

Emergent Behaviour of Aspects in High Performance and Distributed Computing

SASA SUBOTIC AND JUDITH BISHOP
University of Pretoria, South Africa

In this paper we discuss the characteristics of Aspect Oriented Programming (AOP), and the development of new or existing applications using AOP. Aspects are emerging everywhere, and there is a particular need to introduce and practice them strategically in areas such as parallel and distributed computing. We will show that this attractive technology is useful for solving the problems of code scattering and tangling in high performance parallel computing. The performance related aspects are most noticeable for high performance applications. Then we discuss AOP impact in the distributed environment. Programming in a distributed environment is a complex task where object orientation has been of limited success in managing crosscutting concerns such as synchronization, scheduling, fault tolerance and security. Our aspect research vehicle at the Polelo Research Group is the Algon distributed framework (ALGORithms on the Net). The focus here is to show how AOP can be applied to an Algon application aiming at a better separation of concerns. Basic examples are logging, debugging and performance related aspects associated with this system. We argue that this modern programming paradigm allows existing systems to be re-designed or modified as a viable approach to improve the functionality and flexibility of the system.

Categories and Subject Descriptors: C.2.4 *Distributed applications*; C.4 *Design studies, Fault Tolerance, Measurement Techniques, Performance Attributes, Reliability*; D.1.3 *Distributed Programming, Parallel Programming*; K.3.2 *Computer Science Education*;

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Aspect Oriented Programming (AOP), AspectJ, High Performance Computing (HPC), Distributed Computer Systems (DS), Algorithms on the network (Algon)

1. INTRODUCTION

Computer scientists and information technologists are fond of the word *paradigm*. A paradigm in the computing environment is a model used to describe the thinking about something or to show how something can be produced. Object orientation has become one of the most popular programming paradigms. The OOP paradigm is based on the simulation of real world objects that comprise a system. The program consists of classes that describe objects that encapsulate all the state and behaviour of the associated real world entities. However, even with this approach, there is some functionality that could be encapsulated in an object and then scattered over a series of objects [Gabor 2004]. This problem of scattering and tangling created a shift in thinking from objects toward aspects. Aspects encapsulate a state and behaviour that is not an intrinsic part of a real world entity and form a central unit of the aspect oriented paradigm.

Aspect oriented programming is not a replacement for object orientation. Simply stated, it is a collection of additional concepts that are added to OOP. Most of the work on investigation of this paradigm has mainly been done in research laboratories around the world. Aspects present new challenges to software development practice, and there is a need to introduce them in different fields of computer science. Since there are many existing courses around the world addressing concurrency, distributed systems and high performance computing, we anticipate that one of the outcomes of this paper will be to stimulate thinking in terms of aspects in this direction.

In this paper our goal is to provide the reader with an understanding of aspect oriented programming and an appreciation for the different areas of research and study within this important field. In order to illustrate how AOP applies to a specific example, certain aspects were considered and applied to a case study. We move beyond the classical examples (in the early part) of aspects by introducing Aspect Oriented Algon (research study). Within this research study we have addressed both advantages and limitations of aspect orientation.

Following this introductory part, section 2 describes the general idea behind aspect oriented paradigm. Section 3 describes the additional behaviour that emerges when aspects are used in high performance and distributed computing. Section 4 explains how the Algon distributed framework can benefit by making use of this new approach. Section 5 considers related and future work. Finally, section 6 presents our conclusion.

Author Addresses: Department of Computer Science, University of Pretoria, South Africa; sasa@tuks.co.za and jbishop@cs.up.ac.za.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, that the copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than SAICSIT or the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2005 SAICSIT

2. ASPECT ORIENTATION: AN EMERGING PARADIGM

2.1 Aspect Oriented Programming

Aspect oriented programming is a new programming paradigm that specifically targets the management of crosscutting concerns. AOP enables developers to clearly separate crosscutting concerns that would otherwise be intertwined throughout an implementation.

As a simple illustration, consider Figure 1, showing the difference between the OOP and AOP paradigm in managing crosscutting concerns. The OOP based approach creates a certain level of code scattering and tangling by forcing the core modules to embed the crosscutting concern's logic. On the other hand, with the AOP based approach you can make changes to an aspect or even replace it without affecting the other parts of the application.

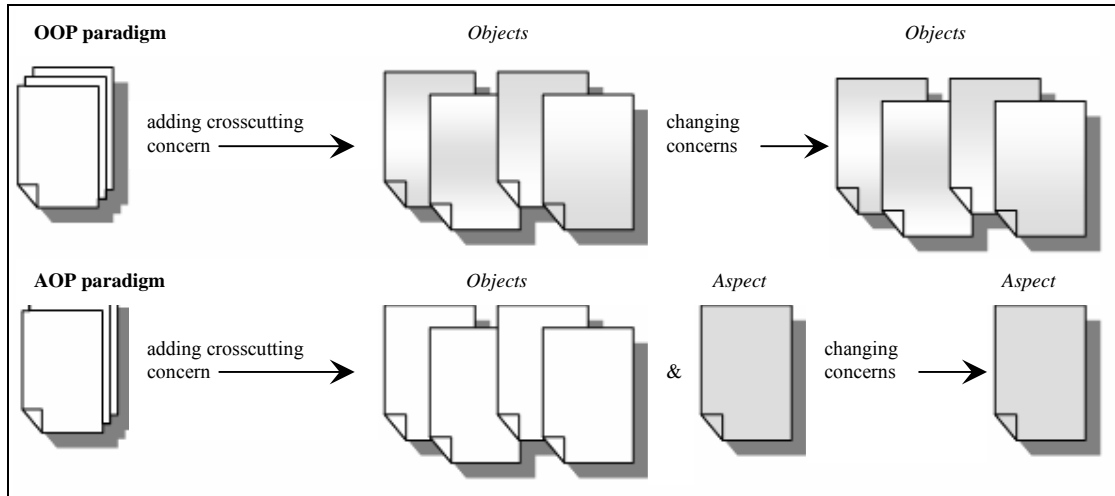


Figure. 1. Management of crosscutting concerns with OOP and AOP

AOP aims at improving the quality of the software by decreasing the level of code scattering and tangling. These two phenomena are also known as symptoms of non-modularization [Laddad 2003]:

- *Code scattering* is when a single issue is implemented in multiple modules.
- *Code tangling* is when a module is implemented to handle multiple concerns simultaneously.

A core concern captures the central functionality of a module, and crosscutting concerns capture system-level requirements that cross multiple modules. Logging, tracing, profiling, policy enforcement, pooling, caching, authentication, authorization and transactional management are some crosscutting concerns that can be nicely addressed with aspects. Table 1 identifies certain aspects for certain domains. From the given table we see that aspects could refer to location, communication, synchronization, etc.

<i>Domain:</i>	<i>Distributed Computing</i>	<i>Image Processing</i>
Aspects:	What the objects do	Operations on pixel maps
	Their location	Control structure
	Communication	Memory usage
	Synchronization	Sharing

Table. 1. Example domains and key aspects of concerns in each, adopted from [G Kiczales et al. 1997]

There are mainly two ways of using AOP. One way is to separate concerns that cut across the functional component. Another way of using aspects is to modify existing application in order to integrate a feature which has not been foreseen during the design phase of the software engineering process. Both ways will immediately yield a cleaner design and substantial code savings.

Aspect can address both *functional* and *non-functional* behaviour. Implementing a functional behaviour using aspects requires that at least certain structure for the function already exists in the system. On the other hand

implementing a non-functional behaviour would have no impact on the design or implementation of the existing system. Worrall *et al* [2004] provide an example of the functional versus non-functional concern by considering a simulation of a CD-ROM drive. Two aspects, added to the CD-ROM drive, are the ability to write CDs and the ability to count the number of bytes read. The first one would be functional requirement of the new drive. On the other hand implementing a byte counter would be a non-functional requirement. Categorizing aspects further, we differentiate among two types of aspects:

- *Developmental aspects*, which are used during the development phase, and then removed.
- *Production aspects*, which are used together with the final product.

AOP promotes clear design and reusability by enforcing the principle of separation of concerns. This separation of concerns provides cleaner assignment of responsibilities, higher modularization and easier system evolution, and leads to a system that is easier to maintain and understand. In the rest of this paper we will show that along with reusability this challenging methodology is also useful for introducing performance into applications.

2.2 The AspectJ Programming Language

There are many implementations of AOP for many languages. AspectJ from Xerox PARC [AspectJ Team] is one such implementation of aspect oriented programming. It is built on top of the programming language Java and provides additional mechanisms to modularize crosscutting concerns. This aspect oriented extension to Java is the most widely adopted programming language supporting AOP.

In AspectJ applications, Java classes are used to implement core modularity and *aspects* (class-like constructs) are used to implement crosscutting modularity. In an AspectJ application everything revolves around *join points*. These are points in the execution of a program, where crosscutting concerns are woven in. According to AspectJ terminology there are two types of crosscutting:

- *Static crosscutting* describes crosscuttings that influence the interfaces of the involved types and does not modify the execution behaviour of the system. AspectJ provides the following mechanisms to achieve this kind of influence :
 1. Introduction – introduces changes to the classes, aspects and interfaces of the system.
 2. Compile-time declaration – adds compile-time warnings and errors when certain usage patterns are captured.
- *Dynamic crosscutting* describes crosscuttings that influence the behaviour of an application. AspectJ provides the following language constructs to achieve this kind of influence :
 1. Pointcut – a constructor that selects join points and collects the context at those points based on different conditions.
 2. Advice – a method like construct, that executes before, after or around join points picked up by a pointcut.

With these additional constructs, the Java developer can add new functionality in the system without changing any code in the core modules. AspectJ retains all the benefits of Java and therefore it is platform independent. It is used today for many real world projects. Its major practice is for enhancing the middleware platform, adding and improving security features to existing applications, and particularly for monitoring and improving performance [Laddad 2003]. Figure 3 provides a simple, yet powerful example of an AspectJ program.

3. EMERGENT BEHAVIOUR OF ASPECTS

Aspects are emerging in many areas of computer science. With aspects we can always influence the behaviour of the existing applications. We may change certain behaviour, or we may add additional ones, without even changing any code in the core modules of the existing application.

Emergence [Wikipedia] is defined as a process of complex pattern formation from simple rules. This section illustrates emergence from chaos, as introduced by OOP in managing crosscutting concerns, to order by using AOP. Figure 2 shows the influence of object-oriented programming on distributed systems and high performance computing, and how AOP and aspects can build on it. We see that many entities operate in an environment, forming more complex behaviour. This emergence of complexity can be nicely addressed with aspects. The resultant emergent behaviour is not a behaviour of certain aspects, nor a behaviour of the core modules. It is the combination of the two that triggers it.

Emergent behaviour also occurs when the number of interactions between aspects in a system increases, thus potentially allowing for the creation of many new types of behaviour. However, a large number of interactions of multiple aspects can in fact work against the emergence of interesting behaviour. In the next two subsections we provide simple illustrations on how to avoid such unexpected or unpredictable behaviour.

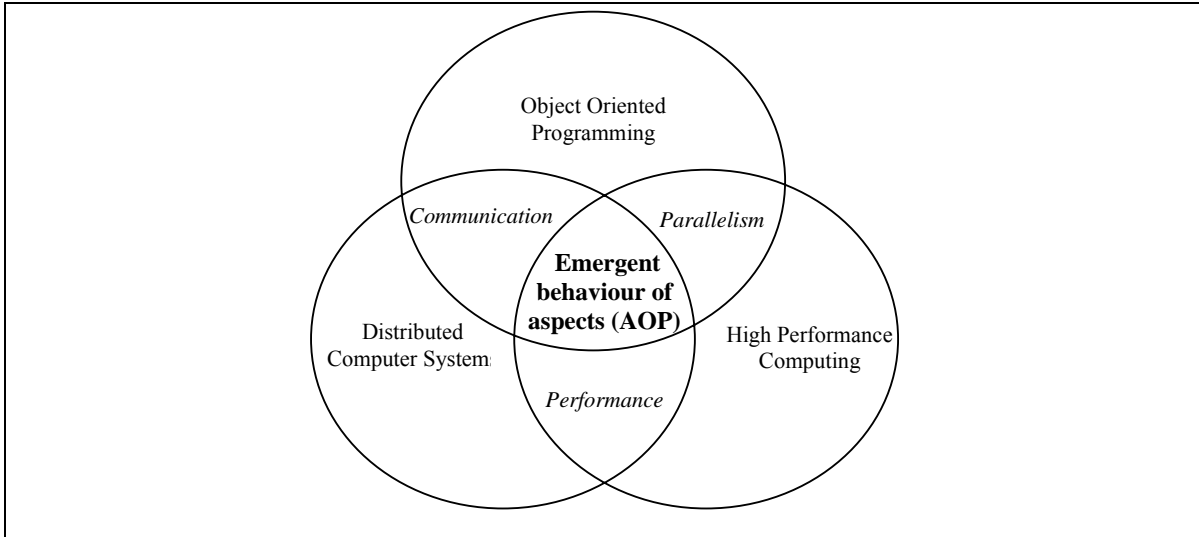


Figure 2. Emergent Behaviour of Aspects in DS and HPC

3.1 Aspects in High Performance Computing

Building a software system with high performance, high scalability and high security is a very big challenge. The main idea behind high performance systems is to use more than one computer or a computer with more than one processor to solve a problem. As a consequence, we expect to achieve greater computational speed. Areas requiring this 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. The problem of code tangling appears in many implementations of parallelised algorithms [Harbulot and Gurd 2004]. It is therefore important to investigate the use of AOP for parallel applications.

Simple examples of aspects here are tracing and performance analysis. In fact, for high performance computing applications, performance is an aspect that comes up very often. We need to deal with both communication and computation separately and sum the components to form the overall parallel time. For example, on entering a computationally intensive method - start the clock, and on leaving this method - stop the clock, and calculate the difference and display overall time performance (see Figure 3). A second concern is that high performance applications are not easy to understand and follow, due to their parallel nature, so developers often want to add some statements in the program to see what happens during execution. This kind of issue is a universal need in today's computing environment.

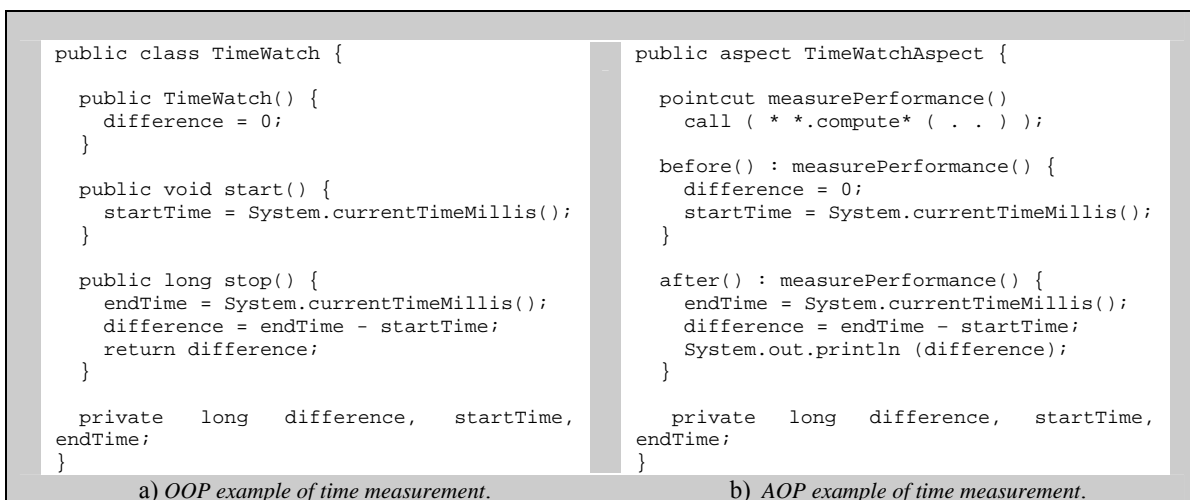


Figure 3. Implementation example of time concern for measuring performance.

Table 1 in section 2.1 provides certain aspects for an image processing domain. Here, we are going to have a look at parallel Mandelbrot Set application – an 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 and it is possible to spread the computation across several processors. Computation is embarrassingly parallel with respect to each pixel. This kind of computing task has two primary concerns: the underlying mathematical model (complex iterating function) and the high performance execution (measuring the time) of the resulting computation. The first concern is the computationally intensive part of the program. The core concern of an application will exactly consist of implementing this functionality. This will be encapsulated inside the class MandelbrotP.java.

The second concern is the measurement of execution time, a feature that is usually wanted when dealing with intensive computing algorithms, in order to see the benefits of parallelising. This execution time measuring of the parallel Mandelbrot set should not interfere with the core component. Figure 3 illustrates an implementation of this concern with Java and AspectJ. With a Java-based approach such a concern would typically require the user to insert code between the lines of the main algorithm. Thereafter, if the user wants to remove this concern he/she needs to modify the main algorithm again. On the other hand, by using AspectJ and aspects it is possible to introduce this concern of performance measurement in the main application, and then remove it - if necessary, without even modifying the main algorithm. This aspect, TimeWatchAspect.java, will measure the time at different points in execution of the program. It activates ‘startWatch’ before advice and ‘endWarch’ after advice. The aspect is activated any time when the method name starting with the word ‘compute’ is called. In addition to ‘startWatch’ and ‘endWatch’ we may want to incorporate some kind of a delay inside the application. This is illustrated in Figure 4. The delay is implemented as advice and can be switched on and off as required.

We still need to test this whole example in parallel, along with AspectJ, and we will need to use MPJ, which is a Java library for a message passing interface, dedicated to parallel execution. We will also try to investigate the possibility of adding MPI code within an aspect. A thorough investigation of aspects in HPC is provided in Harbulot [Harbulot 2002, Harbulot and Grud 2004].

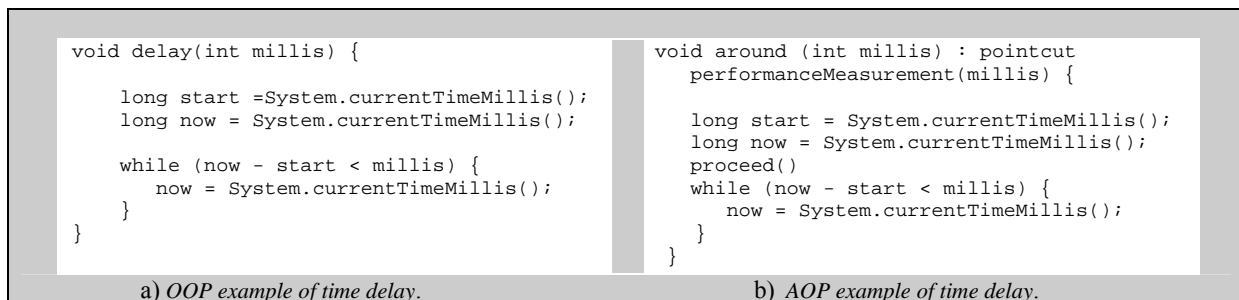


Figure. 4. Implementation example of delay feature

Sometimes, for educational purposes, we want to trace the execution of a parallel program to see what each processor is doing, which one is delayed, and at which point of execution, etc. For that reason, the developer might add some extra statements for logging, debugging and tracing purposes. These can be addressed nicely with aspects. However, certain unexpected and unpredictable behaviour can emerge. We may get poor performance results, due to interaction of these aspects with the performance aspect. The performance concern will not output valid performance results, because concerns like logging, tracing and debugging affect performance by introducing a non trivial input-output cost. To achieve desired behaviour, for all high performance applications, it is important to remove all tracing and debugging code before collecting any kind of performance results. This can be achieved by disabling certain aspects from the final product, which is always possible, as AOP has the ability to weave in and out any aspects from the final application.

In principle it is possible to use AOP to separate core mathematical code from performance measuring code. This section covered the aspect oriented implementation of two concerns. One development concern (debugging-tracing) and the other a production concern (performance-time). These two aspects cannot stand alone without the core concern, but they are not dependent on one another.

3.2 Aspects in Distributed Computer Systems

A distributed system is a collection of independent computers that appears to its users as a single coherent system [Tanenbaum 2002]. The most important issues that we need to consider when dealing with such systems are the following: communication, processes, naming, synchronization, consistency and replication, fault tolerance, and security.

Distributed computer systems are characterized by a substantial amount of complexity and therefore preservation of modularity is not possible for all concerns. In CORBA (industry-defined standard for DS) and RMI (remote method

invocation) a number of problems have not been solved elegantly within such a distributed framework. 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 [Laufer 2003]. They may be addressed with aspects, but with caution, as serious shortcomings could emerge in handling of the failure semantics in a distributed environment.

Table 1 in section 2.1 provides certain aspects that are identified for the distributed domain where the AOP approach should be useful. In addition, the most noticeable examples of crosscutting concerns for distributed applications are the following:

- *Synchronization* - implements all code for synchronization inside an aspect and distribute it to all existing classes that are responsible for supporting it.
- *Remote Access* - tends to cut across the implementation and could be factored out from all existing classes to an aspect.
- *Fault Tolerance* - when part of the system fails, a new behaviour emerges which uses certain aspects along with the rest of the system.
- *Security* - implements all code for security policy inside an aspect and distributes it to all existing classes that are responsible for enforcing that policy.

All of the above issues and particularly synchronization and security tend to cut across the implementation, and they are primary source of complexity in distributed applications. AOP may present the potential to address these issues appropriately and to enforce a formal separation between the main functionality and these issues in distributed applications.

For example, consider security as one of the most difficult principles in distributed systems. Security in such systems consists of many components including authentication, authorization, encryption and auditing. The first two components are closely related. Authentication verifies that you are a legitimate user, and authorization establishes access permissions for that user. In OOP you will perform authentication in one module, pass the authenticated users to modules that need authorization, and then those modules will perform the authorization [Laddad 2003]. With AOP we can create aspects addressing authentication and authorization and distribute them to all modules that need to perform these kinds of operations.

However, a combination of these aspects and interaction with different modules can exhibit an emergent behaviour that is not desirable. The order of executing aspects, applied to the same point in execution, is unpredictable. Authorization cannot be performed without first performing authentication, and therefore the authorization aspect must be executed after the authentication aspect. Also, we are not allowed to disable the authorization aspect, as authentication alone is not sufficient. To enforce the desired behaviour we must consider rules and ways to control precedence of an aspect. Unless you specify precedence, the order of execution is determined arbitrarily and may result in unexpected behaviour.

4. RESEARCH STUDY: ASPECTS IN ALGON

Algon (ALGORithms on the Net) is a research vehicle that drives our investigation of aspect orientation. Algon's main focus is to provide a framework for incorporating algorithmic software components into a distributed system. It incorporates performance comparisons of the various distributed algorithms, and even further supports the interchangeability of the middleware on which it runs. However, the design, and ultimately the implementation, of the system evolved in a rather ad hoc fashion. There is a need for better separation of concerns and the introduction of AOP.

Algon is entirely implemented in Java, due to Java's popularity and sustainability for the development of distributed applications, and provides classes which can be incorporated into new or existing application. Now, with AspectJ we can build on the existing Java application, as the language provides aspects which can be incorporated into such an application. Usage of the advanced features of the Java programming language, together with AspectJ constructs, will certainly contribute to the ideal separation of concerns.

Algon followed the approach of the traditional layered system architecture. The use of technology to isolate and encapsulate the concerns seems to make Aspect Oriented Programming the ideal alternative to a layered systems architecture. However, this aspectization has only limited application in the construction of distributed applications. Algon developers experienced that many kinds of concerns that the Algon system addresses are neither orthogonal nor non-functional. They argue that Algon functional requirements are wrapped up in the functionality of the application and the extra dimension of error that must be dealt with simply can not be adequately expressed in an orthogonal way [Worrall 2004].

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 algorithmic code with the application

code and when required – injecting the implementation of the distributed algorithm at numerous points within the source code. This injection process is conceptually similar to the insertion of timing code or logging code into the program.

It is important to note that there exist two distinct levels for aspect implementations in Algon. The first (and higher) level implementation is probably the reason for the current investigation – aspects can be used to integrate the Algon framework into an application. The second level of implementation includes implementing Algon (or parts of it) using AOP; separating out crosscutting concerns.

In the former case, it should be possible for an application to access a distributed resource in which case Algon is integrated with the resource in such a way as to hide the distribute nature of the resource, as well as concurrency-related concerns. Such an application does not currently exist, and will only be part of the next phase of Algon's implementation.

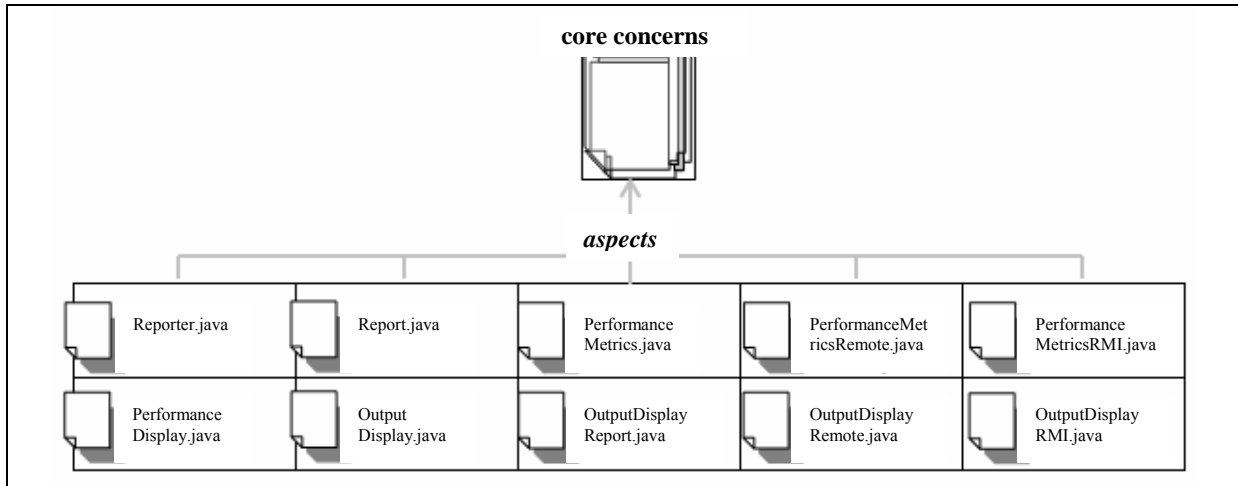


Figure 5. Performance concerns in the Algon system

In the second case, refer to Figure 5. A number of Java classes forming Algon were identified with the aim of bringing out crosscutting concerns. The greatest and most obvious of these concerns were introduced by the implementation of performance-related aspects associated with the system.

```

public void getRequestSet()
{
    UpdateQueue.uq.report(
        new OutputDisplayReport(algName, "Request set requested"));
    requestSet = this.configure();
    UpdateQueue.uq.report(
        new OutputDisplayReport(algName, "Configuration Complete"));
}

```

Figure 6. Code snippet (taken from Algorithm.java) illustrating performance-measuring code.

In Figure 6 a snippet from Algorithm.java is taken, illustrating performance-measuring code. A few other cases also exist. In MEScheduling.java we found time delay implementation, which can be placed inside an aspect as already shown in Figure 4 in section 3.1.

```

public static boolean VERBOSE = (System.getProperty("verbose") != null);
    .
    .
    .
    if (VERBOSE)
        System.out.println( . . . );
    .
    if (VERBOSE)
        System.out.println( . . . );

```

Figure 7. Some debugging information scattered around the Algon system.

Figure 7 highlights a further highly re-occurring scenario. The statement ‘System.out.println(...)’ is scattered across different modular components. There are certain logging APIs which are used over the statement System.out.println (), but AOP promise to be a better approach. This is because with AspectJ you do not need to modify classes. All you need to do is to add aspect to the system.

Figure 8 illustrates this behaviour in detail and provides an AOP approach. In traditional approaches the logging calls will be all over the core modules. On the other hand, with an AspectJ-based approach the aspect separates the core modules and logger object, where the created aspect simply weaves the logging invocations into the core modules when they are needed.

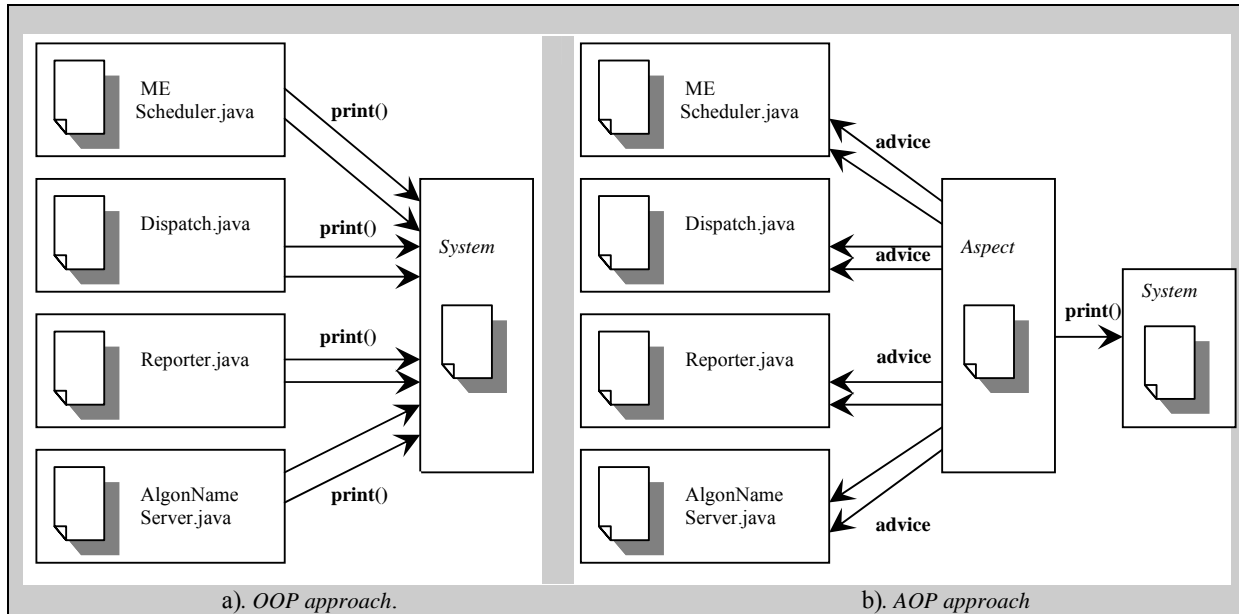


Figure 8. Difference between current and AOP approach in handling debugging information

Following this investigation, we conclude that the AOP approach has obvious and attractive 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.

5. FUTURE WORK

Investigation of aspects in Algon has been done at a conceptual level. The actual implementation of this approach will follow, and also further work is needed to re-design Algon to make use of aspects. As part of an industrially oriented project, with the Polelo Research Group, Algon is to be augmented with parallel algorithms. The new system will be known as parallel Algon or AlgonP. These algorithms require high speed computing and a framework for execution in parallel. Algorithms of this nature are also environment dependant and functionality is required to determine the best algorithm for a given environment. Based on discussion in this paper and aspects in HPC and Algon, there is a place for investigation of AlgonP to incorporate aspects.

Aspects are emerging in others research areas as well, such as aspects in concurrency, aspects in operating systems, aspects in databases and aspects for composition of web services.

6. CONCLUSION

Although AOP does not solve any new problems, we proposed situations and approaches where AOP can be useful for solving existing problems in a better and efficient way with much less time and effort. The main purpose was to stimulate thinking in terms of aspects along with objects and to promote wide acceptance of this new paradigm. By using AOP you will not only benefit from its advantages, but you will also become a better OO programmer.

We show that distributed computer systems and high performance computing are promising fields where aspects could really make a difference. We illustrated how we can use AOP techniques and weave performance concerns in high performance and distributed applications.

The research work focused on the question whether and how AOP can improve distributed application development by looking at Algon distributed framework. We only touched on the appearance and use of aspects in Algon in this paper and the further work is needed for detailed investigation.

7. ACKNOWLEDGMENTS

Many thanks to John Muller, former member of the Polelo Research Group, for his initial work on Aspect Oriented Algon which formed a solid basis for AOP research. The financial assistance of the National Research Foundation towards Algon research is hereby gratefully acknowledged.

8. ABOUT POLELO RESEARCH GROUP

The Polelo Research Group, under Prof Judith Bishop, covers distributed systems, high-performance computing language understanding, web-based computing and software engineering. It is a part of the Department of Computer Science at the University of Pretoria. This specific project, on the Investigation and Practice of Aspect Orientation, is part of research being undertaken at the Honours level.

9. REFERENCES

- ASPECTJ TEAM, *web site for aspectj language*, www.aspectj.org
- ASPECT ORIENTED SOFTWARE DEVELOPMENT, *web site*, www.aosd.net
- BISHOP, J., RENAUD, K., AND WORRALL, B. 2002. Composition of Distributed Software with Algon – Concepts and Possibilities. In *Proceedings of Software Components*, Grenoble, Electronic Notes in Computer Science No: 65.
- DRIVER, C., AND CLARKE, S. 2003. Distributed Systems Development: Can we Enhance Evolution by using AspectJ?, *Proceedings of the 9th International Conference on Object Oriented Information Systems*, Geneva, Switzerland, September 2003. Springer: Lecture Notes in Computer Science, Vol 2817.
- FRANCE, R., RAY, I., AND GHOSH, S. 2004. Aspect Oriented Approach to early design modelling, *IEE Proceedings Proc.-Softw.*, Vol. 151, No. 4 August 2004
- GABOR, L., AND MURPHY, J. 2004. Using Aspect Oriented Programming for performance improving of J2EE applications, *Buletinul Stiintific al Universitatii 'Politehnica' din Timisoara*, Romania, Periodica Politehnica, Transactions on Automatic Control and Computer Science, Vol.49 (63), 2004, ISSN 1224-600X
- HARBULOT, B. 2002. An Investigation of Aspect Oriented Programming, *Msc Thesis*, Department of Computer Science, University of Manchester, England
- HARBULOT, B., AND GURD, J.R. 2004. Using AspectJ to Separate Concerns in Parallel Scientific Java Code, Published in the *Proceedings of the 3rd international conference on Aspect – Oriented Software Development (AOSD)*, March 2004
- LADDAD, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, Greenwich, CT.
- LOPES, C.V., AND KICZALES, G. 1997 D: A Language Framework for Distributed Programming, Lopes - *PhD Thesis*, College of Computer Science, Northeastern University.
- LAUFER, K., THIRUVATHUKAL, G.K., AND ELRAD, T. 2003. Enhancing the CS Curriculum with Aspect Oriented Software Development (AOSD), *Working Paper*, September 2003.
- KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.M., AND IRWIN, J. 1997. Aspect Oriented Programming, In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP '97 - Object Oriented Programming 11th European Conference*, Finland.
- KIENZLE, J., AND GUERRAOU, R. 2002. AOP: Does it make sense? The case of concurrency and failures, *Proceedings of ECOOP – Object Oriented Programming the 16th European Conference*, June 2002
- PULVERMULLER, E., KLAEREN, H., AND SPECK, A. 1999. Aspects in Distributed Environments, *Proceedings of the GCSE*, Sep 1999.
- RENAUD, K., BISHOP, J., LO, J., VAN ZYL, P., AND WORRALL, B. 2003. A Framework for Supporting Comparison of Distributed Algorithm Performance. In *Proceedings of 11th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2003)*, February, Genoa, Italy
- RENAUD, K., BISHOP, J., LO, J., AND WORRALL, B. 2004. Algon: From Interchangeable distributed algorithms to interchangeable middleware. In *Proceedings of Software Composition 2004*, Aprile, Barcelona, Spain
- TANENBAUM, A.S., VAN STEEN, M. 2002. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ.
- VAN ROY, P. 2004. Aspect Oriented Programming for Distributed Systems: its use, its effect on language design, and its limits, *Belgian Symposium and Contact Day on Aspect Oriented Programming and Software Evolution*, May 2004
- WIKIPEDIA, *the free encyclopedia - Emergence*, en.wikipedia.org/wiki/Emergent_behaviour
- WILKINSON, B., AND ALLEN, M. 2005. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Prentice Hall, Upper Saddle River, NJ.
- WORRALL, B., RENAUD, K., AND BISHOP, J. 2004. Limitations of the Aspect-Oriented Approach in Distributed System Architectures, Polelo Technical Report, July 2004