Algon: Supporting Distributed System Programmers

K V Renaud[a] J M Bishop[b] B Worrall[b] J Lo[b] P van Zyl[b]

[a] *University of Glasgow.* karen@dcs.gla.ac.uk
[b] *University of Pretoria.* {jbishop,bworrall,jlo}@cs.up.ac.za

Abstract

Programming in a distributed environment is a complex activity. Programmers need to be aware of issues unrelated to their domain of expertise, and are often unprepared for the challenges that distribution brings. Chief among these are issues of the application of distributed algorithms in differing distributed systems. We have tackled this problem by adopting the technique of separation of concerns. We have developed a framework called Algon which enables complex distributed algorithms to be incorporated into existing or new distributed systems as required. For each type of algorithm, a scheduler is defined that handles the communication between distributed sites, and can be instantiated together with any one of a range of suitable algorithms that solve the problem. Algon therefore frees ordinary programmers from needing to know the intricacies of complex distributed algorithms.
Furthermore, Algon supports the evaluation of different algorithms in the distributed system by making it possible to exchange algorithms without recompilation in order to support comparisons. Algon also provides a performance visualisation mechanism which the programmer can use to compare the performance of different algorithms so that the algorithm which best suits the particular distributed system is used.
This paper concentrates on the use of Algon by the application programmer. We explain how the Algon framework operates and how the programmer can use it to interchange and compare algorithms.
Keywords: *Distributed Systems, Distributed Algorithms, Interchangeability, Components, Middleware Independence, Separation of Concerns, Programmer tool*
Computing Review Categories: *D.2.7, D.2.13, D.2.12, C.1.4, C.2.4, D.1.3, D.4.7*

1 Introduction

Many programmers have been trained to develop on workstations and find the complexity of the distributed paradigm hard to handle. Middleware technologies hide much of the detail involved in achieving language interoperability and simplify maintenance, but they also introduce easily missed complexities that isolated systems seldom exhibit [7]. In addition to the normal cognitive and abstract nature of the programming activity itself, concerns in a distributed environment include, for example, nondeterminism, contention and synchronisation issues [10]. Programmers may be faced with a need to guarantee distributed mutual exclusion, achieve distributed termination, or detect deadlock, for example. There already exists a rich base of research from the 1980s and 1990s for solving such problems. A range of algorithms has been classified according to their function and each algorithm achieves the expected result in a different way and with different performance characteristics. The exact implementation of the algorithms in a particular programming language is often left unspecified [22, 23]. The programmer is therefore faced with

1. deciding which algorithm is best to use, and
2. actually implementing the algorithm in a distributed fashion.

While many aspects of programming distributed applications are challenging, one of the essential issues that many programmers find most difficult to deal with is the incorporation of required distributed algorithms into their systems. For example, one of the simplest algorithms for guaranteeing mutual exclusion, Ricart-Agrawala [21], involves:

1. sites sending requests and replies to other participating sites,
2. comparing timestamps, and
3. keeping queues of waiting sites.

Other algorithms providing better performance have even greater complexity. Although systems exist for illustrating and comparing the functioning of algorithms [2, 4, 9], their primary function is educational and they are not intended to be components for development.

Rather than trying to educate the vast number of programmers involved in developing distributed systems about the functioning, performance and complexity of various algorithms, we propose that the *separation of concerns* technique be applied [17, 11]. Separation of concerns simplifies the programmer's task by enabling him or her to deal with various aspects of the programming process *separately*. Programmers can then concentrate on

specific tasks individually, and remove difficult and complex tasks from their realm of responsibility and control. Separation of concerns also allows the programmer to decompose software into smaller, more manageable parts that are easier to keep up to date with evolving needs [16]. This technique has been applied to other aspects of distributed applications [6], and to separating algorithms from parallelism[25] but not as yet to distributed algorithms.

We propose that as much complexity related to distributed algorithmics as possible be hidden from the programmer in its own component level. This is achieved within a framework called Algon[1], which includes:

- a *library* of algorithms to be used as and when required;
- a *framework* for incorporating an algorithm into the system;
- a *tool* for evaluating different algorithms based on their performance within the distributed application.

Algon provides programmers with a variety of distributed algorithms so that they can experiment and thereby select the algorithm with the best performance for their particular application. The Java programming language, and its use in distributed systems, is the focus of this paper, mainly due to its popularity and suitability for the development of distributed applications.

Algon is implemented entirely in Java, due to its popularity and suitability for the development of distributed applications, and provides classes which can be incorporated into new or existing programs. The Algon concept and its associated design pattern was first proposed in [3]. It has since been successfully implemented, and a performance comparison tool has been developed [20]. The purpose of this paper is primarily to discuss the use of the Algon framework by Java application programmers and to release the system for general use.

Several advanced features of the Java programming language were used, which contributed to the success of the ideal of separation of concerns, and these will be described in the rest of the paper. Section 2 describes the architecture and general design rationale. Section 3 describes the mechanisms used to make the middleware level interchangeable. Section 4 explains how Algon is used to support decisions about algorithm choice. Section 5 considers related work, and Section 6 concludes.

2 The Algon Concept

2.1 Basic Infrastructure

The abstract architecture for Algon-extended application systems is shown in Figure 1. The application-specific code, P uses an interface to the component, C, using the classic algorithm type, In order to distribute C's behaviour, the system is extended by adding:

[1] Algon stands for *Algorithms On the Net*.

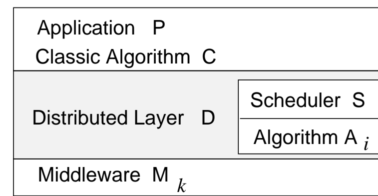

Figure 1: *Basic Algon Infrastructure*

1. a distribution layer, D, which consists of:
   (a) a scheduler S, and
   (b) an algorithm A_i.

   The distribution layer, D, is selected specifically to match the family of algorithms represented by A_i.
2. a middleware backbone M_k, which facilitates communication with other participants. It can use any suitable communication structure such as Java RMI, CORBA IIOP, DCOM or .NET.

For a specific classic distributed algorithm i, the interface I_i is used by the scheduler to interact with all algorithms implementing that interface. This makes it easier to introduce new algorithms and to specify, at runtime, the algorithm that should be used.

For example, say the system has four nodes, and these nodes need to all access a shared resource — some reading and some writing. This obviously calls for the use of a distributed mutual exclusion algorithm. The component, C, would be a ReaderWriter component. The Scheduler, S, would be the MEScheduler — the Mutual Exclusion Scheduler. The algorithm A_i could possibly be the Maekawa or the Ricart-Agrawala algorithms — depending on the algorithm the system developer specifies should be used.

When the abstract architecture was implemented it became obvious that some auxilliary components were required in the system. These components address two problems:

1. The participating applications, with their algorithms, need to be identified uniquely to all other applications in order for them to be able to construct the request sets necessary for their correct functioning.
2. The Algon system aims to directly support monitoring of the system. The developer needs to be able to monitor system activity from a central point. It is necessary to provide the developer with a central display and control interface so that he/she can not only observe but also directly *control* activity in the system.

The first problem can be addressed either statically or dynamically. Assigning a name to each application statically is the simple solution, but this is not realistic in an industrial setting. Therefore we had to assign unique names dynamically. The obvious solution here is to use a Name Server, and the `AlgonNameServer`, shown in Figure 2, was created to serve this purpose. An application using Algon

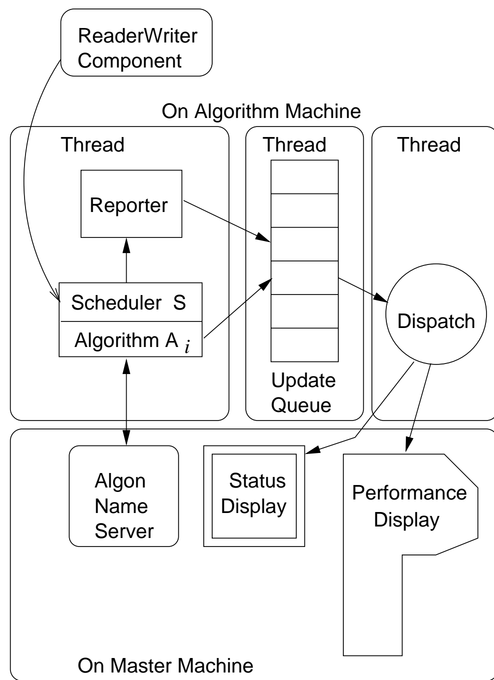

Figure 2: *Minimising the Effect of Performance Measurement*

will instantiate a component, *C*. This component will instantiate the Scheduler, *S*, which will instantiate the algorithm, and then register itself with the `AlgonNameServer`. The name server will store the information about the participating application/algorithm and the IP address it is running on, and then assign a unique identifier to the application.

When the applications want to start using *C*, they will ask the `AlgonNameServer` for the identifiers and IP addresses of the other participating applications and then start constructing request sets with those identifiers.

The second problem is addressed by having an `OutputDisplay` class. This is instantiated and bound by the `AlgonNameServer` so that all applications can send status information to one central output display, shown in Figure 3.

2.2 Performance Measurement

The output display mentioned in the previous section provides the distributed system developer with a coarsely-grained snapshot of system activity but it cannot provide any data to support realistic comparison between algorithms. The algon performance measurement component was created to this end. In order to ensure that the measurement of system performance was done with minimal impact on the application a detached queueing system was used, as shown in Figure 2.

Algon reporting makes use of three classes — `Reporter`, `UpdateQueue` and `Dispatch`. The `Reporter` class is twinned with the `Scheduler` and the `Scheduler` reports on all interaction with the algorithm to the `Reporter`. The `Reporter` maintains all status information

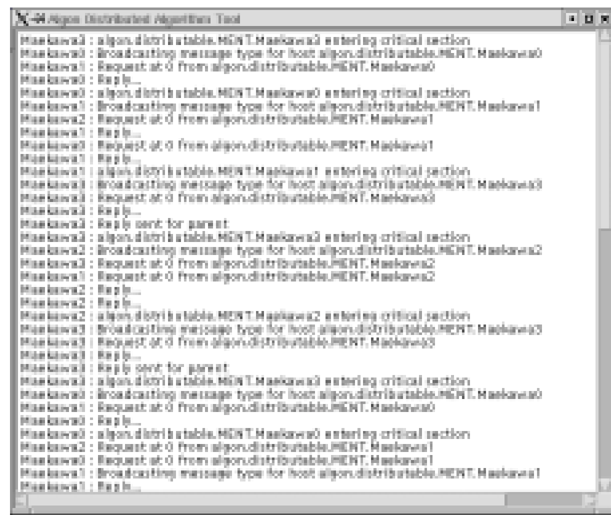

Figure 3: *Output Display*

and constructs reports to be sent to the performance display tool. The reports are inserted into the `UpdateQueue`, which runs in a separate thread so that the `Reporter`, having placed the report on the queue, can directly return control to the `Scheduler`. The `Dispatch` class watches the queue and when it detects a new report it removes it and sends the report to the performance display tool. The reports to the output display are also routed via the queue to minimise the negative impact of the reporting on the performance. This dissociates the time-consuming contact with the performance display tool from the `Scheduler's` activity which impacts directly on the application's performance.

The performance display is shown in Figure 4.

2.3 Dealing with Algon Failure

The failure semantics of distributed systems is different from centralised systems — hence the need for distributed algorithms. Distributed algorithms have been developed specifically to cope with such failures, and will report such failures in the Algon system by means of thrown Exceptions. However, it is important that the Algon layers do not introduce a whole new family of exceptions into the system caused by a failure of one of its components. The system could fail in the following ways:

1. The Performance Display component: The system is protected from the failure of this component since the `Dispatch` class will not report any exceptions it deals with, but will simply stop sending reports to the tool. The functioning of the `Scheduler`, which reports to the tool via the `Reporter` and `UpdateQueue`, will not be affected by this failure. The `Dispatch` class will continue to remove reports from the queue and then discard them so that the application will not be disrupted by the failure.

2. The failure of the `OutputDisplay`, while defeating the purpose of Algon, will not cause the application to fail.

Figure 4: *Performance Display Tool*

Once again the status reports will merely cease to be available.

3. The failure of the `AlgonNameServer` during system setup will cause the system to fail but failure after the initial setup will not affect the system in any way. We are aware that this single point of failure may be seen as a weakness in the system. It is, as always in distributed systems, necessary to weigh up the advantages of having a dynamic registration mechanism with a single point of failure against a static inflexible failure-resistant naming system. We felt the former to be the better design choice. It is a relatively simple matter to replicate the `AlgonNameServer`, and this will be done if the need arises.

The whole Algon distributed layer and associated components have been developed with the philosophy that if an exception is caught the system will try to continue to function regardless so as to not interfere with the functioning of the application. Algon will therefore run with reduced Algon functionality in order not to sabotage the continued running of the controlling application.

3 The Middleware Layer

In our initial implementation we used Java RMI as the middleware layer. Our intention was to concentrate on interchangeability of algorithms and providing the programmer with a visualisation of the performance of these algorithms before focusing on the comparison possibilities of the middleware layer that Algon could support.

It is desirable to provide a middleware-independent core of Algon classes because that makes it far simpler to extend Algon by incorporating another middleware implementation. This is important because it is useful to understand the impact the middleware is having on the system

Making Algon middleware-independent is no easy task. To explain the difficulty consider the use of `java.rmi` as the middleware layer. If we wish to use the java tool `rmic` to facilitate remote invocation of objects representing distributed algorithms the stub class has to extend the `java.rmi.server.UnicastRemoteObject` class. This provides the necessary server semantics required in order to support remote invocation. It also has to implement the `java.rmi.Remote` interface. All methods of any class implementing the `java.rmi.Remote` interface have to throw the `java.rmi.Remote` exception. This is an interesting and crucial design decision by Java's designers.

By giving remote invocation similar semantics to local invocation in Java the designers have attempted to provide a measure of distribution independence. Parameter passing in Java happens in one of two ways: call-by-reference for object instances and call-by-value for primitive types. Remote invocation adds another way — call by deep copy — provided by object serialization. Since Java attempts to hide the remote nature of the invocation the programmer does not always know exactly how the parameters are being passed, something he/she should know in order to refine code and make it more efficient.

The attempt to make the remoteness of the invocation transparent is bound to fail, however, because of the very real difference in the failure semantics of local and remote invocation. The programmer is therefore forced to make provision for possible failure of distributed components of his system, or the network linking him with that system, for remote invocations. Java forces the programmer to do this by ensuring that `java.rmi.Remote` exceptions are thrown by all methods in a class which will be used remotely.

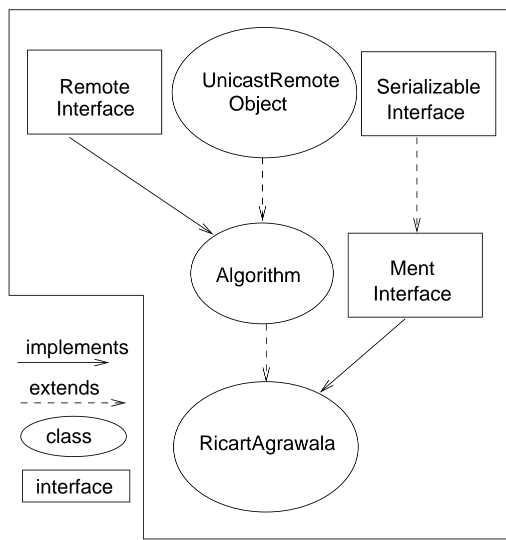

Figure 5: *Algorithm Inheritance Structure*

3.1 Using the Mutual Exclusion Interface

The inheritance structure of the mutual exclusion algorithms used in Algon is shown in Figure 5. Each algorithm implements some interface which represents all algorithms of a specific functionality type. For example, the `Ment` interface will be implemented by all `Mutual Exclusion No Token` algorithms. The Ment interface is defined as follows:

```
public interface Ment extends Serializable {

  public void sendRequests(SchedulerInterface si) ;
  public void reply() ;
  public void request(long time,
    SchedulerInterface si, Ment m) ;
  public void getRequestSet() ;
  public void sendRelease() ;
  public void release() ;
}
```

In the original Algon system the Algorithm class simply extended the `java.rmi.server.UnicastRemoteObject` class. This ensured that the algorithms could have stubs generated for them and those could be distributed via the `java.rmi` middleware infrastructure. However, if we want to make the system middleware independent we can no longer have a `java.rmi`-specific implementation at such a high level in the system. We need to rather become middleware-specific at as low a level as possible and to generalise the upper inheritance structure.

3.2 Making Ment Middleware Independent

This is not as simple as it appears. It would appear that the solution would be to have middleware-specific implementations of each algorithm, which simply extends the inheritance structure as shown in Figure 6. However, this solution is flawed for two reasons:

1. Multiple implementation inheritance is not permitted in Java, therefore `RicartAgrawalaRmi` is not permitted to inherit from both `RicartAgrawala` *and* `UnicastRemoteObject`.

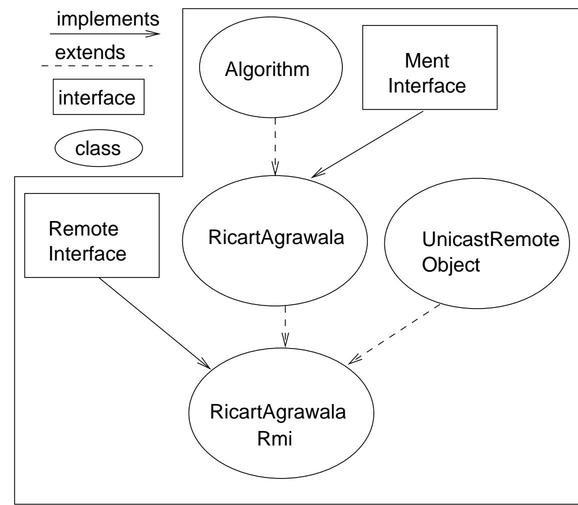

Figure 6: *Extended Algorithm Inheritance Structure*

2. Java's exception handling mechanism has two provisos:

   (a) You *must* declare any exceptions that may be thrown and such a declaration must be part of a method signature.

   (b) A subclass which is overriding a method that throws an exception must declare that it throws that exception too — either the same exception, or a subclass of that exception [14]. On the other hand, a method in a subclass cannot throw an exception unless its superclass method throws an exception of the same type. This feature has both advantages and disadvantages. On the one hand it is valuable to know what exceptions could be thrown by the methods of a Java class. On the other hand, the current restrictions prevent programmers from easily adapting systems in response to evolving needs.

   `RicartAgrawalaRmi` must thus throw at least a `RemoteException` from all methods since it implements the `Remote` interface. Therefore the methods it overrides in `RicartAgrawala` must also throw a `RemoteException`, but this would defeat the desired middleware-independence of `RicartAgrawala`.

The only solution to the inheritance problem is to make use of delegation rather than inheritance. Thus a pairwise inheritance structure was established, where each algorithm has a middleware-dependent place-holder which is used as a stub class in the middleware and relays all algorithm-specific calls to the actual algorithm, as shown in Figure 7.

This deals with the multiple inheritance problem, but does not deal with the exception-handling problem, since both the middleware-independent algorithm

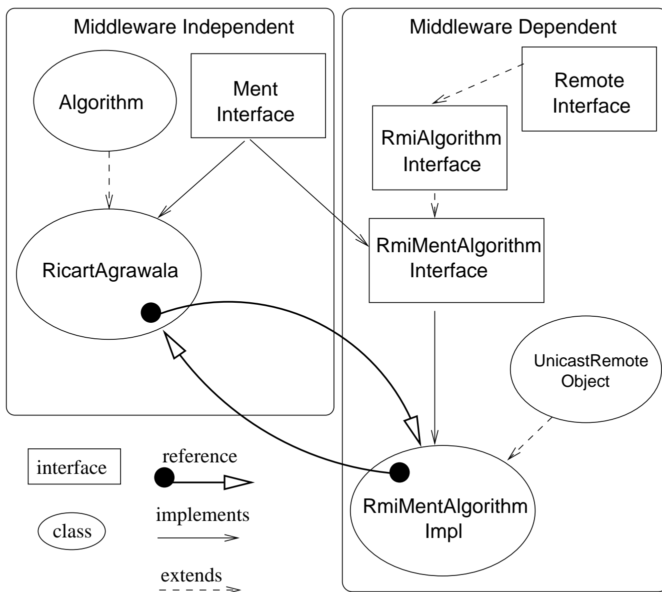

Figure 7: *Pairwise Middleware-Independent Algorithm Structure*

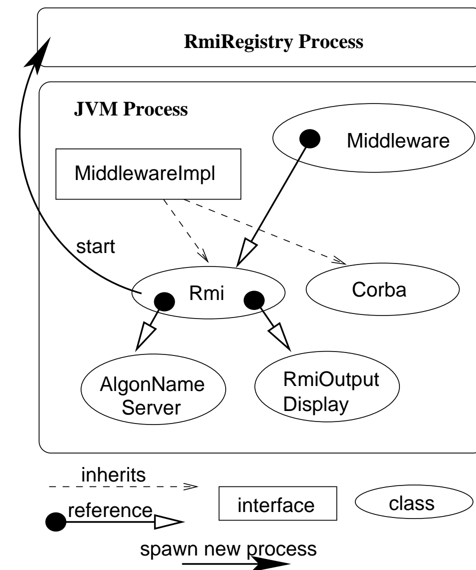

Figure 8: *Algon Middleware Layer*

`RicartAgrawala`, and the middleware-dependent algorithm, `RicartAgrawalaRmi`, must implement the `Ment` interface and since `RicartAgrawalaRmi` must throw `RemoteExceptions` that means that `Ment` also has to throw `RemoteExceptions`, once again defeating the middleware independence of the algorithm.

It was found that by declaring that all methods in the `Ment` interface throw an exception of type `Exception`, it was possible to override this at the middleware-dependent level by having all methods throw middleware-dependent exceptions, such as `RemoteExceptions` in the case of `java.rmi`. This is because `RemoteException` is a subtype of `Exception`;

Algon was then extended by adding a `Middleware` interface which could be used by the system to invoke middleware-type functions. This interface provides Algon with a uniform way of discovering, accessing and manipulating algorithm instances. This layer is illustrated in Figure 8. Algon detects the user's middleware preference, instantiates the specific middleware implementation and provides a static reference which can be used by any class in Algon that needs to use middleware functionality. During instantiation all the middleware-specific components will be started up — such as the RmiRegsitry in the case of `java.rmi`, for example. This frees the Algon user from the minute details required to ensure that all components that the specific middleware implementation requires are in place.

3.3 Simplifying the Process

The final step in making Algon middleware-independent was to define a new exception, called `MiddlewareException`, which could be used by all middleware layers to throw middleware-related exceptions.

Since the place holder classes are essentially middleware-dependent proxies for the actual algorithms they are very simple to generate automatically. Algon therefore provides a tool for `java.rmi` called `AlgonRmic` which generates the place holder for an interface, such as `Ment` for example, compiles it, and runs `rmic` on it so that the stubs and skeletons are automatically ready for use by any algorithm implementing that interface.

4 Use of Algon

Two mechanisms are used to tailor runtime behaviour:

- Configuration files are used to specify information that cannot be discovered from the runtime environment. The configuration file `hosts.prop` holds the following information:
  - The IP address of the master site, where each application algorithm can expect to find the Algon Name Server.
  - The number of algorithms that should be participating in the system.
  - The class name of the algorithm that the system should be using.

  An example is:

  ```
  # hosts.prop
  # Thurs 13 March 2003
  Master=163.20.213.1
  numAlgs=4
  Algorithm=algon.distributable.ment.Maekawa
  ```

- Runtime variables are used to specify behaviour that is tailorable per site. For example, if the developer wishes to obtain more information about the runtime acitivity of Algon at the site the application is running on, the application is started as follows:

```
java -Dverbose App
```

The following settings are available:

– generation of status output statements to the command line to signal critical changes in the system and assist debugging.
– choice of whether to measure performance or not, displayed as shown in Figure 4.
– which middleware is to be used
– whether the Algon output should be sent to the command line or to the graphical user interface, as shown in Figure 3.

5 Related Work

Classifying Algon in order to draw comparisons with related work is not trivial. It has some similarities to reflective systems, and certainly applies separation of concerns techniques. It is also a very specialised programmer tool. A reflective system typically reasons about itself, and then acts upon itself based upon such reasoning [13]. This definition has been applied fairly loosely to many different systems. Reflective systems are composed of a base level and a meta-level, with changes at the meta-level causing changes to the base-level's behaviour [26]. Algon cannot be classified as a reflective system since it does not react to its own behaviour, but rather to a runtime configuration setting. The configuration file cannot really be classified as a meta-object. Thus we cannot compare Algon to reflective systems. We therefore classify Algon as a *tool* which applies a *separation of concerns* technique to *algorithmic concerns*.

Some research has been done into providing programmers with tools which separate behavioural features of software from functional features [6, 8]. The technique has been applied to a variety of different concerns, including real-time constraints [1], distribution and replication [6], exception handling [5], location control [15] and synchronisation [12]. 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 object [8, 19].

2. treating the concern as being orthogonal and freeing the programmer completely from it [18].

3. providing the programmer with a library which encapsulates the complexity [6]. The library typically contains functions which can be invoked by the programmer when required.

The first approach is usually done for reflective purposes but we do not feel that Algon is adding reflective capabilities to the system. 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 [6]. Garf provides the programmer with an extensible library for adding behavioural features to distributed programs. Garf, whilst being innovative for its time, 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 and rather than by providing a library of functions to be used blindly, it recognises the differing nature of distributed systems and offers programmers the capacity to tailor their systems accordingly.

6 Conclusions

This paper has described a new programmer tool which enables the comparison of algorithm performance by everyday distributed system programmers. The tool does not require a knowledge of complex algorithms, and algorithms are interchanged with a minimum of effort. Algon is available for download at URL `http://www.cs.up.ac.za/algon`.

References

[1] M. Aksit, J. Bosch, W. van der Sterren, and L. Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters. In *[24]*, pages 386–407, 1994.

[2] M. Ben-Ari. Interactive Execution of Distributed Algorithms. *ACM Journal of Educational Resources in Computing*, 1(2es), Summer 2001.

[3] J. M. Bishop, K. V. Renaud, and B. Worrall. Composition of Distributed Software with Algon — Concepts and Possibilities. In *Workshop on Software Composition. SC 2002.*, Grenoble, France, April 6-14 2002. ETAPS.

[4] S. Burdette, T. Camp, and B. Bynum. Distributed BACI: A Toolkit for Distributed Applications. *Concurrency and Computation: Practice and Experience*, 12(1):35–52, January 2000.

[5] C. Dellarocas. Toward Exception Handling Infrastructures for Component-Based Systems. In *International Workshop on Component-Based Software Engineering. Proceedings of the 1998 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Kyoto, Japan, April 25-26 1998.

[6] R. Guerraoui, B. Garbinato, and K. R. Mazouni. Garf: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4):32–39, Oct./Dec. 1997.

[7] N. Kaveh and W. Emmerich. Deadlock Detection in Distributed Object Systems. In V. Gruhn, editor, *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, Vienna, Austria, Sept. 2001.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *European*

Conference on Object-Oriented Programming. ECOOP 2001, Budapest, Hungary, June 2001.

[9] B. Kolehofe, M. Papatriantifilou, and P. Tsigas. Distributed Algorithm Visualisation for Educational Purposes. In *4th SIGCSE/SIGCUE Conference. ITiCSE'99*, Cracow, Poland, June 1999.

[10] J. Kramer. Distributed Software Engineering. In *16th ICSE Conference*, Sorrento, Italy, May 1994. Invited State of the Art Report.

[11] C. Lopes and W. Hursch. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, February, 1995.

[12] J. Lu, M. Zhang, M. Xu, and D. Yang. A two-layered class approach for the reuse of sychnronization code. *Information and Software Technology*, 43:287–294, 2001.

[13] P. Maes. Concepts and Experiments in Computational Reflection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '87*, pages 147–155, Orlando, Florida, 1987, 1987.

[14] A. Mikhailove and A. Romanovsky. Supporting evoltuion of interface exceptions. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques.*, number 2022 in Lecture Notes in Computer Science, pages 94–110, 2001.

[15] H. Okamura and Y. Ishikawa. Object Location Control using Meta-level Programming. In *[24]*, pages 299–319, 1994.

[16] H. Ossher and P. Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, October 2001.

[17] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[18] T. Prentezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, Department of Computing Science, University of Glasgow, May 2000.

[19] K. Renaud. JavaCloak: Considering the Limitations of Proxies for Runtime Specialisation. In *Proceedings of the 2001 Annual Conference of Computer Scientists and Information Technologists*, Pretoria, South Africa, Sept. 2001.

[20] K. Renaud, J. Lo, J. Bishop, P. van Zyl, and B. Worrall. Algon: A Framework for Supporting Comparison of Distributed Algorithm Performance. In *Proceedings of the Eleventh Euromicro Conference on Parallel. Distributed and Network-Based Processing*, pages 425–432, Genoa, Italy, 5-7 February 2003.

[21] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion Algorithms. *Communications of the ACM*, 24(1):9–17, Jan. 1981.

[22] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw Hill, 1994.

[23] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.

[24] M. Tokoro and R. Pareschi, editors. *Object-Oriented Programming, Proceedings of the 8th European Conference ECOOP'94. Lecture Notes in Computer Science*, volume 821, Bologna, Italy, July 1994. Springer Verlag.

[25] P. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton-Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998.

[26] I. Welch and R. J. Stroud. Kava — Using Byte code Rewriting to add Behavioural Reflection to Java. In *COOTS '01 — Proceedings of USENIX Conference on Object-Oriented Technology*, San Antonio, Texas, USA, January 29 – February 2 2001.