A Framework for Policy Driven Auto-Adaptive Systems using Dynamic Framed Aspects

Phil Greenwood, Lynne Blair

Computing Department, InfoLab21, South Drive,
Lancaster University, Lancaster, UK, LA1 4WA
{greenwop, lb}@comp.lancs.ac.uk

Abstract. This paper describes and evaluates a framework that allows adaptive behaviour to be applied to systems by using a combination of dynamic AOP, parameterisation and policies. Our approach allows the operator to create policies to define adaptive behaviour based on Event-Condition-Action rules. The combination of dynamic AOP with parameterisation aids reuse and allows aspects to be generated to suit the current system conditions; these aspects can then be woven at run-time to adapt the application behaviour.

This approach is evaluated in two ways; firstly performance measurements are presented to show that such behaviour does not add a substantial overhead to the target system. Secondly, Aspect-Oriented software metrics are applied to the adaptations applied to illustrate their reusability and flexibility.

Keywords: dynamic AOP, frames, dynamic adaptation, auto-adaptive, parameterisation, policies, aspect interaction.

1 Introduction

Many modern applications are required to operate in environments that are susceptible to change. There are numerous properties that could vary during an application's run-time, from the available hardware resources to the user requirements. For a system to continue operating optimally, it is vital that the system adapts to suit the current conditions. Aspect-Oriented Programming (AOP) [1] is now a common and recognised approach to implementing system adaptability.

A traditional approach to system adaptation is to stop the target system, apply the changes, and then restart the target system [2]. Critically, any stopping of the system is undesirable for domains that require high levels of availability. More recently, focus has shifted to dynamically adaptable systems. These types of systems [3-7] do not require the target system to be taken off-line for adaptations to be made, thus maintaining high levels of availability.

These dynamically adaptable systems can be split into two categories. The first category of such systems rely on human system operators to decide when and what adaptations should take place; we shall call these *manually-adaptive* systems.

The second category of dynamically adaptable systems are able to apply adaptations automatically, typically by monitoring the target system and selecting the most appropriate adaptation when it is required. This category is what we shall call *auto-adaptive* systems. There are a variety of systems that achieve this in different ways [3-5, 7].

Several benefits can be gained from implementing an auto-adaptive system:

- the system is able to react much quicker to changes in the environment;
- the chances of errors being introduced to the system are lowered due to reduced human contact with the system;
- improved levels of availability.

The majority of previous research has focused on the use of component-based development to implement adaptive systems [8]. The loosely coupled nature of components eases their dynamic replacement and makes them a suitable choice for adaptive systems. However, one fundamental problem with component-based programming is that cross-cutting concerns cannot be easily encapsulated; this reduces maintainability and causes large disruption when a concern spread over a number of components requires adapting.

We propose that dynamic AOP (dAOP) [9] is an ideal technique to prevent these problems and provides suitable mechanisms for developing adaptive systems. dAOP has previously been used to adapt system behaviour at run-time [3, 4, 7]; we aim to exploit this technique further. Our motivation is to improve the reusability and flexibility of the adaptations and apply them automatically without incurring a substantial overhead to the target system. Improving the reusability will have a variety of benefits associated with the development of adaptations, such as reducing the implementation effort. Additionally, by increasing the flexibility of the adaptations it is possible to customise them to the state of the target system at the time they are applied. This allows the target system to be adapted in a more suitable and precise manner. We hypothesize that by combining policies [5, 10], parameterisation techniques [11] and reflection [12] with dAOP we can achieve these aims.

Our goals in this work are to develop a flexible framework capable of adding auto-adaptive behaviour retrospectively to existing systems. More specifically this framework should:

- require no change to the underlying target system;
- be able to extend/modify adaptations;
- create flexible and reusable adaptations;
- ensure correctness and compatibility between the aspects and the system;
- not add a substantial overhead to the target systems performance.

The evaluation presented will show that these aims have been met without incurring a substantial overhead. Performance measurements are presented to illustrate the overhead from performing a variety of operations to implement this auto-adaptive behaviour. To illustrate that the adaptations implemented are both reusable and flexible, AO software metrics are applied to measure these properties.

The remaining structure of this paper is as follows. Section 2 gives an introduction to auto-adaptive systems and the background of the main technologies being used in

this work: policies, Framed Aspects, dAOP and reflection. Section 3 then details the implementation used and section 4 evaluates this work by giving a series of performance measurements and evaluates the reusability/flexibility of the adaptations used. Our implementation is compared to a number of related systems in section 5. Section 6 then describes potential future work. Finally, section 7 concludes this paper by summarising its key findings.

2. Background

This section will describe the background of the various technologies used in this work, including a brief description of existing adaptive systems.

2.1 Adaptive Systems

There are several examples of adaptive systems implemented in different ways to meet different goals, such as component based solutions [13, 14], middleware implementations [15], or AOP based solutions [3, 7, 16]. One similarity they all posses is the ability to alter their behaviour dynamically to suit the current context of use.

Adaptive component-based middleware systems [17] are currently a very active research area. A component is a module of code that is well defined with a set of required and provided interfaces [18]. Typically this style of adaptive system ensures that components are independent of each other, i.e. that components are loosely coupled. This makes it easy to dynamically replace unsuitable components with components which are more suitable.

However, these component-based solutions do posses certain inherent limitations. Firstly, such solutions cannot capture crosscutting concerns. If a particular type of behaviour needs altering and a number of components co-operate to implement this, each component involved has to be modified and then replaced. This is obviously inefficient, reduces the maintainability of the system and can possibly cause great disturbance in the system. Another restriction is that component-based solutions only allow coarse-grained changes, in that the finest granularity of change is the component level. They do not allow changes to elements contained within the component such as methods, attributes, etc. Again this could result in inefficient changes since whole components must be replaced to make a small change.

Implementing the framework using an AOP based solution will remove these restrictions. A finer-granularity of change can be applied to each component of the system. In addition to replacing a whole component, the behaviour of a single method or an attribute can also be modified; allowing much more precise and efficient changes.

Using AOP will allow the structure of the adaptations to be kept at run-time and so ease the process of applying and later removing adaptations. Additionally, the joinpoint model of AOP techniques provides us with a natural mechanism to define adaptation points in the system. Several dynamically adaptable systems have already been

implemented using AOP; section 5 will compare and contrast these approaches with our framework.

AOP can be used to implement an auto-adaptive system in two contrasting ways. Either *the required changes* can be encapsulated or *the adaptation process* can be encapsulated. Examples of this latter method are described in [19] and [20]. These implementations allow the programmer to create the application logic with little thought required on how to make the system adaptable as long as good software design practices are followed. The adaptation logic is encapsulated inside an aspect and woven to the base-system to make it adaptable. These aspects typically contain hooks to allow adaptations to be added at run-time. Auto-adaptive systems that employ this approach are generally implemented using a static AOP implementation such as AspectJ [21]. This limits their overall dynamicity as changes to the adaptation process or the addition of new points of adaptation are not possible without stopping the system. Our approach follows the former approach, where the required adaptations are encapsulated inside *dynamic* aspects. This approach provides greater flexibility and achieves high levels of adaptability, whilst still achieving good levels of separation of concerns.

2.2 Policies

To achieve *auto*-adaptation, a method to define the conditions when changes should take place is needed. Policy rules are widely used to implement this type of behaviour, a policy rule being defined as a rule governing the choices in behaviour of a managed system [22]. Event-Condition-Action (ECA) rules are a commonly used technique, often encountered in active-databases [23]. ECA rules are so named as they are processed when a defined *event* occurs. When this event is triggered the *condition* specified is evaluated. If the condition is true then the defined *action* is executed. This format fits well with the structure of our framework and the ECA model can be easily extended to be used with dAOP.

These policies can also be used to define relationships that may exist between the adaptations. Defining these relationships within the policies keeps all information regarding system behaviour in one location. If not specified, certain relationships could cause undesirable interactions to occur between aspects. This is especially relevant in auto-adaptive systems where the aspects are woven automatically. A set of examples are given in section 3.6 to illustrate such interactions and how our policy specification can prevent them.

Formal analysis of aspect interaction is an area that has not been widely addressed in the AOSD community [24]. One key exception is the work of Douence et al [25] which involves the formal identification of aspects that interact at the same joinpoints and then the resolution of the interaction problems identified. Our approach aims to address the occurrence of such interactions solely at an implementation level.

2.3 Framed Aspects

Framed Aspects are the amalgamation of Frame technology [11] with AOP. Frame technology was originally conceived in the late 1970s to provide a mechanism for creating reusable segments of code using a combination of meta-variables, code templates, conditional compilation and parameterisation. A developer initially creates a specification from which the "framed" code is automatically processed to generate concrete customised code. Using a generative process in this way allows code to be reused in many different systems and contexts making frames ideal for the creation of code libraries or software product lines.

Frames alone cannot cleanly encapsulate crosscutting concerns. Any evolution of the software may require changes to be made across a number of modules and could cause architectural erosion [26]. Frames also require that the variation points are explicitly specified in the code; this can cause difficulties reading and understanding the code. By combining Frames and AOP, system features which are crosscutting, can be encapsulated within a single module.

Framed Aspects [27] is one such approach to amalgamate Frame technology with AOP. The Framed Aspects implementation is a customised version of XVCL (XML-based Variant Configuration Language) [28] for use with AOP which allows features to be composed together in a non-invasive manner. In section 4, Framed Aspects and XVCL are compared using a quantitative metric to illustrate the differences between the techniques. Figure 1 shows the modules required to implement Framed Aspects.

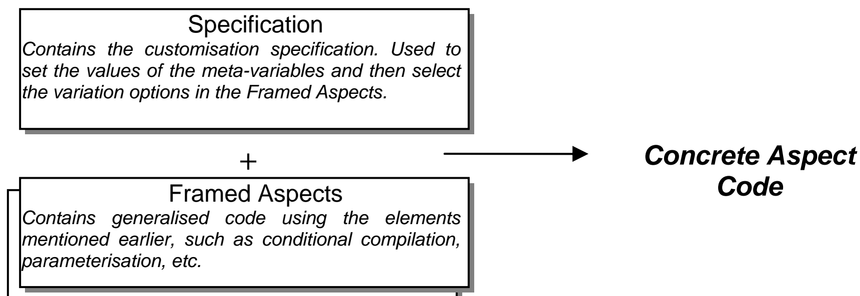

Fig. 1 Framed aspect composition and the required modules.

For aspects to be successfully parameterised and generalised, variation points within the aspect need to be identified using an appropriate Frame construct. [27] explains the Framed Aspect process and methodology in more detail and contains an example to show how a simple cache aspect can be generalised using frames, in order to make it reusable. In section 3.5 of this paper a variation of this cache example is used to show how Framed Aspects can be used in an auto-adaptive environment.

Framed Aspects were chosen for their power and lack of constraints regarding what elements can be parameterised. This is an extremely valuable and flexible property when applying the Framed Aspect approach to adaptations. In our proposed framework the Framed Aspect specification (see Figure 1) is generated automatically at run-time based on the current state of the target system. This allows the generated adaptations to be customised to the current needs of the target system.

2.4 Dynamic AOP

Dynamic AOP is a branch of AOP that allows aspects to be woven dynamically to the base-code of a system at load-time or run-time. This eases tasks, such as testing, patching systems and adapting system behaviour. To clearly illustrate the benefits of dynamic AOP, a comparison with static AOP follows.

Static AOP typically weaves the aspects at compile time. This is less flexible than dAOP in that the aspects and pointcuts need to be pre-determined and defined before the application is deployed. However, static AOP allows much more efficient weaving and advice execution. For example, the overhead for executing AspectJ advice is almost identical to a standard Java method call [29]. In contrast, when a new aspect is woven dynamically using AspectWerkz [30] a significant overhead is experienced and advice execution time is higher due to the reliance on reflection [31]. This is clearly a trade-off between flexibility and performance. Our focus on auto-adaptive systems and the benefits of a flexible framework led us to pursue dAOP techniques. The effects on performance will be discussed in section 4.

The potential problem of undesirable interactions between aspects is more prominent in dAOP when compared to static AOP. When using static AOP, it is more likely that any issues can be identified earlier and problems can be prevented. Some AOP techniques allow relationships and aspect priorities to be defined, such as the precedence rules used in AspectJ [21]. However, we have found that the majority of the rules are not ideal or explicit enough to prevent some of the interaction issues that can occur in an auto-adaptive system. To resolve this we have implemented our own relationship definition technique that will be described in section 3.

A number of different dAOP techniques exist which could have formed the basis for our framework, such as AspectWerkz [30], JAC [32], JBOSS AOP [33] or Prose [9] (see [34] for a comprehensive comparison of these techniques). We selected AspectWerkz due to a number of factors:

- AspectWerkz is a general purpose technique which places no restrictions on the type of system it can be used with;
- it has an extensive joinpoint model, important when implementing an adaptive system since this means fewer restrictions on the points in the system where adaptations can be applied;
- aspects in AspectWerkz are standard Java classes, which eases the creation of the aspect code as programmers will not have to learn a new language;
- AspectWerkz not only allows new aspects to be introduced at run-time but also allows new pointcuts to be added to the target system – an important feature for implementing unanticipated adaptations;
- it maintains aspect structure, which eases the removal and querying of woven aspects.

As mentioned above, aspects used within AspectWerkz are implemented as standard Java classes and advice is implemented as methods within these classes. AOP structures (such as pointcuts, weave model, etc) can be specified in one of three ways: as annotations within the aspect classes, in a separate AOP XML file, or (as used in our approach) specified programmatically at run-time.

2.5 Reflection

Reflection plays an important role in our framework and underpins various features used in dAOP in general. Reflection allows the introspection of a system and the gathering of information at run-time [35]. By using reflection in our framework we can determine structural information such as field types and method signatures or more dynamic information such as the values of fields/parameters. This reflective information is used in our approach, in conjunction with the Framed Aspects, to generate the concrete aspect code; this process is described in more detail in section 3.

2.6 Summary

To summarise, the framework for our implementation of auto-adaptive systems combines a number of technologies:

- ECA-based policies to define behaviour choices;
- Framed Aspects to give flexibility and reuse;
- dAOP to allow changes to be made dynamically at run-time;
- Reflection to gather run-time information.

Using Framed Aspects, dAOP and reflection together allows the operator to dynamically weave code that is specifically generated for the target system's current status. By combining these technologies together with ECA rules, allows explicit control of *when* and *how* the aspects should be generated and woven, a feature not available in current dAOP techniques.

3. Framework Implementation and Behaviour

This section describes the implementation details of our framework. It will cover how the various technologies are combined and describe the various elements which contribute to implementing and managing an auto-adaptive system. Examples will also be given that illustrate how auto-adaptive behaviour can be applied and how undesirable aspect interactions can be prevented.

3.1 Policies

As mentioned earlier, policies are needed to ensure that the system will be adapted in the desired way and at the appropriate times. The policies in our solution are defined in an XML file (as it is relatively easy to be read by both humans and machines) and follows the structure of ECA rules:

- the *event* is some joinpoint being reached;
- the *condition* is some test performed on the state of the target system;
- the *action* involves the weaving or removing of an aspect.

Defining these ECA rules separately from the actual adaptations makes the framework much more maintainable as the behaviour specification is not tangled with other code. This also improves the reuse of the adaptations since defined policies tend to be system specific, whereas the adaptations should be as generic as possible.

The policies are initially processed when the system starts and are constantly monitored for changes so that any modification can have an immediate effect. From these policies, a set of aspects are generated and woven dynamically (see section 3.2). When the specified conditions occur, these generated aspects initiate the execution of the specified actions. Figure 2 is an example of a policy and the attributes that it must contain.

```
<policy name="CachePolicy">
        <weave-condition property="ExecutionTime"   condition="more-than" value="2000"/>
        <remove-condition property="PacketsReceived" condition="more-than" value="20000"/>
        <points pointcut="execution(* autoadaptive.Test*.test*(..))"/>
        <aspect-type type="Cache" frame="true" advice="cacheAdvice"/>
</policy>
```

Fig. 2 Example of the XML used to define a policy.

A description of each of these fields and their function follows below. These fields are used to define the required behaviour, i.e. when an adaptation should be performed and what adaptation should take place. Note that a number of additional fields will also be used to define the relationships between the adaptations to prevent undesirable interactions; these are described in detail in section 3.6.

Policy Name

This is a required field to give the policy a unique identifying name. The policy name is used to refer to and identify the policy whenever actions need to be performed upon it, such as it being removed or edited.

Weave Condition

This field requires three parameters to be specified that are combined to form the condition that triggers the actions of the policy. The three parameters specify the property to monitor, the operator for the condition, and the trigger value (e.g. the policy in Figure 2 checks whether the execution time is more than 2000 milliseconds).

Hardware resources can be observed including memory properties, CPU usage, hard disc usage, network properties and the execution time of a particular method. These physical properties can be monitored in any target system. In addition to these types of properties, run-time properties of the system can also be monitored, such as field values, parameter values and return values of methods. Obviously, these run-time properties are entirely dependent on the target system and the run-time properties it implements.

Weave-conditions can be combined using a set of AND/OR logical operators. These are implemented by listing the weave conditions sequentially and/or nesting conditions within another. Listing the conditions sequentially is equivalent to providing a set of conditions where any of them can occur for the action to be executed. Whereas nesting the conditions allows the programmer to create a set of conditions where each of the conditions must be true for the action to be executed.

Remove Condition
This is an optional field that allows the programmer to specify a condition that triggers the *removal* of the adaptations associated with the policy. This field requires a similar set of parameters as the ones used to define the weaving conditions. Any system property can be used as part of this condition, i.e. different properties can be used in the remove condition and the weave condition. Care has to be taken to prevent inconsistent conditions from being defined. For example the weave and remove conditions could intersect, and the aspect just woven would then be immediately removed. To protect the operator from these conditions, the remove condition is evaluated *before* the aspect is woven. If it holds true then the aspect is not woven to prevent unnecessary operations. These conditions can also be listed sequentially or nested to specify complex conditions.

Points
In order to correctly monitor the target system for the specified conditions, the programmer needs to identify the points that are of interest. The extensiveness of the AspectWerkz joinpoint model allows the operator to naturally pick a wide variety of points and so we have chosen to use this format directly. This model was also extended slightly to allow certain events occurring within the framework to be captured, such as a particular aspect being woven or the actions of a policy being executed. However, such pointcuts are merely syntactic and are actually converted to AspectWerkz pointcut patterns at run-time.

Aspect Type
The required aspect-type field allows the programmer to specify the aspect that has to be woven when the weave conditions are met. The parameters of this field allow the programmer to use two groups of aspects – Framed Aspects or concrete aspects (i.e. aspects which are already compiled and ready to be woven). If a concrete aspect is specified (indicated by setting the frame attribute to false), when the conditions are met the aspect can simply be woven. However, if a Framed Aspect is specified (indicated by setting the frame attribute to true), a concrete aspect needs to be generated using the current system state, as described in section 3.2. An optional attribute included with this field (the pointcut attribute) allows the programmer to specify where the aspect should be woven. This enables the aspect to use a different pointcut from the one which the aspect monitoring for the weave/remove conditions occurring is attached. Finally, the advice attribute is used to specify which advice should be executed within the associated aspect.

3.2 Framed Aspects

Due to the nature of adaptive systems and the unpredictable state of the system at the time changes are required, it is often desirable to modify the adaptations to suit the current state. As described in section 2.3 Framed Aspects allow the creation of code templates via parameterisation; these templates can be reused in a variety of different systems and scenarios. The same Framed Aspect can be used multiple times and in

different systems but have different behaviour depending on the specification provided. Our framework uses the system state to generate this specification; therefore as the state evolves so does the specification.

In our implementation Framed Aspects are used for two purposes. The first is to generalise and construct the aspects used to observe the system behaviour for the weave conditions; these aspects are called *monitoring aspects*. The second purpose is to generalise the aspects that alter the system's behaviour, which we call *effector aspects*.

3.2.1 Monitoring Aspects

All the information required to generate and then weave the monitoring aspects is specified in the policies. The policies are processed to generate aspects that monitor the system for the specified conditions. Once the monitoring aspects have been woven, the conditions specified are evaluated each time the associated *event* (joinpoint execution) occurs. If the *condition* is true, then the *actions* associated with this policy are carried out.

If a policy is removed from the policy definition file at run-time, the system is updated to reflect this change by removing *all* aspects that implement this deleted policy (i.e. both monitoring and effector aspects). Also, if significant changes (such as a change to the conditions or actions) are made to a policy at run-time then aspects associated with this policy are removed, regenerated and rewoven. Additionally, if an entirely new policy is added at run-time, this is processed to create a new monitoring aspect which is then woven dynamically to the target system.

3.2.2 Effector Aspects

The second purpose of the Framed Aspects is to enable concrete *effector aspects* to be generated that alter the target system behaviour. These concrete aspects are specifically suited to the current target system state. All that is needed is a specification that will bind a particular system attribute (or a function of a system attribute) to a particular parameter in the Framed Aspect. This will be illustrated through an example in section 3.5.

Whilst any type of Framed effector aspect can be created, the difficult part is *effectively* parameterising the aspect code. The right balance has to be reached between generalising the aspect enough so it can be effectively reused/customised and making it too complex to use. Guidelines can be suggested to aid the programmer develop Framed Aspects that are both flexible and usable. These issues are discussed in more detail in [11].

The specification to assign the system run-time data to a Framed Aspect meta-variable is specified in a user-defined binding class that must implement the `IFramedAspect` interface. A series of probes and help-classes are provided to ease the collection of run-time data and the setting of these meta-variables.

3.3 System Management

In order to maintain and control the system, our framework includes an overall management system, responsible for:

- processing the defined policies at start-up and during run-time;
- generating monitoring and effector aspects to implement these policies;
- ensuring aspect relationships are maintained to prevent undesirable interactions;
- weaving new aspects and also removing aspects when necessary;
- maintaining a list of aspects that have been woven to the base-system to determine the state of the system and to ensure relationships are maintained;
- monitoring for changes in the policies and making the necessary dynamic changes.

The AspectWerkz API allows the basic retrieval of the currently woven aspects. However, to allow more sophisticated queries to be made, this retrieval functionality is extended in our framework. Information can also be retrieved regarding an aspect's origin, such as its parent policy and other related aspects (such as requires and incompatible-with). This extra information eases the process of maintaining system consistency when weaving an effector aspect to the target system.

3.4 Dynamic Weaving

When using AspectWerkz, new aspects can be attached to existing pointcuts with relatively little overhead. However, when a new pointcut needs to be added, extra operations need to be performed that increases the overhead. AspectWerkz uses load-time modification of classes to weave aspects; hooks are added to the byte-code as classes are loaded. If a new pointcut is to be added to a class which has already been loaded, the "HotSwap" mechanism [36] must be used to reload the affected classes. Unfortunately, this method does have a relatively significant overhead as will be explained in section 4. However, our belief is that the ability to weave new aspects and add new pointcuts is vital. Adaptations unanticipated at development time may be required, potentially needing to be woven at points where no pointcuts currently exist. Aspects can also be detached from pointcuts with relatively little overhead; however removing pointcuts results in a similar overhead to adding pointcuts.

When a pointcut is either added or removed, each class affected by the pointcut needs to be HotSwapped. Although this operation is similar to the process of replacing components as described earlier in section 2.1, the disturbance is in fact much less. HotSwapping of classes does not block or halt any invocations and the system is not suspended. Therefore this process removes these disturbances experienced in component solutions [8].

Note that one potential solution to prevent this overhead completely would be to instrument every possible joinpoint. However, a problem with this solution is that this to adds an overhead. Each time a potential joinpoint is reached a check is performed to gauge whether an aspect should be executed. Although the overhead of each individual check is not as large as the overhead of a single HotSwap operation, the overhead will be significant over long periods of time since many checks will have to be performed. This accumulative effect is undesirable for long-running systems. Addi-

tionally, instrumenting every possible joinpoint will cause a significant rise in the memory foot-print of the target system.

3.5 Dynamic Cache Example

To illustrate how the framework operates and how it can be used to implement auto-adaptive behaviour, the following example involves creating a dynamic cache to demonstrate all the key features of our approach.

The GUIDE system [37] is an interactive tour guide which runs on a wireless hand-held device. The purpose of this system is to create tours of Lancaster depending on the tourist's interests and to deliver information regarding tourist attractions around Lancaster. Due to the mechanism used to deliver data to the device, a desirable feature would be to introduce a cache when delays are experienced on the wireless network. However, due to the resource constrained nature of the device it is not always desirable to have the cache present due to the memory constraints this imposes. Our approach allows a policy to be created dynamically that defines the conditions that should trigger the addition/removal of the cache aspect.

3.5.1 Policy Definition

The following policy (Figure 3) defines a set of conditions and actions that implement the above dynamic cache behaviour. When this policy is processed, a monitoring aspect is generated automatically by the auto-adaptive framework; this aspect's role is to observe the execution time of the specified method.

The policy is given the name CachePolicy, which has to be unique to prevent name conflicts. The condition specified causes action to be taken when the average execution time of the specified method exceeds 500 milliseconds. This condition is applied to methods matching the pattern * guide.Client.get(String). If the execution time exceeds 500 milliseconds then, as specified, a concrete caching aspect is generated from the Framed cache aspect. Significantly, this policy also defines a remove-condition that identifies the conditions when the caching aspect should be removed. In this example, when there is less then 50 Mb of free memory the cache will be removed and so make available more memory to the rest of the GUIDE system. As a pointcut pattern has not been specified to define where the cache aspect should be woven, the cache aspect will be woven to the same pointcut as the monitoring aspect which implements this policy.

```
<policy name="CachePolicy">
    <weave-condition property="ExecutionTime" condition="more-than" value="500"/>
    <remove-condition property="FreeMemory" condition="less-than" value="50"/>
    <points pointcut=" execution(* guide.Client.get(String))"/>
    <aspect-type type="Cache" frame="true" advice=”cacheAdvice”/>
</policy>
```

Fig. 3 Policy to monitor the execution time and to weave a Cache Framed Aspect.

3.5.2 Framed Monitor Aspects

To aid the implementation of the policies, a library of monitoring aspects is provided that can monitor various properties of the system state. Such properties have already

been listed in section 3.1 when describing weave conditions. The monitoring aspects have been parameterised so that concrete aspects can be generated to suit the policies.

The Framed code in Figure 4 monitors the execution time of the specified method and is enclosed by an ifdef Framed Aspect command. If the property to be monitored is the execution time, then this section of code will be selected and included in the concrete monitoring aspect. All the monitoring aspects are built in this manner by using conditional compilation to select and choose the appropriate segments of code.

```
<ifdef item-in="property" value="ExecutionTime">
        long ExecutionTime= ProcessInformation.getExecutionTime();
        if(ExecutionTime<@Operator><@TestValue>){
            <adapt frame="ImplementChanges.frame"/>
        }
</ifdef>
```

Fig. 4 Framed monitor aspect code.

The *if*-statement in Figure 4 has been parameterised (using the value of @ command) to allow the operator and test value to be substituted so the framework can monitor for the desired conditions. The `adapt` frame command allows code to be inserted that is encapsulated in other frames. Figure 5 shows the contents of the ImplementChanges frame, this code is responsible for implementing the required actions when the specified conditions occur.

```
<ifdef item-in="frame" value="true">
  <@Type> aspect= new <@Type>();
  aspectName<@Name>= AW.weaveEffector(<@Name>, this.getClass().getName(), <@Type>,
      <@Expression>, <@Frame>, aspect);
</ifdef>
```

Fig. 5 Framed code to weave the effector aspect.

A number of elements in Figure 5 have been parameterised to customise the actions taken. These parameters are used to define the type of effector aspect to weave, the pointcuts where it should be woven and also information regarding its origins, i.e. the policy and monitor aspect that requested it to be woven. It is important to emphasise that the parameterisation of these aspects leads to highly customisable code.

3.5.3 Framed Effector Aspects

The Framed cache aspect is shown in Figure 6. As can be observed, the basic structure of the code contains a class that will act as a caching aspect, which itself contains a subclass (CacheDS) which is the data structure for storing cached items.

Firstly, the parameterised value that specifies the classname allows multiple instances of the same Framed Aspect to be generated. The Framed cache could be used in other policy definitions; if the classname were not parameterised then this would cause naming conflicts within the JVM. The next parameterised values specify the cache size and percentage to delete allowing the programmer to bind a property of the system state to these parameters. For example, the programmer could specify that the cache size/percentage to delete should be based on the amount of free memory.

Finally, elements within the CacheDS class are also parameterised to allow different types to be stored within the cache. The type to be cached can be retrieved via reflection at run-time and then bound to the TYPE parameter to create a cache to store

the specific type. Care has to be taken to ensure that the aspects are only applied to methods that use the same types to prevent type incompatibilities from occurring.

```
<frame name="Cache">
        public class <@CLASSNAME/>{
          private int CACHE_SIZE= <@CACHE_SIZE/>;
          private int PERCENTAGE_TO_DEL= <@TO_DEL/>;
          private HashTable cache= new HashTable(CACHE_SIZE);

          public Object cacheAdvice(final JoinPoint jp) throws Throwable{
            <@RES_TYPE/> key= (<@RES_TYPE/>) jp.getRtti().getParameterValue()[0];
            CacheDS result= (CacheDS)cache.get(key);
             if(result!=null)
               return result;
             else{
               result= new CacheDS(jp.proceed());
               cache.put(key,result);
             }
            return result.getData();
          }
          class CacheDS{
            private <@TYPE/> data;
            private long timestamp;
            public CacheDS(<@TYPE/> d){
               data= d; }
            public <@TYPE/> getData(){
               return data;
            } } }</frame>
```

Fig. 6 Framed cache code.

```
set= objFactory.createFrameTypeSetType();
set.setVar("CACHE_SIZE");
int size= (MemoryInformation.getFreeMemory()/100)*10;
set.setValue(size);
setList.add(set);
```

Fig. 7 Code to bind the amount of free memory to the CACHE_SIZE parameter.

Figure 7 shows the binding of the amount of free memory to the CACHE_SIZE parameter to create the frame specification (TO_DEL parameter is set in a similar way).

With all these elements implemented, the original client system executes as before but with the monitoring advice being executed around any methods matching the pointcut pattern specified in the policy. When the average execution time of any of these methods rises above 500 milliseconds, an instance of cache aspect is generated and woven.

Figure 8 shows the generated code of the concrete cache aspect that has been customised for the current run-time conditions. As can be observed, all the parameterised elements have been substituted for concrete values. The cache size and percentage to delete have definite values which are based on the amount of free memory. The classname is created which is a combination of the policy name and the Framed Aspect it is generated from. The types used in the CacheDS class have also been substituted for the type to be cached. In this example, this is the return type of the method being monitored (determined using reflection).

Note that some of the policies will be statically defined and loaded when the system starts. However, a dAOP approach is still essential since if, later during the execution of the system, policies are removed then the aspects that implement these policies have to be dynamically removed. A further benefit of dAOP is that the Framed

Aspects in our framework rely on run-time information to generate the concrete aspects. dAOP is needed to weave these run-time generated aspects.

```
public class ResponseTime_Cache{
  private int CACHE_SIZE= 200;
  private int PERCENTAGE_TO_DEL= 2;
  private HashTable cache= new HashTable(CACHE_SIZE);

  public Object cache(final JoinPoint jp) throws Throwable{
    //cache implementation
  }
  class CacheDS{
    private Document data;
    private long timestamp;
    public CacheDS(Document d){
      data= d;
    }
    public Document getData(){
      return data;
    }
} }
```

Fig. 8 Concrete cache aspect.

Summary of Process

To summarise, the general order of events for creating/applying an adaptation in our framework is as follows:

- the policies are parsed;
- Framed Aspects are used to generate a concrete monitoring aspect for each policy;
- these aspects monitor the specified system conditions and when necessary execute the appropriate actions;
- the actions involve a combination of generating concrete aspects from the specified Framed Aspects, weaving aspects or removing aspects;
- policies are continually monitored for changes and the system is dynamically altered to reflect these modifications.

3.6 Interaction Issues

This section will examine some of the potential undesirable interactions that can occur between aspects and will discuss how they can be prevented. The examples presented extend the previous cache example.

3.6.1 Incorrect Execution Order

Suppose the policies of the target system are altered such that, in addition to the cache aspect, an authentication aspect is now required. If this authentication aspect is woven to the same joinpoint as the cache, an undesirable interaction could occur.

In situations where multiple aspects are woven to the same joinpoint, an advice chain is generated that represents the order in which the advice will be executed. If the proceed method is called from within some advice and the end of the advice chain has not been reached, the next advice in the chain is executed. However, once the last

advice in the chain is reached, calling the proceed method will result in the advised joinpoint being executed. This advice chain may be broken at any point by an item of advice *not* calling the proceed method. This results in any remaining advice, and the advised joinpoint, not being executed.

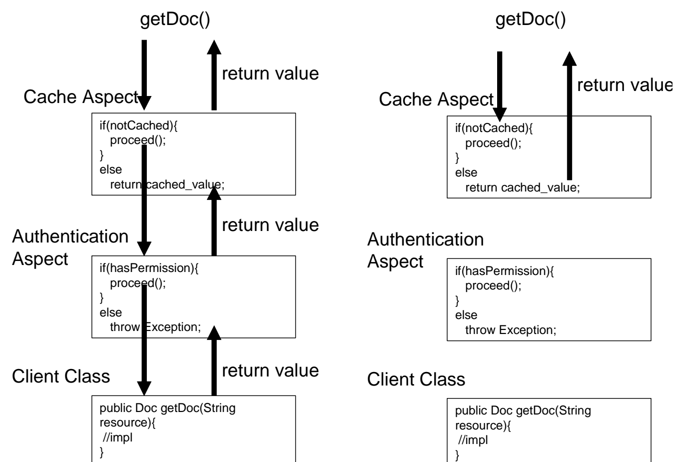

Fig. 9 Flow diagram that shows the sequence of events when an item is added to the cache (left) and when an item is retrieved from the cache (right).

As can be observed in Figure 9 (left), situations where the required resource has *not* been cached the advice chain is executed normally, with both the authentication advice and advised method being executed. However, if the required resource *is* cached (Figure 9 right) then the advice execution chain is broken as the cache advice does not call the proceed method. This causes the authentication aspect not to be executed and users could be granted access to resources they are not authorised to.

This is obviously an undesirable situation and a solution is needed to prevent this type of interaction from occurring. Our approach allows priorities to be assigned both explicitly and implicitly using the rank element of the policies. Firstly, both aspects can be assigned definite priorities (via their respective policies) so that the desired execution order can be achieved, as shown in Figure 10.

```
<policy name=”AuthenticationPolicy>
  …
  <rank value=”1”/>
</policy>
<policy name=”CachePolicy>
  …
  <rank value=”2”/>
</policy>
```

Fig. 10 Policy definitions with both aspects being given a definite rank.

Both related policies in Figure 10 have been given a definite rank, with the authentication policy having a priority of 1 and the cache policy having a priority of 2. This means that the aspects associated with the authentication policy will always be executed before the aspects associated with the cache policy, regardless of the order in which the aspects were woven.

However, this method for defining priorities is not very flexible or dynamic. For example, when a new policy is added at run-time, the priority of the other policies may need to be modified to assign the new policy the desired priority. This could turn into a time consuming (manual) activity if several policies need to be edited so that the new policy can be assigned the correct priority. A more flexible solution that is supported by our framework is to assign the new policy a priority that is *relevant* to the other policies. As shown in Figure 11, this will allow the new policy to be assigned the correct priority without having to edit any other policies.

```
<policy name="AuthenticationPolicy>
  …
  <rank value="++CachePolicy"/>
</policy>
<policy name="CachePolicy>
  …
  <rank value="1"/>
</policy>
```

Fig. 11 The same policies as Figure 10 but the authentication policy is given a rank relative to the cache policy.

The policy shown in Figure 11 specifies that the aspects woven by the authentication policy should be executed *before* the cache aspects. Conversely if the aspects needed to be executed *after* a policy, the ++ should be replaced with --.

Although in Figure 11 the authentication policy rank is only relative to one other policy, multiple relationships can be specified if necessary. This allows complex ordering to be defined with a new aspect able to be inserted between a pair of existing aspects or before/after a set of aspects.

3.6.2 Incompatible Aspects

A scenario could occur where a set of aspects are incompatible with another set of aspects and, in order to operate correctly, one set needs to be absent/removed from the target system. In such a case, the framework is responsible for ensuring that no combination of incompatible aspects are woven simultaneously to the target system.

Suppose in our example that the policies are modified again so that an encryption aspect must be woven at the same joinpoint as the cache and authentication aspect. The purpose of this new aspect is to ensure the privacy of the users of the target system by encrypting the objects retrieved by the user. However, this aspect is incompatible with the cache aspect for two reasons. Firstly, if the cache aspect is executed before the encryption aspect, the objects would be cached with no encryption applied. Secondly, even if encrypted objects were cached this would still be inappropriate as there is no guarantee that the user accessing the cache has the correct key to decrypt the object due to the possibility of keys expiring[1].

It is clear from this scenario that if the encryption aspect is required then the cache aspect should be removed from the target system (assuming that the encryption aspect has precedence over the cache aspect). This relationship can be specified using the incompatible-with element of our policy structure shown in Figure 12.

[1] Cryptographic keys often have an associated expiry time, after which new keys have to be generated.

```
<policy name="EncryptionPolicy>
  …
  <rank value="++CachePolicy"/>
  <incompatible-with aspectname="Cache" frame="true" remove-aspect="true"/>
</policy>
```

Fig. 12 Specifies that the cache aspect is incompatible with the encryption aspect.

Figure 12 shows the encryption policy has been assigned a relatively higher priority than the cache policy. This ensures the encryption aspects will always have precedence over the cache aspect. The incompatible-with element is used to specify that the encryption policy and its aspects are incompatible with the cache framed aspect.

Before any aspects are actually woven, checks are carried out to ensure that the current configuration of the target system allows the encryption aspect to be woven e.g. the framework checks for the presence of any incompatible aspects. If an incompatible aspect is detected, then the highest priority policy gets precedence. In this example, the encryption policy has priority over the caching policy; hence the cache aspect is removed to ensure the incompatible-with relationship is not violated.

3.6.3 Requires Aspect

Alternatively, an aspect may actually *require* the presence of another aspect for it to operate correctly. Suppose that for accounting purposes, logged authentication became necessary, i.e. both authentication *and* logging aspects needed to be woven. Our approach allows a 'requires' relationship to be specified between the authentication aspect and the logging aspect. This can be used to ensure that both aspects are woven at the same time and to the same pointcuts.

Our policy specification can be used to define this type of relationship as shown in Figure 13. The element displayed can be used multiple times to define all the aspects/advice that are required for the policy to be successfully implemented.

```
<policy name="AuthenticationPolicy>
  …
  <requires-aspect aspectname="LoggingAspect" frame="false" advice="log"/>
</policy>
```

Fig. 13 Specifies a 'requires' relationship between the authentication and logging aspect.

Adding this requires-aspect element will result in the logging aspect being woven to the same pointcut and at the same time as the authentication aspect. The logging aspect will have the same priority as the authentication aspect, but will be executed after it due to it being specified later in the authentication policy.

If the required aspect is to be woven at a different joinpoint to the current policy, the optional pointcut attribute can be included within the required aspect definition.

3.6.4 Resolution Aspects

The final relationship between aspects to be examined here concerns resolution aspects [38]. A resolution aspect is used to resolve conflicts between potentially incompatible aspects and allows them both to operate together.

Continuing with our running example, this type of relationship exists between the logging aspect (that was required by the authentication policy) and the encryption aspect. The encryption aspect was introduced to provide some form of privacy to the

users of the target system. However, this requirement would be broken by the logging aspect due to it storing the entries it makes in plain-text.

A resolution aspect can be woven to allow these two aspects to co-exist without conflicting. To achieve this, an aspect is required that encrypts the contents of the log file generated by the logging aspect. However, this log file encryption aspect is only required when *both* the logging and encryption aspects are present. A set of conditions can be defined that specifies when the resolution aspect is required. A special weave condition property, AspectWoven, can be used to check whether the specified aspects are woven. The policy definition to achieve this is shown in Figure 14.

```
<policy name="ResolutionPolicy">
  <weave-condition property="AspectWoven" condition="equals" value="LoggingAspect">
    <weave-condition property="AspectWoven" condition="equals" value="EncyptionAspect"/>
  </weave-condition>
  <points pointcut="actionexecute(AuthenticationPolicy)||actionexecute(EncryptionPolicy)"/>
  <aspect-type name="LoggingResolution" joinpoint="execution(* *.RemoteObject.get*(..))" ad-
            vice="encryptLog"/>
  <rank value="--AuthenticationPolicy"/>
</policy>
```

Fig. 14 The resolution policy to fix the interaction between the logging and encryption aspect.

As can be observed, two weave-conditions have been defined, one nested inside the other to implement a logical AND operation between them. The first weave-condition specifies that the logging aspect needs to be present; the second specifies that the encryption aspect has to be present. Only when both of these conditions are met is the operation to weave the logging resolution aspect triggered. The remaining elements in Figure 14 are used to specify the aspect to be woven (in this case the LoggingResolution aspect) and the priority that the aspect should take. It should be noted that the points element in this example specifies a pointcut unique to our framework (i.e. an extension to the AspectWerkz joinpoint model, as discussed in section 3.1). The pointcut specified in Figure 14 attaches the monitoring aspect to events occurring in other policies; in this case when the actions of AuthenticationPolicy or EncryptionPolicy are executed. Other policy events supported include the removal of adaptations associated with a particular policy and the execution of a particular policy monitoring aspect.

As the target system executes, the conditions could alter so that either the logging or the encryption aspect is removed. This will result in the resolution aspect no longer being needed. The policy in Figure 14 can be extended by specifying the conditions under which the logging resolution aspect should be removed. This will allow the system to adapt itself according to the current conditions. These extensions are shown in Figure 15.

```
<policy name="ResolutionPolicy">
      …
  <remove-condition property="AspectWoven" condition="notequals" value="LoggingAspect"/>
  <remove-condition property="AspectWoven" condition="notequals" value="EncyptionAspect"/>
</policy>
```

Fig. 15 Policy extensions to remove the logging resolution aspect when it is no longer required.

The changes made to the resolution policy specify that the logging resolution aspect should be removed when either the logging aspect *or* the encryption aspect is not present. This logical OR operator is implemented by listing the conditions serially.

3.6.5 Other Issues

Outside policy definitions, there are additional issues that need to be addressed by the framework to ensure that policies are correctly and consistently applied. The first issue involves the actions taken by the framework when a policy cannot be correctly applied due to some incompatible aspects with higher priorities already woven. Under these circumstances, to prevent aspects being deployed in a state which is incompatible, none of the actions specified by the policy are performed. If the state alters and the incompatible aspects are no longer present, the aspects will then be deployed automatically. However, the incompatible relationship only comes into effect when *both* sets of incompatible aspects should be woven to the target system at the *same* time.

A more subtle issue that could arise is the problem of intra-policy inconsistencies. We have already addressed how inter-policy consistency is addressed through the use of priorities, but intra-policy inconsistency is more difficult to solve. For example, a policy could specify that it requires a particular aspect, yet at the same time specify that it is also incompatible with that same aspect. This would obviously cause issues when the policy came to be applied. To prevent such issues arising, when the policies are first loaded, checks are carried to ensure such relationships have not been specified. If any intra-policy inconsistencies are present, the affected policy is not loaded and the user warned of the problems detected.

3.6.6 Interaction Summary

This subsection has, with examples, covered some of the undesirable interactions that can occur between aspects that are applied dynamically and automatically at run-time. We have addressed how these can be prevented with the specification of the execution order, incompatible aspects, requires aspects and resolution aspects. It is vital that these interactions are prevented to allow the target system and the aspects to operate correctly. Our policy specification allows users to specify a variety of types of relationships that could exist between aspects. These relationships are enforced at run-time by the auto-adaptive framework to ensure that the system operates as intended. It is worth noting that these undesirable interactions are addressed on an aspect-by-aspect basis. Although this may seem to be a limiting factor, it is necessary to deal with conflicts at this level to ensure precise relationships can be defined. If necessary, this approach could be extended to allow conflicts to be defined on a per-policy basis. Furthermore, the use of aspect interaction catalogues [39] could aid the process of both identifying and specifying such conflicts.

3.7 Ensuring Weaving Consistency

When multiple aspects are required to implement a particular adaptation, it is important that transient inconsistencies do not occur. To prevent this, aspects must be woven atomically to ensure that all required aspects are woven simultaneously and there is never a time during the adaptation process when a required aspect is absent.

In our framework, before an adaptation that involves weaving multiple aspects is applied, all incompatible-with relationships are queried to ensure none of these will

be broken. If the adaptation passes these checks then the process of weaving the aspects begins. This initially involves modifying the byte-code of the classes that the aspects affect. After this process the affected classes still require HotSwapping to allow the modified byte-code to come into affect. The affected classes are HotSwapped concurrently to ensure that the woven aspects become active simultaneously and so avoiding the issue of transient inconsistencies.

4. Evaluation

To illustrate the extent to which we have met our objectives, a series of evaluation techniques have been used. A number of performance measurements have been carried out to show the overhead added from using our framework. In order to assess the aims relating to reusability, flexibility, and maintainability of the adaptations, an AO metric is applied to measure these properties. In addition, to demonstrate the effectiveness of aspects and more specifically Framed Aspects on these properties, certain elements used within the framework were re-implemented using OO and XVCL to act as a comparison to our implementation. The target system used in this evaluation is the GUIDE system described in section 3. The cache aspect presented will also be used throughout this section to measure appropriate operations. It should be noted that the actual functionality of the aspect being applied does not affect the overhead experienced.

4.1 Performance Discussion

As discussed earlier, there is an overhead added to systems that are implemented using dAOP. The framework presented allows programmers to adapt system behaviour in various ways and also allows the adaptations themselves to be customised via parameterisation. This extra flexibility inevitably adds an overhead since the data used to create the concrete aspects must be gathered, the aspects need to be generated and finally the generated aspects must be woven dynamically. Also, the execution of the advice that implements the adaptations introduces their own delay. This section will illustrate the extent of these overheads; all tests were run on Java 1.5, on a Pentium M 1.4Ghz processor, 512Mb RAM machine.

4.1.1 Weaving a New Aspect to a New Pointcut

The readings shown in Table 1 present the length of time taken to weave an entirely new aspect to a new pointcut. This action involves generating a concrete aspect (from a Framed Aspect), modifying the byte-code of the classes to implement the pointcut and then HotSwapping these classes in order for their new implementation to take effect. The time required to generate a concrete aspect from a Framed Aspect is dependent on the number of Framed Aspect commands used; as the number of commands increases so to will the required generation time. To keep this time consistent in each of the measurements, the same Framed Aspect will be used. The framed cache aspect described earlier in section 3.5 will be used as we feel this represents a typical

Framed Aspect with a typical number of commands. To obtain a range of measurements, the pointcut is simply altered to affect a wider range of classes.

Table 1. Times for weaving a new aspect to a new pointcut including aspect generation time.

Affected Classes	0	1	2	3	4	5	6	7	8	9	10	11	12
Time (ms)	0.40	22	30	43	61	72	89	103	132	133	147	149	166

The figures show how the time to weave a new aspect to a new pointcut increases as the number of affected classes increases. This can be expected due to the number of operations required to modify the byte-code of *each* of the affected classes. One result to note is that when zero classes are affected, this does not necessarily mean that no classes are affected by the pointcut, but could mean that no classes affected by the pointcut have been loaded yet. The execution time for this value is significantly lower than the rest of the values due to the fact that no class requires byte-code modifications (and consequently no HotSwapping). Instead the byte-code of any affected class will be modified as it is loaded, the overhead of this operation is examined in section 4.1.3.

4.1.2 Weaving a New Aspect to an Existing Pointcut

The results shown in Table 1 required byte-code modifications and HotSwap operations to be executed. However, these operations do not have to be carried out when a new aspect is woven to an *existing* pointcut, thus reducing the total weave time. The times for executing this operation (which again includes the time to generate a concrete aspect from a Framed Aspect) are shown in Table 2.

Table 2. Times for weaving a new aspect to an existing pointcut.

Affected Classes	1	2	3	4	5	6
Time (ms)	0.298	0.307	0.304	0.307	0.311	0.317
Affected Classes	7	8	9	10	11	12
Time (ms)	0.320	0.322	0.327	0.330	0.331	0.330

As can be observed from these results the time required to weave a new aspect to an existing pointcut is significantly lower than when weaving a new aspect to a new pointcut. This reduction is because the hooks to implement the pointcut have already been added. In these tests the aspect is simply generated and then attached to the pointcut. A slight increase in the execution time can be observed as the number of affected classes increases, this is due to checks having to be performed for each affected class to ensure the pointcut is implemented.

It is important to differentiate the results between these two operations as they will be executed with differing frequencies. Under normal operating conditions, this second operation (weaving a new aspect to an existing pointcut) will be executed more frequently since, once a monitoring aspect has been woven, all subsequent effector aspects woven to the same pointcut will reuse the existing pointcut. As can be seen in the examples presented in section 3, it is likely that effector aspects will be woven to the same pointcut as its assocItated monitoring aspect.

4.1.3 Class Loading

Mentioned earlier in section 4.1.1 was the fact that a new pointcut could be added which does not currently affect any loaded classes. It is expected that when a class affected by a pointcut is loaded it would take longer to load than a new class not affected by any pointcuts. To examine this a simple test was performed. This first involved loading a class affected by a pointcut and then loading the same class when not affected by any pointcut (both were loaded through the modified AspectWerkz class-loader). The results are shown in Table 3.

Table 3. Results of loading classes affected by a pointcut compared with ones not affected.

Class Modifcations	No modifications	Pointcut Added
Load Time (ms)	1.6	23

As can be observed it takes a substantially longer length of time to load a class which requires modifications than the time required to load a class which requires no modification. This delay is introduced due to the modification of byte-code to implement the required pointcut and the checks to determine whether this modification is necessary. An interesting point to note is that the time to load and modify a new class is almost identical to the time to modfiy and HotSwap a single class (see Table 1), 23ms compared with 22ms.

4.1.4 Removing an Aspect

The next set of results present the average time required to remove an aspect. This required two sets of readings to be taken. The first measured the time to remove an aspect from the target system and the second when both an aspect *and* a pointcut are removed.

Both of these operations are required for the following reason. When an aspect is removed from the target system it may be inappropriate to remove the pointcut to which it is attached as the pointcut may be still in use by other aspects. However, when the pointcut is no longer in use then it is appropriate to remove the pointcut and avoid any unnecessary future overhead. Tables 4 and 5 show the measured times to execute these operations.

Table 4. Times for removing an aspect from a pointcut.

Affected Classes	1	2	3	4	5	6
Time (ms)	0.285	0.284	0.292	0.295	0.302	0.294
Affected Classes	7	8	9	10	11	12
Time (ms)	0.303	0.306	0.308	0.314	0.315	0.307

Table 5. Times for removing an aspect and a pointcut.

Affected Classes	1	2	3	4	5	6
Time (ms)	8	17	19	20	26	27
Affected Classes	7	8	9	10	11	12
Time (ms)	33	35	37	40	48	45

The same pattern which occured in the add aspect measurements also occurs in the above results. The times of both sets of results increase as the number of affected

classes also increase. Again this is due to operations having to be executed for each affected class. As before, there is a significant difference between the times recorded for removing just an aspect compared to when both an aspect and a pointcut need to be removed. This difference is again due the operation involving byte-code modifications and HotSwapping.

The measured times to remove just an aspect are comparable to the times to add just an aspect, since both of these operations are similar in complexity. However, there is a difference in execution times between removing an aspect and pointcut compared with adding an aspect and pointcut. This can be explained due to the increased processing required to generate and add the byte-code to implement a pointcut. Removing a pointcut involves the simpler task of only removing the byte-code.

4.1.5 Advice Execution

The final set of measurements involve the execution time of a piece of advice within the auto-adaptive framework. This will be the most frequently executed operation and so it is vital that the overhead is at a minimum. The results displayed in Table 6 show the average execution time of an an empty regular Java method, an empty advice executing in AspectWerkz and an empty advice executing in our framework. These results are provided to give a comparison of overhead from our framework.

Table 6. Measured execution times of advice/methods.

Implementation	Regular Java Method	AspectWerkz Advice	Auto-Adaptive Framework Advice
Time (ns)	5	610	920

As can be observed, the execution of AspectWerkz advice is substantially slower than the execution of a regular method. This is due to the reflective information which is gathered when the pointcuts are reached and the increased levels of indirection. There is also a difference between the execution time of advice in our framework and with the execution of advice in the regular version of AspectWerkz. This increase is caused by extra context information being collected by our framework whenever joinpoints are met. This information is then used by the framework to determine what actions (if any) should be executed.

It is always desirable to improve system performance and our framework is no exception. AOP and in particular dAOP are still in their infancy. As the technology gains in maturity, the efficiency of approaches used should improve, and should result in improved performance in our framework. This has already been shown in the major performance improvements in a more recent version of AspectWerkz [40].

4.1.6 Summary

This section has provided performance figures for significant operations performed within our framework. Although some of the figures are relatively high, namely those that involve byte-code modification and HotSwap operations, these are operations that will be performed the least frequently. Conversely, the operations that will be performed the most frequently, such as advice execution or adding a new aspect to an existing pointcut, have a much lower overhead and so should minimise the impact on

the overall overhead. Obviously there will be some overhead from using the framework but this is a trade-off from gaining increased flexibility and adaptive behaviour.

The main factor that affects the scalability of the framework is the pointcut definitions. If a very specific pointcut is specified (i.e. one with no wildcard tokens) then the size of the target system will not affect the overhead experienced. However, if a generalised pointcut (i.e. one with wildcard tokens) is specified then the number of classes present in the target system will affect the overhead (as per the presented results). Furthermore, the pointcut type can affect the performance. For example, weaving an aspect to a call pointcut is more expensive then an execution pointcut. An execution pointcut only requires the target side to be advised whereas a call pointcut requires all classes where the identified method is called to be advised. As such, the performance overhead is entirely customisable by the user. Users can be guided to avoid unnecessary overhead by using the most appropriate pointcut type (e.g. using execution pointcuts instead of call pointcuts) and providing more concise pointcut patterns.

4.2 Software Properties Discussion

In order to be able to quantitatively assess the reusability, maintainability and flexibility of the adaptations being applied to the target system, a suitable metric had to be chosen which can measure the desired properties. A number of metrics have previously been proposed for Object-Oriented methodologies [41, 42] to assess them through empirical studies. However, applying these existing metrics to AOSD implementations is unsuitable due to the way AOSD alters the coupling and cohesion between components [43]. A metric is needed that takes these differences into account, and allows AOSD implementations to be evaluated using empirical studies.

The majority of previous work in developing metrics for AOSD techniques has focused upon qualitative assessment rather than quantitative [44]. To assess the software properties objectively, the AO quantitative assessment framework developed at PUC-Rio was selected [45, 46]. This framework relies on both a quality model and a metric suite. The quality model maps the relationships between the attributes to be evaluated and the various metrics in the metric suite. The metric suite defines various metrics that capture information regarding software attributes such as separation of concerns, coupling, cohesion and size in terms of physical properties of the code. Existing metrics were extended and customised to suit AO software whilst allowing them to still to be applied to OO software. Furthermore, entirely new metrics were also introduced which again can be applied to both AO and OO software to offer comparisons[2] between the two paradigms. These changes and new metrics are detailed in [46]. The list of metrics and the attributes that they map on to are shown in Figure 16 and Table 7 which are taken from [47].

---

[2] An important point to note is that it is difficult to directly compare AO and OO. For example, the notion of a pointcut does not exist in OO. As such, only equivalent features can be directly compared.

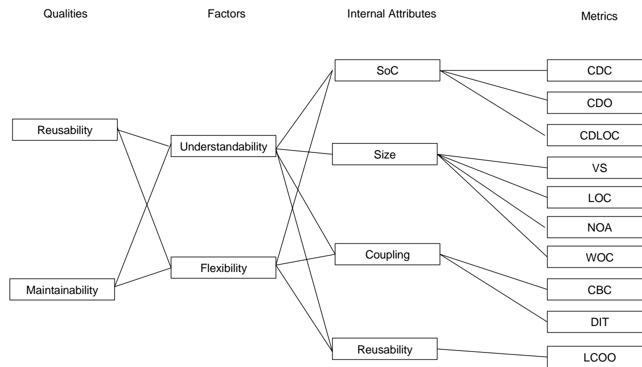

Fig. 16 The quality model used to assess the reusability and maintainability.

Table 7. Lists the metrics used and the attributes they map to.

Attributes	**Metrics**	**Definitions**
Separation of Concerns	Concern Diffusion over Components (CDC)	The number of classes/aspects that a given concern is spread over and the number of class/aspects that access them.
	Concern Diffusion over Operations (CDO)	The number of methods/advices that a given concern is spread over. Additionally, the number of methods/advices that access them are counted as this contributes to the implementation of the concern and so contributes to the diffusion.
	Concern Diffusions over LOC (CDLOC)	The number of transition points for each concern; a transition point is where there is a concern switch within the code.
Coupling	Coupling Between Components (CBC)	The number of classes that are used in attribute declarations, formal parameters, return types, throw declarations, and local variables. Each class used is only counted once
	Depth Inheritance Tree (DIT)	How far down the inheritance tree an aspect/class is.
Cohesion	Lack of Cohesion in Operations (LCOO)	The number of method/advice pairs per class that do not access the same instance variable.
Size	Lines of Code (LOC)	Number of lines of code in a class/aspect.
	Number of Attributes (NOA)	Number of attributes in a class/aspect.
	Weighted Operations per Comonent (WOC)	A sum of the number of methods/advices in each of the classes/aspects and the number of parameters.

4.2.1 Applying the Metrics

Initially, the metrics were applied to the concrete cache aspect (generated from the Framed cache aspect) used earlier in section 3.5. In order to offer a comparison to the AO implementation the cache was re-implemented using OO. In addition to re-implementing the cache using OO, it was also necessary to modify the base-code of the target system to allow the OO cache to be executed correctly. The modifications to the target system are taken into account in the metric readings.

As an extension to comparing the OO and AO implementations we also applied the metrics to the Framed version of the cache aspect. The aim of this was to illustrate the benefits of using Framed Aspects. To allow the metrics to be applicable to the

Framed Aspect code a number of the metrics had to be altered slightly. Below is a summary of the assumptions/changes made:

– Concern Diffusion over Components: include the number of frames that contribute to implement a given concern.
– Coupling Between Components: do not count parameterised types.
– Lines of Code: include all frame commands in the count.
– Weighted Operations per Component: include the number of Framed Aspect commands.

Again, to provide a comparison to the Framed Aspect adaptations we also implemented the adaptations using XVCL. The above assumptions to apply the AO metrics to Framed Aspects are applicable to XVCL.

```
public URL getResource(String res){
  URL value= cache.checkCache(res);
  if(value==null){
    value= server.getDoc(res);
    cache.cacheObject(res,value);
  }
  return value;
}
```

Fig. 17a Changes made to the target system.

```
public URL checkCache(String key){
  if(cache.get(key)==null)
    return null;
  else
    return ((CacheDS)cache.get(key)).getData();
}
public void cacheObject(String key,URL value){
  CacheDS result= new CacheDS(value);
  cache.put(key,result);
}
```

Fig. 17b The methods required to implement caching behaviour when OO is used.

The differences between the AO and OO implementation and the necessary changes made to the target system base-code are highlighted in Figure 17a. Once the changes had been made to the adaptations and base-code, we applied the metrics to both versions. The measurements recorded are shown in Tables 8 and 9.

Table 8. Metric readings for the cache concern implemented using AO.

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_AO				2	0	0	10	1	2
Cache_AO				7	0	0	34	3	2
Cache AO Total	1	4	1	9	0	0	44	4	4

Table 9. Metric readings for the cache concern implemented using OO.

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_OO				3	0	0	16	2	2
Cache_OO				3	0	0	37	3	5
Cache OO Total	2	6	8	6	0	0	53	5	7

As can be seen there is an improvement in all but one of the metrics when using AO to implement the caching feature. The only metric where AO fairs worse is the Coupling Between Components (CBC) metric. This is due to the extra types required for the aspect to be successfully implemented, such as the JoinPoint and MethodRtti types that allow access to the parameters provided to the advised method. Coupling to these types does not affect the aspect's reusability. The two most significant improvements when using AO is found under the Concern Diffusions over LOC (CDLOC) and Weighted Operations per Component (WOC) metrics. The large increase of the CDLOC metric in the OO implementation is due to the changes to the base-code that are necessary to call the cache concern at the appropriate places. As a result, there are concern switches between the original concern and the cache concern. There is also a significant 43% increase in the WOC metric when using OO, since more operations are required to implement the cache when using OO compared to AO, hence the complexity is increased.

To examine the impact of Frames, the metrics were also applied to the Framed implementation of the cache aspect. The results from applying the metrics to this module are shown in Table 10.

Table 10. Metric readings for the cache concern using Framed Aspects.

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_AO				2	0	0	10	1	2
Framed Aspect				5	0	0	36	3	8
Framed AOP Total	1	4	1	7	0	0	46	4	10

As can be observed, there is very little difference between the Framed Aspect implementation (Table 10) and the normal AO implementation (Table 8). However, the two metrics values that do alter are significant ones: WOC and CBC. The WOC increases due to the frame commands used in the Framed Aspect increasing the complexity. However, the CBC metric drops by 22%. This is significant since it proves that reusability and flexibility is improved when using Framed Aspects due to a reduction in coupling between the cache aspect and other components. An important point to make when comparing the separation of concerns metrics of the Framed Aspect implementation to the normal Aspect-Oriented implementation is that there is no increase in these values. This indicates that Framed Aspects maintain the same levels of separation of concerns and so do not detract from the advantages gained from using AOP.

As mentioned earlier, an alternative approach to parameterising/generalising code is XVCL [28]. To illustrate the differences between XVCL and Framed Aspects, we have implemented the cache concern using XVCL. We have applied XVCL to the OO cache implementation, as XVCL has not been specifically designed for use with an AOP approach and so it would be inappropriate to apply XVCL to the AO cache. Table 11 shows the results from applying the metric to the XVCL code.

The most obvious change with the XVCL implementation is the increase in the number of components: a 200% increase compared to the AOP implementations and a 50% increase compared to the standard OO implementation. This increase was due to the introduction of a Cache_Hook module to separate the tests performed on the

cache to determine whether the requested resource has been cached from the base code. This separation allowed the cache concern to be omitted when necessary, like the AO version of the cache.

Table 11. Metrics results from an XVCL OO cache implementation.

Resource	CDC	CDO	CDLOC	CBC	DOIH	LCOO	LOC	NOA	WOC
Base_XVCL				2	0	0	16	1	4
Cache_Hook				2	0	0	15	1	4
Cache_XVCL				1	0	0	39	3	13
XVCL Total	3	6	12	5	0	0	70	5	21

As can be seen, the other separation of concerns metrics (CDO and CDLOC) are much higher compared to any other implementation. Although the Cache_Hook component was introduced to improve separation, this is only partially achieved since XVCL requires commands to be added to components to identify where the concern needs to be inserted. This is achieved in Framed Aspects with the use of pointcuts and the weaving process, so there is a complete separation between the base-code and the concern to be applied. As expected, the metrics show that XVCL does not separate cross-cutting concerns as well as Framed Aspects. However, comparing the Coupling Between Components (CBC) metric with all the previous results, the XVCL implementation reduces the coupling the most. However, the CBC metric found in the AOP and Framed Aspect metrics are artificially high due to them requiring extra components to implement the aspects and to retrieve the parameter values passed to the advised method. The coupling to these components does not reduce the reusability or flexibility. As can be observed, the XVCL implementation significantly increases both the lines of code (a 52% increase compared to Framed Aspects) and weighted operations per component metrics (a 110% increase compared to Framed Aspects). This is due to the increased number of XVCL commands required to integrate the cache concern to the appropriate points in the base-code.

From these metric results we can conclude that Framed Aspects do contribute to improving the reusability and flexibility. However, it is also clear that the maintainability does suffer when using Framed Aspects. The effects of this can be limited by providing appropriate tool support to aid the user manage the Framed Aspects. Such tool support is described in section 6.

The results presented also show that Framed Aspects are a more suitable solution for our needs than XVCL. Framed Aspects improve the separation of concerns, and reduce the complexity and the lines of code required to implement a concern when compared to XVCL.

Additionally, we can also conclude from the adaptations used in our case-studies that AOP is a suitable programming paradigm to implement an auto-adaptive system. In specific cases when AOP may not be the most optimum paradigm, standard HotSwap can be used to allow the target system to be adapted whilst still using OO.

5. Related Work

This section will compare a number of related AOP based auto-adaptive systems with our framework to illustrate the differences and benefits of our implementation. To recall, our overall aim was to implement a framework that can make a system adaptable to suit its current state. This was then divided into a number of sub-aims, a set of which are used as a guide for comparing our work with other related work:

- easily add dynamic adaptive behaviour to existing systems;
- be able to extend/modify adaptations;
- create flexible and reusable adaptations;
- ensure correctness and compatibility between the aspects and the system.

As mentioned in section 2.1, a number of approaches have been implemented that use AOP to encapsulate the *adaptation process*. One of these approaches [19] uses the fractal component model to provide customization. However, since AspectJ is used to identify adaptation points within the system, this prevents new adaptation points being added once the system has been started and limits the dynamic behaviour. Hence, this approach cannot be accurately compared with our work.

A notable piece of related work is JAC which will be briefly described below. However, we focus mainly on QuO and Prose based implementations, since these systems are the most closely related to our work and implement similar features.

5.1 JAC

JAC [32] is a dAOP approach that allows the parameterisation of aspects. JAC enables parameters to be passed into the aspects using configuration files; this allows the aspect to be customised for a particular scenario. Unlike our solution, JAC does not allow complete parameterisation (i.e. Framed Aspects allow any element within the code base to be parameterised) and so the aspects cannot be completely customised.

JAC also provides an interesting feature known as Wrapping Controllers, which allow the programmer to create entities that deal with aspect relationships. This gives the programmer control over when and which aspects get executed, useful to prevent incompatible aspects from being executed together. The specification to implement these relationships is done at a low-level which increases the complexity of this definition process. In comparison, the relationships in our approach are defined at a higher-level and so can ease the definition of such constraints.

5.2 QuO

QuO (Quality Objects) [3] is a domain specific implementation focusing on adapting systems that are implemented using middleware. The aim of this work is to improve Quality of Service (QoS) provided by such systems by applying adaptations to the system. QuO perceives QoS as an aspect and weaves the aspects that alter the QoS properties of the system into the boundary between the middleware and the applica-

tion. The aspects are defined in QuO's own Aspect Specification Language (ASL), which employs a joinpoint model consistent with traditional AOP approaches. QuO also implements its own Contract Definition Language (CDL) to assign conditions to 'regions' of code which are used to specify when advice should be woven.

A limitation of this is that QuO requires the various definition files all to be compiled to generate the adapt-ready application before it is executed. In comparison, our framework allows true dynamic adaptations. Any change made to the policies at run-time will result in aspects being removed and new aspects woven to reflect the changes made. Changes can also be made to the Framed Aspects used; this will affect any future aspects that are generated from the altered Framed Aspects.

QuO's Contract Definition Language specification appears to be highly extensible with few restrictions being placed on the programmer regarding what properties of the system can be monitored. Similarly our approach allows the programmer to extend the attributes monitored, with a number of hardware resources and system attributes available to be monitored by default.

A feature in QuO currently lacking is the ability to customise the code via parameterisation. This would improve the reuse of the code and allow the code to be customised for a particular system/scenario. The use of Framed Aspects allows any part of the aspects used in our framework to be generalised; this is an extremely powerful feature that makes the aspects completely customisable.

QuO allows the order of advice execution to be specified via an aspect precedence definition. However, there is no way of specifying any other more complex relationships that may exist between the aspects used. In contrast, our approach allows dependencies and relationships to be specified in four flexible ways. Incompatible relationships, required relationships, priorities of policies/aspects (either explicitly specified or assigned relative to another policy) and combinations of woven aspects to trigger the weaving of a resolution aspect can all be provided.

5.3 Prose

A large amount of research has been performed using Prose [9] to implement adaptable systems. The majority of this work has involved the adaptation of networked/mobile systems. MIDAS [7] is one example of a system implemented using Prose. The aim of MIDAS is to allow the distribution of Prose aspects to mobile nodes and alter the node behaviour to suit the current location. As Prose is a dAOP technique the aspects can be woven at run-time to achieve dynamic adaptation.

Another system that also relies on Prose is JADDA (Java Adaptive components for Dynamic Distributed Architectures) [16]. Although this solution is component based and mainly achieves dynamic adaptation by altering the architecture by introducing new xADL (XML Architecture Definition Language) [48] files, it still relies on Prose for some functionality to fully configure the system. JADDA allows the basic parameterisation of Prose aspects by placing them into an aspect template and generating concrete aspects when required. String values defined in the xADL file are used to complete the aspect template to generate a concrete aspect.

Jadabs [49] is a dynamic lightweight container for small mobile devices which again makes use of Prose to allow the behaviour of a component to be altered at run-

time. However, when Jadabs is running on resource constrained device, the heavy-weight nature of Prose makes it unsuitable; Nanning [50], a more light-weight but less functional AOP technique, is used in these conditions.

As Prose allows dynamic behaviour, these related systems (MIDAS, JADDA, and Jadabs) all permit new aspects and pointcuts to be introduced dynamically at run-time and have an immediate effect on the target system. This enables new aspects to be created whilst the target system is executing and then woven dynamically.

Prose has been heavily used for adapting distributed system behaviour according to context information based on the location of nodes. This is comparable to our approach in that when nodes change, location aspects are either woven or removed to alter their behaviour. However, our approach is more flexible in that our framework can be applied to any type of system not just distributed systems. Also, multiple attributes can be monitored and combined to form complex requirement/behaviour specifications. Our approach can also be easily extended to monitor other new system attributes by adding to the framed monitoring aspects.

The parameterisation used with Prose/JADDA is very basic with only a small number of elements parameterisable, such as the classname and method name. An additional limitation in the JADDA approach is that conditional compilation is not implemented. Framed Aspects allow any element within the code to be parameterised allowing much more flexibility and customization.

Finally, with the exception of priorities to allow advice ordering, there is no explicit support within Prose for the definition of relationships between aspects. We believe that the ability to define complex relationships is a very important feature to support when implementing an auto-adaptive system. This was highlighted in section 3.6, where relationships were used to handle some of the undesirable interaction problems that can occur in auto-adaptive systems.

6. Future Work

The core implementation of the auto-adaptive framework is now complete with several extensions currently in progress to improve the functionality of the framework, as discussed below.

We believe the use of a GUI tool will help improve the creation/management of the policy files and the Framed Aspects. Currently, the policies are relatively easy to create although it is recognized that they may become difficult to manage when a large number of them are present. Also, errors can be easily introduced when specifying the policies. Tool-based support will aid the management of the policies by presenting the policies in a more manageable format to the user. Additionally, the tool will be able to restrict the values assigned to certain attributes within the policies and perform checks to ensure the policies specified are correct with respect to the target system (i.e. the attributes to monitor exist) prior to run-time.

One of the current problems of Framed Aspects is the readability of the code. As the Framed code has XML frame commands interspersed amongst standard Java code, the readability as a whole is affected. By implementing a GUI these commands could be formatted to improve readability of this combination of Java code and XML.

This tool could also be used to check for any incompatibility problems that may occur when the policy and Framed Aspects are applied to the system.

Currently no semantic and type checking is performed to ensure type-safe aspects are generated from the Framed code. This problem is limited slightly in our approach as the specifications (used to generate the concrete aspects) are created programmatically based on information gathered by the framework. Whilst this reduces the potential for errors, some checks are still desirable to eliminate other problems.

A limitation of the current implementation is that the framework can only control the behaviour of a system local to the machine where the framework is executing. For example, if the behaviour of a node gets adapted in a particular fashion, it may be necessary to adapt other nodes with which it communicates in a similar way to ensure they remain compatible. This type of behaviour would require a distributed policy to be created/modified to ensure a set of nodes all followed certain behaviour and all remained compatible when one of the nodes was adapted.

7. Conclusion

The overall goal of this work has been to develop a flexible framework capable of adding auto-adaptive behaviour to existing systems. The solution we have proposed involves policies, Framed Aspects, dynamic AOP and reflection. By adopting these four technologies in combination, our framework provides good support for various different properties of auto-adaptive systems, properties that are not currently supported in existing systems. The use of Frame technology, reflection and dynamic AOP enables the dynamic weaving of code that has been specifically generated based on the target system's current status. Furthermore, the inclusion of ECA rules in our framework allows the explicit control of *when* and *how* the generated aspects should be woven, a feature not available in current dynamic AOP techniques.

We believe we have achieved our goal and that our framework represents an advance in this research area with respect to other systems. The combination and diverse properties of the various technologies used has allowed a framework to be created that provides highly dynamic and flexible behaviour. Our framework can be applied to any existing system with no implementation changes necessary to the target system. However, an understanding of the existing system is required so that the correct adaptations and behaviour can be implemented.

As shown in our evaluation results, the use of Framed Aspects gives high levels of flexibility and reuse in the adaptations being applied to the system. The aspects generated can be customised to suit a particular scenario due to the generalised aspect code.

As our framework generates the concrete aspects dynamically while the system is running, any changes made to the Framed Aspects will be reflected in the aspects woven to the system. Also, our approach allows changes to be made to the policies at run-time that will also be reflected immediately in the system behaviour. These techniques allow true dynamic behaviour in that the adaptations themselves can be altered and these changes will take immediate effect. By delaying the processing of the Framed Aspects to run-time and generating the concrete aspects using run-time in-

formation, we have illustrated a new novel use of Frame technology. This dynamic behaviour adds an overhead to the execution of the target system but our results show that this overhead is acceptable compared to the benefits gained from such features.

The problem of unanticipated aspect interaction has also been discussed and perhaps is more critical in our work due to the fact that aspects will be woven automatically and dynamically. To help to eliminate this problem, the elements in the policy file allow relationships to be specified. Requires relationships, incompatible-with relationships and priorities can all be defined to facilitate the correct operation of the system.

References

1. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Longtier, J. Irwin. *Aspect-Oriented Programming*. in *Proceedings European Conference on Object-Oriented Programming*. 1997. Jyväskylä, Finland: Springer-Verlag, pp. 220-242.
2. Gupta, D., *A Formal Framework for On-Line Software Version Change.* IEEE Transaction on Software Engineering, 1996. **22**(2): pp. 120-131.
3. Duzan, G., J. Loyall, R. Schantz, R. Shapiro, J. Zinky. *Building Adaptive Distributed Applications with Middleware and Aspects*. in *3rd International Conference on Aspect-Oriented Software Development*. 2004. Lancaster, UK: ACM, pp. 66-73.
4. Falcarin, P., G. Alonso. *Software Architecture Evolution Through Dynamic AOP*. in *1st European Workshop on Software Architectures (EWSA), co-located with ICSE 2004*. 2004. St Andrews, Scotland: Springer-Verlag, pp. 57-73.
5. Greenwood, P., L. Blair. *Policies for an AOP Based Auto-Adaptive Framework*. in *NODe Conference in conjunction with Net.ObjectDays*. 2005. Erfurt, Germany: Lecture Notes in Informatics (LNI), pp. 76-93.
6. Hirschfeld, R., K. Kawamura, *Dynamic Service Adaptation.* ICDCSW, 2004. **2**: pp. 290-297.
7. Popovici, A., T. Gross, G. Alonso. *A Proactive Middleware Platform for Mobile Computing*. in *ACM/IFIP/USENIX International Middleware Conference*. 2003b. Rio de Janeiro, Brazil: ACM, pp. 455-473.
8. Warren, I., *A Model for Dynamic Configuration which Preserves Application Integrity*, in *Computing*. 2000, Lancaster University: Lancaster.
9. Popovici, A., T. Gross, G. Alonso. *Dynamic Weaving for Aspect-Oriented Programming*. in *1st International Conference on Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands: ACM, pp. 141-147.
10. Lupu, E.C., M. Sloman, *Conflicts in Policy Based Distributed Systems Management.* IEEE Transaction on Software Engineering, 1999. **25**(6): pp. 852-869.
11. Basset, P., *Framing Software Reuse - Lessons from Real World*. 1997: Yourdon Press Prentice Hall.
12. Demers, F.-N., J. Malenfant. *Reflection in logic, functional and object-oriented programming: a short comparative study*. in *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*. 1995, pp. 29-38.
13. Hillman, J., I. Warren. *An Open Framework for Dynamic Adaptation*. in *6th International Conference on Software Engineering (ICSE 2004)*. 2004. Edinburgh, Scotland: IEEE Computer Society, pp. 594-603.
14. Moreira, R.S., G.S. Blair, E. Carra. *Supporting Adaptable Distributed Systems with FORMAware*. in *4th International Workshop on Distributed Auto-adaptive Reconfigurable Systems (DARES 2004) - ICDCS 2004*. 2004. Tokyo, Japan, pp. 320-325.
15. Duran-Limon, H., Blair, G.S., Coulson, G., *Adaptive Resource Management in Middleware: A Survey.* IEEE Distributed Systems Online, 2004. **5**(7): pp. 1.
16. Falacarin, P., G. Alonso. *Software Architecture Evolution Through Dynamic AOP*. in *1st European Workshop on Software Architectures (EWSA), co-located with ICSE 2004*. 2004b. St Andrews, Scotland: Springer-Verlag, pp.
17. Blair, G., G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, N. Parlavantzas. *Reflection, Self-Awareness and Self-Healing in OpenORB*. in *Proceedings of the first workshop on Self-healing systems*. 2002. Charleston, South Carolina: ACM, pp. 9-14.

18. Szyperski, C., *Component Software: Beyond Object-Oriented Programming*. 1998: Addison Wesley.
19. David, P., T. Ledoux. *Towards a Framework for Self-Adaptive Component-Based Applications*. in *4th IFIP International Conference on Distributed Applications & Interoperable Systems*. 2003. Paris, France: Springer-Verlag, pp. 1-14.
20. Yang, Z.e.a. *An Aspect-Oriented Approach to Dynamic Adaptation*. in *Proceedings of the first workshop on Self-healing systems*. 2002. Charleston, South Carolina: ACM, pp. 85-92.
21. Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. *An Overview of AspectJ*. in *ECOOP 01*. 2001, pp. 327-353.
22. Keeney, J., V. Cahill. *Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework*. in *Workshop on Policies for Distributed Systems and Networks*. 2003: IEEE Computer Society, pp. 3-14.
23. Diaz, O., A. Jaime, *EXACT: An Extensible Approach to Active Object-Oriented Databases.* VLDB, 1997. **6**(4): pp. 282-295.
24. Filman, R., T. Elrad, S. Clarke, M. Aksit, *Aspect-Oriented Software Development*. 2004: Addison Wesley.
25. Douence, R., G. Muller, M. Sudholt. *A Framework for the Detection and Resolution of Aspect Interactions*. in *GPCE 2002*. 2002. Pittsburgh USA, pp. 173-188.
26. Van Gurp, J., J. Bosch, *Design Erosion: Problems and Causes.* Journal of Systems and Software, 2002. **61**(2): pp. 105-119.
27. Loughran, N., A. Rashid. *Framed Aspects: Supporting Variability and Configurability for AOP*. in *8th International Conference on Software Reuse*. 2004. Madrid, Spain, pp. 127-140.
28. XVCL. *XVCL Homepage*. [Web Site] 2005 [cited 2005; Available from: http://fxvcl.sourceforge.net.
29. Hilsdale, E., J. Hugunin. *Advice Weaving in AspectJ*. in *3rd International Conference on Aspect-Oriented Software Development (AOSD)*. 2004. Lancaster, UK, pp. 26-35.
30. Vasseur, A. *Java Dynamic AOP and Runtime Weaving - How Does AspectWerkz Address It?* in *Dynamic Aspects Workshop held in conjunction with AOSD 2004*. 2004. Lancaster, UK, pp. 135-145.
31. Coulson, G., P. Grace, G. Blair, L. Mathy, D. Duce, C. Cooprt, W. K. Yeung, W. Cai. *Towards a Component-Based Middleware Framework for Configurable and Reconfigurable Grid Computing*. in *Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 2004: IEEE Computer Society, pp. 291-296.
32. Pawlak, R., L. Seinturier, L. Duchien, G. Florin. *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*. in *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. 2001b. Kyoto, Japan: Springer-Verlag, pp. 1-24.
33. Burke, B. *JBoss Tutorial*. in *3rd International Conference on Aspect-Oriented Software Development*. 2004. Lancaster, UK, pp. 1.
34. Greenwood, P., L. Blair. *Using Dynamic AOP to Implement an Autonomic System*. in *Dynamic Aspects Workshop*. 2004. Lancaster, UK, pp. 76-88.
35. Maes, P. *Concepts and Experiments in Computational Reflection*. in *Conference proceedings on Object-oriented programming systems, languages and applications*. 1987. Orlando, USA: ACM, pp. 146-155.
36. Dmitriev, M., *Safe Class and Data Evolution in Large and Longed-Live Java Applications*. 2001, Glasgow.
37. Cheverst, K., N. Davies, K. Mitchell, A. Friday, C. Efstratiou. *Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences*. in *Proceedings of CHI 2000*. 2000. Netherlands, pp. 17-24.
38. Pang, J., L. Blair. *An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Feature*. in *22nd International Conference on Distributed Computing Systems Workshops*. 2002. Vienna, Austria: IEEE Computer Society pp. 445-450.
39. Sanen, F., E. Truyen, B. De Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, S. Clarke, *Study On Interaction Issues*. 2006, Katholieke Universiteit Leuven: Leuven, Belgium.
40. AWBench. *AWBench: AOP Benchmark*. [Web Site] 2004 [cited 2005 April 2005]; Available from: http://docs.codehaus.org/display/AW/AOP+Benchmark.
41. Chidamber, S., C. Kemerer, *A Metrics Suite for Object-Oriented Design.* IEE Transactions on Software Engineering, 1994. **20**(6): pp. 476-493.
42. Fenton, N., S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. 1997: London: PWS.
43. Garcia, A.F., U. Kuleska, S. Sant'Anna, C. Chavez, C. Lucena. *Aspects in Agent-oriented Software Engineering: Lessons Learned*. in *6th International Workshop on Agent-Oriented Software Engineering*. 2005. Utrecht: Springer, pp. 25-37.
44. Hannemann, J., G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2002. Seattle, Washington, USA: ACM, pp. 161-173.

45. Garcia, A.F., *From Objects to Agents: An Aspect-Oriented Approach*, in *Computer Science Department*. 2004, PUC-Rio: Rio de Janeiro. pp. 319.
46. Sant' Anna, C., A. Garcia, C. Chavez, C. Lucena, A. von Staa. *On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework*. in *Brazilian Symposium on Software Engineering*. 2003. Manaus, Brazil, pp. 19-34.
47. Garcia, A., S. Sant'Anna, E. Figueiredo, U. Kuleska, C. Lucena, A. Von Staa. *Modularizing Design Patterns with Aspects: A Quantative Study*. in *4th International Conference on Aspect-Oriented Software Development (AOSD)*. 2005. Chicago, USA, pp. 3-14.
48. Dashofy, E. *A Highly-Extensible XML Based Architecture Description Language*. in *WICSA*. 2001, pp. 103-112.
49. Frei, A., G. Alonso. *A Dynamic Lightweight Platform for Ad-Hoc Infrastructures*. in *Third IEEE International Conference on Pervasive Computing and Communications*. 2005. Kauai Island, Hawaii, pp. 373-382.
50. Nanning. *Nanning AOP*. 2005 [cited 2005 April 2005]; Available from: http://nanning.codehaus.org/.