Reflection, Self-Awareness and Self-Healing

Gordon S. Blair
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK
+44 1524 65201
gordon@comp.lancs.ac.uk

Geoff Coulson
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK
+44 1524 65201
geoff@comp.lancs.ac.uk

Lynne Blair
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK
+44 1524 65201
lb@comp.lancs.ac.uk

Hector Duran-Limon
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK
+44 1524 65201
duranlim@comp.lancs.ac.uk

Paul Grace
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK
+44 1524 65201
p.grace@lancaster.ac.uk

Rui Moreira
Computer Engineering Department
University Fernando Pessoa
4249-004 Porto, Portugal
+351 22 507 1300
rmoreira@ufp.pt

Nikos Parlavantzas
Computing Department
Lancaster University
Lancaster, LA1 4YR, UK
+44 1524 65201
parlavan@comp.lancs.ac.uk

ABSTRACT

There is a growing interest in the area of self-healing systems. Self-healing does however impose considerable demands on system infrastructures—especially in terms of openness and support for reconfigurability. This paper proposes that the self-awareness inherent in reflective technologies lends itself well to the construction of self-healing systems. In particular, the paper examines the support provides by the Open ORB reflective middleware technology for the construction of this increasingly important class of system.

Categories and Subject Descriptors
C.3.11 [**Software Architectures**]: Patterns (reflection).

General Terms
Design

Keywords
Middleware, reflection, self-awareness, self-healing.

1. INTRODUCTION

There is growing interest in the distributed systems community in the general area of self-repairing, self-healing or self-organizing software systems [1, 2]. This work is partially stimulated by industrial initiatives such as IBM's autonomic computing [3]. To quote from their web site, "IBM invites the world, our customers, competitors and colleagues to accept the Grand Challenge of building and deploying computing systems that regulate themselves and remove complexity from the lives of administrators and users". This is an extremely challenging and long-term vision but one that has considerable potential in terms of masking out failure or environmental changes, and also dealing more generally with the evolution of systems to changing user needs or platform capabilities. The approach is particularly attractive for emerging application domains such as mobility and ubiquitous computing.

Self-healing systems do however place particular demands on the underlying infrastructure. In this paper, we are particularly interested in the demands in terms of *openness*. In other words, to support the healing process, it is necessary to have access to various aspects of the system structure and to be able to reconfigure such aspects at run-time. It is also important that such changes do not endanger the overall integrity of the (running) system. More specifically, this paper explores the extent to which reflection, and its inherent property of self-awareness, provides natural support for self-healing systems. In particular, we investigate the Open ORB architecture developed at Lancaster University and discuss the potential of this reflective middleware technology to support self-healing systems.

The paper is structured as follows. Section 2 introduces the three technologies underpinning Open ORB, namely reflection, component technologies and component frameworks. Section 3 then presents Open ORB, highlighting its multi-model reflective architecture. Following this, section 4 discusses self-healing in Open ORB. In particular, the section examines support for self-adaptation in Open ORB and also considers 3 examples of self-healing systems. Finally, section 5 summarises the discussion and introduces some areas demanding further investigation.

2. BASELINE TECHNOLOGIES

2.1 Reflection

Reflection [4] is now widely adopted in language design, as witnessed for example by the Java Core Reflection API [5]. Reflection is also increasingly being applied to a variety of other areas including operating system design [6], concurrent languages [7] and increasingly distributed systems, e.g. as [8] or [9]. Crucially, there is now a growing community working on the area of reflective middleware [10].

The main motivation for this research is to overcome the "black-box" philosophy of many existing middleware platforms by providing more openness; and to achieve this in a *principled* (as opposed to ad-hoc) manner through a comprehensive reflective architecture [11]. The key to the approach is to offer a meta-interface, or *meta-object protocol (MOP),* supporting access to the engineering of the underlying platform. This MOP provides operations to inspect the internal details of a platform (*introspection*), and by exposing the underlying implementation, it is also possible to insert behaviour, e.g. quality of service monitors. In addition, the MOP typically provides operations to alter the underlying middleware (*adaptation*), e.g. changing the implementation of the underlying transport protocol to operate efficiently over a wireless link or inserting a filter to reduce the bandwidth requirements of a media stream.

More generally, middleware platforms typically offer two (complementary) styles of reflection:

- *Structural reflection* is concerned with the underlying structure of systems, e.g. in terms of the set of interfaces supported (cf. introspection in [5]). More advanced possibilities include support for adapting the behaviour of objects and *architectural reflection* [11]. In the latter approach, the MOP provides access to the architecture of the system, e.g. in terms of components and connectors.
- *Behavioural reflection* is concerned with activity in the underlying system, e.g. in terms of the arrival of invocations. Typical mechanisms include interceptors (as found in CORBA) and dynamic proxies in Java [5]. Some research has also been carried out on providing access to underlying resources and associated resource management [12].

A significant number of experimental platforms have now emerged including Open ORB [11] (Lancaster University), Dynamic TAO, LegORB and UIC [13] (all University of Illinois at Urbana-Champaign), Flexinet [14] (APM, Cambridge), Open CORBA [15] (Ecole des Mines de Nantes) and OOPP [16] (University of Tromsø).

2.2 Components

In parallel with the above developments, there has been increasing interest in the role of *components* in distributed systems. According to Szyperski [17], a component can be defined as "a unit of composition with contractually specified interfaces and explicit dependencies only". In addition, he states "a software component can be deployed independently and is subject to composition by third parties". A key part of this definition is the emphasis on *composition*; component technologies rely heavily on composition rather than inheritance for the construction of applications, thus avoiding the fragile base class problem (and the subsequent difficulties in terms of system evolution) [17]. To support third party composition, they also employ explicit contracts in terms of *provided* and *required* interfaces. The overall aim is to reduce time to market for new services through an emphasis on programming by assembly rather than software development (cf. manufacturing vs. engineering).

In terms of middleware, most emphasis has been given to *enterprise (or server-side) component technologies*, such as Enterprise Java Beans (EJB) or the CORBA Component Model (CCM). In such technologies, components execute inside a *container*, which provides *implicit* support for distribution in terms of support for transactions, security, persistence and resource management. This offers an important separation of concerns in the development of business applications; i.e. the application programmer can focus on the development and potential re-use of components to provide the necessary business logic, and a more "distribution-aware" developer can provide a container with the necessary non-functional properties. Containers also provide additional functionality including life-cycle management and component discovery.

2.3 Component Frameworks

The application of component frameworks forms the third key technology underpinning the OpenORB architecture. Component frameworks are defined by Szyperski as "collections of rules and interfaces that govern the interaction of a set of components plugged into them" [17]. Essentially, component frameworks are reusable architectures that embody domain-specific constraints and strategies for composing components. For example, in Open ORB we employ a protocol component framework that describes how protocol stacks can be assembled from "plugged-in" components.

The main contribution of component frameworks is that they provide a means of enforcing desired architectural properties and invariants by constraining the interactions among their plug-ins in a domain-relevant manner. The enforced properties can be both functional (e.g., how some functionality is decomposed among plug-ins) and extra-functional (e.g., modifiability or performance of plug-in assemblies). As additional benefits, component frameworks simplify component development through design reuse, enable lightweight components, and increase the system's understandability and maintainability.

Component frameworks in Open ORB play a twofold role. First, they help structuring the middleware architecture into a set of specialized and focused domains (e.g., composing communication protocols or distributed bindings), that are each based on a component framework. The component frameworks have clearly identified dependencies and can easily be recombined into new architectures. Second, component frameworks are used to constrain the scope of dynamic reconfigurations and ease the task of integrity maintenance. Specifically, assemblies of components conforming to a component framework are "reified" by components that expose component-framework specific meta-interfaces for reconfiguration. This has the advantage that the meta-interfaces can exploit the domain-specific knowledge embodied in the component framework to enforce a desired level of integrity across reconfiguration operations. Furthermore, the desired level of integrity and consistency can be suitably traded-off against the degree of afforded flexibility.

To give a concrete example, we employ a multimedia streaming component framework, which accepts media filter plug-ins. The associated meta-interface allows clients to reconfigure a media filter graph with minimum perceived disruption of the media stream by exploiting a (domain-specific) buffering mechanism. Component frameworks are closely related to the notion of architectural style, which has similarly been exploited to achieve style-specific adaptation for self-repairing systems [2].

2.4 Analysis

Our research indicates that reflection, component technologies and components frameworks are highly *complementary*. Reflection provides the necessary level of openness to access the underlying platform architecture whereas components provide an appropriate structuring mechanism. The compositional approach inherent in components also provides a clean basis on which to re-configure the underlying architecture. Finally, component frameworks have the potential to impose appropriate constraints on this adaptation process.

3. THE OPEN ORB ARCHITECTURE

3.1 Overall Approach

The overall goal Open ORB is to develop a more configurable and re-configurable middleware technology through a marriage of reflection, component technologies and component frameworks. In particular, Open ORB is structured as a set of (configurable) component frameworks and reflection is then used to discover the current structure and behaviour, and to enable selected changes at run-time. The end result is a flexible middleware technology that can be specialised for a range of application domains including mobile and ubiquitous computing, and real-time systems. We are also currently investigating if the techniques can be used in areas such as programmable networks and to support longer-term evolution of software in for example the banking sector.

One of the key aspects of the Open ORB architecture is its 'multi-model' approach to structuring meta-space. In particular, meta-space is partitioned into a number of complementary meta-space models covering both structural and behavioural aspects. The motivation of this approach is to provide a separation of concerns and hence to reduce the complexity of the overall meta-interface. This is particularly important in distributed systems given the wide range of concerns that must be considered (in comparison to the design of a single programming language for example). The structure of meta-space is captured by figure 1 below.

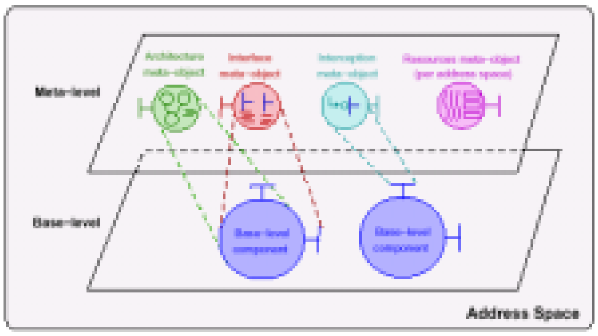

Figure 1: The structure of meta-space.

We consider this structure in more detail below.

3.2 The Meta-Space Models

3.2.1 Supporting Structural Reflection

In reflective systems, structural reflection is concerned with the content and structure of a given component [7]. In our architecture, this aspect of meta-space is represented by two distinct meta-models, namely the *interface* and *architecture* meta-models. These represent a separation of concerns between the external view of a component (i.e. its set of interfaces), and its internal construction (i.e. its software architecture).

The *interface meta-model* provides access to the external representation of a component in terms of the set of provided and required interfaces. In particular, it is possible to enumerate all provided (or required) interfaces offered by a given component, or to discover the type signature associated with a given interface. This meta-model therefore provides a capability similar to introspection facilities in the Java reflection API, allowing a programmer to interact with a dynamically discovered component.

The *architecture meta-model* then provides access to the implementation of the component as a software architecture, consisting of two key elements: a *component graph* and an associated set of *architectural constraints* (cf. components frameworks as introduced above). The concept of the component graph is central to this design, and is represented by a set of components (more specifically interfaces) connected together by *local bindings*, where a local binding represents a mapping between a required and provided interface in a single address space. Distribution can be added by introducing (distributed) binding components into the graph (cf. connectors in the software architecture literature). An extensible set of binding types is supported offering interaction models such as remote invocation, publish/subscribe, continuous media flows, group communication, etc. Normally this structure would be hidden from a user of a component. However, the architecture meta-model can be used to both discover and also adapt this structure at run-time.

If unconstrained, this is a rather dangerous approach to advocate. Consequently, we extend the software architecture to include a set of architectural constraints. A type management system offers one level of constraints, i.e. a new component must be a valid substitution of the old component (cf. subtyping of the respective interfaces). This is however not enough; it is also important to take a more global view of the architecture in determining the validity of adaptations. For example, changing a compression component may require a similar change to the peer decompression component. Similarly, it may be necessary to preserve a given architectural style over time such as pipes-and-filters. Our approach is to record such constraints explicitly in the architecture and to ensure that adaptations preserve the architectural rules before committing the changes (cf. atomic transactions).

Note that the approach described above is applied *recursively* in that a component within a component graph may itself have architecture, accessed via *its* architecture meta-model (i.e. at a meta-meta- level relative to the uppermost component. For example, a binding component within a graph may have a structure consisting of stubs and protocol components. This recursion terminates with *primitive components*, which have no visible underlying structure, and whose internal implementation details are inaccessible to the programmer.

3.2.2 Supporting Behavioural Reflection

Behavioural reflection focuses on activity in the underlying system [7]. More specifically, Open ORB distinguishes between actions taking place in the system, and the resources required to support such activity. These two aspects are represented by the *interception* and *resources* meta-models respectively.

The *interception meta-model* is arguably the most straightforward in the Open ORB design. In keeping with a number of reflective middleware proposals, this meta-model enables the dynamic insertion of *interceptors*. Such interceptors are associated with interfaces (more specifically, local bindings) and enable the insertion of pre- and post- behaviour. This applies equally to all styles of interface supported in Open ORB (operational, continuous media, etc). This mechanism is useful, for example, to dynamically introduce monitoring or accounting into a running system. Similarly, interceptors can be used to introduce additional non-functional behaviour, such as security checks or concurrency control.

The *resources meta-model* in contrast is quite unique to the Open ORB design, offering access to underlying resources and resource management [12]. We strongly believe that for many classes of application (including multimedia applications) it is just as important to be able to adapt resource usage and management policies as to evolve the basic structure of the system, e.g. when now operating in a mobile environment.

The resources meta-model is based around the abstractions of *resources* and *tasks*. Resources can be either primitive (e.g. raw memory or OS threads) or complex (e.g. buffers or user-level threads multiplexed on to kernel-level threads). They are created by *resource factories* and managed by *resource managers*, the latter typically building complex resources by adding value to, or combining, primitive resource instances. For example, a user level scheduler is a resource manager that builds user level threads from OS threads. Tasks are then the logical unit of activity in the system with the precise granularity varying from configuration to configuration. For example, there could be a single task dealing with the arrival, filtering and presentation of an incoming video stream, or alternatively this could be divided into a number of smaller tasks. Importantly, tasks can span component boundaries and are thus orthogonal to the structure of the system. Tasks are essentially the unit of resource allocation, i.e. tasks have a pool of resources to support their execution.

3.3 Implementation

Initial implementations of the Open ORB architecture were developed using Python, due to the support for rapid prototyping inherent in this language [18]. More recently, the architecture has been re-implemented with the explicit goal of provide a high performance implementation of our reflective middleware technology. To this end, we have defined a lightweight and efficient reflective component technology based on a subset of COM. The resultant *Open COM* technology is then used to construct configurable and re-configurable families of middleware. More specifically, a given middleware instance is constructed as a set of component frameworks. As an example, figure 2 illustrates our current implementation of a CORBA-compatible platform.

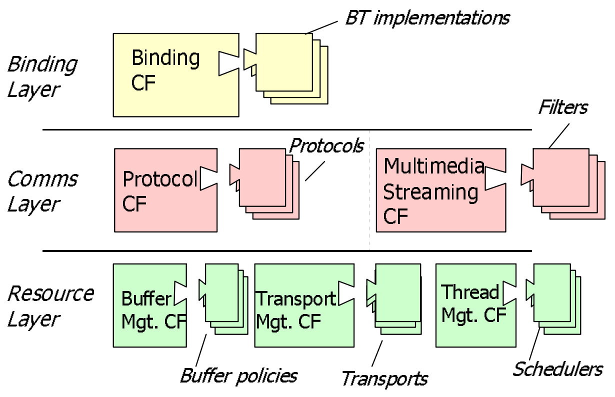

Figure 2: CORBA-compatible implementation of Open ORB.

Further details of this implementation can be found in [19].

4. OPEN ORB and SELF-HEALING

4.1 Self-Adaptation

Open ORB supports the ability to discover meta-information about the current system, both in terms of its structure and ongoing behaviour. These aspects can also be adapted by using the appropriate meta-interfaces. This however is not sufficient to support self-healing. There are essentially two approaches to adaptation that can be supported by this approach:

- Applications or system services can support monitoring and adaptation as an external service, or
- Components for monitoring and adaptation can be injected into meta-space to provide such a service.

It is the latter approach that is most interesting in terms of self-healing systems. We have previously explored an approach to such self-adaptation in Open ORB. In particular, we have developed styles of management component that can be introduced (dynamically) into the various meta-space models (see table 1 below).

Table 1. Styles of management component.

Monitoring	
Event Collector	Observe behaviour of underlying functional components and generate relevant QoS events.
Monitor	Collect QoS events and report abnormal behaviour to interested parties.
Control	
Strategy Selectors	Select an appropriate adaptation strategy (i.e. strategy activator) based on feedback from monitors.
Strategy Activators	Implement a particular strategy, e.g. by manipulating component graph.

Policies for monitoring and strategy selection are expressed as timed automata, which then map directly on to management components which then act as timed automata interpreters at run-time. They then interface to other components in the system using event notification, i.e. they register for events of interest, receive events, react to them and then emit events to interested parties (cf. reactive objects [20]). This use of timed automata also allows us to carry out formal analysis of the behaviour of the QoS management subsystem in isolation, and also when composed with a model of the rest of the system.

Further details can again be found in the literature [21].

4.2 Examples of Self-Adapting Systems

We present three contrasting examples to illustrate the potential of this reflective approach in supporting self-healing systems:

1. *Self-adaptive stream binding.* Through its general binding mechanism, Open ORB can support stream bindings representing continuous media flows in the system. Building on this capability, we have previously experimented with a self-adaptive audio binding. In the experiments, we use the architecture meta-model to gain access to the buffer component at the receiver end and then monitor when this buffer becomes either full or empty. Depending on the current context, we can then either increase the buffer size or change the transmission quality of the audio [21]. This example has also recently been extended to adapt both the audio transmission strategy and also the resource usage/ management (via the resources meta-model) thus illustrating how self-adaptation can span multiple meta-models [22].
2. *Self-adaptive mobile middleware.* One of the key requirements of mobile middleware is the ability to interoperate with nearby services and users [13]. Mobile applications and services are implemented on a range of middleware platforms (e.g. RPC, message-oriented and event-based), therefore, the middleware must be able to adapt itself to the current environment in order for interactions to continue, allowing classes of mobile applications to be developed independently of fixed middleware types. Therefore, we have developed a dynamically reconfigurable binding framework, which allows the middleware behaviour to change between SOAP, IIOP and publish-subscribe functionality. Furthermore, in order for self-adaptation to take place the middleware must be aware of its current context; that is, what types of services are currently available. For this purpose, we have implemented a service discovery framework, which can change between multiple service discovery personalities (e.g. SLP and UPnP), allowing all available services to be discovered. This information then drives the appropriate reconfiguration of the binding framework.
3. *Self-adaptive network architectures.* We have recently become interested in extending our approach to open/ programmable network architectures. Our approach here is to uniformly implement all layers of the programmable networking architecture—including the router's OS software, fast-path packet handling, per-flow/ per-application packet handling, and signaling—in terms of the Open COM / Open ORB technologies. For example, we use (optimised) Open COM components as fast-path schedulers, queues, etc; and we employ Open ORB as a signaling engine. This approach has the potential to endow programmable networks with extremely rich and comprehensive self-healing functionality. For example, we can apply generic self-monitoring and adaptation techniques (e.g. using reflection and the self-adaptation pattern of section 4.1) to areas as diverse as congestion management (e.g. adding a RED-based congestion manager to a router on the basis of congestion monitoring), and self-healing physical topologies (e.g. automatically managing redundant physical paths in mission critical disruption-prone networking environments).

The first two examples have been fully implemented, and work is currently ongoing to port the Open COM technology to a heterogeneous network of standard PC-based and Intel IXP1200-based programmable routers [23] to enable experimentation in the third area outlined above.

5. CONCLUSIONS

This paper has presented an analysis of reflection and its potential in supporting self-healing systems. Openness and reconfigurability are clear prerequisites for self-healing systems, and it is already well-recognised that reflective technologies enhance such properties of a system. Furthermore, the examples presented in section 5.2 above have strengthened our belief that reflection coupled with an appropriate framework for self-adaptation provides precisely the right technology for the construction of sophisticated self-healing systems. We also feel that middleware is the right place to locate such techniques given the unique role of middleware in providing platform-independent programming models for the construction of distributed applications. We thus conclude that the self-awareness inherent in reflective middleware technologies such as Open ORB are naturally supportive of self-healing systems.

This is however a preliminary analysis and further work remains to be done to test this hypothesis more fully. We are particularly interested in studying more advanced mechanisms for self-adaptation including for example biologically-inspired approaches such as machine learning and neural networks. We are also interested in combining such approaches with studies of context to provide context-aware adaptation.

6. REFERENCES

[1] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., David S. Rosenblum, D.S., Wolf, A.L., "An Architecture-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems, Vol. 14, No. 3, pp. 54-62, May/June 1999.

[2] Schmerl, B., Garlan, D., "Exploiting Architectural Design Knowledge to Support Self-repairing Systems", Proc. 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 2002.

[3] Autonomic Computing Home Page, http://www.research.ibm.com/autonomic/.

[4] Kiczales, G., J. des Rivières, D.G. Bobrow, "The Art of the Metaobject Protocol", MIT Press, 1991.

[5] Sun Microsystems, "Java Reflection", URL: http://java.sun.com/j2se/1.3/docs/guide/reflection/index.html

[6] Yokote, Y., "The Apertos Reflective Operating System: The Concept and Its Implementation", Proc. OOPSLA'92, ACM SIGPLAN Notices, Vol. 28, pp 414-434, ACM Press, 1992.

[7] Watanabe, T., Yonezawa, A., "Reflection in an Object-Oriented Concurrent Language", In Proceedings of OOPSLA'88, Vol. 23 of ACM SIGPLAN Notices, pp 306-315, ACM Press, 1988; Also available as Chapter 3 of "Object-Oriented Concurrent Programming", A. Yonezawa, M. Tokoro (eds), pp 45-70, MIT Press, 1987.

[8] McAffer, J., "Meta-Level Architecture Support for Distributed Objects", Proc. Reflection 96, pp 39-62, G. Kiczales (ed.), San Francisco; Available from Dept of Information Science, Tokyo University, 1996.

[9] Okamura, H., Ishikawa, Y., Tokoro, M., "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework", Proceedings of the Workshop on New Models for Software Architecture, November 1992.

[10] Kon, F., Costa, F., Blair, G.S., Campbell, R., "The Case for Reflective Middleware: Building Middleware that is Flexible, Reconfigurable, and yet simple to Use", CACM, Vol. 45, No. 6, 2002.

[11] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., "The Design and Implementation of OpenORB v2", IEEE DS Online, Special Issue on Reflective Middleware, Vol. 2, No. 6, 2001.

[12] Duran-Limon, H., Blair, G.S., "The Importance of Resource Management in Engineering Distributed Objects", Proc. 2nd International Workshop on Engineering Distributed Objects (EDO'2000), California, USA, Nov. 2000.

[13] Roman, M., Kon, F., Campbell, R.H., "Reflective Middleware: From the Desk to your Hand", IEEE DS Online, Special Issue on Reflective Middleware, Vol. 2, No. 5, 2001.

[14] Hayton, R., Herbert, A., Donaldson, D., "FlexiNet: A Flexible Component-oriented Middleware System", Proc. 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, Sintra, Sept. 1998.

[15] Ledoux, T., "OpenCorba: A Reflective Open Broker", Proc. Reflection'99, Saint-Malo, France, Springer-Verlag, LNCS, Vol. 1616, 1999.

[16] Andersen, A., Eliassen, F., Blair, G.S., "A Reflective Component-Based Middleware with Quality of Service Management", Proceedings of PROMS'2000 (Protocols for Multimedia Systems), Cracow, Poland, 2000.

[17] Szyperski, C., "Component Software: Beyond Object - Oriented Programming", Addison-Wesley, 1998.

[18] Costa, F. Duran, H., Parlavantzas, N., Saikoski, K., Blair, G.S., Coulson, G., "The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications", In Reflection and Software Engineering, Cazzola, W., Stroud, R. and Tisato, F. (Eds), Springer-Verlag, LNCS Vol. 1826, pp 79-98, 2000.

[19] Coulson, G.,, Blair, G.S., Clarke, M., Parlavantzas, N., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", ACM/ Springer Distributed Computing Journal, Vol. 15, No. 2, pp 109-126, April 2002.

[20] Manna, Z., Pnueli, A., "The Temporal Logic of Reactive and Concurrent Systems", Springer-Verlag, New York, 1992.

[21] Blair, G.S., Andersen, A., Blair, L., Coulson, G., Sánchez, D., "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform", IEE Proceedings Software, Vol. 147, No. 1, pp 13-21, February 2000.

[22] Duran-Limon, H., "A Resource Management Framework for Reflective Multimedia Middleware", PhD Thesis, Computing Department, Lancaster University.

[23] Intel IXP1200; http://www.intel.com/IXA/.