers may operate in different network domains e.g. Internet, Wi-Fi, or ad-hoc wireless networks. To cope with the application and environmental demand a number of dynamic (re)configurations may be required: (i) new mobile users with limited bandwidth may join, requiring a fragmenter aspect to be configured to split data before being sent; (ii) when monitoring users a persistence aspect may be deployed to keep track of authorised users connected to the server; (iii) data may be required to be encrypted to protect the users' privacy; (iv) the authorisation module may be replaced with an updated one filtering users by privileges (e.g. by administrator, game user, etc).

## 2.2 Built-in Interaction

In aspect-component middleware, aspects (which are themselves implemented as component modules) are composed with the base components (hereafter termed components) using AO-Connectors [5, 12, 14, 15]. A list of advices is attached to the connector between a receptacle and a provided interface. This capability is illustrated in Figure 1.

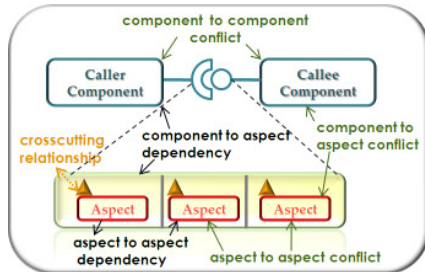


Figure 1: AO-Component Interaction

We now identify and classify the types of interaction that can occur in this local aspect-component model; these fall into three categories: dependency, conflict and indirect interactions.

### 2.2.1 Dependency Interaction

Dependency represents a required interaction between aspects i.e. one aspect instance must be present for the other to operate. In an aspect-component model, such dependencies may occur from the different parties involved; we now show the potential dependencies between different elements in the model:

**Component to component dependency.** The 'caller' component passes invocations through its required interface to the 'callee' component's provided interface. Required interfaces are dependencies that need to be fulfilled in order to guarantee correct semantics of the component model.

**Component to aspect dependency.** In general one assumption is that components must be present first, and aspects being non-functional services are composed later. However, aspects are often

integral, with components depending upon specific aspects. For example, a synchronisation aspect may be required to guarantee the real-time properties of the system, where the communication component is dependent on the synchronisation aspect.

**Aspect to aspect dependency.** Similar to *component to component dependency*, aspects may depend upon each other. For example, an encryption aspect depends upon the corresponding decryption aspect to operate effectively.

### 2.2.2 Conflict Interaction

A conflict interaction represents a negative interaction [6] between two aspects, where the operation of one aspect is detrimental to the other. The causes of these conflicts in the aspect-component model can be classified in terms of:

**Component to component conflict.** The interaction between two components can conflict following reconfiguration. This can be due to a number of factors: incompatible component types, invalid state of component e.g. replacing a component generating unique users' identifiers and the latter generates ids' that were previously created by the old component.

**Component to aspect conflict.** The composition of an aspect with the base system may also cause semantic conflicts. For example, weaving the encryption aspect at the communication component join point may cause the system's throughput to fall below the required level.

**Aspect to aspect conflict.** Composing multiple aspects at a join point is a common source of conflict. This can be caused from mainly from:

- **Ordering.** The order in which aspects are composed influence their execution order. Consider a join point having two security aspects, an authentication aspect and an authorisation aspect. The order in which the aspects are invoked influence the correct system execution. Invoking the authorisation aspect before the authentication aspect may give access to non-authenticated users.
- **Mutual-exclusion.** This involves two aspects that implement concerns having contradictory semantics such that either one of them can be used but not both. For example, in the gaming scenario, when multiple users are playing a live-game with the server using a real-time aspect to guarantee proper updates to be sent across the users, reconfiguring the system by adding a synchronisation aspect may cause both aspects to block; the real-time aspect waiting for the synchronisation and vice-versa, resulting in deadlock at the server.

### 2.2.3 Indirect Interaction

A more complicated interaction that can occur is the indirect interaction between aspects involving multiple oblivious parties. For instance, if an aspect "AC-1" depends on another aspect "AC-2", which itself depends on another aspect "AC-3" but AC-1 has semantic conflicts with "AC-3". Such indirect interactions are harder to detect, and require reasoning at the AO-Connector level to find indirect interactions.

## 2.3 External factors affecting Aspect Interactions

Importantly these interactions are influenced from the following input:

- **Application-specific influences** arising from specific constraints and requirements of the application. For example, composing an encryption and logging aspect depends on the

application-specific requirements. In an untrusted environment, all data needs to be encrypted before the logging aspect reads the data, to preserve the data safety while in a trusted environment unencrypted data suffice.

- *Domain-specific interactions.* Aspect compositions differ across domains. Each domain imposes specific policies about compositions, for example the combination of two aspects may cause incorrect synchronisation in a real-time domain e.g. applying a caching aspect with a synchronisation aspect is likely to make the software system miss real-time deadlines. In a non-real-time domain the interaction of the two aspects does however no cause any interaction concern. Moreover, aspect weaved at the aspect-connector may also contain remote reference such that the order in which they are woven in one domain may differ to that woven in a different domain. Applying the same weaving order in a different domain may result in conflict semantics. In such cases, the local node policy needs to be checked to ensure the correctness of the (re)configuration.

### 3. SEMANTIC COMPOSITION MODEL

In this section we describe our semantic model for supporting the detection of interaction issues in the aspect-component model and resolution of emerging runtime semantics by supporting the following dimensions: (i) describing semantic aspect interactions; (ii) attaching metadata to entities in the aspect-component model; (iii) using a resolution engine and policies to resolve inconsistencies. Each of the dimensions is now examined in turn.

#### 3.1 Aspect Interaction Semantic Metadata

In order to detect possible semantic conflicts and dependencies each aspect-component is attached with metadata that describes and explains its functionality. This is used to inform the selection and deployment of the aspect—i.e. to help manage compositional and reconfiguration interaction between aspect-components in the aspect-component model as illustrated. These descriptions are written in the format as illustrated in the BNF form of Figure 2.

The *aspect-component interaction* model consists of the aspect scope, composition rules, and interaction policies. The *aspect scope* refers to the aspect-component instance of whether it is deployed on the local host, or is remote, or is replicated. More specifically, an aspect-component is assigned to a particular composition policy and has a specific aspect-scope in which it is defined. The *composition policies* deal with the types of policies options to constrain aspect instances on each particular (or multiple) address space. The *interaction policies* defines the aspects in which the underlying aspect-component is either dependent on or conflicts with, or the set of conditions that can lead to indirect interactions.

*Dependency-specific interactions* define the coordination-rules and enforcement rules that the aspect-components need. The coordination and enforcement-rules specify the aspect parties with which the composed aspect must coordinate with. For example for an encryption aspect, the underlying aspect must specify enforcement rules for a decryption aspect to be also added to the system when the encryption composition takes place and coordination rules specifying that both should be added to preserve the system consistency.

The *conflict-specific interactions* refer to the set of orders and mutual exclusive aspects than an aspect must be composed with respect to other aspects. Finally, the *indirect-interactions* define the set of conditions on how the aspect can be composed when

dependency-interactions and conflict-specific interaction occur in the system.

```

aspect-composition-interaction ::= < { aspect-scope, composition-specific-policies, interaction-policies}
aspect-scope ::= local | remote | replicated
composition-specific-policies ::= application-specific-policies | domain-specific-policies
interaction-policies ::= < { dependency-specific-interactions, conflict-specific-interactions, indirect-interactions} >
dependency-specific-interactions ::= < coordination-rules, enforcement-rules >
conflict-specific-interactions ::= < ordering-conflict, mutual-exclusion >
indirect-interactions ::= < dependency-specific-interactions, conflict-specific-interactions >
coordination-rules ::= refers to named aspect components that need to be coordinated
enforcement-rules ::= refers to named aspect components that need to be weaved/unweaved for aspect to operate correctly.
ordering-conflict ::= {expressions describing on how aspect order(s) conflict with other aspects}
mutual-exclusion ::= {expression describing conditions on how aspects may be affected if other aspects are composed}

```

Figure 2: BNF Semantic Composition Model for Aspect Interaction

#### 3.2 Attaching Metadata

Aspect-components are considered as black-boxes which provide advices in the form of operations within the provided interface (but hide their implementation). Three implications of this black-box property are:

1. Metadata can be attached to the interfaces and receptacles of components and aspects, as they are the only access points available to other aspect-components at be inspected and inform runtime decisions;
2. After aspect-components have been woven, they are *invoked* through their operations such that metadata is also required to be annotated at the aspect-component operations to detect runtime interactions; This is because when reconfiguration is performed at runtime, already weaved aspect metadata might be required to detect semantic interactions (as discussed section 3.3) with the reconfiguration aspect(s) and at the join point the aspect is accessed through its operations;
3. The tagged metadata needs to be kept separate from the main source functionality. This is because, an aspect represents crosscutting functionality such that adding descriptions by extending the implementation, e.g. through a new interface, will restrict its applicability to different applications and domains because it couples the consistency checking with the aspect-component functionality. Thus, keeping metadata separate allows both the core functionality and metadata to be reconfigured independently and transparently from each other.

#### 3.3 Semantic Resolution Engine & Policies

A Semantic Resolution Engine (SRE) provides the tool to query and reason about the annotated aspect-components; and resolve possible sources of inconsistency that may result from a dynamic reconfiguration. The latter retrieves the associated aspect-component metadata as illustrated in Figure 3, by getting the annotation file path from the aspect-component and parsing the *Aspect Metadata* file (retrieved from the *Aspect Metadata Repository*) to extract respective semantic composition tags for the aspect-component (structured as described by the BNF semantic composition model from Figure 2). Then, the SRE check the reconfiguration aspect against a set of composition policy on each reconfigured address space (referred as a *node*) to ensure reconfiguration follow the specified domain or running application policies. The composition policy uses a ‘condition-action’ approach to ensure the associated metadata and the join

point aspects metadata are valid by not causing any domain or application inconsistency in the node in which the reconfiguration take place. In case the validation is successful the (re)configuration is allowed to proceed. However, in case of any interaction issue is found, based on the policies specification the necessary remedy action is taken. Two alternatives of remedy actions can be taken by the SRE in terms of: either stopping reconfiguration from proceeding by calling the *rollback* operation to drive the system to the state prior to when the reconfiguration started; or if appropriate resolution policies are specified these can be deployed by the SRE and the reconfiguration can proceed (e.g. resolving the correct order of advices).

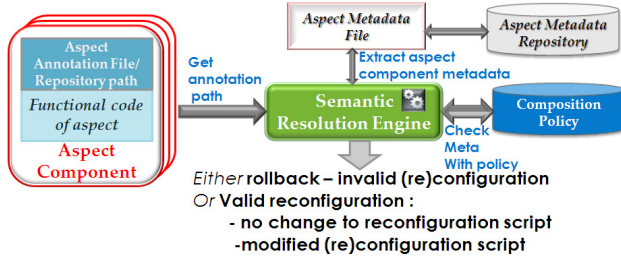


Figure 3: Semantic Resolution Framework

## 4. VALIDATION

In this section we validate our approach using AO-OpenCom [14]. We first provide some background on AO-OpenCom and then validate the extent to which our Semantic Composition Model (SCM) achieves the stated goals of semantic composition resolution, transparency and flexibility. Finally we measured the overhead of deploying the SCM.

### 4.1 AO-OpenCom

AO-OpenCom is an extension of the OpenCom [2] component model and provides a distributed AO composition service while allowing aspectual compositions to be dynamically reconfigured. The purpose of AO-OpenCom is to build on OpenCom and its associated reflective meta-models and component frameworks architectures [2], to provide a distributed AO composition service, and to allow aspectual compositions to be dynamically reconfigured. The programming model employs components to play the role of aspects—i.e. an aspect is simply an OpenCom component. The AO-OpenCom aspect framework comprises a set of components that are instantiated across each host.

The set of components is as follows (see Figure 4):

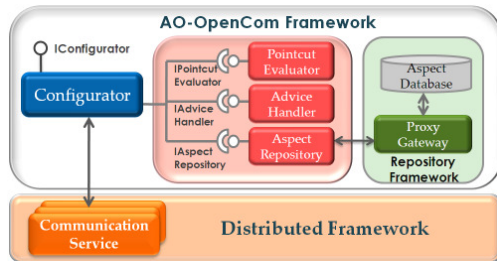


Figure 4: AO-OpenCom platform Architecture

The **Configurator** manages the other components in the framework as it is responsible for accepting and handling *(re)configuration requests* that will apply to a set of hosts. The Configurator also caches join point information it receives from Pointcut-Evaluators in case similar behaviour needs be applied in

the future. The **Aspect-Repository** holds a set of instantiable aspect-components e.g. the cache aspect, encryption aspect, etc. The **Pointcut-Evaluator** evaluates the pointcuts provided by the Configurator and returns a list of the matching join points found within the local address space. Finally, the **Aspect-Handler** acts on instructions from the Configurator to weave advices at join points as well as supporting the invocation of remote aspects.

The main API provided by an AO-OpenCom-enabled instance for AO (re)configuration is as follows:

```
Configurator.reconfigure(pc, command, aspect);
```

The *pc* argument specifies a pointcut that picks out the join points in the target nodes at which the desired reconfiguration should occur. The *command* argument offers options for the action to be taken at the identified join points: the ‘add’ action is used to weave the specified aspect at the join points; ‘remove’ is used to remove it, and ‘replace’ is used to add the specified aspect after removing an existing aspect of the same type that is assumed to be already there. The *aspect* argument can be a direct reference to a local aspect-component, or an indirect reference to an aspect stored in an Aspect-Repository, or a reference to an already-instantiated remotely-accessible singleton aspect. The *aspect weaving order* and the *type* of aspect in terms of (before, after, around) are also specified in the aspect argument.

### 4.2 Applying the SCM to AO-OpenCom

To ensure the semantic consistency, the SRE and the Composition-Policy aspect are both encapsulated as an aspect and weaved at the AO-connector component join point connecting the Configurator and the pointcut component as an ‘after’ advice in the AO-OpenCom platform. Moreover, the *Aspect Metadata* file of the SCM is implemented in an XML file with each aspect annotated with the path to the XML metadata file.<sup>1</sup>

### 4.3 Qualitative Validation

To illustrate the semantic reconfiguration consistency, we consider the following reconfiguration: the application programmer needs to reconfigure the online gaming system by adding an encoder to all mobile wireless node member; a fragmenter and a logger aspect have previously been woven at that join point. To perform this reconfiguration, the application programmer would provide a reconfiguration request by writing code as shown in Figure 5 (the code is simplified for presentational purposes). The *Configurator.reconfigure()* call takes the given pointcut and aspect specifications and also specifies that the specified aspect should be *added*.

```
Pointcut pc = new Pointcut( "mobile-wireless-nodes*", "communication*", "ICommunication", "communication*");
List aspectList = new LinkedList();
aspectList = new ArrayList();
Aspect aspectEncryption = new Aspect(encryption, before);
aspectList.add(aspectEncryption);
Configurator.reconfigure(mobile_app, pc, add, aspectList, perNode);
```

Figure 5: Aspect Reconfiguration specification example

This reconfiguration specification however fails to capture the semantics of the reconfiguration in terms of the: ordering between the three aspects with the fragmenter aspect needing to be woven *before* the encryption aspect; dependencies involved by weaving an encryption aspect requiring the corresponding decryption

<sup>1</sup> Since AO-OpenCom also supports remote aspects [14], the respective URL path to the XML file *Annotation Metadata Repository* is provided for remote aspects.

aspect to woven in a coordinated manner to ensure good running of the application; conflicts due to the weaving of the logging and encryption aspects.

### 4.3.1 Semantic Interaction Resolution

The encoder aspect in the AO-OpenCom *Application Repository* is tagged with appropriate metadata describing its semantic interactions, that is: the encoder aspect interface is tagged with the location path of the xml file containing the metadata having: the *type* of the aspect, *dependency-interaction tags* specifying a corresponding decoder aspect is dependent and must be woven when the encoder is applied; *conflict-interaction tags* specifying two constraints are specified in terms of: encoder aspect conflicts with a logging aspect in an untrusted domain if woven *after* the logging aspect; the encryption aspect can be allowed in an untrusted domain and can be allowed based on the underlying domain policies.

The Composition-Policy aspect, as illustrated in Figure 6, then contains the ‘condition-action’ rules in terms: (i) a fragmenter must be woven *before* data is encrypted and similarly containing another policy describing that the decryption aspect must be woven *before* the reassembler aspect; (ii) the weaving of fragmenter and reassembler aspects must be coordinated across nodes to ensure both needs to be woven as they are dependent; and (iii) the weaving of encryption and decryption aspects must be coordinated across nodes to ensure that encrypted messages can be decrypted.

```

<policy name = "encryption" reconfiguration-actions = "add">
  <condition>
    <joinpoint type = "fragmenter" condition-logic = "and" type = "logger"></joinpoint>
    <indirect-interaction application-type = "none" domain-specific = "none">
      </indirect-interaction>
    </condition>
  <action>
    <dependency-action>
      <aspect-dependency type = "decryption" ></aspect-dependency>
      <coordination-aspect type = "coordination-protocol" ></coordination-aspect>
    </dependency-action>
    <ordering-conflict-action>
      <aspect-order type = "fragmenter" order = "1"></aspect-order>
      <aspect-order type = "encryption" order = "2"></aspect-order>
      <aspect-order type = "logger" order = "3"></aspect-order>
    </ordering-conflict>
    <indirect-interaction-action>
      .
      .
    </indirect-interaction-action>
  </action>
</policy>

```

Figure 6: Composition Policy Example

When *Configurator.reconfigure()* is called on the Configurator of one of the nodes (referred as the ‘initiator’), the latter calls the Pointcut-Evaluator to locate all the target join points. On returning the located join points, the SRE aspect gets *invoked*. Using the target join points, metadata, that is from the fragmenter and logging aspects metadata, together with the encryption metadata, the SRE first parses their respective metadata to detect any semantic interactions. With the encryption aspect containing metadata with an order conflict for weaving with aspect *type* fragmenter, the SRE checks the Composition-Policy Aspect to determine if the constraint is *valid* for the reconfigured node and checks for any other application-specific or domain-specific restrictions. With the fragmenter metadata constraint *matching* the Composition-Policy Aspect metadata, and no application-specific or domain-specific constraints for this case, the SRE aspect instructs the AdviceHandler to weave a corresponding *reassembler-aspect* with the *encryption-aspect* woven in the *second order* to ensure the reconfiguration can be successfully done and thus semantically consistent. Moreover, since the original pointcut specification has been updated, the Configurator

caches an updated version of the pointcut specification. In case remedy policies were not specified, the reconfiguration would be aborted with the rollback operation deployed in case any changes made.

### 4.3.2 Transparency

The approach naturally supports a *selectively transparent* approach as the SRE aspect and the Composition-Policy aspect can be pre-configured at application start-up time so that the application programmer who wishes to initiate a run-time reconfiguration needs only to make the appropriate call to *Configurator.reconfigure()*. This achieves complete transparency of consistency-related mechanisms from the code to invoke a reconfiguration. At the other extreme, the programmer can be explicit specifying the SRE and Composition-Policy aspects should be put in place for each reconfiguration. In this case, both aspects are woven on-the-fly (if they are not already present) before proceeding to perform the requested reconfiguration. Note that this extreme is still *partially* transparent as the programmer is protected from the low level details of actually weaving the SCM aspects.

### 4.3.3 Flexibility

The use of a separate *Aspect Metadata* file (e.g. using XML when applied to AO-OpenCom) to attach semantics of the aspect-components allows new metadata updates to be applied without having to recompile existing source-code. Moreover, our approach adds the SCM as an independently-deployable service which can be used for both local and distributed (re)configuration. This means that the SCM imposes no overhead when it not used, and can be dynamically woven/unwoven where and when required. We also believe that the approach, being based upon applying metadata and behaviour at common architectural elements (i.e. interfaces), can be applied generally to other AO middleware not just AO-OpenCom; indeed we see important future work in the deployment of our model in a wider range of systems.

## 4.4 Quantitative Evaluation

We next evaluate the overheads incurred by the SCM to perform dynamic reconfiguration. The baseline for our experiments is as follows; we reconfigure aspects at one join point using AO-OpenCom without SCM (in this case there are no conflicts to detect). This was performed locally on one node and then repeated with the aspects to be reconfigured spread across 4 client nodes with another node acting as the initiator of reconfiguration. Each node ran on a separate Core Duo 2 processor 1.8 GHz PC with 2GB RAM, using the Java-based version of the AO-OpenCom platform. Each measurement was repeated ten times and the mean value was calculated to discount anomalous results.

We then performed four separate reconfiguration cases and measured the performance overhead compared to the base-line measure above. In each of these cases we increased the scale of the experiment by increasing the number of aspects woven at the join point. Figure 7 shows the results of these four cases: (i) SCM manages reconfiguration where there is no conflict on a local node; (ii) SCM manages reconfiguration where there is no conflict across 5 nodes; (iii) we introduce an aspect conflict and SCM manages reconfiguration on a local node; iv) SCM manages a conflicted reconfiguration across 5 nodes. We identify the following results:

- The overhead of using SCM increases as the number of nodes the reconfiguration executes across increases to 5.

- As the number of aspects to be weaved at a join point increases the percentage overhead of SCM decreases.
- The overhead to weave 'one' aspect is the worst case. This is mainly attributed to the retrieval of the Composition-Policy aspect metadata and once obtained the metadata can be parsed for subsequent checks, causing significant decrease of overhead.
- Furthermore, it can be observed that on a single node the use of SCM added an average overhead of 15% when no conflicts were managed; there was an extra 1.84% when aspects with a semantic concern were woven on the node.
- The overhead of the SCM is mainly attributed to the use of XML and the parsing of the file structure before the semantics metadata are retrieved.

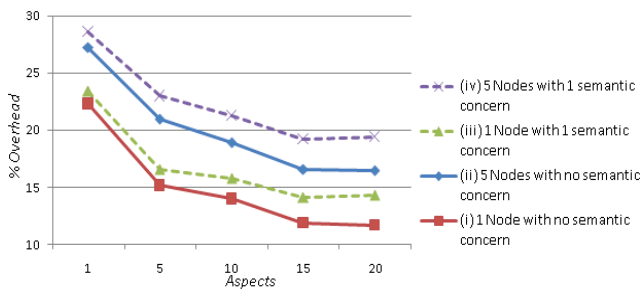


Figure 7: Overhead of SCM to perform reconfiguration per join point in the AO-OpenCom platform

## 5. RELATED WORK

Few AO middleware platforms have addressed the challenges of performing consistent semantic (re)configuration. CAM/DAOP [12], Spring AOP [13], JBoss AOP [1] and DyReS [15] are prominent examples of AOP middleware platforms but do not consider validation for aspect composition.

Other prominent platforms such as JAC [9] and Prose [8] and DyMac [7] are limited to solving aspect semantic by checking aspect ordering only. custAOMware [6] is a runtime AO component middleware allowing aspect interactions to be evaluated. The interaction model allows the detection of conflicts, dependencies and resolution of aspects by storing and accessing aspects metadata in the runtime kernel repository. However, compared to our approach, the platform offers only aspect configuration; the semantic validation of distributed dynamic reconfigurations is not supported. Furthermore, the approach does not easily support dynamic aspect metadata updates, requiring the programmer to work with low-level kernel APIs. Spoon [10] is a Java program-transformation framework that uses AOP and compile-time (CT) reflection to ensure semantic consistency of the middleware components. However, this approach can only provide compile-time validation and is therefore unsuitable for adaptive software.

Outside the domain of AO middleware there a number of language-based AO approaches. Composing Around advice (CompAr) [11] is a language-based AO approach to detect and solve aspect-composition issues. However, the approach only detects aspect-ordering interactions issues. Douence et al., [3] offers a similar approach for the automatic detection and explicit resolution of aspect interactions. However, the approach only allows for static analysis of aspect interactions.

SEmantiC Reasoning Tool (SECRET) [4] uses a similar reasoning mechanism as our semantic composition model by analysing all advices at a join point before composing them.

Similarly, SECRET annotates advices with semantics which are reasoned at the join point using conflict patterns to detect and resolve semantic interactions. However, the approach is language dependent requiring advices to be written in Aspect-J language for the conflicting patterns to be detected. Our approach differs from these related approaches, in that we introduce a general semantic composition model within the field of AO middleware which can be applied to each of the aforementioned platforms.

## 6. CONCLUSION AND FUTURE WORK

In this paper we have demonstrated the need to consider aspect semantics to better support and ensure consistent composition and reconfiguration in dynamic AO middleware. We have illustrated our general approach to semantic composition model for validating distributed dynamic reconfiguration, catering for semantic differences in terms of application-specific and domain-specific conditions. Moreover, our solution also prevents the combinatorial explosion that may result as a consequence of coupling metadata with the core aspect functionality allowing aspects semantics to be dynamically evolving without changing the source-code of running aspects.

Turning to future work, we first plan to investigate using our approach in a self-managing autonomic environment in which reconfiguration requests are initiated by the platform itself as opposed to the user. Then, we also plan to extend our semantic philosophy to use an ontology model such that the concepts between applications and domain can better be understood when building large-scale distributed middleware applications.

## 7. REFERENCES

- [1] Burke, B., "JBoss AOP Tutorial", 3rd International Conference on AOSD, Lancaster UK, 2004.
- [2] Coulson, G., Blair, G, Grace et al., "A Component Model for Building Systems Software". In *Proc. SEA, USA*, 2004.
- [3] Douence, R., et al., "Composition, reuse and interaction analysis of stateful aspects". 3<sup>rd</sup> Conference AOSD, 2004.
- [4] Durr, P., "Reasoning About Semantic Conflicts Between Aspects". 2nd EIWAS Workshop in Software, 2005.
- [5] Grace, P., et al., "A Reflective Framework for Fine-Grained Adaptation of Aspect-Oriented Compositions". In *Proc 7th Software Composition*, 2008.
- [6] Greenwood, P., et al., "Interactions in AO Middleware". In *Proc. Workshop on ADI, ECOOP* 2007.
- [7] Lagaisse, B., et al., True and Transparent Distributed Composition of Aspect-Component. In *Proc. Middleware, 06*.
- [8] Nicoara, A., et al., "Dynamic AOP with PROSE". In *Proc. 17<sup>th</sup> International Conference ASMEA*, 2005.
- [9] Pawlak, R., et al., "JAC: an aspect-based distributed dynamic framework." In *Journal Software Practice & Experience*, '04.
- [10] Pawlak, R., "Spoon: annotation-driven program transformation - the AOP case". In *Proc AOM*, 2005.
- [11] Pawlak, R., et al., "CompAr: Ensuring Safe Around Advice Composition". In *Proc. FMOODS, LNCS Springer* 2005.
- [12] Pinto, M., "Supporting the Development of CAM/DAOP Applications". *Software Practice & Experience*, Vol 37, '07.
- [13] Spring website, 2009. <http://www.springsource.org/>
- [14] Surajbali, B. et al., "Augmenting reflective middleware with an aspect orientation support layer. In *Proc. ARM*, 2007.
- [15] Truyen, E., et al., "Support for distributed adaptations in aspect-oriented middleware". In *Proc. 7th International Conference on Aspect-oriented software development*, 2008.