ReCycle: Resolving Cyclic Dependencies in Dynamically Reconfigurable 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
In aspect-oriented middleware systems, the aspect modules are typically composed as chains of aspects within the connectors (or bindings) that join the base software components. However, this approach can lose or hide information about the dependencies between multiple aspects in the chain; this is particularly important when dynamically reconfiguring such a system at run-time. Without knowledge of these dependencies the system could reconfigure a new aspect with a dependency to a prior aspect in the chain resulting in a cyclic dependency and subsequent deadlock. Furthermore, the problem is harder to detect with the presence of remote aspects within the connectors as their dependencies are hidden across address spaces. To resolve cyclic dependencies that may occur when reconfiguring both local and remote aspects we propose the use of a *re*configuration *cycl*ic dependency r*e*solution (ReCycle) model. This approach can be employed generally in dynamic AOP middleware platforms, and in this paper we evaluate it within the AO-OpenCom middleware.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: D.2.11 Software Architectures – Languages (interconnection), Patterns.

General Terms
Design, Management

Keywords
Middleware, dependency, aspect, dynamic reconfiguration.

1. INTRODUCTION
Aspect-oriented middleware platforms provide solutions to create distributed component-based systems into which aspect modules representing cross-cutting concerns can be woven. Aspects are made up of individual code elements that implement the concern (*advices*), which are deployed at multiple positions in a distributed system (*join points*) that are expressed by *pointcuts*—a particular form of composition language. AO-OpenCom[11], AspectOpenCom[4] ,CAM/DAOP [3], FAC [9], FuseJ [13], DyMAC [5], and DyReS [14] are examples of aspect-oriented middleware which allow aspects to be composed and adapted at runtime. The aspect runtime composition of aspects in such AO middleware platforms differs from the standard component to component binding (where there is a direct reference from the provided interface to the required interface). In these AO middleware aspects are advice components which are woven non-invasively at their connector (between the required and provided interfaces of the base software components) in advice chains with the aspect reference stored in the advice chain. Then, the aspects are *invoked* from the connector chain when a call or execution occurs from the call or execution of the provided or required interface.

Unlike components, the dependency of an aspect to another aspect is not explicitly defined, such that an aspect within a chain may have a dependency with another aspect located earlier in the chain, and cause a *cyclic dependency* while performing reconfiguration. The potential problem of cyclic dependency is that it may cause the running system to enter into a *deadlock* after performing reconfiguration, when an invocation occurs at the join point. The cyclic dependency problem is hard to detect since an AO-Connector, maintains both local and remote advices. For an AO-Connector containing solely local advices, inspection of the AO-Connector can reveal the possibility of cyclic dependencies. However, this is non-trivial when the AO-Connector contains both local and remote advices, since for remote advices the visibility of the methods invoked by remote advices is located in the remote address space from where the AO-Connector is.

In this paper, we present a *re*configuration *cycl*ic dependency r*e*solution (ReCycle) model for dynamic aspect-oriented, component-based middleware; this provides the capability to describe the various kinds of built-in dependency inconsistencies that affect aspect configuration and reconfiguration at runtime. This is coupled with a graph-based tool which detects and resolves cyclic dependency inconsistencies at run-time while reconfiguration is performed.

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

- *Resolution of reconfiguration cyclic dependency.* We show that cyclic dependency inconsistencies can be resolved for one case-study with minimal performance overhead.
- *Transparency.* We apply consistent reconfiguration with minimal developer effort or change to the underlying component model.
- *Flexibility.* New dependency consistency can be described dynamically to evolve with the running application or domain context without breaking the implementation details of the instantiated aspect. Moreover, the approach can be applied in different compositions approaches and tools; for example we show how both node-local and distributed reconfiguration cyclic dependency consistency can be avoided in this paper.

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

2. ASPECT RECONFIGURATION

In aspect-component middleware, aspects (which are themselves implemented as component modules) are composed with the base components (hereafter termed components) using AO-Connectors [4, 8, 11, 12, 14]. AO-Connectors are the architectural element offering aspectual composition (weaving) of aspects between a receptacle and a provided interface of components. AO-Connectors maintain the meta-data containing references to aspects instances in an advice chain. For example, it maintains details of all advised aspects and their types and allows these to be queried to determine the operations they support and the aspects currently advising them. It also supports the runtime manipulation of the chain to add a new advice, or remove or reorder aspects in the chain of advices.

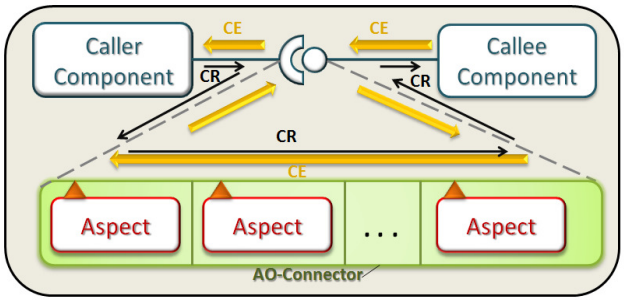

Figure 1: Aspect-Component Model

A list of advices is attached to the connector between the required and the provided interface. This capability is illustrated in Figure 1, which shows *a caller component* connected to a *callee component*, and an AO-Connector containing a list of aspects that get called. Where a call comes from the caller component (arrows marked CR) then the aspects in the chain are executed first or otherwise in case an execution is triggered from the Callee component, the aspect chain is executed in the reverse order, as highlighted with arrows marked CE) in Figure 1.

We now identify and classify the types of dependency inconsistencies that can occur in the aspect-component model.

2.1 Use case scenario

To motivate the requirement to resolve cyclic dependency for AO reconfiguration we present its occurrence within the distribution framework stack. The AO composition is as follows (see Figure 2): when the message handler is called on the communication module, the following aspects are enforced, before the execution of the communication module operation:

i.) *Selecting the format of transportation*. Format selection handles the formatting of the message such that it can be serialised and deserialised for remote invocations and replies.

ii.) *Selection of the transportation*. Transport selection creates a transport listener and transport request and binds them to a socket.

iii.) *Deployment handler for the message transfer*. The deployment handler creates the skeleton and binding for the message transfer as well extracting the object name in the URI to lookup the correct instance in case of a normal method call.

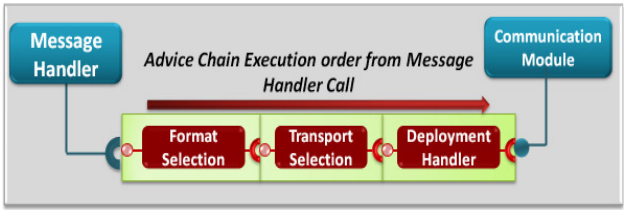

Figure 2: Distribution Stack AO Composition scenario

Whenever, the Message Handler component calls the Communication Module, the list of advices within AO-Connector chain gets invoked and executed in the following order:

Format Selection Aspect → Transport Selection Aspect → Deployment Handler Aspect.

2.2 Cyclic Dependency Occurrence

To cope with the application and environmental demand the following two dynamic (re)configurations may be required: (i) new users with limited bandwidth may join, requiring a *Compression aspect* to be configured to split data before being sent; (ii) data may be required to be encrypted using a *Security aspect* to protect the users' privacy.

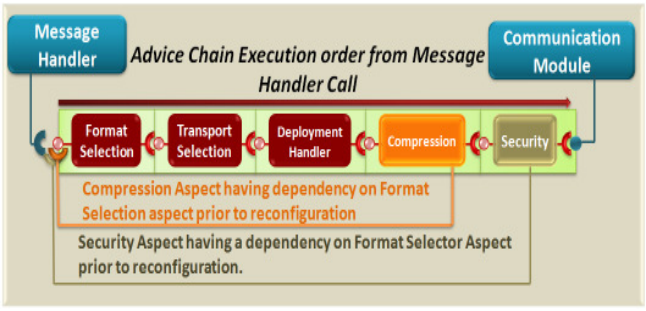

Figure 3: Reconfiguration with Cyclic Dependency Occurrence

The reconfiguration proceeds by weaving the Compression Aspect after the Deployment Handler Aspect in the AO Connector chain and the Security Aspect woven after the compression aspect in the chain, as illustrated in Figure 3. However, both the Compression aspect and the Security Aspect may have a dependency on the Format Selection aspect prior to the reconfiguration, causing cyclic dependencies to occur at the AO Connector such that calls may not return back to the Security Aspect, causing a deadlock to occur if the reconfiguration is allowed to proceed. The cyclic call dependency for Figure 3 when called proceeds as follows:

Format Selection Aspect → Transport Selection Aspect → Deployment Handler Aspect → Compression Aspect → Format Selection Aspect.

A more complicated cyclic dependency occurrence is when remote aspects are attached to the AO-Connector. In the case of remote aspects, they may have dependencies with other aspects located on different address spaces, causing the dependency to be unnoticed while performing reconfiguration.

2.3 Analysis

An aspect represents a crosscutting functionality that may be referenced and shared by other software modules in a running system. That is AO middleware typically just add/reconfigure at runtime without knowledge of the chain or taking into consideration the existing aspects dependencies that may already be present. So doing, as described above, can potentially lead to cyclic dependencies. Furthermore, creating two versions of the aspect by replicating the aspect functionality is not a feasible solution either and can potentially result in an exponential growth in versions of the same aspect. To solve the above problems, we propose the ReCycle model.

3. ReCycle: RECONFIGURAION CYCLIC DEPENDENCY RESOLUTION MODEL

In this section we describe our ReCycle model to support the detection of cyclic dependencies that may result in the configuration and reconfiguration of aspects as well as its resolution by supporting the following dimensions: (i) describing aspect dependency; (ii) attaching metadata to entities in the aspect-component model; (iii) using graph based detection with a resolution engine capable of parsing the AO-Connector to detect the occurrence of any cyclic dependency inconsistencies. Each of the dimensions is now examined in turn.

3.1 Aspect Dependency Metadata

In order to detect cyclic dependencies each aspect-component is attached with metadata that describes and explains its functionality as well as the dependency they may have on other aspect-components. This is used to inform the deployment of the aspect—i.e. to help manage compositional and reconfiguration cyclic detection between aspect-components in the aspect-component model as illustrated. These descriptions are written by the AO middleware developer in the format as illustrated in the BNF form of Figure 4.

```
aspect-instance :== <{ aspect-scope, aspect-required-interfaces, aspect-provided-interfaces }>
aspect-dependency : == <{ (dependency-aspect-instance | dependency-aspect-type),
                          dependency-aspect_scope, <AO-Connectors> }
aspect-scope  :== local | remote
AO-Connectors  :== { list of ao-connectors dependent aspects are connected}
aspect-required-interfaces :== {list of required interfaces attached with aspect-component}
aspect-provided-interfaces :== {list of provided interfaces attached with aspect-component}
dependency-aspect-instance :== { expressions describing dependency aspect instance}
dependency-aspect-type :== { expressions describing dependency aspect type }
dependency-aspect_scope :== { expressions describing dependency aspect scope }
```

Figure 4: ReCycle Model BNF Metadata representation

The *aspect-instance* defines the aspect-component instance aspect scope, list of aspect required interfaces of the aspect-component instance and list of provided interfaces for the aspect-component.

Aspect-dependency defines the list of aspect-instance aspect-type to which the aspect is dependent on as well as the AO-connector to which it is currently bound with.

The *aspect scope* refers to the aspect-component instance of whether it is deployed on the local host, or is remote.

The *AO-Connectors tag* refers to the list of connectors to which the aspect-component instance or type is bound with. This can be zero in case there is no connection dependency for the aspect-component.

3.2 Attaching Metadata

As described in our previous work in [12] tagged metadata needs to be kept separate from the main source functionality. This is because:

1. aspect-components are considered as black-boxes which provide advices in the form of operations within the provided interface (but hide their implementation);
2. aspects represent 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.

Keeping metadata separate allows both the core functionality and metadata to be reconfigured independently and transparently from each other.

Metadata is attached to the aspect-component interfaces and receptacles at load-time, as they are the only access points available to other aspect-components to be inspected and inform runtime decisions. Then to provide for runtime reconfiguration, since aspect-components are *invoked* through their operations, aspect-component operations also need to be annotated. This is because when reconfiguration is performed at runtime, already woven aspect metadata might be required to detect cyclic dependencies at the join point the aspect is accessed via its operations.

3.3 ReCycle Model

A ReCycle model provides the tool to query and reason about the annotated aspect-components; and resolve possible sources of cyclic inconsistency that may result from a dynamic reconfiguration. The latter retrieves the associated aspect-component metadata as illustrated in Figure 5, 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 dependencies tags for the aspect-component (structured as described by the BNF Cycle Metadata representation from Figure 4). Then, the ReCycle model builds a graph using the aspect-component instance and its dependencies if they are connected for the corresponding AO-Connector involved with the reconfiguration. After the graph is built, the graph is traversed from the root, the aspect-component contained in the first-order to the end of the graph.

In case the validation is successful the reconfigured AO-Connector, chain list is first stored in a reconfiguration repository having transactional capabilities and the reconfiguration is then allowed to proceed. However, in case of any cyclic dependency issue found, based on the composition policy, two alternative remedy actions can be taken by the ReCycle model in terms of:

either the ReCycle Configurator stopping reconfiguration from proceeding by calling the *rollback* operation to drive the system to the state prior to when the reconfiguration started by restoring the AO-Connector chain from the reconfiguration repository; or if appropriate resolution policies are specified these can be deployed by the ReCycle model and the reconfiguration can proceed (e.g. removing the cyclic connector or adding a null binder to return the call and exiting the cyclic loop). If a connector is removed or updated, the associated AO-Connector meta data is updated for the respective aspect-component (by updating the aspect component associated AO-Connector tag meta data).

Moreover, to avoid the potential occurrence of semantic interactions, the Semantic Resolution model from [12] may be called by the ReCycle model to reason about the resolved reconfiguration interaction. In case a semantic conflict is detected and no policies are defined, the reconfiguration gets aborted by calling the rollback operation. Otherwise if appropriate resolution is defined, the semantic valid reconfiguration is allowed to proceed while ensuring with the ReCycle model it does not result in any cyclic dependencies.

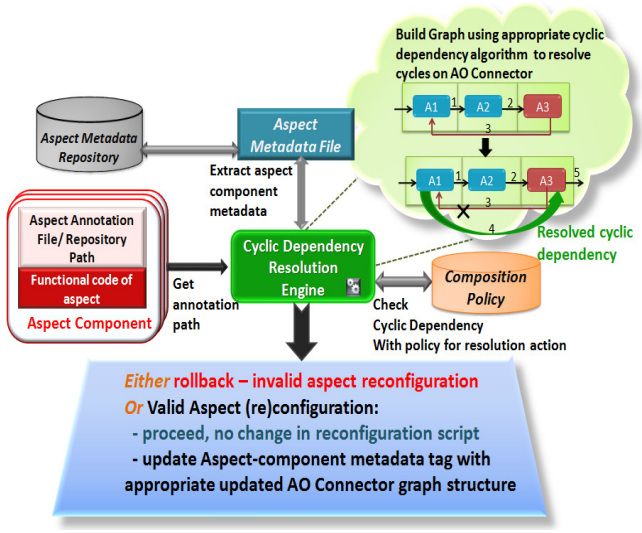

Figure 5: ReCycle model to resolve Cyclic Dependency

4. VALIDATION

In this section we validate our approach using AO-OpenCom [11]. We first provide some background on AO-OpenCom and then validate the extent to which our ReCycle model achieves the stated goals of cyclic dependency resolution, transparency and flexibility. Finally we measured the performance and resource overhead of deploying the ReCycle model.

4.1 AO-OpenCom

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 6):

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 indentified 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.

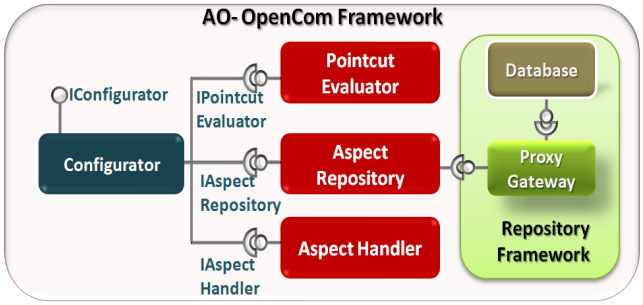

Figure 6: AO-OpenCom platform Architecture

4.2 Applying the ReCycle Model to AO-OpenCom

To ensure semantic consistency, the ReCycle model and the Composition-Policy modules are both encapsulated as aspects and woven at the AO-connector component join point connecting the Configurator and the pointcut evaluator component as an 'after' advice in the AO-OpenCom platform. Moreover, the *Aspect Metadata* file of the ReCycle model is implemented in an XML file with each aspect annotated with the path to the XML metadata file.[1]

4.3 Qualitative Validation

To illustrate the ReCycle model preserving reconfiguration consistency, we consider the use case scenario reconfiguration. To

[1] Since AO-OpenCom also supports remote aspects [11], the respective URL path to the XML file *Annotation Metadata Repository* is provided for remote aspects.

perform the reconfiguration outlined in Section 2.1, the application developer would provide a reconfiguration request by writing code as shown in Figure 7 (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.* This reconfiguration specification however fails to capture the cyclic dependency by adding the two aspects at the AO Connector as shown in Figure 3.

```
Pointcut pc = new Pointcut ("*", "Communication-Module", "ICommunication", "communication*");
List aspectList = new LinkedList();
aspectList = new ArrayList();
Aspect aspectCompression = new Aspect(Compression, after);
Aspect aspectSecurity = new Aspect(Security, after);
aspectList.add(aspectCompression);
aspectList.add(aspectSecurity);
Configurator.reconfigure(pc, add, aspectList);
```

Figure 7: Aspect Reconfiguration specification example

4.3.1 Resolution

The Security and Compression aspects in the AO-OpenCom *Application Repository* is tagged with appropriate metadata describing its dependencies on other aspect-components, that is: the Security and Compression aspects interface is tagged with the location path of the xml file containing the metadata having the *aspect-dependency tags* specifying a corresponding Compression and Security aspects each have a dependency with the Format Selection aspect and with an active AO-connector.

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 ReCycle aspect gets *invoked.* The latter evaluates the AO-Connector to build a *aspect dependency graph* and using the annotation metadata from the Format Selection aspect, the graph is updated to detect any cyclic dependencies that may occur.

In this case, a cyclic dependency is detected such that the Cyclic Resolution Dependency Engine checks with the Composition Policy or any 'condition-action' policies to resolve such a cyclic dependency.

The Composition-Policy aspect, as illustrated in Figure 8 specifies the 'condition-action' rules in terms if a cyclic dependency is located and aspect-instance is Security aspect, and the latter aspect has a dependency connection with a Format Selection aspect, then the connection needs to be removed, as the messages format are already set. (Otherwise if the connector cannot be removed based on the Composition-Policy specification then the reconfiguration is aborted to avoid the occurrence of cyclic dependency.)

The Cyclic Resolution Dependency Engine aspect then instructs the AdviceHandler to remove the AO-Connector connecting the Security with the Format Selection aspect, thus resolving any potential cyclic dependencies issue for this reconfiguration scenario. In case remedy policies were not specified, the reconfiguration would be aborted with the rollback operation deployed for any changes.

```
<policy name="security-formatSelection" reconfiguration-action="add">
    <condition>
        <cyclic-dependency aspect-instance="security" condition-logic="and"
                connector-instance="bound" aspect-dependency="format-selection" />
    </condition>
    <action>
        <cyclic-dependency-action connector-instance="unbound"
                aspect-dependency="formatSelection" />
    </action>
</policy>
<policy name="compression-formatSelection" reconfiguration-action="add">
    <condition>
        <cyclic-dependency aspect-instance="compression" condition-logic="and"
                connector-instance="bound" aspect-dependency="format-selection" />
    </condition>
    <action>
        <cyclic-dependency-action connector-instance="unbound"
                aspect-dependency="formatSelection" />
    </action>
</policy>
```

Figure 8: Composition Policy Example

4.3.2 Transparency

The approach naturally supports a *selectively transparent* approach as the ReCycle aspect and the Composition-Policy aspect can be pre-configured at application start-up time so that the application developer 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 developer can be explicit specifying the ReCycle 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 developer is protected from the low level details of actually weaving ReCycle.

4.3.3 Flexibility

The use of a separate *Aspect Metadata* file to attach dependencies of the aspect-components allows new metadata updates to be applied without having to recompile existing source-code. Moreover, our approach adds the ReCycle as an independently-deployable service which can be used for both local and distributed reconfiguration. This means that ReCycle 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 AOM not just AO-OpenCom; indeed we see important future work in the deployment of our model in a wider range of systems.

4.4 Overhead of ReCycle

We next evaluate the overheads incurred by ReCycle to perform dynamic reconfiguration. The baseline for our experiments is as follows; we reconfigure aspects at one join point using AO-OpenCom without ReCycle (in this case there are no cyclic dependencies to detect). This was performed as follows:

- the compression aspect and security aspect both instantiated on a local aspect repository;
- the compression aspect instantiated on the same local node as the join point (AO-Connector) and the security aspect instantiated on a remote node;

- both the compression aspect and security aspect instantiated on separate remote nodes from the join point.

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. The cyclic dependency algorithm used is the *single-source negative-weighted acyclic-graph shortest-path algorithm* [6] and the results of the experiment are shown in Table 1.

It can be observed that on a single node the use of ReCycle added an average overhead of 5.6% when no conflicts where managed; there was an extra 8 % when aspects with a cyclic were woven on the node. The overhead of the ReCycle is mainly attributed to the use of XML and the parsing of the file structure before the proper metadata are retrieved, which accounts for the extra overhead of using ReCycle when detecting cyclic dependency.

Table 1. Overhead of using ReCycle in AO-OpenCom

Reconfiguration:	Reconfiguration Time in (ms)		
	Setup A	**Setup B**	**Setup C**
Without ReCycle	390	1356	2651
With ReCycle with no cyclic dependency	411	1432	2810
With ReCycle with cyclic dependency	*442*	*1541*	*3024*

Setup A – Security and Compression Aspect woven locally.
Setup B – Security as remote aspect and compression as local aspect.
Setup C – Both Security and Compression woven as remote aspects.

A final point to note is that overhead of the ReCycle is determined by the cyclic graph detection algorithm. An optimised algorithm detection could be used to reduce the induced overhead of ReCycle in detecting cyclic dependency.

5. RELATED WORK

There are several cyclic dependency algorithms developed to detect cyclic cycles among software modules at runtime. JooJ [7] checks source code of java classes to detect for cyclic dependencies among java classes. However, JooJ requires the developer intervention to resolve the occurrence of any detected cyclic dependencies. Our approach differs from Jool in that the reconfiguration is entirely managed by the ReCycle Configurator and in case of cyclic dependencies based on the attached metadata the Configurator can apply appropriate resolution or rollback from invalid reconfiguration without the developer assistance.

ByeCycle [15] is a tool that is very similar to JooJ in that it checks for cycles among java packages. As a result only packages on which classes depend on are analysed to detect cyclic dependencies, such that internal invocations occurring within classes are not detected. AOR [1] tackles cyclic referential dependencies by reverting the dependency between modules such that the references points in one direction only. However, this can potentially lead to semantic interactions concerns, whereby one aspect could be in mutually exclusive of another. ReDac [10] uses a configuration framework to detect cyclic dependencies while composing components. The configuration framework works for multi-threaded component. However, the approach does not detect cyclic dependencies in the connector component.

With respect to AOM platforms: CAM/DAOP [3], FAC [9], FuseJ [13], DyMAC [5], and DyReS [14] none of the existing platforms provide mechanisms to detect the occurrence of cyclic dependency while composing and reconfiguring the platforms.

6. CONCLUSION AND FUTURE WORK

In this paper we have demonstrated the need to consider the occurrence of cyclic dependencies in aspect chains to better support and ensure consistent reconfiguration in dynamic AO middleware. We have illustrated the ReCycle model, a general approach for validating distributed dynamic reconfiguration, catering for potential cyclic dependencies following a dynamic distributed reconfiguration. Moreover, our solution does not change the implementation of the aspect-component, which would result in breaking the encapsulation of its functionality; this allows aspects dependencies to be dynamically evolving without changing the source-code of running aspects. The essence of our approach is that ReCycle can be encapsulated as an aspect to resolve any occurrence of cyclic dependency at configuration and reconfiguration. This means that ReCycle model can be independently woven and unwoven as required. We believe this gives the approach strong flexibility and generality that will allow it to be deployed in a number of AO-Middleware platforms not just AO-OpenCom.

Turning to future work, we first plan to investigate extending our approach to cover cyclic dependency in multi-threaded aspects environments. Then, we also plan to integrate our semantic resolution model [12] and the ReCycle model to ensure consistent aspect reconfiguration when building large-scale distributed middleware applications.

7. REFERENCES

[1] Apel, S., Kastner, C., Batory, D. 2008. Program refactoring using functional aspects. In Proc. 7th Conference. on Generative programming and component engineering. ACM Press, New York, 161-170.

[2] Coulson, G., Blair, G., Grace, P., et al. 2008. A Generic Component Model for Building Systems Software. In ACM Transactions on Computer Systems, Volume 26, Issue 1, February 2008. ACM Press, New York, Article 1.

[3] Fuentes, L., Pinto, M., Troya. J.M.. 2007. Supporting the Development of CAM/DAOP Applications. In Journal Software Practice & Experience John Wiley, Vol. 37. 21-64.

[4] Grace, P., Lagaisse, B., et al. "A Reflective Framework for Fine-Grained Adaptation of Aspect-Oriented Compositions". In Proceedings of 7th Software Composition, 2008.

[5] Lagaisse, B. and Joosen, W. 2006. True and Transparent Distributed Composition of Aspect-Component. In Proceeding Middleware Conference. ACM Press, New York, NY, 42-61.

[6] Lingas, A., Lundel, E., Efficient approximation algorithms for shortest cycles in undirected graphs. In Elsevier Publications Volume 109, Issue 10, 30 April 2009, 493-498.

[7] Melton, H., Tempero, E., 2007. JooJ: real-time support for avoiding cyclic dependencies. In Proc. conference on Computer science. Vol. 62. ACM Press, New York, 87-95.

[8] Pawlak, R., Duchien, G., et al. 2004. JAC: an aspect-based distributed dynamic framework. In Journal Software Practice, Volume 34, Issue 12, 1119 - 1148.

[9] Pessemier, N., et al., "Component-based and Aspect-oriented Systems". In Proc. Software Composition, 2006.

[10] Rasche, A., Polze, A. ReDAC Dynamic Reconfiguration of Distributed Component-Based Applications with Cyclic Dependencies. In Proc. IEEE on OO Real-Time Distributed Computing, 2008.

[11] Surajbali, B., Coulson, G., Greenwood, P., and Grace, P. 2007. Augmenting reflective middleware with an aspect orientation support layer. In Proc. 6th Int. workshop Adaptive and Reflective Middleware, ACM Press, Article 1.

[12] Surajbali, B., Grace, P and Coulson. G. 2009. Surajbali, B., Grace, P and Coulson. G. 2009. A Semantic Composition Model to Preserve (Re) configuration Consistency in Aspect Oriented Middleware. In Proc.8th International workshop Adaptive and Reflective Middleware, ACM Press, Article 6.

[13] Suvee, D., et al., "A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development". In Proc. 9th International SIGSOFT Symposium on CBSE, 2006.

[14] Truyen, E., Janssens, N, Sanen, F., et al., 2008. Support for distributed adaptations in AOM. In Proc. of the 7th Conference AOSD. ACM Press, New York, 120-131.

[15] Wuestefeld K., Rodrigo Oliveira, B., Beck, K., "ByeCycle", http://byecycle.sourceforge.net/.