

Comparing Dynamic AO Systems

Ruzanna Chitchyan
Ian Sommerville

Computing Department, Lancaster University, Lancaster, UK
{rouza,is}@comp.lancs.ac.uk

Abstract

In this paper we present a comparative analysis of several currently available Java based dynamic AO systems. The comparison is built on how these systems deal with general dynamic reconfiguration problems. Through this exercise we hope to better understand weather and how do the specific AO change support mechanisms (e.g. total, actual, or collected weaving) affect dynamic system reconfigurability.

1 Introduction

Many software systems remain in operation for several decades. Among them there is a class of systems which is required to be continuously operational, as their interruption will result in economic loss (e.g. telecommunications systems), environmental damage (e.g. nuclear plants) or even loss of human life (e.g. life support systems). The maintenance and upgrading of such systems have to be carried out without shutting the systems down, i.e. dynamically.

We call a system *dynamic* if it provides support for changing its organisation as a concurrent activity to the application providing services. We also say that a system *supports dynamic applications* if the organisation or functionality of applications based on the system can be changed without the application being interrupted. While both of these properties could combine, providing a *dynamic system supporting dynamic applications*, each of them could also exist independently. Furthermore, a system is a *dynamic aspect-oriented system* if it additionally accommodates dynamic change with *crosscutting* concerns (i.e. concerns simultaneously affecting multiple system or/and application units).

We maintain that the issue addressed by all dynamic systems is that of dynamic reconfiguration, however in the context of aspect-oriented software development the traditional solutions of configuration paradigm based on components, ports and bindings are unsuitable. Traditionally, components explicitly provide ports for binding, i.e. are pre-designed for a specific composition, while aspects cannot always expect pre-planned design support. Consequently, research in dynamic AO has resorted to alternative mechanisms.

In the present paper we look at a Java-based subset of dynamic AO systems, Java being currently the most widely used language in the AO community. Generalising from this set of systems, we identify some alternatives for dynamic change support mechanisms, depending on (a) when the change is incorporated (or *woven*) and (b) how the weaving is accommodated.

We also present a set of generic criteria for dynamic reconfiguration and evaluate the selected systems with respect to it, highlighting how the systems differ, depending on their change support mechanisms. Through this exercise we hope to better understand weather and how do the specific AO change support mechanisms affect dynamic system reconfigurability.

The rest of this paper is structured as follows: section two discusses the above mentioned change support mechanisms in more detail, section three presents the generic reconfiguration criteria, section four discusses how our selected systems compare with respect to the introduced criteria, finally section five provides summaries and conclusions.

2 Change support criteria¹

Having found traditional reconfiguration solutions unsatisfactory, dynamic AO research with Java has resorted to byte-code modification (e.g. [1]) and reflection based (e.g. [2]) solutions. Depending on what stage change is woven into base programme, these solutions can be classified as using:

- Load-time weaving;
- Just in Time (JIT) compiler weaving;
- Dynamic proxies.

Load-time weaving based systems perform byte code transformation at *class loader* level: the code is transformed as it is loaded into the VM. The class loader based approaches either subclass from the Java abstract `ClassLoader` class – this is a standard Java mechanism – or completely replace the root class loader (JMangler [3], AspectWerkz [1] approach) the later however breaches the Java security mechanism. When sub-classing is used, the *problem of advising system classes* arises. This is caused due to Java security mechanism, where applications loaded by user-defined class loaders are prohibited access to Java system classes loaded by the system class loaders. Moreover, the namespace visibility constraints enforced by class loader hierarchies present additional problem when a crosscutting concern needs to be able to access independently loaded modules. Due to all these tradeoffs often custom class loaders, deviating from standard security mechanisms, are preferred (e.g. JBoss [4]).

With JIT compiler level modification the bytecode is loaded into the VM without transformation. The alteration takes place when the JIT compiler compiles the bytecode into a native code. Consequently, for this approach, the JIT compiler needs to be augmented with additional functionality.

The Dynamic Proxy² approach is solely a Java Reflection based solution: no code transformation takes place, only standard Java language mechanisms are used.

Both class loader and JIT based weavings require modification of bytecode. Depending on *how* the code is transformed to accommodate change, the systems can be further classified³ as those using:

- *Total* hook weaving: augment the entire code at each possible join point with a hook to which additional behaviour could reference;
- *Actual* hook weaving: weave hooks only to a set of points of actual interest, not to every possible point of potential interest;
- *Collected* weaving: weave in the code (rather than hooks) for additional behaviour at the points of interest, with the resulting code collecting the aspects and base in one unit.

Each of the above alternatives has its strengths and weaknesses. JIT, for instance, could be used for both *collected* and *hook weaving*. While the *collected weaving* will improve the performance⁴ of the AO system (due to reduction of number of indirect references), it will also tightly bind the aspect and base code, making *unweaving* for advice removal more complicated.

¹ Much of the discussion in this section is synthesized from documentation and publications for the approaches discussed in section 4 of this paper.

² A dynamic proxy is a class generated at runtime that implements one or more separate interfaces. A call to the methods of an instance of the proxy will be re-directed to an object implementing `InvocationHandler`.

³ Adopted from [13-14].

⁴ Assuming that inlined code is executed more than ones, or the inlining takes lesser time than indirect referencing.

The *hook weaving* options, on the other hand, are expensive from the performance point of view, but preserve separation of aspect and base code and simplify future attachment/detachment of aspects to any potential point of interest without need for repeated base code transformation.

The summary table below grades the various weaving options, with A given for best and C for worst results:

Table1. Summary of tradeoffs for alternative weaving approaches

Weaving approach \ criteria	Evolvability	Excessive Code	Performance
Total hook weaving	A	C	C
Actual hook weaving	B	B	B
Collected weaving	C	A	A

It is worth noting that in practice a combination of the discussed mechanisms could be used in a single system.

In this section we have discussed several alternative implementation mechanisms, but how do these mechanisms help to address the dynamic reconfiguration problems? To address this question, we first define a set of generic reconfiguration criteria in the following section, then evaluate systems with the above implementations against these criteria.

3 Criteria for comparison

The criteria presented below are gathered from a body of work on dynamic reconfiguration [5-9], which, as discussed earlier, is directly related to the problems being addressed by dynamic AO systems:

1. *Separation of application functionality from structure (or level of coupling)* determines whether the software system can be defined in terms of loosely coupled units. If yes, then the developers can have 2 different – functional and structural – views on it. The functional view pertains to the state and behaviour of a unit, while the structural view presents the system as a graph of interconnected components (with no care about their functionality).
2. *Preservation of application integrity* is self-evidently concerned with integrity preservation. This criterion can be further unravelled into *change action synchronisation* and *persistent state preservation*. The action synchronisation should ensure that the components involved into the reconfiguration process are made mutually consistent, and persistent state preservation is to ensure that any unprocessed messages of the changing component, and critical application data for the application survives the reconfiguration.
3. *Application contribution to the reconfiguration process* defines how much does the application need to contribute towards the system reconfiguration process. While ideally the application will be completely unaware about reconfiguration, in reality it could need to perform some activities to ensure that its correctness is preserved.
4. *Reconfiguration specification* considers how are the required changes presented. Ideally this will be a declarative specification, telling what needs to be changed, rather than how to do the changes. This criterion contributes towards understandability of the changes and their analysability.

5. *Efficiency*: this can be broken down to *application disturbance* and *time delay*. The application disturbance can be measured in terms of number of components affected by the reconfiguration.
6. *Programmed change robust to evolution*: where both programmed and evolutionary changes coexist, the programmed change must minimise its assumptions about the existing configuration, as the configuration could be unpredictably changed due to evolutionary requirements.

Although it might seem that AO systems should have additional reconfiguration criteria to address due to the wider impact and unanticipated nature of aspects, we are of the view that the above criteria are sufficiently broad to accommodate aspect-specific issues. This is because other reconfiguration actions could potentially have wider impact on system modules (e.g. structural change) or cause unexpected interactions, as aspects could.

4 Systems for comparison

The systems selected for evaluation in this paper are AspectWerkz, JBoss, PROSE, and Nanning. While all these systems are considered to be dynamic, they provide varying level of dynamic change support. A brief introduction to each of these systems is provided (section 4.1) followed by a discussion on how they perform against the dynamic reconfigurability criteria (section 4.2).

4.1 Outline of the systems

AspectWerkz [1, 10, 11] is a dynamic AOP framework for Java that uses *load time actual hook weaving* with custom-enhanced core java class loading architecture hooked in directly after the bootstrap class loader. It can perform bytecode transformations on classes loaded by all but the bootstrap loaders¹. The framework has several architectures to support different versions of Java, however the recommended version (that for Java 1.4 release) is presented in the Figure 1².

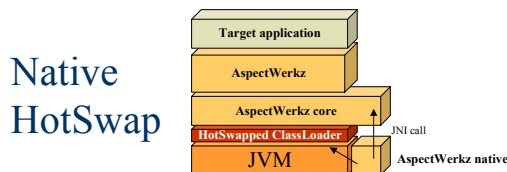


Figure 1. AspectWerkz Native HotSwap for Java 1.4 architecture. The JVM is launched with a native JVMPi extension. This extension, when loaded, hotswaps the java.lang.ClassLoader to AspectWerkz core and native JVMDI API. Thus, the system uses a single JVM, and a non-intrusive way of plugging in AspectWerkz in Java 1.4.

JBoss [4, 12] is a reflective and reconfigurable Java application server [4] which uses the JBoss AOP framework – where aspects are implemented as interceptors – to provide a set of crosscutting services (e.g. persistence, remote access etc.). JBoss uses *load time weaving* with a custom class loader (called *unified class loader*) that:

- loads all classes into a flat namespace, for the components to be able to share non-system classes, and

¹ The core *theoretical concept* of hooking in the class loader hierarchy is inspired by the JMangler project, but more recent versions of AspectWerkz (after 0.8) do not use JMangler at all.

² This figure is used from [11].

- inserts hooks into the bytecode to allow interceptor attachment for interception for field, constructor, and method manipulation.

By default JBoss uses *actual* hook weaving, in which case it is not possible to hot-deploy any AOP constructs to the units with non-augmented joinpoints. However, the system also provides an option for *total* hook weaving for any unit marked as *advisable*.

The general architecture of JBoss is presented on Figure 2.

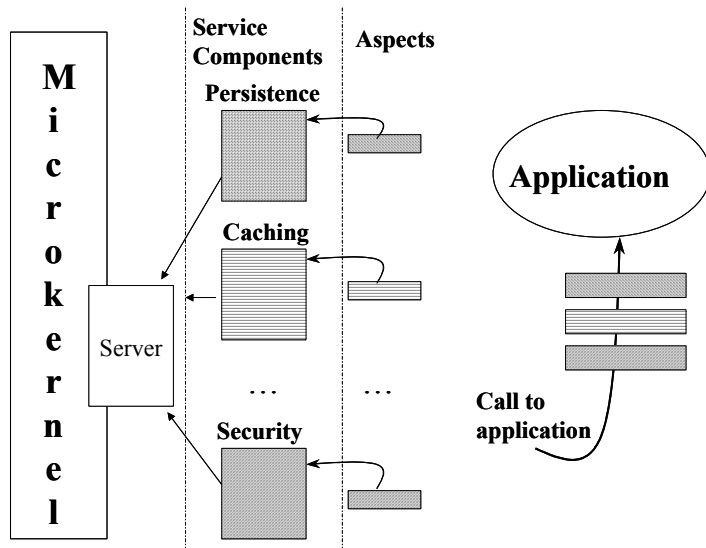


Figure 2. The JBoss architecture. The system consists of a *microkernel layer* which provides most of the “application server” functionality; a set of *deployable services* that can be dynamically added or removed, as required; an aspect layer where aspects are implemented as interceptors; and an application layer which contains the user applications written in plain Java [12]. The aspect layer is used to link the application layer to the JBoss services, allowing container type functionality to be added to plain Java objects. Both *dynamic proxies* and *interceptors* are used in this process.

The **Prose** [13-17] framework uses *JIT compiler weaving*, and allows both *total* and *actual* hook weaving. The weaving scope is specified through a (pattern of) class names and a specific Boolean variable, if the variable is set to false only the classes matching the given name (pattern) are augmented, otherwise these matching classes are left unchanged, and the rest of the code is augmented.

The system consists of two main layers – the Execution Monitor and the AOP Engine – demonstrated on Figure 3.

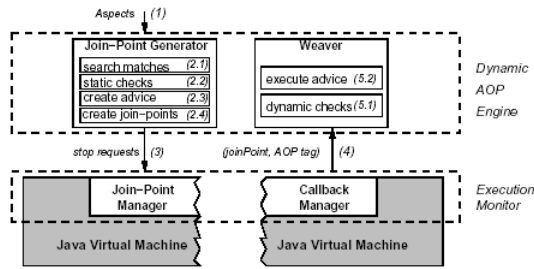


Figure 3¹: Prose architecture. In the upper layer the AOP Engine accepts aspects (1) and transforms them into basic entities like joinpoint requests (2.1-2.4)², then activates these joinpoint requests by invoking methods of the Execution Monitor (3). The Execution Monitor is integrated with the JVM and serves to activate joinpoints at JIT time and to notify the AOP Engine when a joinpoint has been reached (at run time (4)). When notified, the Engine then executes the advice (5) for that joinpoint.

Although Prose can be extended by adding new *types* of joinpoints, it can only be done statically and requires quite a number of changes all across the system.

The **Nanning Aspects** [2] framework uses the *Proxy* facilities of Java Reflection API (Figure 4) and interceptors to provide aspect-oriented-type functionality at run time. Aspectised objects in Nanning consist of sets of interfaces, target objects, and interceptors. The interceptors are used for *around-advise-like-function*, i.e. an interceptor is called when the method is called and is responsible for further propagation of the call either to other interceptors, or to the initially called method. Currently only method-call pointcuts can be advised.

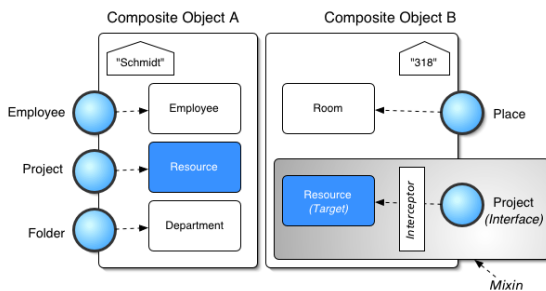


Figure 4³: Objects in Nanning.

No language extensions are used for any of the discussed systems.

4.2 Comparative Evaluation per Criteria

In the following discussion the systems themselves, as well as applications build with these systems are discussed.

¹ Figure reused from [13].

² 2.1 – inspect all the currently loaded classes and gather all matching requests (e.g. methods with given signature); 2.2 – perform static check (e.g. given method exists etc.); 2.3 – define the client data to be passed back by the weaver then the joinpoint is actually reached (e.g. the advice); 2.4 – generate joinpoint requests.

³ Figure reused from [2].

4.2.1 Separation of application functionality from structure

The **AspectWerkz** framework itself is dynamic only in so much as the `java.lang.ClassLoader` is hotswapped to AspectWerkz core at framework load time. After this change the framework remains static; so the *framework itself* is not dynamically reconfigurable. It does however provide a certain set of *dynamic application support* features: advices can be added, removed and re-structured and the implementation of introductions can be swapped at runtime.

In AspectWerkz applications the advice and introductions are written in plain Java and their target classes can be regular plain Java objects. Aspects can be defined using either an XML definition file or Runtime Attributes. In the recent version of the framework (AW 0.9 release candidate) an aspect with its related pointcuts, introductions and advice can also be defined within a single aspect class. Whichever way aspects are defined, they need to be referenced to from within an XML¹ definition file that states which aspects apply to which classes. It should be noted that until recently no new aspects or pointcuts could have been introduced into a running application, neither the existing ones could have been removed, but the advice and introduction defined on existing pointcuts could have been taken out, replaced or updated. However, the very recent work on this system [18] is building towards allowing dynamic aspect deployment, though the issues of un-deployment have not yet been addressed.

Thus, at runtime, aspects are tightly coupled to the application and pointcuts are tightly coupled to the aspects in which they are designed. On the other side, the coupling of advice and introductions to the aspects is relatively loose.

JBoss middleware components are based on *Java Management Extension (JMX)* standard [19], which defines an architecture for dynamic management of resources. As a result, each service, managed through JMX, can be added/removed from the running server without affecting the rest of the system. Thus, unlike AspectWerkz, *JBoss itself* is a *dynamic system* with well-defined structure and encapsulated functional modules where AOP framework is also a deployable module.

Applications developed with JBoss AOP are also dynamic, in that at runtime they can obtain, discard, and re-structure services they use through the JBoss AOP framework. In the AOP framework the advice are implemented as interceptors, introductions provide new interface types which can be implemented either from within an interceptor, or through a mixin class, with pointcuts defined in XML configuration file and no distinct entity for aspect concept to be mapped to. Due to this structure, changes in application structure - caused by attaching or detaching interceptors - are separate from the application code and reflected only in the corresponding XML configuration file.

As presented in Figure 3 for **Prose**, the Execution Monitor is embedded in the JVM while the AOP Engine is application specific. The Execution Monitor provides a Model for its JoinPoint API to the AOP Engine, so different engines could be used with the same monitor. Nevertheless, to the best of our knowledge, dynamic replacement of the Engine is not supported, and neither is the run-time change of the selected weaving mode. Thus, the system itself is not dynamic, but does provide *dynamic application support*: aspects can be woven and unwoven to/from the running system.

Aspects and Crosscuts – constructs containing advice and pointcut type information – are Java classes extending provided framework classes. These constructs are loosely coupled to the base application as their joinpoints are extracted and (un-) registered for interception at load time with no explicit binding requirements.

The runtime mechanism in core **Nanning** is entirely based on dynamic proxies. The *AspectInstance* class implements this runtime support for AO by containing aspects which *introduce* and *advise* the *AspectInstance*, as well as handling all calls to aspectised objects

¹ Depending on which aspect definition style is used, this file could contain more or less additional information.

(through its implemented *InvocationHandler* interface). Here *introductions* are implemented as mixins, *advice* as interceptors, and proxies are used to insert interceptors at run time between the call and the actual execution of a method. Pointcuts are instances of *Pointcut* class, defined and stored in aspects. This approach decouples the object implementations from their provided interfaces, since all references to an instance are directed to a proxy.

In short, summary of discussion for these criteria is presented below:

Table 2. Summary for criterion 1: Separation of application functionality from structure.

AspectWerkz	JBoss	Prose	Nanning
<p>Framework itself is not intended for dynamic reconfiguration.</p> <p>Aspects are coupled to the application and pointcuts to the aspect but advice and introductions are loosely coupled to aspects.</p>	<p>JBoss itself is a dynamically reconfigurable system with loosely coupled functional modules.</p> <p>Interceptors, Introductions and mixins are loosely coupled to the applications, but closely dependant on pointcuts and so XML definition files.</p>	<p>Prose framework itself is not dynamically reconfigurable, though its main two components (AOP Engine and Execution monitor) are relatively loosely coupled.</p> <p>Prose aspects are loosely coupled to the base application, but Crosscuts are tightly coupled to aspects.</p>	<p>Proxies in Nanning help to decouple the object implementations from their provided interfaces.</p> <p>Pointcuts are coupled to the aspects, which in turn are coupled to AspectInstance and AspectSystem classes.</p>

4.2.2 Preservation of application integrity

Since for AspectWerkz the class loader hotswap is the only dynamic step in the framework configuration, and that step is carefully designed and synchronised, the integrity of the framework itself cannot be jeopardised.

For the applications based on the framework the following features are helpful in preserving their integrity:

- both advice and introductions are synchronised (except when an introduction provides a marker interface with no implementation);
- both advice and introduction updates are generally thread save;
- default implementation of *JoinPointController* provides clear instructions for *consecutive* execution of each advice/introduction for each join point;
- default implementations for advice and introduction persistence are provided.

The first three points above support *change actions synchronisation*, and the last point is helpful for *persistent state* preservation.

On the other hand, there are no in-built measures for synchronisation of the application state *as a whole* or its persistent state when replacing or updating advice and introduction implementations¹, except the requirement that the replacing introduction implementation has to implement the same interface as the replaced one.

The AOP services in **JBoss** currently provide a set of pre-packaged crosscutting functionality (e.g. transactions, security, etc.) which are additive in nature and so do not interrupt the existing functionality of the applications. All the updates are addressed “in one action”,

¹ Suppose an advice has a state variable defined, when it is replaced with a new advice, the variable in the new advice will not be re-initialised to the value of the one being updated.

triggered by the update of the changed configuration specification file. Furthermore, since objects are not aware of interceptors, they do not need to be adjusted for their change. However, guards for preserving semantic integrity of applications need to be incorporated in interceptors (e.g. persistence should not be introduced half-way through a transaction, etc.). Although in the provided set of JBoss AOP services these guards (e.g. threading and synchronisation) could be taken care of, a wider use of AO for non-standard processing could become conflict-prone in this respect.

One of the requirements for the **Prose** system during its development was “secure and atomic weaving”. The atomicity is supported through blocking the advice execution in the hook methods till the weaving operation for the whole system is completed.

Atomicity of weaving ensures simultaneous update of affected units. An option for transactional weaving for several (un-) weave operations is also provided, allowing weaving operations to be prepared, but not committed until a separate commit instruction. This option could be used, for instance, to synchronise weaving on distributed systems, or to indirectly assist in persistent state preservation. While persistent state preservation is not supported explicitly, a developer aware of potential persistent state violation, can provide specific checks before committing (several) changes. For instance, suppose “withdraw from account A and credit to B” is being processed; after the withdrawal has been executed a transaction support aspect is dynamically woven in. Due to atomic weaving all affected parts of the application will acquire transaction support from the point of the weaving onwards, but for the completeness and correctness of the first semantic transaction the withdrawal needs to be accounted for as well. Via the “commit” operation, the developer will be able to instruct the committing of transactional aspect, for instance, after completion of the started semantic transaction.

On the other hand, since the dynamic support for Prose allows only for weaving and unweaving of aspects in transactional manner, the issue of integrity of aspects themselves does not raise.

Since in **Nanning Aspects** all references to an object instance are directed to a proxy, an object implementation can be changed during runtime with all its references still remaining intact. Thus change of advice or introduction implementations will not affect the applications as long as the interfaces remain in use. Besides, the Proxy is a Serializable class which is helpful for change action synchronisation.

The summary of discussion for this criterion is presented in Table 3.

Table 3. Summary for criterion 2: preservation of application integrity.

AspectWerkz	JBoss	Prose	Nanning
The integrity of framework itself is well preserved. There is also some support for change synchronisation and persistence in applications, but a holistic support mechanism is missing.	Pre-packaged aspects can cope with change synchronisation and persistent state preservation, however the wider use of AO could be conflict-prone if integrity guards have to be cared for in interceptors.	Since insertion and withdrawal are performed through registering and un-registering points of interest to events, addition or removal of aspects does not significantly disrupt the application, but there is no explicit support for persistent state preservation.	Provides some support for synchronisation of change and persistent state preservation.

4.2.3 Application contribution to the reconfiguration process

As already mentioned, in the **AspectWerkz** framework the application has a significant role to play in preserving its integrity. In particular since there are no semantic integrity preservation constraints at the framework level, these must be provided at the application level. To achieve

this the application might need to provide custom implementations for a number of default features, such as:

- *JoinPointController* concerning business logic for handling e.g. advice redundancy, dependency, or compatibility, or special exception handling.
- pluggable container, e.g. for an application-specific persistence policy;
- in cases when a thread is being hotswapped into an advice, a special method for thread resumption needs to be called manually;
- when replacing an introduction, the new one is required to implement the same interface as the one being updated.

Since the intention of the **JBoss** AOP is to augment the application with additional functionality without its knowledge, the application should not be required to provide any contribution at all. However, in reality the application sometimes needs to provide certain support for dynamic aspect reconfiguration. For instance in order to be able to make an object remotely accessible through JBoss Remoting aspect, the object to be aspectised must have a default constructor in its class definition and also the parameters and return values for the remotely invoked method must be Serializable, etc. These and similar issues must be explicitly dealt with by the applications.

In **Prose** the weaving and unweaving are transactions. However, as already discussed in the example for *preservation of application integrity* subsection, the system does not explicitly address the issues of persistent state preservation and the application programmer should provide appropriate checks at the application level.

Although **Nanning** uses some mechanisms for change synchronisation, applications could be required to contribute to the reconfiguration process if, for instance, some private non-persistent data needs to be processed before change.

The summary of discussion for this criterion is presented in Table 4.

Table 4. Summary for criterion 3: application contribution to the reconfiguration process.

AspectWerkz	JBoss	Prose	Nanning
The application has a significant role to play in the reconfiguration process, triggered by need to preserve its integrity.	The applications need to provide certain support for reconfigurability with aspects.	The applications need to provide certain support for reconfigurability with aspects.	The applications might be required to provide certain support for reconfigurability with aspects.

4.2.4 Reconfiguration specification

In **AspectWerkz** the change specification could be considered imperative, as the steps to be taken need to be provided in a prescriptive pieces of Java programme. And although all used methods are provided by the system, part of the specification requires some additional coding (e.g. how to re-order the advice applied to a pointcut.).

In **JBoss** change specification is provided through XML files, thus it is declarative. XML is easily analysable as well as understandable to the readers. Not only change specification is provided via XML, but also the pointcuts and class metadata. While it is helpful to have all this data available in one place, the file could become very large and so less readable.

The types of change supported by **Prose** are aspect weaving and unweaving. Both of these changes can be specified declaratively either through a graphical user interface of the Prose

WbProse tool, or through command line tool for both local and remote weaving/unweaving operations.

Configuration of **Nanning's** aspects is provided either externally in a declarative XML file or in Java code and via run-time attributes. While XML configuration allows to locate everything in a single file, it also could become very large with large portion of the system's behaviour being defined in it. Nanning documentation [2] cautions against this "XML hell", suggesting partial use of imperative in-code configuration.

The summary of discussion for this criterion is presented in Table 5.

Table 5. Summary for criterion 4: reconfiguration specification.

AspectWerkz	JBoss	Prose	Nanning
Could be considered imperative as some "how to change it" code could be required, though some declarative XML is also used.	Declarative, via XML.	Declarative, via set of simple commands (e.g. insert) or GUI-based tool.	Could be a mix of imperative and declarative styles.

4.2.5 Efficiency

In **AspectWerkz** the disturbance due to dynamic change is limited to actually updated advice or introductions, with no effect on already loaded code.

The overhead of bytecode modification is rather light when no class has bound aspects, or binding occurs only once per class in each class loader hierarchy. The documentation [1] suggests that an advice or introduction adds an overhead of only ~ 0.00025 ms per call¹.

In **JBoss** the additional bytecode added by the instrumentation for interceptor attachment have some time delay hit. This could be minimised by fine-tuning the AspectManagerService mbean (which is pre-defined to instrument all possible points for pointcuts) to disable instrumentation of some unused pointcuts (e.g. all Filed access points). However, this will disallow use of interceptors for non-augmented points, and if their use is required later on, the base code will have to be re-augmented. When no re-transformation is required, disturbance due to interceptor attachment/detachment is rather limited.

In **Prose** the disturbance measure is high for 2 reasons:

- due to transactional nature of weave/unweave operations, all affected units in the system as well as units that will attempt to use them, will be prevented from progression until the completion of the change operation.;
- since there are no other update operations than weave/unweave for the whole aspect, every change to the woven aspects will require unweaving and reweaving, thus affecting all units advised by the changing aspect.

On the positive side, Prose provides both run-time and insertion-time filtering. The insertion time filtering is used to prevent joinpoint activation during aspect insertion. This is a more efficient way of filtering, as no run-time overhead will be introduced due to non-required joinpoint activation and evaluation at run time. The *response time* of dynamic weaving is also negligible [13], though PROSE is relatively slow, compared to other approaches.

In **Nanning** disturbance will be limited to the currently updated aspectised object. And although here, like in AspectWerkz, *actual hook weaving* is used the response time will be significantly higher due to extensive use of reflection and presence of the proxy layer.

The summary of discussion for this criterion is presented Table 6.

¹ Measured on Pentium 4 2.56 Mhz, 215 RAM.

Table 6. Summary for criterion 5: Efficiency.

AspectWerkz	JBoss	Prose	Nanning
Limited disturbance and good response time.	Disturbance is limited, but there is a trade off between disturbance and response time.	Rather high disturbance, but negligible response time for dynamic weaving.	Limited disturbance but high response time.

4.2.6 Programmed change robust to evolution:

Ideally, an aspect should be able to introduce a change into the base without any undesirable side effects. However, by now it is well known that some issues of the AO technology make it difficult to guarantee side-effect free evolutionary aspectisation. Such issues arise, for instance, due to use of generic patterns (e.g. with wildcards) in pointcut specifications¹ or unintended interaction between aspects applied to the same joinpoint, etc. These generic pitfalls are present in all of the considered systems, as all of them use, for instance, patterns for pointcut specification.

Applications developed with the **AspectWerkz** framework support change in terms of addition, removal and updating of advice and introductions to already defined pointcuts, and more recently addition of new aspects with their pointcuts. **Prose** supports change through registration or change of pointcuts and dynamic weave/unweave of aspects applied to them; and **Nanning** - through change in object implementation and new interfaces. The **JBoss** AOP framework supports addition and removal of aspect services independently of each other. These changes could be considered as planned if set of potentially useful aspects have been developed and provided with the applications, to be used as and when needed; or as evolutionary if these aspects have been developed later on, in accordance with changing requirements; or, in case of **AspectWerkz**, **Prose** and **JBoss**, if they need to be applied to the initially non-augmented parts of these systems. Evolutionary change requiring other types of change support, on the other hand, could require updating and extending the frameworks.

With respect to the frameworks themselves, JBoss is suited for evolutionary change due to use of Configuration paradigm in its design, where loosely coupled service components can be modified without affecting the rest of the system. The **AspectWerkz**, **Prose** and **Nanning** frameworks themselves have not been developed for change, though, due to use of OO for their implementation, subclassing and interfaces could be used for static evolution.

The summary of discussion for this criterion is presented in Table 7.

Table 7: Summary for criterion 6: Programmed change robust to evolution.

AspectWerkz	JBoss	Prose	Nanning
Supports addition, removal and updating of advice and introductions for applications, and more recently also addition of new aspects with their pointcuts. The framework itself is not designed for change.	AOP framework supports change in terms of use of aspectual interceptors. The server itself supports evolutionary as well as planned change.	Supports change through change of pointcuts and aspects in applications. Aspect Engine can be replaced, though the framework itself is not planned for dynamic change.	Supports change through proxies. The framework itself is not planned for evolutionary change.

¹ For instance, items from newly added modules of code, irrelevant to the initially defined pointcut, could be caught by it.

5 Summary and Conclusions

In this paper we have discussed the problem addressed by the dynamic AO systems – dynamic reconfiguration with crosscutting concerns – and talked about mechanisms used to achieve it. We have provided a set of criteria for evaluating whether the problem is addressed effectively and discussed how several dynamic AO systems compare in this respect.

We have questioned how do the change implementation mechanisms affect the dynamic reconfigurability of systems where they are used in?

From the above discussion we can conclude that *hook weaving* (AspectWerkz, JBoss, Prose) allows for *loose coupling* of base and aspect code. When *actual* hook weaving is used (AspectWerkz, and possible option for JBoss and Prose) the reconfiguration *efficiency* of the systems is low for aspectising the none-augmented code (due to high *disturbance* and re-augmentation *time delay*) but the *application performance* is better, compared to *total* hook weaving (option in JBoss, Prose). The later option provides constant efficiency due to absence of non-augmented code, but poorer general application performance due to unnecessary processing from unused hooks. Total hook weaving also supports *programmed change robust to evolution* best as it has readily prepared hooks at possible joinpoints for any potential use. None of the systems uses *collected weaving*, which could be explained by the additional complexity required for dynamically unweaving the in-lined code.

The change implementation mechanisms do not seem to have a direct bearing on criteria such as *preservation of application integrity*, *application contribution to the reconfiguration process* and *reconfiguration specification*. Yet, these mechanisms could, of cause, be used to help address some issues (such as achieving *change synchronisation* by blocking execution in hooks for the whole duration of weaving process, as in Prose) in individual system implementations.

A notable trend in dynamic Java-based AO systems (including AspectWerkz, JBoss, Prose, as well as systems not discussed in this paper: e.g. JAC [20]) is that most of them turn to byte-code transformation rather than reflection. Although core Nanning Aspects does not resort to anything but reflection, frameworks based on top of core Nanning (such as cache, preveyley, etc.) do use byte-code manipulation. The likely cause of this is limited power of Java Reflection which supports introspection but not structural change (except for dynamic method hotswap in Java HotSpot VM).

These are our generic conclusions from the presented theoretical discussion. While a practical evaluation of the systems will clearly be beneficial we have postponed it to a future paper due to the lack of space. Nevertheless, it should be noted that an objective practical evaluation will have to be carried out against several scenarios, as each scenario could be better suited to a particular system's implementation. Moreover, we are aware that it could be difficult to make attributions due to the problems in identifying whether a specific result is achieved due to the underlying change mechanisms, or due to the implementation particularities of a given system.

References:

- [1] J. Boner and A. Vasseur, "AspectWerkz Web Site, <http://aspectwerkz.codehaus.org>," 2004.
- [2] J. Tirsén, J. Larsson, R. Lillsjö, J. Lind, and L. AB, "Nanning Aspects web pages , <http://nanning.codehaus.org/overview.html> AND <http://nanning.snipsnap.org/space/Overview>," 2003.
- [3] G. Kniesel, P. Costanza, and M. Austermann, "JMangler - A Framework for Load-Time Transformation of Java Class Files," in *First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, 2001.
- [4] M. Fleury and F. Reverbel, "The JBoss Extensible Server," presented at International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil, 2003.

- [5] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1293-1306, 1990.
- [6] I. Warren, "A Model for Dynamic Configuration which Preserves Application Integrity," in PhD Thesis, Computing Department. Lancaster: Lancaster University, 2000.
- [7] C. Hofmeister, E. White, and J. Purtilo, "Surgeon: A packager for Dynamically Reconfigurable Applications," *IEE Software Engineering Journal*, vol. 8, pp. 95-101, 1993.
- [8] J. M. Purtilo and C. R. Hofmeister, "Dynamic Reconfiguration of Distributed Programs," presented at 11th International Conference on Distributed Computing Systems, Texas, 1991.
- [9] P. Oreizy, N. Medvidocic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution," presented at International Conference on Software Engineering, Kyoto, Japan, 1998.
- [10] J. Boner, "Self-defined aspects in AspectWerkz - new definition model <http://blogs.codehaus.org/people/jboner/>," 2003.
- [11] A. Vasseur and J. Bonér, "AspectWerkz: A deeper view in the new "online mode" architecture, <http://aspectwerkz.codehaus.org/downloads/papers/AspectWerkz-online-architecture.ppt.zip>," 2003.
- [12] B. Burke, A. Chau, M. Fleury, A. Brock, A. Godwin, and H. Gliebe, "JBoss Aspect Oriented Programming," The JBoss Group, <http://www.jboss.org/developers/projects/jboss/aop>, 2003.
- [13] A. Popovici, G. Alonso, and T. Gross, "Just In Time Aspects: Efficient Dynamic Weaving for Java ." presented at 2nd International Conference on Aspect- Oriented Software Development, Boston, USA, 2003.
- [14] A. Popovici, T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, G. Kiczales, Ed., 2002, pp. 141-147.
- [15] A. Popovici, A. Frei, and G. Alonso, "A proactive middleware platform for mobile computing," presented at 4th International Middleware Conference, Rio de Janeiro, Brazil, 2003.
- [16] A. Popovici, G. Alonso, and T. Gross, "Spontaneous Container Services," presented at 17th European Conference for Object-Oriented Programming, Darmstadt, Germany, 2003.
- [17] A. Popovici, G. Alonso, and T. Gross, "PROSE website <http://prose.ethz.ch/Wiki.jsp?page=AboutProse>," 2003.
- [18] J. Boner and A. Vasseur, "AspectWerkz Source Code <http://aspectwerkz.cvs.codehaus.org/viewcvs.cgi/aspectwerkz/?root=aspectwerkz>," 2004.
- [19] "Java Management Extensions (JMX) Overview, <http://java.sun.com/products/jmx/overview.html>," Sun Microsystems Inc., 2003.
- [20] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java," presented at 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001.