Aspect-Oriented Programming for a Distributed Framework

Saša Subotić, Judith Bishop, Stefan Gruner

Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa

ABSTRACT

Aspect Oriented Programming, is changing the way software is developed in various domains of computing and software engineering. In this article we discuss the main characteristics of AOP with a particular focus on distributed systems. The testbed for our investigation is our distributed algorithm framework, Algon. Our experiments support the hypothesis that the use of AOP will result in a better separation of concerns and thus in better maintainability and portability of such systems (in comparison with a classical OOP development approach). However, we detected some limitations of the approach when tackling functional concerns. The article serves as an introduction to AOP for the distributed systems domain, and also as a case study of its use in practice.

KEYWORDS: Aspect-Orientation, Software Development, Distributed Systems, High-Performance Computing

1 INTRODUCTION

A *paradigm* in the field of software development is a particular method of thinking or a particular approach to representing the world in the structures of computer programs or software specifications. We can currently observe a paradigm shift from classical object oriented programming and design (OOP) toward aspect oriented programming and design (AOP). This shift is seen as an *e*volutionary (not *re*volutionary) process, as described by Kuhn [11].

The main reason for this paradigm shift is the discovery that even in OOP, with its generally good characteristics as far as software modularity, separation of concerns, maintainability and reusability are concerned, there are plenty of examples of unfortunate code tangling and code scattering which undermine the principle of abstraction. Moreover, these phenomena are inherent to the very problem to be solved by programming, and not just the results of inexperienced implementation attempts carried out by novice programmers.

Aspects are the key concept of AOP. They may be regarded as pseudo objects that encapsulate internal state and behavior which address issues belonging to many classes in the system. Aspects are woven into an existing program, forming a repository for the cross-cutting concerns of the system. Using aspects can fundamentally change the way in which software systems are seen and developed. While their application in sequential systems has been studied for more than a decade, less has been said about distributed systems, which is the subject of this paper.

Some years ago, AOP was mentioned by the MIT Technology Review as one of the top ten emerging areas of technology that are expected to "profoundly impact our economy and our lives" [28]. Since then, the body of experience reports from *case studies* about AOP is growing world-wide [12] [18], and models for aspects in various domains are being investigated, for example, persistence [21] and security [26]. However, little is yet reported about the application of AOP as far as South Africa is concerned, and the adoption of AOP even in academia proceeds only slowly in this part of the world — though at least one South-African research project relating AOP to system security is known to be underway [17]. In this context, the purpose of our article is to present AOP as a useful technology in one of the other emerging areas of national importance, namely high performance and distributed computing.

Over the past five years, our research group has developed a framework for the insertion and interchange of distributed algorithms, called Algon [2]. Algon has served as a useful tool in understanding the nature of distributed algorithms, and in comparing and monitoring their performance [22]. In this article we shall discuss both the introduction of simple aspects into such a distributed framework, but also the advantages and limitations of converting 'hard-coded' algorithms into inserted aspects in their own right.

1.1 Structure of this Paper

Because AOP is relatively 'new'[1] to South Africa, the following main part of our article starts with section 2 by presenting the key ideas underlying the aspect oriented paradigm (including available tools) and provides some hints to the effective use of AOP. Sections 3 and 4 describe the behavior that can emerge when aspects are used in high performance computing and distributed systems, and illustrate how AOP applies

Email: Saša Subotić ssubotic@bmm.com, Judith Bishop jbishop@cs.up.ac.za, Stefan Gruner sgruner@cs.up.ac.za

[1] despite of being mentioned already in 1997 [9]

to these specific computing domains. Section 5 explains how the distributed computing framework Algon (developed by members of our research group) can benefit from AOP as well: There we investigate the idea of a two-level implementation of Aspect Oriented Algon, which can be regarded as the 'seed' of small case study of aspect oriented software re-engineering and system evolution. Section 6 concludes the article and suggests some ideas for further work.

2 THE ASPECT ORIENTED PARADIGM

AOP [9] is especially designed to support the management of crosscutting concerns at system level. It enables developers to clearly separate crosscutting concerns that would otherwise be intertwined throughout an implementation [4].

When dealing with AOP it is necessary to distinguish *core* and *crosscutting* concerns:

Crosscutting concerns capture system-level requirements that cross a larger set of modules: Logging, tracing, profiling, policy enforcement, pooling, caching, authentication, authorization and transactional management are some crosscutting concerns that can typically (and elegantly) be dealt with by means of AOP.

Core concerns capture the central functionality of individual modules and can sufficiently be dealt with in OOP by means of objects, respectively classes.

For example a credit card processing system's core concern would process payments, while its crosscutting concerns would handle logging, security or performance.

It is important to note that the nature of these crosscutting concerns is *dynamic* which means that they become especially relevant at system runtime. If crosscutting concerns were only *static* then they could be conveniently dealt with in OOP by means of multiple inheritance. The key idea of AOP is that whenever some precondition is fulfilled anywhere in the entire system, an aspect related to that precondition will be triggered 'on the fly' to perform some additional activities which would otherwise have to be implemented explicitly by procedure/function/method calls in all the modules which are able to trigger that aspect.[2] Recent research is aiming at the *automated separation* of such concerns [19].

As a simple illustration, consider Figure 1 which sketches the difference between the OOP and AOP paradigms in managing changes to crosscutting concerns. The figure illustrates how the OOP approach forces the core classes to include the implementation of the crosscutting concern's logic which results in a certain level of code scattering and tangling. In AOP, on the other hand, it is possible to change an aspect or even replace it without affecting the other parts

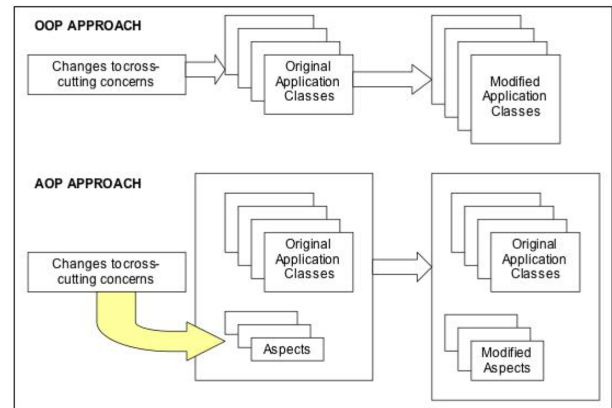

Figure 1: Crosscutting concerns in OOP and AOP

of the application. Classes and aspects can be implemented largely independent of each other, even by different members of a distributed software development project (and over a period of time in the evolution of the project); hence AOP is well compatible with Finkelstein's *view-points* approach [6] to software engineering.

Thus, AOP aims at improving the quality of the software by decreasing the level of code *scattering* and *tangling*, known as primary symptoms of non-modularization [13]:

Code scattering occurs when a single issue is implemented in multiple modules.

Code tangling occurs when a module is implemented to handle multiple concerns simultaneously (which can not be solved by means of inheritance).

The implications of such non-modularization are poor traceability and code quality, lower productivity and code reuse, and difficult software evolution. All these features can be improved with AOP which decreases the level of code scattering and tangling.

2.1 Categorization of Aspects

There are two main ways of using aspects. The one is to separate concerns that cut across the functional component, and another is to modify an existing application in order to integrate a new feature. However, aspects are concerns that are dependent on a particular given problem domain. Thus, careful domain analysis with the corresponding categorizations is likely to be beneficial for understanding the nature of aspects in a particular domain [15], and as already mentioned above there are attempts to automate the this development processes [19].

Aspects can address both *functional* and *non-functional* system properties. Implementing a functional property using aspects requires that at least rudimentary structures supporting the function in question already exist in the system. On the other hand, the injection of non-functional properties should have no impact on the design or implementation of the existing system in the process of software evolution [1] or software maintenance.

[2] A reviewer of this article pointed out that this flexible feature of AOP could also be mis-used for 'masking' existing defects, instead of eliminating them.

For example in a simulation of a CD-R, the ability to write CDs would be regarded as a functional property, whereas the ability to count the number of bytes read would be regarded as a non-functional feature. A byte counter could be implemented completely independently as an aspect.

2.1.1 Development Aspects versus Production Aspects

Categorizing aspects further, AOP practitioners differentiate between the following:

Development aspects which are only used for auxiliary purposes during the development phase of a software system (and are then removed);

Production aspects (or: product aspects) which are combined together with other modules for the final product system.

Typical development aspects are aspects for logging, tracing, debugging, profiling and testing. All these aspects can be used in the development process, and they can be removed or unplugged from the final application, as the core functionality is not affected by them: The 'Mandelbrot' example described in section 3 of this paper shall further illustrate this topic. Moreover, a development aspect can be used as an ancillary part of system, such that one can design the functionality to be optionally deployed and enable it when needed. The usage of this type of aspect is comparatively risk-free.

Typical production aspects, on the other hand, inject runtime code for performance monitoring, updating and notification, error handling or security procedures. Production aspects are an essential part of the final delivery, and getting them wrong can have serious consequences as far as the correctness of the implemented system is concerned. Consequently, when adopting AOP on the basis of OOP incrementally, AOP novices should first start with development aspects (to learn the paradigm and understand the system), and then proceed to implementing crosscutting concerns by means of production aspects.

2.1.2 Advantages and Disadvantages of AOP

AOP promotes clear design and reusability by enforcing the principles of abstraction and separation of concerns. AOP explicitly promotes separation of concerns, unlike earlier development paradigms. This separation of concerns provides cleaner assignment of responsibilities, higher modularization and easier system evolution, and should thus lead to software systems which are easier to maintain. The process is to collect scattered concerns into compact structure units, namely the aspects.

On the other hand, AOP cannot be elegantly applied to every possible problem situation: Consider for example, the limitation of AOP described in [10], suggesting that transaction management, through crosscutting, is hard to factorize out into a separate aspect. AOP is mostly suited for large scale software development projects, especially as we want to emphasize in this article, distributed systems. However it brings with it certain difficulties as far as testing and debugging are concerned. This is due to the *side effects* which stem from the dynamic (ad-hoc) injection of runtime code and which could, in the worst case, lead to semantic ambiguities in the control-flow of an aspect oriented program. Different aspects can actually interfere at the same points of in-weaving. Thus, AOP *can* violate the principle of encapsulation, although in a rather systematic and well-controlled way [13].[3]

2.2 AOP Languages and Environments

The aspect oriented programmer of today can choose amongst a variety of AOP languages and frameworks which integrate with all the major programming languages of today, namely Java, C++, C$^\sharp$, Smalltalk, Ruby, Scheme. There is even an aspect-oriented approach to XML specifications. All of these can be found on the AOP website [29]. The key purpose of an AOP language is to specify structured runtime transformations on a program [25]. The transformations involve the dynamic insertion (or removal) of runtime code at (before or after) well-defined program points. The most common locations for such 'on the fly' code modifications are function or procedure calls [26].

2.2.1 AspectJ

AspectJ, originally from Xerox PARC, but now part of the Eclipse initiative supported by IBM, is currently the most widely adopted programming language supporting AOP and was also used for our case studies which are described in section 3. AspectJ is built on top of the programming language Java [3]. It provides mechanisms to modularize crosscutting concerns as explained above. In AspectJ programs, Java classes are used to implement the core characteristics, and aspects (understandable as pseudo classes) are used to implement crosscutting concerns in a modular fashion.

In an AspectJ application, everything revolves around *join points*. These are points in the control flow graph of a compiled program, where crosscutting concerns are woven in. According to AspectJ's terminology there are two types of crosscutting — see Figure 2 for illustration:

Static crosscutting describes crosscutting that influence the interfaces of the involved types and does not modify the execution behavior of the system. AspectJ provides the following two mechanisms to achieve this kind of influence:

- **Introduction** introduces changes to the classes, aspects and interfaces of the system.
- **Compile-time Declaration** adds compile time warnings and error messages for the case that certain occurrences of patterns are captured.

Dynamic crosscutting describes crosscutting that influence the behavior of an application. AspectJ

[3]Remember a similar debate between Dijkstra and Knuth about the well-controlled application of 'harmful' GOTO statements some decades ago, when the then new paradigm of Structured Programming was arising.

```
public aspect ExampleAspect {
   // Dynamic crosscutting
   pointcut doSomething()
   call ( * ExampleClass.do* ( . . ) );

   before() : doSomething() {
   . . . advice body
   }
   after() : doSomething() {
   . . . advice body
   }
   // Static crosscutting
   declare warning : <pointcut>:<message>;
   declare error : <pointcut> : <message>;
   declare precedence : Aspect1, Aspect2;
   declare parents : [Child] extends [Class]
   private float ExampleClass._dataMember;
}
```

Figure 2: Two types of crosscutting in AspectJ

provides the following two language constructs to achieve this kind of influence:

Pointcut is a constructor that selects join points and collects the context at those points based on different conditions.

Advice declares a quasi-method which will be executed before, after or around join points picked up by a pointcut whenever the according preconditions are fulfilled.

With these additional constructs, the Java developer can add new functionality in the system without changing any code in the core modules (classes). AspectJ retains all the benefits of Java and is therefore platform-independent. As far as compatibility is concerned it is important to note that

- every syntactically correct Java program is also a syntactically correct AspectJ program, and
- every successfully compiled AspectJ program can be executed on a standard Java Virtual Machine.

After these preliminary explanations we are now prepared to consider the application of AOP in distributed and high-performance computing (HPC) in the following sections.

3 ASPECTS IN HPC

Building a software system with high performance, high scalability and high reliability is a challenge. The main idea behind high performance systems is to use more then one computer or a computer with more than one processor to solve a problem. Areas requiring such computational speed include mostly numerical and scientific applications. Traditional mechanisms for controlling parallel execution in numeric and scientific applications do not provide an easy separation of concerns, so that the problem of code tangling is likely to appear in many implementations of parallel or distributed solutions [8]. Research into the separation of concerns in grid applications is also well on its way [14]. Simple examples of aspects for HPC are profiling, tracing and performance analysis.

Profiling in a distributed context cannot be achieved with standard profiling software incorporated into the usual operating systems for ordinary PCs: The distributed nature of the runtime application requires more refined approaches to performance profiling. In such scenarios it is necessary to deal with computation and communication separately, and sum up the components to calculate the total CPU time across the entire cluster. An example of profiling by means of AOP is sketched in Figure 3, whereby *time* is the observable on which the profiling is solely focused in this example.

```
// a) OOP example of profiling
public class TimeWatch {

   public TimeWatch() {
      difference = 0;
   }
   public void start() {
      startTime = System.currentTimeMillis();
   }
   public long stop() {
      endTime = System.currentTimeMillis();
      difference = endTime - startTime;
      SEND(difference);
   }
   private long difference,startTime,endTime;
}
------------------------------------------------
// b) AOP example of profiling
public aspect TimeWatchAspect {

   pointcut measurePerformance()
      call ( * *.compute* ( . . ) );

   before() : measurePerformance() {
      difference = 0;
      startTime = System.currentTimeMillis();
   }
   after() : measurePerformance() {
      endTime = System.currentTimeMillis();
      difference = endTime - startTime;
      SEND(difference);
   }
   private long difference,startTime,endTime;
}
```

Figure 3: Distributed Profiling in a) OOP and b) AOP

The pseudo code statement ‘SEND(·)’ in the figure represents a remote procedure call (or similar operation of network communication) which transmits the *local* profiling result across the network to a special profiling node which will eventually calculate the *global* performance. Due to their parallel (concurrent, distributed) nature, high performance applications are intrinsically difficult to comprehend, such that developers often want to add some logging (tracing) state-

ments in the program to see exactly what happens during program execution. With AOP it is possible to separate the core mathematical code from the performance measuring code. The following subsection provides an example to this technique.

3.1 Example: Concurrent Mandelbrot Set

Our concurrent Mandelbrot set program is a HPC example of processing a bitmap image. The computation can be divided into a number of completely independent parts which are known as embarrassingly parallel in that it is possible to spread the computation across several processors, and the computation is 'embarrassingly parallel' with respect to every pixel of Mandelbrot's set.

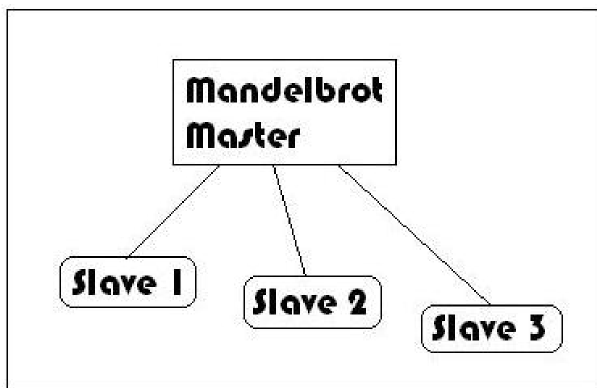

Figure 4: Architecture of the Mandelbrot Example

600 x 600 timing	Slave 1	Slave 2	Slave 3
logging on	106 820ms	107 000ms	189 969ms
logging off	8 422ms	8 531ms	86 156ms

Table 1: Results from the Mandelbrot experiment

The architecture of our example is sketched in Figure 4, with three slave processors computing the Mandelbrot set and one master processor steering the work and gathering the results, whereby Java RMI was used to implement the communication between the master node and its three slaves. As mentioned above, this application covers the aspect oriented implementation of two different concerns:

- Logging (tracing) is an example of a *development* aspect that shall be used only temporarily: we will say that it can be 'switched on' and 'switched off' *ad libitum*;
- Performance time measurement (profiling) shall here be regarded as a *product* aspect which belongs to the system *intrinsically* and may thus never be 'switched off'.

Note that these two aspects cannot stand alone without the core concern, however they are not dependent on each other. Part b) of Figure 3 sketches an implementation of the *profiling* aspect (time measurement) in Java with AspectJ. Due to lack of space in this paper, the program code of the *logging* aspect of this example is *not* depicted.

To observe the concurrent behavior of the program without the availability of AOP, the programmer would typically insert the logging code between the lines of the main algorithm. Thereafter, the programmer must remove these lines from all over the place to create the product to be delivered. In AOP, as explained above, it is possible to concentrate the program code of this concern of behavior observation in a single development aspect and to remove (or 'switch off') this aspect later, without any need to muddle in the core routines of the application. This has been done for our Mandelbrot experiment which we are discussing in this section.

The most interesting result of our Mandelbrot experiment is related to the observation of *interference* between the two above-mentioned aspects, namely the development aspect (for the occasional purpose of logging the program's behavior) and the product aspect (for the intrinsic purpose of profiling the program's runtime). Table 1, which is logically related to the master node of Figure 4, presents the data resulting from the experiment.

The result data clearly demonstrate that the time measurements delivered by the profiling product aspect are strongly influenced by the presence or absence of the auxiliary development aspect (which can be switched on or off as mentioned above). This is because the profiling aspect does not only assess the 'core' of the Mandelbrot application itself but also the auxiliary development aspect as soon as it is being switched on. AOP novices might feel surprised by this kind of unexpected interdependence between seemingly different aspects. Therefore it is important to keep in mind that in order to achieve the desired system behavior, auxiliary development aspects must be eventually switched off to make the product aspects work undisturbed and as intended.

4 ASPECTS IN DISTRIBUTED SYSTEMS

Generally speaking, a distributed system is a collection of independent processors that appears to its users as a single coherent system [24]. Important issues that we need to consider when dealing with such systems are: communication, processes, naming, synchronization, consistency and replication, fault tolerance, integrity and security. Distributed computer systems are characterized by a substantial amount of complexity and a strict preservation of modularity is not possible for all concerns.

In particular fault tolerance, fault detection, and eliminating the central point of failure, are examples of concerns that are not easily addressed as these issues tend to span multiple components in distributed applications. However, they may be addressed with aspects, but with caution, as serious shortcomings could emerge in handling of the failure semantics in a distributed environment [27]. In addition to what

has already been said, the most noticeable examples of crosscutting concerns (aspects) at system level for distributed systems are:

Synchronization to implement all code for node synchronization within an aspect and to inject it into all existing modules that rely on inter node communication,

Remote Access which also is an aspect tending to cut across the whole system and could therefore be factored out from existing modules,

Fault Tolerance which can be implemented by emergency aspects which trigger as soon as parts of the computation network are failing,

Security at system level, such as filtering of malicious messages, buffer overflow control, periodical password checking, session time counting, and the like.

AOP possibly has the potential to address these issues appropriately and to enforce a formal separation between the main functionality and these issues in distributed computer applications [16] [20] [5] as well as in classical single processor systems. The following subsection illustrates this by an example of failure handling in a distributed environment.

4.1 Example: Failure Handling

AOP provides the means to manage the handling of network failure in a coherent way. In a distributed environment this is an important task, as maximum availability of the service must be achieved. When a service is down, it is a reasonable strategy to reattempt the operation a certain number of times. Figure 5 sketches an aspect which implements such a simple way of failure handling. In such a scenario it is also important to know how many times to reattempt a temporarily unavailable operation in order to achieve approximately 99% availability. This is relatively easy to test using AOP by simply tuning the corresponding aspect. Table 2 provides the results of hundred simulations for three different failure scenarios, with the relevant *number of re-attempts* aspect being 'switched on' (in the terminology of above) in every case.

Availability (pure)	25%	50%	75%
No. of re-attempts: aspect switched on	15 $1-0.75^{15}$	7 $1-0.5^{7}$	4 $1-0.25^{4}$
Availability approx.	98.66%	99.22%	99.61%
100 simulations	100%	100%	100%

Table 2: Results from Failure Handling Experiments

```
import java.rmi.RemoteException;

public aspect FailureHandling {
   final int MAX = 7;
   Object around() throws RemoteException
  :call(* *.get*(..)  throws RemoteException){
      int tryagain = 1;
      while (true) {
         try {
            return proceed();
         }
         catch (RemoteException e) {
            SEND(``Found''+e);
            if (tryagain++ > MAX) {
               throw e;
            }
            SEND(``\tReattempt'');
         }
      }
   }
}
```

Figure 5: Aspect of Failure Handling, according to [13]

5 AOP AND ALGON

The research vehicle that drives our investigation of aspect orientation is the Algon system. Algon —for *alg*orithms *o*n the *n*et— is a framework for the investigation and exchange of algorithms in distributed systems [2]. Algon hides the complexity of distributed algorithms (such as mutual exclusion, synchronization, etc.) from application programmers in a separate component layer. It also features performance comparisons of the various distributed algorithms [22], and it also supports the interchangeability of the middleware on which it runs [23].

Algon meets the requirements of both the application programmer and the system builder. Furthermore, it also addresses the needs of the maintenance team, in monitoring the performance of the system during maintenance and evolution. The application programmer interacts with Algon via an API, and uses it as a library. The programmer also tailors the configuration files to reflect the current distributed system architecture. The system developer will monitor the use and performance of the chosen middleware and distributed algorithm so as to choose the best configuration. This happens before system delivery. Once the system has been delivered, the software maintenance team can make use of the Algon performance display to monitor the performance of the particular algorithm using the chosen middleware. If the maintenance team wish to make alterations to the chosen algorithm or middleware this can be done by means of a configuration file.

5.1 Overview of Algon's Architecture

Algon's architecture consists of distributed algorithms and scheduling code grouped in a component layer. This component layer consists of at least one distributed algorithm interface (for a family of algorithms), one coded algorithm (implementing that interface) and a scheduler (for that family of algorithms), as depicted in Figure 6. The scheduler is a class that provides transparent access to lower-level Algon features, and it performs configuration. It is specific to a family of algorithms (such as mutual ex-

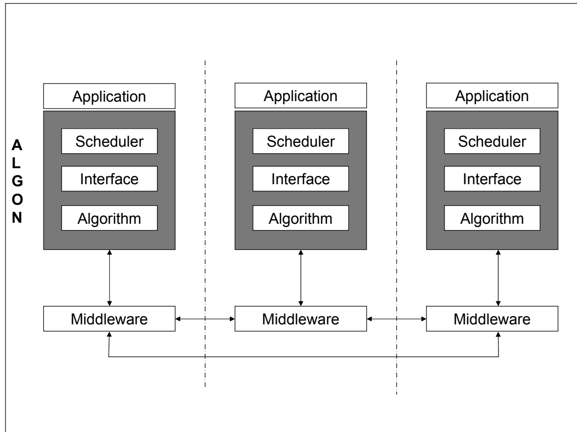

Figure 6: Algon's Architecture according to [23]

clusion) and allows algorithm-dependent state information to be maintained outside of the application logic. The interface abstracts behavior common to functionally equivalent algorithms. The algorithm is programmed from theory and implements an interface. The greatest benefit of interfaces and schedulers is that only small modifications are needed for adding further algorithms to the framework.

5.2 Aspect Oriented Algon

Programming the collaboration among the various processes running on different sites in a distributed application will often require the services of a distributed algorithm, and these algorithms need to be incorporated into the application code in some way. The conventional way would be to intersperse the framework code with the application code as and when required. In AOP, on the contrary, we would inject the implementation of the distributed algorithm at numerous points into the framework code, as it is sketched in Figure 7. The figure illustrates that aspects could be inserted at several parts in the layered architecture. However, examining the existing Algon framework we can find two main ways for re-engineering the system in the aspect oriented paradigm:

1. Aspects can be used to integrate the Algon framework into the user's particular distributed application, in which case Algon is the aspect code;
2. AOP could be used with the Algon framework, with the algorithms and non-functional concerns being the aspects.

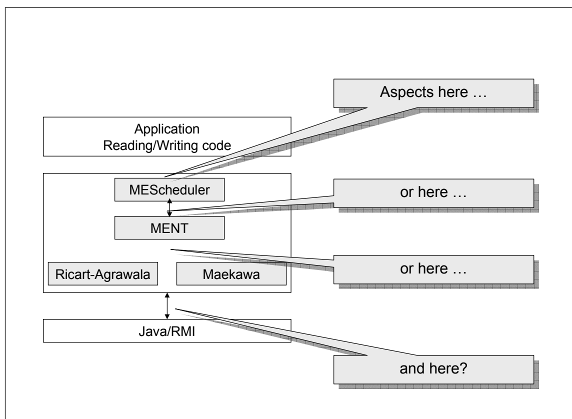

Figure 7: Aspect Oriented Algon

In the first case, the user's particular distributed applications would become to a greater degree *independent* of the underlying Algon platform and could still be used after newer versions of Algon have been released. AOP would contribute toward the evolution and maintenance of distributed code. For the second case, a considerable number of Java classes forming Algon could identified as carrying crosscutting concerns. The most prominent of those concerns is, as usual, the performance measuring code, which exists throughout the system. Figure 8 illustrates the current Algon design in comparison to an AOP approach.

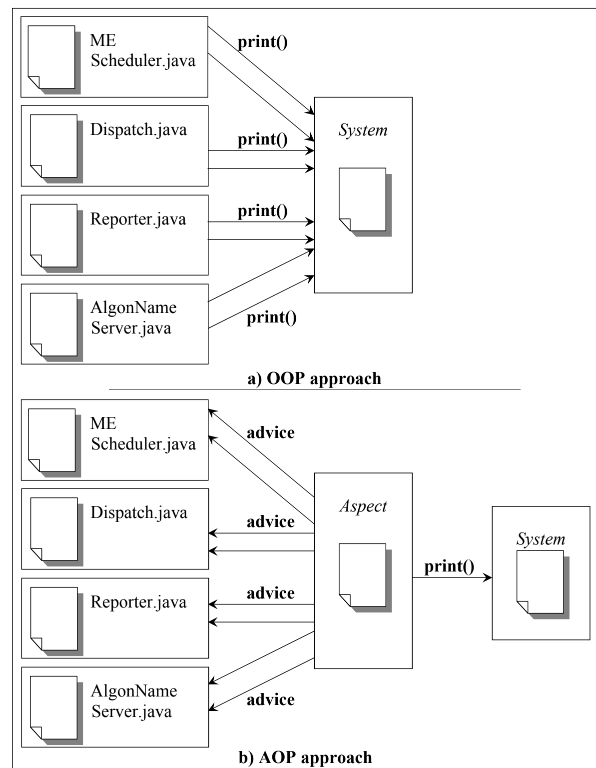

Figure 8: Difference between current and AOP approach

Whereas four debugging points can be seen in part (a) of Figure 8, only one debugging point is needed in part (b). Because the application exists on several computers, the usual profilers cannot be used for the purpose of gathering performance data, and Algon and its aspects would be responsible for the message passing that would get the data to a central place.

Algon can be considered to be a *tool* which applies a *separation of concerns* technique to *algorithmic concerns.* The technique has been applied to a variety of different concerns, including real-time constraints distribution and replication exception handling location control and synchronization [27]. There are basically three approaches to achieving separation of concerns:

1. identifying the specification of concerns and allowing the programmer to specify each concern in a separate aspect,
2. treating the concern as being orthogonal and freeing the programmer completely from it,
3. providing the programmer with a library which encapsulates the complexity.

The first approach is usually applied for reflective purposes but we do not feel that Algon is adding reflective capabilities to the system. Algon has addressed the problem of middleware independence, and not cross-cutting, non-functional concerns across a particular middleware system. Algon applies a modular approach, and does not make use of aspects. The second approach is also not suitable for Algon's purposes since the programmer must obviously be involved in the use of Algon. Algon is most similar to tools that fit into the third category. Algon does provide a library, but offers the programmer an additional level of choice, and supports an informed choice by means of the performance comparison tool.

One approach that fits into the third category and that also addresses distribution issues is *Garf* [7]. Garf provides the programmer with an extensible library for adding behavioral features to distributed programs. Whilst being innovative for its time, Garf has two shortcomings: it was implemented in Smalltalk which limits its applicability, and it does not attempt to offer a choice between different implementation techniques. Algon addresses distribution issues, as does Garf, but from an algorithmic perspective. Rather than providing a library of functions to be used blindly, Algon recognizes the differing nature of distributed systems and offers programmers the capacity to tailor their systems accordingly.

5.3 Limitations of the aspect approach

Whereas the AOP approach has obvious and attractive advantages in dealing with many non-functional aspects of complex systems, we argue that it should be used with caution when dealing with functional aspects of systems, especially with regard to distributed systems. Indeed our experience in building Algon has demonstrated that this kind of aspect simply cannot be addressed by AOP but should rather be dealt with using a layered approach. The problem is that aspects are perfect for dealing with *non-functional* and *orthogonal* concerns. However the kinds of concerns the Algon system addresses are *neither* orthogonal nor non-functional. They are wrapped up in the functionality of the application and the extra dimension of failure handling simply cannot be adequately dealt with in an orthogonal fashion. More details of this discussion are found in [27].

6 CONCLUSION AND FUTURE WORK

Following the above investigation, we conclude that the AOP approach has obvious advantages in dealing with many non-functional aspects of complex systems, but that it should be used with caution when dealing with functional aspects of system, especially with regard to distributed systems. We have shown situations and approaches where AOP can be convenient for solving existing problems in a better and efficient way. We conjecture that this is especially true for distributed systems and high performance computing, especially in terms of fault tolerance and performance monitoring. Aspects can provide performance monitoring where profilers cannot, in a distributed system. It remains future work to implement the suggested AOP modifications in our distributed algorithms framework Algon, such that we can expect deeper insights into the finer details of aspect oriented code migration and software evolution.

Acknowledgments

Thanks to the Algon team over several years for their inputs, especially Basil Worrall and John Müller. Financial support by South-Africa's National Research Foundation (NRF) is gratefully acknowledged. Last but not least thanks to the anonymous reviewers for their critical and helpful comments and remarks.

REFERENCES

[1] Walter Cazzola, Sonia Pini and Massimo Ancona, "AOP for software evolution: a design oriented approach". In Proc. *ACM SAC*, pp. 1346-1350, Santa Fe, 2005.

[2] Judith Bishop, Karen Renaud and Basil Worrall. "Composition of distributed software with Algon: concepts and possibilities". *ENTCS*, vol. 65, no. 4, pp. 1–12, 2002.

[3] Andy Clement, George Harley, Matthew Webster and Adrian Colyer. *Eclipse AspectJ: aspect oriented programming with AspectJ and the Eclipse AspectJ development tools*. Addison Wesley Prof., 2005.

[4] Antonella DiStefano, Marco Fargetta, Giuseppe Pappalardo and Emiliano Tramontana. "Metrics for evaluating concern separation and composition". In Proc. *ACM SAC*, pp. 1381-1382, Santa Fe, 2005.

[5] Cormac Driver and Siobhn Clarke. "Distributed systems development: Can we enhance evolution by using AspectJ?" *LNCS*, vol. 2817, pp. 368–382, 2003.

[6] A. Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke and L. Finkelstein "ViewPoints: a framework for integrating multiple perspectives in system development". *Internat. Journ. of Softw. Eng. and Knowledge Eng.*, vol. 2, no. 1, pp. 31–58, 1992.

[7] R. Guerraoui, B. Garbinato and K. Mazouni. "Garf: a tool for programming reliable distributed applications". *IEEE Concurrency*, vol. 5, no. 4, pp. 32–39, 1997.

[8] Bruno Harbulot and John R. Gurd. "Using AspectJ to separate concerns in parallel scientific Java code". In Proc. *3rd Internat. Conf. on Aspect Oriented Softw. Development*, pp. 122–131, March 2004.

[9] Gregor Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin. "Aspect oriented programming". In Proc. *ECOOP'97: 11th Europ. Conf. on Obj. Oriented Progr.*, 1997.

[10] Jörg Kienzle and Rachid Guerraoui. "AOP: does it make sense? The case of concurrency and failures". In Proc. *ECOOP 16th Europ. Conf. on OOP*, pp. 37–61, June 2002.

[11] Thomas S. Kuhn. *The structure of scientific revolutions.* University of Chicago Press, 1962.

[12] Axel Anders Kvale, Jingyue Li and Reidar Conradi. "A case study on building COTS-based system using aspect-oriented programming". In Proc. *ACM SAC*, pp. 1491-1498, Santa Fe, 2005.

[13] Ramnivas Laddad. *AspectJ in action: practical aspect oriented programming.* Manning, Greenwich, 2003.

[14] Paulo Henrique M. Maia, Nabor C. Mendona, Vasco Furtado, Walfredo Cirne and Katia B. Saikoski. "A process for separation of crosscutting grid concerns". In Proc. *ACM SAC*, pp. 1569-1574, Dijon, 2006.

[15] Marius Marin, Leon Moonen and Arie van Deursen. "A classification of crosscutting concerns". In Proc. *ICSM'05 21st Internat. Conf. on Softw. Maintenance*, pp. 673–676, 2005.

[16] Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana and Giuseppe Ursino. "Handling run-time updates in distributed applications". In Proc. *ACM SAC*, pp. 1375-1380, Santa Fe, 2005.

[17] Keshnee Padayachee, J.H.P. Eloff and Judith Bishop. "An aspect oriented implementation of e-consent to foster trust". In Proc. *SAICSIT'06*, Erinvale, October 2006.

[18] Odysseas Papapetrou and George A. Papadopoulos. "Aspect oriented programming for a component-based real-life application: a case study". In Proc. *ACM SAC*, pp. 1554–1558, Nicosia, 2004.

[19] Giuseppe Pappalardo and Emiliano Tramontana. "Automatically discovering design patterns and assessing concern separations for applications". In Proc. *ACM SAC*, pp. 1591-1596, Dijon, 2006.

[20] Elke Pulvermüller, Herbert Klaeren and Andreas Speck. "Aspects in Distributed Environments". In Proc. *GCSE'99 Internat. Symp. on Generative and Comp.-Based Softw. Eng.*, pp. 37–48, London, 1999.

[21] Awais Rashid and Ruzanna Chitchyan. "Persistence as an aspect". In Proc. *AOSD'03*, pp. 120–129, 2003.

[22] Karen Renaud, Johnny Lo, Judith Bishop, Pieter van Zyl and Basil Worrall. "A framework for supporting comparison of distributed algorithm performance". In Proc. *PDP'03 11th Euromicro Conf. on Parallel, Distr. and Network-based Processing*, pp. 425–432, Genoa, February 2003.

[23] Karen Renaud, Judith Bishop and Basil Worrall. "Algon: from interchangeable distributed algorithms to interchangeable middleware". *ENTCS*, vol. 114, pp. 65–85, 2005.

[24] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems: principles and paradigms.* Prentice Hall, 2002.

[25] John Viega and Jeffrey M. Voas. "Can aspect oriented programming lead to more reliable software?" *IEEE Software*, vol. 17, no. 6, pp. 19–21, 2000.

[26] John Viega, J. Bloch and P. Chandra. "Applying aspect oriented programming to security". *Cutter IT Journal*, vol. 14, pp. 31–39, February 2001.

[27] Basil Worrall, Judith Bishop and Karen Renaud. "Limitations of the aspect-oriented approach in distributed system architectures". Tech. rep., Polelo Research Group, University of Pretoria, July 2004.

[28] "Emerging technologies that will change the world". http://www.technologyreview.com/ January 2001.

[29] http://www.aosd.net/