

Policies for an AOP Based Auto-Adaptive Framework

Philip Greenwood and Lynne Blair

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

Abstract. Dynamic AOP has been identified as a useful technique to implement an auto-adaptive framework. To supplement this policies based upon Event-Condition-Action rules are used to specify when adaptations should be applied to the underlying system. However, for use in auto-adaptive systems it is advantageous if these policies allow certain relationship information to be specified to ensure the system never encounters undesirable interactions and adaptations are applied in a correct manner. This paper highlights the need for these relationships by giving a series of examples followed by a proposed solution to these problems.

Keywords: AOSD, adaptive and reflective systems, generative programming, autonomic computing, dynamic software evolution.

1. Introduction

Since many modern systems often have to operate in environments that are susceptible to change; it is vital that such systems adapt themselves accordingly to suit these changing conditions. Often systems that require this adaptive behaviour are also required to maintain high-levels of availability such as telecommunications systems or other safety-critical systems. For such systems to maintain the desired levels of availability, the adaptations should be applied dynamically while the target system remains online; examples of such systems can be found in [1].

Extensions to these dynamic systems are auto-adaptive systems which allow the desired behaviour of a system to be specified. When the behaviour does not conform to this specification, the most appropriate adaptations are applied to the target system. This allows the system to operate as intended and adapt its behaviour automatically when necessary, examples of such systems are described in [2] [3] [4].

A common problem, that often arises when implementing an auto-adaptive system, is unexpected interactions occurring between the applied adaptations [5]. As the adaptations are applied automatically, the system operator may be unaware a potential problem could have arisen. Also, as the adaptations are applied automatically at run-time, it may already be too late for the operator to fix the problem. What is required is a method that allows the operator to specify *how* the system should be correctly adapted including: adaptations that are compatible/incompatible, the order they should be applied and any resolution code that may be able to correct any undesirable interactions.

In related work performed at Lancaster [6], Aspect-Oriented Programming (AOP) [7] has been identified as an ideal methodology for implementing auto-adaptive sys-

tems. By using AOP, we can cleanly encapsulate the adaptations we wish to apply to the target system and easily specify the points (using pointcuts) we wish to alter. A more recent development in the AOP domain is the introduction of dynamic AOP [8]. With standard static AOP techniques, the weaving process normally takes place at compile time. However, by using dynamic AOP, we can weave the aspects to the target system at run-time without having to take the system off-line; this is obviously beneficial when implementing an auto-adaptive system.

Through research described in [6], a set of common problems between dynamic AOP and auto-adaptive systems were identified. Dynamic AOP suffers from similar problems to auto-adaptive systems, such as undesirable interactions between dynamically applied aspects and execution order dependent aspects [9]. Therefore, it was necessary to find a single solution to a common problem found in the two domains. Our aim was to develop a policy definition and resolution scheme for defining the system behaviour and ensuring the aspects that are woven to adapt the system behaviour are done so without causing any interaction issues. The primary contribution of this work is to allow these relationships to be specified at a high-level to prevent these undesirable interactions from occurring.

The remaining sections of this paper are structured as follows. Section 2 describes the problems encountered in dynamic AOP/auto-adaptive systems in more detail and gives examples of such problems. Section 3 then details our proposed solution to the problems described and revisits the examples described in section 2. Our solution is then evaluated in section 4 by qualitatively assessing our work and describing how it can be applied to other AOP implementations. Finally, section 5 concludes this paper by summarising its findings.

2. Problem Domain

The aim of this section is to highlight some of the potential problems that could occur when adapting system behaviour automatically and dynamically. Examples will be given that illustrate undesirable aspect interactions and show a need for a resolution mechanism to be found if an AOP based auto-adaptive framework is to be successfully implemented. First we will give some background to our research on using dynamic AOP to implement an auto-adaptive framework to give the problem some context.

2.1 An Auto-Adaptive Framework

As mentioned earlier, AOP is a useful tool for encapsulating the changes that need to be applied to an auto-adaptive system. Fine-grained changes can be woven to a target system to alter the system behaviour, for example the behaviour of a method can be altered, or replaced, or additional behaviour could be attached when a field is accessed. The framework developed at Lancaster uses dynamic AOP to weave such changes at run-time. Two other technologies used to implement the framework were Frame Technology [10] in the form of Framed Aspects [11] and ECA based policies [12]. An overview of how these technologies are used together will now be given.

2.1.1 Framed Aspects

In order to increase the reuse of the aspects and increase the flexibility of the framework, the aspects used are parameterised using Frame technology [10] in the form of Framed Aspects [11]¹. Frame technology is commonly used in software product lines to implement the variation points between a family of products using a combination of meta-variables, code templates, conditional compilation and parameterisation. These framed elements of code can then be processed and customised using a given specification to generate compilable concrete code.

Framed Aspects allows *any* part of a class to be parameterised, enabling an entire class to be customised to meet any needs required. Additionally, Framed Aspects allow an alternative form of conditional compilation which allows the most appropriate algorithms to be selected for a given concern.

In previous work Framed Aspects were processed using a pre-compile process using a specification created manually. We have adapted this process in our auto-adaptive framework so that the specification is created automatically and at run-time. The current system properties are collected to determine the values that the specification should assign to the various parameters used in the Framed Aspects. The system status is gathered using a series of probes (implemented as aspects and helper-classes) and reflection. These techniques allow the collection of very detailed information such as the current hardware usage, field values and the signature of methods which aspects are to be woven to. This methodology enables concrete aspect code to be generated at run-time which is specifically suited to the current run-time conditions of the target system. The generated concrete aspects can then be woven dynamically to adapt the system behaviour more precisely than if a generic aspect was applied.

2.1.2 Policy Definition

In order for the concrete aspect code to be generated and woven at the correct times and to the correct points in the target system, a mechanism for defining the desired behaviour is needed. In our auto-adaptive framework, XML policies based on ECA rules are used. These policies allow the programmer to define under what system conditions and which pointcuts a particular Framed Aspect should be processed and then woven to. For example, the current processor usage could be monitored and then used to decide which image compression algorithm is currently the most appropriate to use. A more detailed description of these policies will be given in section 3.

2.2 Aspect Interaction

Analysis of the aspect interaction problem is an area that has not been widely addressed in the AOSD community [13] at the design or implementation level. However, this is an area that we aim to partially address at an implementation level; to resolve potential problems before they occur. It has not been our aim to provide a mechanism by which such interactions can be detected although this would of course be valuable future work.

¹ Non-parameterised aspects can also be used but at the expense of reusability and flexibility.

As any number of aspects could be woven automatically and at arbitrary times during the target systems execution, there is a potential for undesirable aspect interaction which could result in significant consequences. To address this, the proposed framework allows the programmer to specify the certain relationships that may exist between aspects prior to the target systems' execution. This allows the target system to be managed and prevents these undesirable interactions from occurring. From our earlier work [14] a number of relationship issues were discovered that need to be declared prior to the target system being executed:

- The order that aspects are executed could affect their compatibility.
- An aspect could require the presence of another aspect for it to operate correctly.
- An aspect may be incompatible with another, so the presence of both could cause undesired interactions.
- A resolution aspect [15] may resolve interaction problems between a set of aspects; this should be woven when that combination of aspects is detected.

A solution to these problems will be presented in section 3; the rest of this section will consist of a set of examples that illustrate the above problems to give a better understanding.

2.3 Problem Examples

The examples described in this section could potentially occur in any system implemented using dynamic AOP, not just an auto-adaptive system [9]. In these examples, the aspects are woven automatically to optimize the behaviour as the condition of the target system alters. Although the highlighted issues are more common when the aspects are applied automatically, the problems are still relevant to systems implemented using manual dynamic AOP. The examples are implemented using the Java based dynamic AOP framework AspectWerkz [16]. To reduce the complexity of the examples the aspects shown are not framed aspects.

2.3.1 Incorrect Execution Order

Suppose the conditions of the target system alter in such a way that it becomes necessary to weave (automatically) a cache and authentication aspect to the same pointcut. It is under these conditions that our first interaction issue arises. Figure 1 shows the base-class and the method that the aspects described in these examples are woven to.

```
public class Client{
    public Doc getDoc(String resource){
        //implementation
    }
}
```

Fig. 1. Shows the class Client and the method getDoc which the aspects will be woven to.

The implementation of the `getDoc` method retrieves the specified resource from a server and then returns the `Doc` object to the callee method. This operation could be susceptible to fluctuations in execution time due to potential delays on the network. Figure 2 shows the concrete cache aspect that will be woven to the pointcut `execu-`

tion(Doc Client.getDoc(String)). The aim of this aspect is to reduce the number of calls needing to be made to the remote server by locally caching the results of the requests and so reducing the observed fluctuations in the execution time.

```
public class Cache{
    private HashTable cache= new HashTable();

    public Object cache(final JoinPoint jp) throws Throwable{
        Object param= jp.getParameterValue()[0];
        if(cache.get(param)!=null)
            return cache.get(param);
        else{
            Object result= jp.proceed();
            cache.put(param,result);
            return result;
        }
    }
}
```

Fig. 2. A cache aspect implemented using AspectWerkz.

The around advice shown in figure 2 is executed whenever the `getDoc` method is executed in the `Client` class. The cache aspect intercepts any calls to the `getDoc` method, extracts the resource parameter being passed and checks the cache for a cached value for that resource. If a cached value exists, this value is immediately returned and the advised `getDoc` method is not called. Otherwise the `getDoc` method is called using the `proceed` method and the value returned is cached; the retrieved resource is then returned to the caller method.

```
public class Authentication{
    public Object auth(final JoinPoint jp) throws Throwable{
        Object param= jp.getParameterValue()[0];
        String username= User.getUsername();
        if(checkPermission(username,param))
            return jp.proceed();
        else{
            throw new InvalidOperationException("user does not have permission");
        }
    }
    public boolean checkPermission(String username,String resource){
        //impl
    }
}
```

Fig. 3. An authentication aspect that checks the current user has permission to access the desired resource.

Suppose after the cache aspect has been woven and is operating normally, a change in the target system context occurs and an authentication aspect is needed to implement a basic security feature, e.g. to ensure that the current user has permission to access the desired resource. Figure 3 shows a basic concrete authentication aspect, which is woven around the same pointcut as the cache, to implement this functionality.

The authentication aspect extracts the parameter that identifies the required resource passed to the `getDoc` method and then also retrieves the username of the person currently using the system. By using the `checkPermission` method, the aspect can verify that the user has permission to access the resource. If the user does have permission, the system carries on operating normally with the advice calling the `proceed` method to allow access to the desired resource. However, if the user does not have permission, an exception is thrown and the user is not permitted to access the resource.

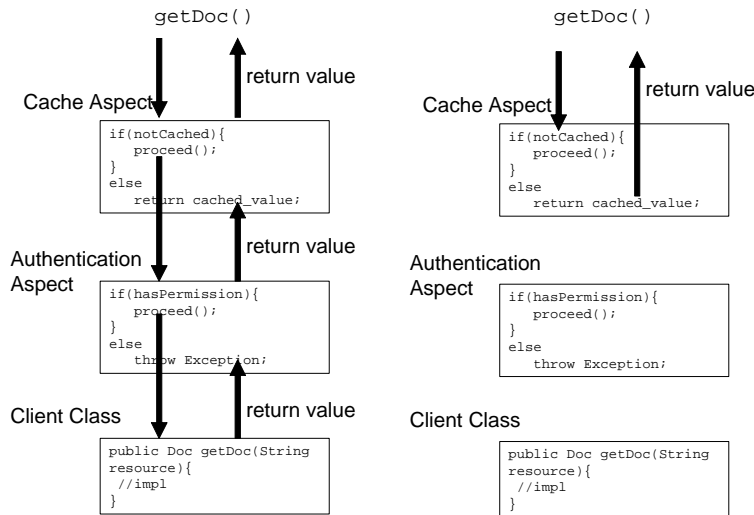


Fig. 4. The advice execution chain: when the resource is not cached (left) the aspects operate correctly; when the resource is cached (right) the authentication aspect will never be called.

A potential interaction problem could occur under these circumstances. For example, we described earlier that when the `proceed` method is called the system continues by executing the advised method. However, if multiple aspects are woven to the same pointcut then the next advice in the chain is executed instead until the final advice is reached, after which the advised method is executed. If any advice in the chain does not call the `proceed` method then the chain is broken; any remaining advice in the chain and the advised method are not executed.

In this example, if the cache aspect is executed first then a potential problem could occur. If the resource to be retrieved has been cached (possibly by a different user), the next user could access a resource they do not have permission for. As the resource has been cached, the `proceed` method will not be called; this will result in the authentication advice not being executed. In turn, the user will not be authenticated and the cached object returned regardless of the user's permission; this behaviour is illustrated in figure 4. To avoid this, a relationship needs to be specified that prevents the cache aspect from being executed before the authentication aspect.

2.3.2 Requires Aspect

Continuing on from this scenario, changes to the system requirements could be made at run-time. For example, a logging requirement could be added to the authentication feature that requires a record to be kept of all resources each user has accessed. This logging requirement can easily be implemented as an aspect and so there exists a *requires* relationship between the authentication aspect and the logging aspect. This requires relationship means that whenever and wherever the authentication aspect is woven the logging aspect should also be woven. To implement this relationship using manual dynamic AOP, the programmer must be aware of it and initiate the weaving of the logging aspect manually. Additionally it is difficult for the programmer to apply this relationship retrospectively and weave the logging aspects to the pointcuts where

the authentication aspect is already woven. This is not an ideal situation; the programmer could easily forget to weave the necessary aspect or miss some pointcuts where the aspect is required.

These problems could be prevented by extending the authentication aspect to include code to implement the logging requirement but this limits reuse and breaks the separation of concerns.

2.3.3 Incompatible Aspects

Conversely to the requires relationship, if two aspects are incompatible, one aspect could rely on the absence of another aspect for it to operate correctly. They could be incompatible in two ways; they could be incompatible in the sense that the actual code is incompatible or the behaviour of one could be incompatible with the requirements of the other. A situation where this type of relationship could occur is if an encryption aspect is to be introduced to add privacy to the system while the cache aspect is still present. The cache and encryption aspect are incompatible in this example for two reasons. Firstly, it would be inappropriate to cache objects that have been encrypted due to the security risk of some third-party using the cached objects to crack the encryption algorithm used. Secondly, the keys used to encrypt objects may expire, so the cached objects will become useless. Altering the execution order of these aspects would still not solve these incompatibility issues; if the cache was executed first the cached objects would not be encrypted and so break the requirements of the encryption aspect.

From this description, it is obvious that the cache aspect should be removed whenever the encryption aspect is woven to prevent both performance and security problems. Without the framework's proposed relationships, the programmer must be aware that the cache aspect is present and remove it manually before the encryption aspect can be woven.

2.3.4 Resolution Aspect

Alternatively, a situation could exist where a set of aspects that are normally incompatible could be made compatible by weaving a resolution aspect [17]. Suppose the state of the system is as follows: the authentication aspect has been woven to check permissions, a logging aspect has been woven to fulfill the requirements of the authentication and an encryption aspect has been woven to add privacy to the system. A potential problem exists due to the logging aspect and the encryption aspect both being woven to the target system. The encryption aspect was introduced to ensure privacy. However, the logging aspect stores the resources accessed in plain-text as well as the username of the person who accessed the resource – this clearly does not ensure the privacy of the users and so breaks the requirements of the encryption aspect. This interaction problem can be easily fixed by weaving a resolution aspect that will encrypt the contents of the log file when both the encryption and logging aspects are present in the target system. Again, when using manual dynamic AOP, the operator must be aware of this relationship and weave the appropriate aspects when necessary.

2.4 Multiple Problems

The examples described above do not take into account the fact that a number of the highlighted issues could occur simultaneously. For example, an aspect to be woven could require another aspect to be present in the system as well as a being incompatible with another. It is important that the proposed policy definition can handle any combination of the issues mentioned so the exact system behaviour can be defined.

When a number of relationships are defined, it is important that no incompatibilities exist within each individual policy defined (intra-policy compatibility). For example, a programmer could define a policy which requires Aspect A to be woven but within the same policy they could also define it is incompatible with Aspect A. Checks should be performed to ensure that the policies defined will not exclude themselves due to simple errors like the one described.

Another issue that needs to be resolved involves determining the precedence of each policy and which relationship specification should be applied over a conflicting one (inter-policy compatibility). Two policies could conflict, with one specifying that it requires a particular aspect whereas another policy specifying that it is incompatible with the same aspect. A method to resolve these types of conflicts and decide which policy should be fulfilled is necessary to allow the system to operate correctly.

Following on from the previous example where the requirements of a pair of policies cannot both be fulfilled simultaneously, the system should take consistent actions regarding how it handles the policy that has been over-ridden. For example, should the system ignore that policy for the remaining execution time of the system, should it implement the parts of the policy that are compatible with the conflicting policy, or should the policy be ignored until the conditions change so that it can be implemented fully without any conflicts? Also, a situation could occur when one aspect is specified as being incompatible with a set of other aspects, but can still be woven as the incompatible aspects have not yet been woven. Depending on priorities, this aspect should be removed when any of the aspects with which it is incompatible are woven.

Finally, as the framework aims to allow dynamic alterations to the behaviour of the target system, the policies themselves should also be able to be altered at run-time. Any changes made to the policies at run-time should have an immediate affect on the target system, i.e. any newly specified relationship should be retrospectively applied (e.g. any required aspect woven, any incompatible aspect removed, the aspect execution order adjusted accordingly or any necessary resolution aspect woven). For this to be achieved, the policies should be able to be altered easily and whole new policies or policy elements should be easily inserted/removed. Inserting a whole new policy should not require massive disruption to previously defined policies.

2.5 Summary

In this section we have given a brief description of our implementation to achieve auto-adaptive behaviour and highlighted several interaction issues that could arise when aspects are applied automatically. Note that although this work has focused on auto-adaptive behaviour, the issues highlighted are still relevant to systems where aspects are applied manually.

3. Implementation

Our implementation of an auto-adaptive framework relies on policies based on Event-Condition-Action (ECA) rules. In our framework, we class an event as some joinpoint being reached, a condition as a test applied to some system attribute and an action as some aspect(s) either woven or removed. Figure 5 shows two basic policies that illustrate our initial format for specifying system behaviour. Although the policies examples shown here are statically defined, they can be altered at run-time while the underlying system is being executed. Any changes made to the policies will be detected by the framework, which will then apply the modified policies accordingly.

```
<policy name="CachePolicy">
  <condition attribute="ExecutionTime" condition="more-than" value="2000"/>
  <points pointcut="Doc Client.getDoc(String)"/>
  <aspect-type name="Cache" advice="cache"/>
</policy>
<policy name="AuthenticationPolicy">
  <condition attributes="fieldvalue" condition="equals" value="secure"/>
  <points pointcut="String Client.mode"/>
  <aspect-type name="Authentication" advice="auth" pointcut="Doc Client.getDoc(String)"/>
</policy>
```

Fig. 5. Shows a basic policy for our auto-adaptive framework.

The CachePolicy above specifies when the execution time of any method with the signature Doc Client.getDoc(String) goes above 2000ms a Cache aspect should be woven around the methods matching that signature. From this policy, a monitor is generated that observes the execution time of the specified method(s) and detects when the conditions occur. Once the condition has occurred, the actions associated to the policy are executed; in this policy a cache aspect is woven. Similarly a monitor is deployed to implement AuthenticationPolicy; a monitor is generated to detect when the String field Client.mode is set to “secure”. When this occurs a concrete authentication aspect is generated. This is then woven around methods with the signature Doc Client.getDoc(String).

3.1 Execution Order

Unfortunately, a problem regarding the execution order highlighted earlier (section 2.3.1) will occur with our two policies. Neither of the policies specifies that a relationship exists between the two. As discussed earlier, the authentication aspect must be executed before the cache aspect. Currently the order of execution is dependent on the order the aspects are woven or more precisely which condition being monitored occurs first. This is undesirable as there is no way to predict which will occur first. One solution to this problem is to add an extra field to the policies to assign a rank to each policy; representing the order the aspects should be executed.

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

Fig. 6. Policies with the added rank field.

With this added field we can ensure that the authentication aspect will be executed before the cache aspect. However, a problem with this solution is that it is static and

may require large disruption when inserting new policies. For example, suppose we wish to add an aspect that should be executed before the cache and authentication aspect, we have to manually alter the rank of both the `AuthenticationPolicy` and `CachePolicy` so that every aspect still had a unique rank. To address this, in addition to giving each policy a definite rank, we allow ranks to be assigned that are relative to other policies. Figure 7 shows the syntax we have used to implement this feature.

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

Fig. 7. Field to give a rank relative to an existing policy.

The policy shown in figure 7 specifies that the aspects woven by the authentication policy should be executed immediately *before* the cache aspects. Conversely if the aspects should be executed *after* a policy the `++` should be replaced with `--`.

3.2 Required and Incompatible Aspects

Based on our earlier discussion (section 2.3), the policy definition should also allow the specification of dependencies between aspects (one aspect requires the presence of another to operate correctly). Similarly, it should also allow the specification of incompatibilities (one aspect will fail to work correctly in the presence of another). The policies were extended to include the required information as shown in figure 8.

```
<aspect-type name="Authentication" advice="auth" pointcut="Doc Client.getDoc(String)">
  <requires-aspect name="Logging" advice="log"/>
</aspect-type>
<aspect-type name="Encryption" advice="encrypt" pointcut="Doc Client.getDoc(String)">
  <incompatible-with name="Cache"/>
</aspect-type>
```

Fig. 8. Policy extensions to include information regarding the required and incompatible aspects.

As can be seen, for required aspects, the programmer can simply list all the aspects that are required for this policy to be implemented successfully. Similarly, a list of incompatible aspects can also be given if required. Both of these fields have an optional pointcut parameter. When used with the `requires-aspect` field this specifies the pointcuts to which the required aspect will be woven, useful if each of the aspects need to be woven at different pointcuts.

The pointcuts parameter when used in the `incompatible-with` field has a different use. This allows the programmer to specify the pointcut pattern where the incompatible aspect has to be woven to actually make it incompatible. This allows the programmer to specify interaction points. For example, a pair of aspects could only be incompatible if they interact directly at a certain pointcut. If the aspects are incompatible with each other regardless of where they interact, this parameter can be assigned the value `*`.

3.3 Resolution Aspect

To weave the resolution aspects at the correct time, the conditions (the combination of aspects) under which they are needed have to be specified. Instead of adding another field to the policy definition we decided to expand the types of attributes used in the `weave-conditions` field of the policies. Using this new attribute the programmer

can define the combination of aspects that will require the resolution aspect to be woven. Figure 9 shows the policy definition to weave the resolution aspect to fix the interaction problem between the logging and encryption aspects (section 2.3.4).

The `ResolutionPolicy` in figure 9 specifies that both the `LoggingAspect` and `EncryptionAspect` have to be woven for the `LoggingResolution` aspect to be needed. Note that the conjunction is achieved by the nested conditions. The rank value is specified so that the `LoggingResolution` is executed *after* any aspect associated with the `AuthenticationPolicy`, so as to encrypt the newly added contents of the log file.

```
<policy name="ResolutionPolicy">
  <weave-condition attribute="AspectWoven" condition="equals" value="LoggingAspect">
    <weave-condition attribute="AspectWoven" condition="equals" value="EncryptionAspect"/>
  </weave-condition>
  <aspect-type name="LoggingResolution" pointcut="execution(Document Client.get(String))"/>
  <rank value="--AuthenticationPolicy"/>
</policy>
```

Fig. 9. Policy definition to weave a resolution aspect.

3.4 Run-Time Checks

In addition to implementing these policy extensions, a number of run-time checks have also been created to ensure the relationships defined are correctly enforced. Each time there is a request to reconfigure the system (aspects woven or removed) this series of checks are performed to ensure no relationships that relate to the current reconfiguration will be broken.

The first check performed is to ensure that all the required aspects to successfully implement a policy/concern are woven. If some required aspects are absent when the weave conditions are triggered then they are woven to the specified pointcut. As aspects can be removed when they are no longer needed, these checks are also performed before any aspect is removed in case another policy still requires that particular aspect. If a policy is detected that still requires the aspect, then the remove operation for that aspect is modified and the pointcut the aspect is attached to is altered to only include the pointcuts of the policies which still require the aspect. Incidentally, the order in which the required-aspects associated with a single policy should be executed is determined by the order that they are listed within that policy. If the programmer wishes to alter this order they can simply rearrange the listings and this will be detected by the framework and will have an immediate effect. Similar checks are performed for the incompatible-aspects; if the presence of the specified aspects is not detected at the specified pointcuts then weaving process goes ahead.

These two checks will operate as intended unless there are conflicts to resolve between policies (inter-policy). For example one policy could require an aspect whereas another policy could be incompatible with it. These kinds of conflicts are only triggered when the aspects associated with both policies are to be woven to the system, i.e. two conflicting policies could co-exist if their associated aspects are never woven to the system at the same time. The way we have decided to solve this is by using the ranks as described earlier (section 3.1). When a conflicting set of relationships are detected, the rank of the policy is used to decide which policy and which set of relationships should have precedence. To prevent incompatibility problems the policy that is over-ridden will have *all* of its aspects removed to prevent them from operating in

conditions that they are incompatible with, this remove operation is described in more detail in [14]. However, these aspects can be re-woven at a later time if the state of the system changes and the incompatible aspects are removed. When an adaptation is taking place, if at any stage the relationship requirements cannot be met, the adaptation is abandoned and any changes (aspects woven or removed) undone.

To ensure that the execution order of aspects is correct, the first time an advice chain is executed the aspects are ordered to reflect their ranks. This chain is then cached for future use, if a rank of a policy affecting that chain is altered the cached chain is invalidated, causing it to be regenerated the next time it is executed.

Finally, static checks are also performed before the policies are loaded, to ensure conflicts between required and incompatible aspects are not specified within the same policy (intra-policy). If any conflicts are detected the programmer is warned and the conflicting policies are not loaded.

4. Evaluation

Evaluation of the general framework relies on the performance of the underlying dynamic AOP approach used (AspectWerkz). Benchmark figures for AspectWerkz can be found in [18]. The evaluation of our framework, when compared to these figures, is largely consistent and summarised in the table below. The middle column shows the fixed overhead that occurs on each execution of related operation, while the end column shows how the execution time increases as the number of affected elements increases. The tests were run on Java 1.5, Pentium M 1.4Ghz, 512Mb RAM.

Table 1. Performance figures summary.

Operation	Fixed Overhead (ms)	Increase (ms)
Weaving a new aspect to a new pointcut.	0.4	13 (per affected class)
Weaving a new aspect to an existing pointcut	0.3	0 (per affected class)
Removing an aspect	0.3	0 (per affected class)
Removing an aspect and pointcut	0.4	3.3 (per affected class)
Empty Method Execution	0	0.000005 (per method)
Empty Around Advice Execution (AspectWerkz)	0	0.00071 (per advice)
Empty Around Advice Execution (our framework)	0	0.00092 (per advice)

However, the main aim of this paper has been to present the use of *policies* within this framework for the specification of particular relationships. Evaluation of these is inevitable more subjective. To demonstrate the benefits of our implementation, we have identified five properties to assess: separation of concerns, reuse, adaptability, support for interaction definition and portability.

4.1 Separation of Concerns

Several elements of this framework need to be well separated to ensure good reuse and ease the implementation stage. The adaptations to be applied to the underlying system are already well encapsulated with use of AOP and frames.

In addition to this, it is also important that there is good separation between the adaptations/policies and also between each of the policies themselves. If the policies and adaptations were not separated then it would complicate the task of creating the adapta-

tions as they would have the relationship specification intertwined. Our approach allows all the relationship definitions to be separated away from the adaptation code, thus removing any system specific information. Compared to other relationship specification techniques found in other AOP languages, such as the precedence keyword used in AspectJ [19] which is tightly integrated into the aspect definition, our relationship definitions are very well separated. This is useful as the relationships and aspect are easier to edit, and the relationships are easier to understand due to them not being tangled within the adaptation definition.

4.2 Reuse

Reuse is an important property in any type of system to reduce the effort to develop a system and to reduce the number of errors present. By allowing the relationships/behaviour of a target system to be specified separately from the affected adaptations the reuse is improved. The majority of the relationships/behaviour will be specific to a target system, so the reuse of the adaptations is improved.

Currently, the policies do not allow the reuse of other previously defined elements, such as allowing certain conditions to be specified once and applied to a number of separate policies. One way to improve the reuse would be to define certain relationships as meta-data within the adaptations themselves. This would still allow the relationships to be separate from the code to some degree and would save having to re-define relationships that must be present every time a particular adaptation is used. This is future work which we plan to perform.

4.3 Adaptability

As the overall aim of this work is to develop an auto-adaptive framework, the policies themselves should also be highly adaptable to reflect the changes in requirements that could occur while the system is executing. We allow for any changes to be made to the policies at run-time and any changes made will be applied immediately to the framework, so the system is also very adaptable in this respect. To ensure no undesirable interactions occur, the framework checks the modified policies for any intra-policy incompatibilities and then determines any other policy that is affected by the changes made. These checks allow the framework determine in what ways the system needs altering. For example, a policy could be modified to require a certain aspect, yet another policy could have specified that it is incompatible with the same aspect. If both policies are active, i.e. their weave conditions have been triggered, the framework will have to determine the best course of action which will be dependent on the ranks of the affected policies.

4.4 Support for Interaction Definition

In this paper we have described in detail the types of relationships that can be defined and how they should be specified. Currently, in the AOP domain there is no technique that allows aspect relationships to be defined in this way. Using our framework the programmer can explicitly define the relationships separately from the aspect definitions. JAC [9] does allow wrapper classes to be defined that can be used to define aspect precedence/relationships programmatically (described in more detail in section

4.5). Other examples of auto-adaptive systems that are implemented using AOP such as MIDAS [20] and QuO [3] do not allow relationships between the adaptations to be specified. However, QuO does allow the precedence between the aspects to be specified. Our approach allows the priority to be specified by using the rank field; this can also be modified at run-time to alter the execution order of the aspects while the system is still executing. The relationship elements (rank, required aspects and incompatible aspects) can be combined to allow extremely accurate relationships between policies to be defined to ensure the behaviour of the system will be correct.

4.5 Portability to Other Dynamic AOP Techniques

A wide variety of dynamic AOP techniques now exist [21]. Although our solution currently only focuses on solving the interaction issues described for AspectWerkz, these issues will still occur in other AOP implementations. The aim of this section is to describe how our policy definitions could be reused in other dynamic AOP implementations.

The actual policy specification can remain the same between implementations; only the methods used to interpret the policies and apply them to the AOP technique need to be altered. One prerequisite of any AOP technique is that information regarding the woven aspects cannot be lost during the weave process. For example a technique like AspectJ [19] would be unsuitable, as information regarding the aspects can be lost when the aspects are woven, such as pointcut definitions etc. We have selected two dynamic AOP techniques to apply our policy definition to, JAC [9] and Prose [20].

4.5.1 Prose

Prose is a dynamic AOP implementation for Java which runs on top of a modified JVM to allow aspects to be woven dynamically and efficiently. Prose, like AspectWerkz, allow the retrieval of the currently woven aspects, which is vital for checking whether any incompatible aspects have been woven or whether it is necessary to weave a required aspect.

However, a complication is encountered in Prose when the active pointcuts need to be retrieved so that an assessment can be made as to whether a direct interaction occurs between a pair of aspects. Using the policies to retrieve the pointcuts may not yield accurate information as not all the specified pointcuts will be active. Unlike AspectWerkz, Prose does not have an aspect manager that allows the querying of the currently active pointcuts. Instead, an object is created programmatically to define the signature of the desired pointcut, which can be easily retrieved. However, the pointcuts can be filtered further at run-time by using the advice method signature to restrict the types of objects the advice should be applied to and the parameter types the advised method must accept. This extra filtering information cannot be retrieved using some convenient retrieval method; instead reflection would have to be used.

In comparison reordering the aspects is much simpler when using Prose due to the fact that a precise priority can be assigned to an aspect using the inherited `setPriority` method. In addition to these definite priorities our framework also allows relative priorities to be assigned to aspects/policies. As these are converted to definite priorities at run-time they remain compatible with the Prose.

4.5.2 JAC

JAC (Java Aspect Components) is another dynamic AOP implementation for Java. The issues regarding aspect interaction and composition are some of the key areas which JAC aims to focus upon. As such JAC has a specific module that aims to prevent the problems that can arise from these issues. These modules, called wrapping controllers (WC) are executed in-between advice executions and ensure the desired relationships are implemented. They can prevent aspects being executed, weave aspects, or reorder aspects to ensure the aspects exhibit the correct behaviour.

The code to implement the relationships is done so at a low-level can be complex and time consuming to create. In comparison our policy definition is relatively straightforward to create relationships that address the similar relationships issues due to them being defined at a higher level. To allow our policy definition to be used with JAC, a method of converting our policies to JAC WC code is required. One way this can be achieved is by using Framed Aspects, which were briefly described earlier in section 2.1.1. The WC implementation can be generalised and parameterised in such a way that it can be highly reusable and then in conjunction with the policy definition can be used to generate the concrete WC to implement the required relationships.

5. Conclusion

In our previous work, an auto-adaptive framework had been developed [14] that used dynamic AOP to encapsulate and apply adaptations to an underlying system at run-time. This implementation relied on the use of policies to define where and when these adaptations should be applied to the system. This paper has proposed an extension to this work highlighting the need for policies to take into account additional information regarding the system operation.

This paper has highlighted several examples where adaptations could be made and the system left in a state where undesirable interactions could occur. A solution to these problems, where relationships between these adaptations could be specified, has been presented.

We proposed a set of extensions to our initial policy definition to include extra fields that allow relationships to be specified between the adaptations. The policies now allow a *rank* to be given to the policy which is associated with any adaptations that the policy applies to the system. This rank is used to determine the execution order of the adaptations and also used to resolve any conflicts between policies.

A list of *required aspects* can be given that ensures all the aspects needed for the adaptation to be correctly implemented are woven. Each of these aspects can be woven to different pointcuts if necessary. Conversely, a list of *incompatible aspects* can also be specified to list a set of aspects an adaptation is incompatible with. Again pointcuts can be associated with these incompatible aspects to define the interaction points that may make them incompatible with the adaptation. It is also possible to specify the circumstances when a *resolution aspect* can be woven to allow a set of aspects normally incompatible to operate together correctly. A *resolution mechanism* has been described that ensures that none of the relationships specified will get broken and the policy with the highest rank having precedence over the others.

The evaluation has discussed how the framework, and specifically the use of policies, address five important characteristics of system implemented using dynamic AOP. Although basic relationship and precedence mechanisms are available in other AOP techniques none are as explicit or as comprehensive as the ones we have proposed. The techniques described in this paper have been mainly created for use in an auto-adaptive environment the principles are also applicable to AOP systems (especially dynamic AOP systems) as a whole. Although our framework does not currently provide a mechanism to detect these undesirable interactions it does provide the programmer a way to specify them and prevent them from occurring in a high-level way, a feature not currently present in other similar systems or techniques.

References

1. Hillman, J., I. Warren. *An Open Framework for Dynamic Adaptation*. in *ICSE 04*. 2004. Edinburgh, Scotland.
2. David, P., T. Ledoux. *Towards a Framework for Self-Adaptive Component-Based Applications*. in *DAIS 03*. 2003. Paris, France.
3. Duzan, G., J. Loyall, R. Schantz, R. Shapiro, J. Zinky. *Building Adaptive Distributed Applications with Middleware and Aspects*. in *AOSD 2004*. 2004. Lancaster, UK.
4. Falacarín, P., G. Alonso. *Software Architecture Evolution Through Dynamic AOP*. in *EWSA 04*. 2004. St Andrews, Scotland.
5. Gupta, D., *A Formal Framework for On-Line Software Version Change*. *IEEE Transaction on Software Engineering*, 1996. **22**(2): p. 120-131.
6. Greenwood, P., L. Blair. *Using Dynamic AOP to Implement an Autonomic System*. in *Dynamic Aspects Workshop*. 2004. Lancaster, UK.
7. Kiczales, et al. *Aspect-Oriented Programming*. in *ECOOP 97*. 1997. Jyväskylä, Finland.
8. Baker, J., W. Hsieh. *Runtime Aspect Weaving Through Metaprogramming*. in *AOSD*. 2002. Enschede, The Netherlands.
9. Pawlak, R., L. Seinturier, L. Duchien, G. Florin. *Dynamic Wrappers: Handling the Composition Issues with JAC*. in *TOOLS-USA*. 2001. Santa Monica, USA.
10. Basset, P., *Framing Software Reuse - Lessons from Real World*. 1997: Prentice Hall.
11. Loughran, N., A. Rashid. *Framed Aspects: Supporting Variability and Configurability for AOP*. in *ICSR 04*. 2004. Madrid, Spain.
12. Diaz, O., A. Jaime, *EXACT: An Extensible Approach to Active Object-Oriented Databases*. *VLDB*, 1997. **6**(4): p. 282-295.
13. Filman, R., T. Elrad, S. Clarke, M. Aksit, *Aspect-Oriented Software Development*. 2004: Addison Wesley.
14. Greenwood, P., L. Blair, N. Loughran, A. Rashid. *Dynamic Framed Aspects for Autonomic Systems*. in *RAM-SE Workshop*. 2004. Oslo, Norway.
15. Blair, L., J. Pang. *Aspect-Oriented Solutions to Feature Interaction Concerns Using AspectJ*. in *Feature Interactions in Telecommunications and Software Systems*. 2003. Ottawa, Canada.
16. Vasseur, A. *Java Dynamic AOP and Runtime Weaving - How Does AspectWerkz Address It?* in *Dynamic Aspects Workshop*. 2004. Lancaster, UK.
17. Pang, J., L. Blair. *An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Feature*. in *ICDCS Workshop*. 2002.
18. AwBench: AOP Benchmark, <http://docs.codehaus.org/display/AW/AOP+Benchmark>, 2004.
19. Kiczales, G., et al. *An Overview of AspectJ*. in *ECOOP 01*. 2001.
20. Popovici, A., T. Gross, G. Alonso. *AOP Support for Mobile Systems*. in *Advanced Separation of Concerns in Object-Oriented Systems*. 2001. Tampa, USA.
21. Chitchyan, R., I. Sommerville. *Comparing Dynamic AO Systems*. in *Dynamic Aspects Workshop*. 2004.