

Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System

Philip Greenwood
Lynne Blair

Computing Department, Lancaster University, Lancaster, UK
p.greenwood@lancaster.ac.uk | lb@comp.lancs.ac.uk

Abstract

As computational complexity of systems continues to increase, the amount of maintenance required to keep them operational will also increase. Autonomic systems have the aim of reducing the amount of maintenance required by performing certain levels of maintenance themselves. This paper outlines the case of using dynamic AOP to implement such a system. The benefits and issues arising from using dynamic AOP will be looked at and discussed.

1 Introduction

Computational complexity is related to the amount of time required to carry out a certain operation [5]. As computing power increases more operations are able to be carried out in the same amount of time and so system complexity will increase. As systems become more complex increasing amounts of maintenance are required to be performed on them in order to keep them operational. This requires vast amounts of money and time for skilled IT workers to maintain these complex systems.

What is needed is a system which still performs all the required complex tasks of modern systems but also does not need the high levels of maintenance that these systems require. It is unrealistic to build a system that requires no maintenance at all; instead autonomic systems are being developed in which systems are able to perform certain levels of maintenance themselves.

Automating the maintenance tasks will, theoretically, improve the availability of the system as human error is the most common cause of the system failure. As a result reducing the amount of human contact is likely to reduce the number of failures. The availability will also be improved as the system does not need to be taken off-line for changes to be made.

As well as reducing the risk of failures and the time spent performing maintenance, autonomic systems will also reduce the operating costs, which is a vital factor in the competitive times we live in. The most obvious saving is the reduced man hours required.

Autonomic systems are generally implemented by using a monitoring component to detect when an adaptation is required. The monitor will then trigger an event to reconfigure the system to suit the current operating conditions. The type of adaptation will depend on the past and present conditions; it could be as simple as changing some parameter or it could require a more complex adaptation where a whole component needs to be swapped to alter the behaviour.

The main aim of Aspect-Oriented Programming (AOP) is to improve the separation of concerns by encapsulating crosscutting concerns. We propose that AOP and more specifically dynamic AOP can be used to encapsulate the adaptations that are required to implement an autonomic system. Combining this property of encapsulation with dynamic AOP allows the adaptations to be neatly contained and be applied at run-time. Additionally, the majority of the concerns that need adapting will also be crosscutting, for example security, synchronisation, caching; AOP will allow these concerns to be cleanly encapsulated. This paper presents a discussion of the key issues relating to the use of AOP for implementing an autonomic system.

The remaining sections of this report are structured as follows. Section 2 describes in more detail the properties of autonomic systems. Section 3 then looks at the various dynamic AOP techniques available and assesses their suitability for this implementation. Section 4 describes a caching example which illustrates how dynamic AOP can be used to implement a basic autonomic system. Section 5 briefly analyses this example then lists some problems of using AOP. Section 6 will then describe other complementary techniques that could be used and examine various other implementations of autonomic systems which use AOP. Finally, section 7 concludes the report and summarises this paper.

2 Autonomic Systems

As mentioned previously, autonomic systems are systems that are able to perform maintenance on themselves. They achieve this primarily by monitoring various variables relating to their running environment. When they detect that something is not behaving as intended, or the environment has changed in such a way that results in the system not running optimally, then they will perform some re-configuration on themselves to correct this.

2.1 Overview

IBM is one of the leading organisations developing autonomic systems; they have outlined four key properties that a system needs to possess to be classed as autonomic [8]:

- Self-Configuring – systems adapt automatically to dynamically changing environments
- Self-Healing – systems discover, diagnose and react to disruptions
- Self-Optimising – systems monitor and tune resources automatically
- Self-Protecting – systems anticipate, detect, identify and protect themselves from attack

For a system to be fully autonomic it is required to possess all of the above features; however a system can be developed with a sub-set of these features to be partially autonomic.

Each of these features will in some way alter the way in which the system will behave. Where the system requires high availability these changes will have to be performed dynamically without taking the system off-line. Hicks et al [6] describe how this can be done using dynamic patches without the need to use redundant hardware (a traditional approach to allow adaptations to be made to systems while maintaining system availability). This solution however, does not go far enough for autonomic systems; it is still the programmer who must select *which* patch should be applied and *when* it should be applied.

There are various implementations [3] [4] [17] [19] of autonomic systems using AOP concepts currently available and they all encapsulate the adaptations differently, we shall examine these implementations in section 5. We propose to use dynamic AOP to encapsulate the required adaptations and dynamic weaving to be able to apply these adaptations at run-time without needing to take the system off-line and without requiring redundant hardware.

2.2 Desired Properties

Next we list a set of fundamental properties required to implement an autonomic system. These properties relate to the process of applying the adaptations to the base-code, encapsulating the adaptations and specifying the relationships/dependencies. In this list we will reason why dynamic AOP is suitable for this use and justify our decision in using it.

Apply Adaptations Dynamically

Dynamic AOP is a natural choice for implementing an autonomic system due to the nature of it being able to apply code retrospectively to a running application. However, one limitation is that new code can only be woven at the points in the base-code which fit the join-point model

of the selected dynamic AOP framework and is limited to the types of advice the language allows (before, after or around). For the majority of cases this model will be acceptable.

Easily Remove Adaptations

Most dynamic AOP frameworks allow the easy removal of previously woven aspects. By simply issuing a command, the advice attached to a particular join-point can be removed. Also, as the byte-code is not modified, the system can be easily stopped and restarted to roll back to the original state of the system. Other problems can arise when this behaviour is not desirable which will be discussed later.

Encapsulate Adaptations

One of the main reasons why AOP has been chosen for this purpose was the fact that the majority of adaptations that are used in an autonomic system tend to be cross-cutting concerns such as caching, security, persistence, etc. AOP has been shown to improve the encapsulation of these types of concerns and also some types of functional concerns.

Specify Relationships

Some AOP frameworks allow, to a certain degree, relationships to be specified between aspects, for example both JAC and AspectWerkz allow this. Both of these frameworks allow the order which aspects should be applied to the base-code to be specified. However, for autonomic systems these solutions do not go far enough. The AOP frameworks solve some problems but autonomic systems need more complex relationships to be specified which will include some sort of conditional relationships.

Implement Fine Grained Changes

All AOP frameworks allow some degree of modification to the method level of a system which is much finer grained than the majority of current autonomic systems. Most current autonomic systems are component based and only allow whole components to be swapped or adapted. This can be inefficient and time consuming to implement when only a small change to a single class is needed.

Apply Adaptations to Various Points in a System

The points at which aspects can be woven with the base-code needs to follow the join-point model of the selected framework. Fortunately, the majority of frameworks have a wide reaching join-point model. These models allow code to be attached to a wide variety of points during a programs execution from exception throwing to field access operations.

3 Dynamic AOP

Static AOP languages have been used to implement systems for sometime now and techniques such as AspectJ [12] are now gaining much maturity. However, dynamic AOP languages have recently started to appear and be widely used.

An aspect created using a dynamic technique is woven at run-time and provides the extremely powerful tool of allowing the application to be modified, by changing the aspects currently woven and weaving new aspects, while the application is still running. However, the downside to this is the performance hit taken due to checks having to take place to determine whether an aspect should be woven at the current point of execution. Further problems arise regarding safety, security and compatibility.

There are currently many dynamic AOP techniques being developed, the majority of which are fairly immature so they still require more development and testing. At this stage in our development process it is not necessary for the chosen dynamic AOP framework to be 100% stable, as it is expected that no AOP framework will exactly fit the properties required so customisations will have to be made to the AOP framework to suit our needs which this paper will outline.

This next section will look at three dynamic AOP frameworks: JAC, AspectWerkz and Prose, each of these techniques possess the desired properties listed in section 2.2 to a lesser or greater extent. As well as describing the implementation details of these techniques and relevant information regarding these desirable properties will also be mentioned.

3.1 JAC

JAC [14] [15] [18] is an AOP framework developed by Renaud Pawlak which is implemented in pure Java and requires no language extensions.

JAC relies on BCEL [2] which is used to alter the byte-code of a Java object at class load-time. BCEL is used to manipulate the byte-code to add references to aspect *wrapping* methods that are used to advise the base-code. The wrapping methods are linked to form a *wrapping chain*; each method calls the next in the chain and if applicable the method is executed. It is the job of the last wrapping method in the chain to call the method which is being wrapped.

One of the problems of AOP which JAC aims to solve is *inter-aspect composition* which is related to the specification of relationships property. JAC allows the user to define a *composition aspect* which can be used to define certain relationships between aspects; a useful property when implementing an autonomic system. The creators of JAC have identified the following issues:

- Checking for aspect compatibility with the application
- Checking for inter-aspect compatibility
- Checking for inter-aspect dependency
- Checking aspect redundancy
- Ordering/selecting the aspects at run-time

The composition aspect allows the programmer to define a set of rules which can prevent the above issues from occurring. However, to prevent these issues from occurring the rules must be defined to be ‘sound and complete’; this is often difficult to achieve. The issues that JAC and the composition aspect aims to solve are similar to those of autonomic systems; decisions have to be made in order to choose the most appropriate adaptations/aspects.

JAC easily allows new aspects to be woven and later removed to existing or new joinpoints at any point during the system’s execution. Also JAC allows the common types of advice to be implemented such as after, before and around advice, so the required fine grained changes can be implemented.

One negative feature of JAC is that field access operations are not part of its joinpoint model, this limits where the adaptations can be applied to the system.

3.2 AspectWerkz

Similar to JAC, AspectWerkz [1], created by Jonas Bonér, allows the creation of dynamic aspects using pure Java and again, does not require any language extensions to implement the aspects.

Like JAC, AspectWerkz uses run-time modification of the bytecode but uses a modified classloader to weave the aspects with the base-code instead. AspectWerkz hooks directly into the bootstrap classloader and can then weave aspects to any classes loaded by the preceding classloaders. A wide range of joinpoints are supported and AspectWerkz uses the same semantics as AspectJ which makes AspectWerkz very simple and easy to use.

Similarly to JAC, AspectWerkz also allows new advice to be attached to existing pointcuts at run-time, allowing new behaviour to be introduced, again a useful property for an autonomic system.

However, AspectWerkz does not allow new joinpoints to be created at run-time; this is a limiting feature for autonomic systems and limits the dynamic property of the framework.

There are two ways around this problem. The first is to reload the classes in a new classloader, but this could result in loss of state information and would require having control of how the classes are loaded. The second solution involves creating a very 'generous' joinpoint which would cover every possible pointcut so that advice could be added at any possible joinpoint. This could introduce a large overhead when only a subset of the possible joinpoints are used.

Just as JAC allows a composition aspect to be defined to perform checks before advice is executed, AspectWerkz allows a similar module to be defined called a JoinPointController. This gives similar functionality as JAC's composition aspect and again allows certain relationships between aspects to be specified.

The joinpoint model of AspectWerkz is slightly better than the one of JAC as AspectWerkz allows field access operations to be specified. This allows the adaptations to be applied to a wider range of point much more easily.

3.3 Prose

Prose [16] developed by Popovici et al is another dynamic AOP framework which again, like the previous two examples, does not require any language extension as the aspects are implemented in pure Java. The major difference between Prose and the previous two approaches is that Prose aspects are woven at run-time using JIT compiler weaving.

Prose relies on two main modules, the Execution Monitor and the AOP Engine. The execution monitor is responsible for activating new joinpoints and notifying the AOP when a joinpoint is reached. The purpose of the AOP engine is to decompose aspects into joinpoint entities and activate them with the execution monitor. The AOP engine is also responsible for executing advice when notified by the execution monitor that a joinpoint is reached.

The creators of Prose have identified four requirements of an AOP framework that they wish to meet:

- Efficiency under normal weaving
- Secure and atomic weaving
- Efficient advice execution
- Flexibility

One of the most relevant properties listed above when implementing an autonomic system is secure and atomic weaving. This is important for several reasons but this is a feature not supported in JAC or AspectWerkz. These issues will be discussed in more detail later. Prose does offer a further advantage over JAC and AspectWerkz through its improved performance due to run-time weaving.

As with AspectWerkz similar difficulties arise in Prose when new joinpoints need to be added, although advice can still be easily added or removed from the Prose framework. Again, this limits its dynamic properties and its effectiveness.

There are also two disadvantages when compared to other languages; it does not allow any specification of the relationships between aspects as seen in JAC and AspectWerkz. Additionally, Prose does not allow 'around' advice to be specified, unlike AspectWerkz and JAC, which is a problem when the behaviour of a method needs to be replaced. This limitation will hamper the development of an autonomic system as fine grained changes cannot be implemented as easily.

The joinpoint model which Prose uses is not as extensive as that of AspectWerkz, but it still allows advice to be attached to method entry/exit events and field access operations. Again, this allows the adaptations to be applied at a variety of places in the system.

As can be seen from these descriptions of a selection of dynamic AOP frameworks, each have their own advantages and disadvantages. Whichever framework is chosen, some modification will be needed to the selected framework to implement all our desired properties.

4 Automated Cache Example

This section will briefly describe a simple example illustrating how an autonomic cache concern has been implemented using JAC. JAC was chosen due to it being the most familiar technique to us at this time and the fact that it permits the addition of new joinpoints dynamically. However, the decision to use JAC for the final autonomic system has not yet been made; it is only being used at this stage to simplify the process and for evaluation purposes. This example will highlight some of the properties described above.

The example will show how a caching aspect is added autonomically and dynamically depending on the current environment conditions. The base-code will simulate a request-response operation where the server object sending the reply will introduce a delay by sleeping for an ever increasing time. Once this delay has reached a threshold, the caching aspect will be introduced to improve performance.

```
1.public String request(int req){
2    Thread.sleep(time);
3    switch (req){
4        case 1: return "One";
5        case 2: return "Two";
6        case 3: return "Three";
7        case 4: return "Four";
8        case 5: return "Five";
9        default: return "Not valid";
10   }
11 }
```

Figure 1. Server Code

The code segment above shows the server code. When the request method is called an integer is passed to the method and the string value of that integer is returned. Line 2 is responsible for introducing the delay; the variable time is gradually increased to simulate network delay so that the response takes longer to be received.

```
pointcut("ALL", "ALL", "request.*", "MonitorWrapper", "monitor", null,
false);
```

Figure 2. Monitor pointcut

The above construct creates the pointcut to monitor the response times from the server. The first three parameters specify the point in execution where the advice should be specified. They specify that any method named 'request' that is located in any class with any number/type of parameters should be the pointcut. The following two parameters specify the method that should be used as the advice and the class its located in, in this case the class is MonitorWrapper and the method is monitor. The last two parameters are not relevant to this example. One problem which arises when defining such a joinpoint is that it will not be known at run-time where the aspects should be applied to. A potential solution is proposed in [10] which uses reflection and parameterisation of the joinpoints to generalise them.

```
1..public Object monitor(Interaction interaction throws Error{
2    long beforeTime= System.currentTimeMillis();
3    Object obj= proceed(interaction);
4    long afterTime= System.currentTimeMillis();
5    long duration= afterTime-beforeTime;
6    if (duration>threshold&&!weaved){
7        weaved=true;
8        Jac.remoteWeaveAspect("CacheTest", "s0", "CacheAC",
9            "c:\\jac\\CacheTest\\cache.acc");
9    }
10   return obj;
11 }
```

Figure 3. Monitor advice

The above monitoring code is a piece of around advice which surrounds the request method described earlier and is woven when the application is started. Lines 2, 4, and 5 are used to

calculate the time taken to execute the wrapped method that is called in line 3. Line 6 implements an if-statement to determine whether the threshold has been reached, if it has then the caching aspect is woven in line 8. The appropriate method from the JAC package is called and parameters are passed to identify the application and the server which the aspects should be woven to. Additionally, the class which implements the aspect and the path to the aspect configuration file are also passed.

```
pointcut (".*", "ALL", "request.*", "CacheWrapper", "checkcache", null, false);
```

Figure 4. Caching pointcut

Again the above code segment declares a pointcut, which specifies where the new caching aspect should be woven to. The request method is again specified as the pointcut, but this time the method will be wrapped by the checkcache method in the CacheWrapper class (see below).

```
1 public Object checkcache(Interaction interaction) throws Error{
2   Integer arg= (Integer)interaction.args[0];
3   Object result= cache.get(arg);
4   if (result==null){
5     result= proceed(interaction);
6     cache.put(arg,result);
7   }
8   return result;
9 }
```

Figure 5. Cache advice

The main aim of the above code is to intercept the calls to the request method, extract the parameter passed to it and examine the cache to see if that value is already stored in the cache. Line 2 gets the parameter passed to the request method, the following line then tries to retrieve the cached value. An if-statement on line 4, checks whether the retrieved value is null or not. A null value here indicates that there is no cached result for the value passed and so the request method needs to be called anyway, this is done so on line 5. The value which is returned from the request method is now cached for later use. Line 8 then returns the correct value to the method caller; this will either be the value retrieved from the cache or the value returned from the request method.

As mentioned at the beginning of this section this is only a simple example; many important issues have been missed for the sake of simplicity and to illustrate only the essential features. Issues regarding removal of the cache aspect and knowing which methods need monitoring have been excluded. Also, issues regarding the cache concern have been omitted such as the cache replacement policy and altering the cache size to suit the conditions. However, the example does illustrate how the required properties are implementable using a dynamic AOP framework, as will be discussed below.

5 Analysis

In the previous sections we have described some of the reasons why dynamic AOP is suitable for implementing an autonomic system. In this section we shall analyse the caching example to show the benefits of using dynamic AOP and then discuss some of the difficulties which could arise from using dynamic AOP. The problems described here will have to be overcome for an autonomic system to be successfully implemented using dynamic AOP.

5.1 Caching Example

In section 3 we listed a set of properties that make dynamic AOP suitable for implementing an autonomic system:

- Apply adaptations dynamically
- Easily remove adaptations

- Encapsulate adaptations
- Specify relationships
- Implement fine grained changes
- Apply adaptations to various points in a system

As can be seen from line 8 in the monitor advice code an entirely new aspect can be introduced dynamically to the system. Although not used in the caching example the aspect could just as easily be removed, thus returning the system to its original state and allow the easy removal of adaptations using the following instruction:

```
Jac.remoteUnweaveAspect ("CacheTest", "s0", "CacheAC");
```

Figure 6. Aspect Removal

It can be clearly seen that the adaptation to introduce the cache to the system is cleanly encapsulated. No external references are required to the base-code and no changes are required to the original code for the cache to work. This autonomic cache could be introduced to any other application with only changes being made to the pointcuts and the variable types stored in the cache. Although a simple example this demonstrates the encapsulation of adaptations and highlights the potential issues with reuse when needing to apply the same aspect to a variety of systems.

The example shows that fine-grained changes at the method level can be made. Not many applications will require changes at finer level of granularity than this. Although this caching example only shows changes at one point of execution it is clear that changes could be made at any point in the system.

The only property not demonstrated in the example is the definition of relationships between adaptations. This property was not shown in the example as only one adaptation was being made and so could not be related to anything else.

5.2 Issues Raised

This section will highlight some of the potential issues which could occur when implementing an autonomic system on a larger scale than the caching example.

Aspect Consistency

The crosscutting nature of aspects will mean that a single aspect could be applied to a number of different objects at a time. Each of these objects that have the aspect woven through it is likely to be in different states of 'use'. When the aspect needs to be removed from the objects this could be difficult due to the objects being in different states. It may be a requirement that for the aspect to be removed in a single operation the objects are all placed in a safe-state; this may not be possible.

If this problem is not considered and the aspects are removed regardless of the state, this could result in aspect inconsistency. This would mean that some objects that were in use could have the aspect still woven through it after the aspect has been requested to be removed while other objects will have had the aspect successfully removed.

When to Refactor Code and Disappearing Aspects

With aspects being added and removed arbitrarily, the system will no doubt at some stage need refactoring. The particular time chosen when this should be performed will need to be carefully selected as to avoid disruption.

Additionally, due to the fact that the majority of dynamic AOP languages do not alter the byte-code or source code in any way, dynamic aspects can be viewed as being temporary or volatile changes, in that once the application is stopped and restarted the system will be back in its original state with no aspects woven. However, a subset of the aspects that have been applied to an application may have to be made 'permanent' so that they are woven to the application whenever it is run.

Aspects of Aspects

Due to the dynamic nature of autonomic systems changes may need to be applied that were not anticipated while the system was being developed. This may also be true for aspects that have already been woven to the system. To use AOP to apply changes to the aspects already woven, the AOP language needs to allow aspects of aspects. However, the majority of dynamic AOP languages do not support this so some other method needs to be found; some static languages do support this such as Hyper/J, DJ and Aspect Collaborations.

Synchronising Aspects and Base-Code

When implementing a system which is able to modify its behaviour it is important to be able to predict how the system will behave once a change has been applied to it. So it is important to know when a new concern, which has just been woven, will be first executed. For example if a method is being executed and an aspect is woven around it, will the after part of the advice be executed? This is a fundamental feature of AOP that needs to be examined as the integrity of the application could be affected if the system behaves in unintended ways.

Describing Aspect Behaviour

Describing what each aspect will do is probably one of the more important properties/goals that needs to be implemented. The data which describes what the aspect does will be used when deciding which aspect needs to be used to alter the system behaviour in the desired way. There are various additional pieces of information that will also have to be stored such as dependencies, precedences, requirements, and other prerequisites.

This information relating to aspect behaviour and requirements could be stored in a meta-layer. This meta-layer could then be used by a change management process which would be able to determine whether the requirements set out in the meta-layer are currently met by the system and whether applying the aspect will affect the system stability or compatibility. Additionally, the meta-layer could support queries of the system regarding how the system has been altered to allow adaptations to be removed when necessary. As well as inspection the meta-layer could also support adaptation of aspect information, to update the requirements as more information becomes available.

Aspect behaviour is one component of system behaviour that needs to be described as well as rules regarding when adaptations should occur being specified. Existing approaches have used static rules, modifying these rules dynamically would be one way to implement an autonomic system that can learn about itself. Again using a meta-level, the rules could be both inspected and adapted at run-time.

Security

The weaving process needs to be secure to prevent aspects being altered or aspects being introduced from an unverified source, since both of these situations could result in the system being left in an insecure state. Therefore some kind of authentication and encryption will be needed to be able to verify the source and to prevent changes being made to the aspects.

Atomic Weaving

Atomic weaving is also important to ensure that an aspect has been successfully woven. If it cannot be determined whether an aspect was fully woven it will be difficult later when trying to determine whether or not other aspects can be woven.

Suppose for example aspect A and aspect B are incompatible with each other. Aspect A is required to be woven with the base-code but it fails mid-way through the weaving process. It is now difficult to determine whether aspect B can be woven when it is needed. If atomic weaving was used and an aspect failed to be completely woven then the system would roll-back to the state it was in before the aspect was woven, this would then leave the system in a definite state.

Combining atomic weaving with some method of monitoring the aspects that have been woven/removed will be very useful when implementing an autonomic system as the current

state of the system can be easily predicted. This is required when determining the compatibility issues between aspects.

Reuse

Being able to create a generic aspect which can be applied to a variety of systems and still maintain system consistency is vital when developing an autonomic system when using dynamic AOP. Although a set of systems may have similarities in their implementation, the specific implementation details will be different. Therefore it will be necessary for these aspects to be customisable; the Framed Aspect solution proposed in [13] may be suitable.

6 Other Techniques

The following section examines other techniques that can be used to compliment AOP in developing such a system, and will also look at other attempts to implement autonomic systems using AOP related techniques.

6.1 Complementary Techniques

HotSwap

Hotswap [9] is a new feature added to Java which allows entire classes to be swapped while the application which uses them is still running. When a replacement command is issued any methods of the old class that are still executing are allowed to finish but any new method calls are directed to the new class. The way HotSwapping would compliment AOP when implementing an autonomic system is obvious; instead of implementing complex aspects to substitute class behaviour, HotSwapping could be used. There is one limitation with using HotSwap; the old and new classes must have the same interfaces when using JDK version 1.4. However, this limitation could be removed in later JVM's to allow more complex changes to be made.

Remote Debugging

With the wide use of distributed systems it is likely that the autonomic system will have distributed elements. Also, due to the nature of distributed systems they exist in an environment which is highly susceptible to change, a situation which is suited to autonomic systems.

Implementing an autonomic system in a distributed environment will need some sort of protocol to issue commands to remote components. The Java Platform Debugger Architecture (JPDA) [11] can be used to achieve this, as it allows remote JVMs to be debugged using the Java Debugger Wire Protocol (JDWP). The Java debugger has many useful features to introspect a running application which can be used for gathering information when deciding what actions need to be taken by the autonomic system. Additionally, the remote debugger allows HotSwapping, so remote classes can be swapped dynamically – another useful feature.

Dynamic Reconfiguration Algorithms

Dynamic reconfigurations have been made to component-based systems for sometime and so algorithms designed to aid the reconfigurations have been developed. The purpose of these algorithms is to examine the proposed adaptation and determine which components need to be directed to a safe state for the adaptation to take place, more information on this can be found in [7]. Elements of these algorithms should also be applicable to dynamically adding, removing and replacing aspects.

6.2 Other Work Using AOP Concepts

Previous work has been done by Yang [19] to implement a system which performs dynamic adaptation using AOP. This approach uses joinpoints to specify *where* the adaptation should take place, and a set of rules to specify the conditions *when* an adaptation should occur. Using rules is a static solution for self-configuring systems; the system should be able to learn the

normal operating conditions and be able to identify when adaptations need to be made. Furthermore, although a small adaptive system that uses rules may be feasible, if a large system is being implemented, it would be nearly impossible to specify all the rules to define all the adaptations that may be necessary.

Another solution to achieve adaptation in applications using AspectJ is proposed by Dantas et al [3]. A number of key components have been identified to implement this solution:

- Base Application
- Adaptability Aspects
- Auxiliary Classes
- Context Manager
- Adaptation Data Provider

The base application contains the core application code without any code to implement adaptations. The adaptability and auxiliary classes co-operate together to implement the adaptations, the aspects specify how the application should be changed and then some tasks are delegated to the auxiliary classes. The context manager is responsible for monitoring the application and triggering any adaptations implemented in the aspects. Finally the adaptation data provider is a set of classes which provide context data to the dynamic adaptations, the same context change could lead to a different adaptation occurring if different data is passed from this module.

The developers claim the architecture pattern is dynamic in that adaptations can be made at run-time and the adaptation data provider can alter the application adaptations. However, it is not possible to add new adaptations at run-time as AspectJ is being used, which is the main limiting factor of this implementation.

An adaptation framework developed by Pierre-Charles David et al [4] is implemented using the Fractal component model. Although this framework does not use AOP directly it still uses the main fundamental concept – separation of concerns. The creators believe that the adaptation process should be kept separate from the main business logic of the application. This is something that we also aim to achieve by using AOP directly; we hope to be able to use our developed framework to automate a variety of systems, and so need to separate the adaptation process from the application.

The Fractal component model uses a controller which all communication passes through; the developers have modified this controller so that additional code can be inserted at the appropriate time and so can alter the application behaviour. This is similar to our approach, but instead we will be using dynamic AOP to insert this additional code.

Prose has been used to implement a system which possessed a certain degree of autonomous behaviour. MIDAS [17] was added to Prose to implement an extension management system. MIDAS was used to control the distribution of aspects in a mobile environment. The given potential scenario for use was in a factory setting, where machines could be moved to different locations. Whenever a machine entered a new location an extension base distributes the aspects to the nodes that contains an extension receiver to allow receive and weave these aspects. The aspects allow the machines to operate in the ways desirable to that particular location. Although this could be classed as an autonomic system, it is quite limited and only suited to a distributed environment. The solution we propose would be able to make more decisions about how to behave and would be suited to a variety of different applications.

7 Conclusion and Future Work

This paper has outlined the case for using dynamic AOP to implement an autonomic system. Both the benefits and problems have been discussed. There are clear advantages in using dynamic AOP for such a system which are illustrated using a simple cache example.

Although there are many benefits from using dynamic AOP, several issues are raised which are mainly related to maintaining system stability. None of the mentioned issues have yet to be fully solved by the AOP community; however some of these problems are present in other related work in dynamic reconfiguration. We will have to find a solution which is suited to dynamic AOP.

The goals and future work of this project will aim to solve the outstanding issues listed in section 5. Although these goals are focused upon aiding the implementation of an autonomic system, the issues mentioned will be present in any application created using dynamic AOP; this work will hopefully contribute to both the autonomic and AOP communities.

References

- [1] AspectWerkz, “AspectWerkz Overview”, <http://aspectwerkz.codehaus.org/>, 2003.
- [2] BCEL, “BCEL Home Page”, <http://jakarta.apache.org/bcel/>, 2004.
- [3] Dantas, A., Borba, P. “Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications with Aspects”, Proceedings of SugarloafPLoP 2003 Conference, 2003.
- [4] David, P., Ledoux, T., “Towards a Framework for Self-Adaptive Component-Based Applications”, Proceedings of FMOODS/DAIS 2003, 2003.
- [5] Gell-Mann, M., “What is complexity?”, Complexity Vol. 1 No. 1, 1995.
- [6] Hicks, M., Moore, J. T., Nettles, S., “Dynamic Software Updating”, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001.
- [7] Hillman, J., “Dynamic Adaptation of Dependable Systems”, 8th CaberNet Radicals Workshop, 2003.
- [8] Horn, P., “Autonomic Computing: IBM’s Perspective on the State of Information Technology”, 2003.
- [9] HotSwap, “HotSwap Project Research Publications”, <http://developers.sun.com>, 2004.
- [10] Hughes, D., Greenwood, P., Blair, L., “Aspect Testing Framework”, FMOODS/DAIS Student Workshop, 2003.
- [11] JPDA, “Java Platform Debugger Architecture (JPDA)”, <http://java.sun.com/products/jpda/>, 2004.
- [12] Kiczales, G. et al, “An Overview of AspectJ”, Proceedings of ECOOP pp 327-353, 2001.
- [13] Loughran, N., Rashid, A., “Framed Aspects: Supporting Variability and Configurability for AOP”, ACP4IS Workshop, 2004.
- [14] Pawlak, R. et al, “JAC: A Flexible Solution for Aspect-Oriented Programming in Java”, Reflection 2001, 2001.
- [15] Pawlak, R. et al, “Dynamic Wrappers: Handling the Composition Issues with JAC”, TOOLS USA 2001, 2001.
- [16] Popovici, A., Gross, T., Alonso, G., “Dynamic Weaving for Aspect-Oriented Programming”, AOSD 2002, 2002.
- [17] Popovici, A., Frei, A., Alonso, G., “A Dynamic Middleware Platform for Mobile Computing”, Middleware 2003, 2003.
- [18] Seinturier, L. et al, “JAC Milestones 2001”, Research Report LIP6 2001/025, 2001.
- [19] Yang, Z., “An Aspect-Oriented Approach to Dynamic Adaptation”, WOSS 2002, 2002.