View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Research Repository

Murdoch
UNIVERSITY

MURDOCH RESEARCH REPOSITORY
http://researchrepository.murdoch.edu.au/

This is the author's final version of the work, as accepted for publication following peer review but without the publisher's layout or pagination.

Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K. and Ueyama, J. (2004) *A component model for building systems software.* In: M.H. Hamza (ed), Proceedings of IASTED Software Engineering and Applications (SEA 04), Cambridge, MA, USA, November 2004.

http://researchrepository.murdoch.edu.au/4846/

© ACTA Press
It is posted here for your personal use. No further distribution is permitted.

A COMPONENT MODEL FOR BUILDING SYSTEMS SOFTWARE

Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, Jo Ueyama
Computing Department, Lancaster University, UK
[geoff,gordon,p.grace,joolia,leek,ueyama]@comp.lancs.ac.uk

ABSTRACT
OpenCOM v2 is our experimental language-independent component-based systems-building technology. OpenCOM offers more than merely a component-based programming model. First, it is a runtime component model and supports dynamic runtime reconfiguration of systems (i.e. one can load, unload, bind, and rebind components at runtime). Second, it explicitly supports the deployment of the model in a wide range of 'deployment environments' (e.g. operating systems, PDAs, embedded devices, network processors). Third, it allows the particularities of different deployment environments to be selectively hidden from/ made visible to the OpenCOM programmer without inherent performance overhead.

KEY WORDS
Ccomponent-based development; reflection; systems programming; embedded systems

1. Introduction

The notion of constructing application-level software from components [1] is well established. For example, there is a substantial component-based software engineering community [2], and numerous component technologies for application development (both standalone and distributed) are available. Examples are: browser plug-ins [3], JavaBeans/ Enterprise JavaBeans [4], and the CORBA Component Model [5]. The general benefits of these technologies are: i) they promote a high level of abstraction in software design, implementation, deployment and management, ii) they facilitate flexible configuration (and, potentially, run-time reconfiguration), and iii) they foster third-party software reuse.

Less established is the notion of using components to build *systems-level software*, like embedded systems, operating systems, communications systems, programmable networking environments, or middleware platforms. Nevertheless, the above-mentioned benefits of componentisation appear just as compelling in this area, and this fact has been recognised by a number of researchers in recent years. For example, k-Components [6], and various JavaBeans-based approaches (e.g. [7]) have been proposed as component models for the construction of middleware platforms; Knit [8], THINK [9], MMLITE [10], and DEIMOS [11] have been proposed for the construction of operating systems (OSs); and Click [12] and Netbind [13] have been proposed for programmable networking environments.

However, these efforts have all been *narrowly-targeted*, both in terms of their intended application domains, and their intended hardware/ OS deployment environments. In particular, most of them have only been deployed on conventional desktop machines as opposed to more 'exotic' environments such as embedded hardware, PDAs, or network processors. In the present work, we propose a *general-purpose* component-based systems building technology. In more detail, our technology addresses the following requirements:

- *Wide applicability*. It should be applicable in a wide range of deployment environments from standard PC/Windows or PC/Unix environments, to resource-poor PDAs, to embedded systems with no operating system, to high speed network processor hardware. This implies small memory footprint, language independence, and policy independence.
- *Policy independence*. It should offer generic mechanisms; it should not prescribe policies, constraints, services or facilities that are specific to particular application domains (e.g. real-time; packet-processing; multimedia; 24x7 availability), or deployment environments (e.g. above-mentioned).
- *Support for runtime reconfiguration*. It should support mechanism-level runtime reconfiguration as required when implementing inherently dynamic target systems such as operating systems, reflective middleware, active networking nodes etc. [14]. Again, though, the policy that controls and manages such reconfiguration should be separable.
- *Selective transparency of deployment-environment-specific features*. Because of the requirement for wide applicability, it must operate in heterogeneous deployment environments with non-standard features—e.g. multiple processing elements, packet processing assists, and specific hardware message channels and memory hierarchies, etc. It should be possible to render such features invisible as far as possible but visible where required (whether for performance or functionality reasons). Where such features are made visible they should be presented in terms of the generic component-based programming model.
- *Separation of concerns*. It should encourage a separation of concerns in what is potentially a very

complex environment. For example, as well as the above-mentioned separation of mechanism and policy, the technology should separate the concerns of i) programming the base functionality of the target system versus the managing its runtime reconfiguration, and ii) providing selective transparency of deployment environment specifics versus writing target systems in terms of the generic component based programming model.

- *High performance*. It must not incur an inherent performance cost. This implies that the 'in-band' execution path of systems [15] must not be dependent on the execution of the component model runtime. In addition, when presenting non-standard features (as above) in terms of the generic programming model, this should incur as small a performance penalty as possible.

Our architectural approach to meeting these requirements is to define a generic run-time component model (see section 2) as a foundation, and then to augment this with the notions of *component frameworks* and *reflective meta-models*.

Component Frameworks (hereafter CFs) [14] are composite components, built in terms of the underlying component model, that accept 'plug-in' components that add to or modify the behaviour of the composite. They serve as architectural place-holders for application-specific or deployment-environment-specific policies, constraints, services or facilities. The idea is that each CF addresses a particular functional domain (e.g. protocol stacking, thread scheduling, packet forwarding, memory management, user interaction, etc.), and embodies policies, constraints etc. that make sense in that domain. For example a CF for protocol stacking would take protocol components as its plug-ins, and could constrain the plug-ins to be composed into linear stacks. Note that the design and implementation of CFs does not require anything beyond the generic services provided by the foundational component model—essentially, CFs are architectural patterns rather than distinct mechanisms.

Reflective meta-models [14] are causally-connected representations of selected aspects of a target system. Their function is to enable inspection and adaptation of the represented aspect. For example, we employ a so-called architecture meta-model that represents the topology of a set of composed components as an architecture graph that can be inspected to discover the topology, and adapted to change the topology (e.g. adding a node to the graph results in the deployment of a new component; removing an arc results in the breaking of an inter-component binding, etc.). The purpose of reflective meta-models is to maintain a clean architectural separation of concerns between system building (often called *base-level programming*) and system management/ configuration/ adaptation (which involves the use of 'meta-interfaces' provided by the meta-models, and is often referred to as *meta-programming*). Reflective meta-models are themselves implemented as CFs—again, their implementation does not require anything beyond the generic services provided by the foundational component model.

Note that the OpenCOM v2 component model builds on and generalises our earlier work on OpenCOM v1 which was used to build middleware platforms [14] in standard OS environments only.

In the remainder of the paper, section 2 overviews the design of the OpenCOM v2 component-based programming model. Then section 3 identifies a set of orthogonal programmer roles that help separate concerns in developing an OpenCOM-based system on a new hardware platform. Finally, section 4 discusses related work, and section 5 offers conclusions.

2. The OpenCOM Programming Model

2.1 Overview

The OpenCOM programming model essentially consists of primitives to load components into units of scope and management called *capsules*; and to 'bind' component interfaces and receptacles. Components may support any number of *interfaces* (described using an extended OMG IDL) and *receptacles* (these are 'anti-interfaces' that express a dependency on an interface provided by some other component) and they may also be composite (i.e. composed of internal sub-components). Importantly, interface-to-receptacle *binding* is a third-party operation: i.e. code that binds a receptacle on one component to an interface on another can reside in any component within the capsule. Components, interfaces and receptacles all support the attachment of arbitrary meta-data in the form of <*name*, *value*> pairs. However, this meta-data is intended solely for the use of higher level CFs which embody policy. It is not interpreted or understood by the component model itself.

The programming model further supports the notions of *caplets*, *loaders* and *binders* as first class entities. Caplets are nested 'sub-scopes' within capsules; loaders provide various ways of loading components into various types of caplets; and binders provide various ways of binding interfaces and receptacles, both within and across different caplet types and instances. Caplets, loaders and binders are themselves implemented as components that are 'plugged-in' to hosting CFs. Plug-in caplets, loaders and binders play a crucial role in i) facilitating the deployment of OpenCOM in a wide range of deployment environments in an uniform yet highly-performant manner, and ii) selectively masking the peculiarities of underlying deployment environments from OpenCOM programmers without compromising performance.

Figure 1 visualises the concepts of capsule, component, interface, receptacle, caplet, loader and binder. It shows a capsule containing three caplets as dotted boxes (the

distinction between root and slave caplets is discussed in section 2.2). Also shown are two loaders: L_1 is associated with the left-hand caplet, and L_2 can be associated with either of the other two; and two binders: B_1 knows how to bind components in the left hand caplet, and B_2 knows how to bind across the two slave caplets. Components are shown as rounded rectangles. Component interfaces are shown as circles, and receptacles as cups.

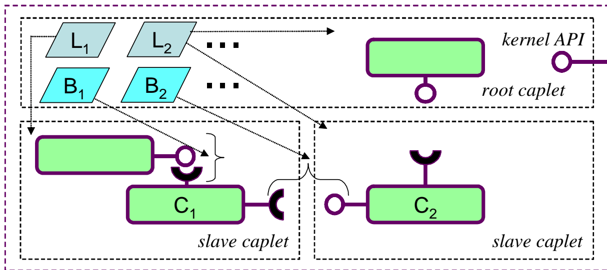

Fig. 1. The OpenCOM Programming Model

2.2 The Caplet CF

Plug-in caplets are hosted by a CF called the Caplet CF. Here are some examples of caplet plug-ins that might usefully be provided:

- in a standard OS environment, caplets might be implemented as separate OS processes that isolate trusted and untrusted components;
- a Java virtual machine might be wrapped as a caplet so that Java-implemented OpenCOM components can co-exist in the same capsule as, say, C++-implemented components;
- in a car area network or similar embedded networked environment, CPUs on the network can be represented as individual caplets, with the whole system being represented as a single capsule;
- in an network processor environment, one caplet might map to a UNIX process on a control processor or host PC, while other caplets map to specialised processor/ memory environments.

The degree of physical distribution across caplets within a capsule is implementation dependent. An extreme position, for example, would be to provide a widely-distributed capsule containing caplets that run on separate machines. The general intention, however, is that the caplets within a capsule should be relatively 'tightly-coupled' so that centralised, per-capsule, state can be held, and bindings between components in different caplets can be assumed to be relatively 'reliable', 'deterministic' and 'fast' (according to application area-specific definitions of these terms).

The Caplet CF's API is as follows:

```
loaded_caplet caplet_cf.load(caplet_type);
status caplet_cf.unload(loaded_caplet);
caplet_instance
caplet_cf.instantiate(loaded_caplet,
                     list of <name, value>);
status caplet_cf.destroy(caplet_instance);
```

Caplets are packaged as standard OpenCOM components and can be loaded and instantiated via the above API. Despite this uniform packaging, it must be emphasised that different caplets may be implemented very differently. In some cases, for example, the caplet component being loaded may be merely a bootstrapper for an arbitrarily complex and deployment-environment-specific caplet creation process.

Each caplet is either a *root* or a *slave*. There is only ever one root caplet in a capsule: this is the 'original' capsule environment that existed before the Caplet CF was loaded. This root caplet is also distinguished in being the only caplet to directly support the OpenCOM runtime's kernel API (which is not discussed in this paper). Typically, the 'core' CFs (i.e., the Caplet, and the Loader and Binder CFs) reside in the root caplet because there they have direct access to the kernel. All other caplets (i.e. those created using the above API) are slaves. By default, slaves allow their hosted components to bind to the core CFs, and to the kernel API, so that these components can create further caplets, and load and bind components in arbitrary caplets. However, the core CFs and the kernel API can be selectively hidden from individual slave caplets—e.g. for security reasons. For example, components in a low-privileged slave caplet may be denied access to the Caplet CF to disallow them from creating further caplets.

2.3 The Loader and Binder CFs

Like caplets, plug-in loaders and binders can also encompass a wide range of functionality. For example, consider the following:

- loaders that know about non-standard component repositories and different component packaging conventions;
- a loader that performs (recursive) dependency-satisfaction: when a component is loaded, any dependencies it has (as denoted by its receptacles) that are not already available in the capsule are automatically pre-loaded;
- a loader that transparently load-balances across a set of caplets it manages;
- binders that exploit alternative 'connectivity' technologies such as interrupts, call-gates, buses, shared memory or message passing services;
- binders that represent different cost/ performance trade-offs;
- binders whose bindings support an interception meta-model [14];

The API for managing plug-in loaders and binders follows the general pattern of the Caplet CF's API shown above. In brief, there are operations to load a new loader or binder plug-in, and to unload a plug-in, etc. The main

API calls for using loader and binder plug-ins are as follows:

```
loaded_component loader_cf.load(component_type);
loaded_component loader_cf.load(caplet,
                         component_type);
loaded_component loader_cf.load(loader_name,
                         component_type);
loaded_component loader_cf.load(caplet,
                         loader_name,
                         component_type);

binding binder_cf.bind(interface_instance,
                         receptacle_instance);
binding binder_cf.bind(binder_name,
                         interface_instance,
                         receptacle_instance);
```

The four *load()* signatures offer 'selective transparency' in the choice of a loader, and a caplet into which to load. More specifically, one can opt for any of the following:

- full transparency (i.e. neither the loader and the target caplet are explicitly named), or
- partial transparency (i.e. the target caplet is specified but the loader is unspecified, or vice versa), or
- no transparency (i.e. both the loader and the target caplet are explicitly named).

(An analogous choice of whether or not to explicitly name a binder is available via the two *bind()* signatures.) In all cases involving transparency, the policy for the choice of loader, binder and target caplet is embedded in a 'policy' plug-in that selects and dispatches to a 'real' loader/ binder/ caplet. Usually, the policy is informed by <*name*, *value*> meta-data that are attached to the various involved entities. For example, when transparently loading a component with an attached meta-data attribute <*CAPLET_TYPE*, *PRIVILEGED*>, a loader policy might dispatch to a loader that shared this attribute name and value.

2.4 Using Loaders and Binders

As a simple example of the (transparent) use of the Loader and Binder CF APIs, consider the following code which loads and binds components C_1 and C_2 in figure 1:

```
component_instance c1, c2;
loaded_component loaded_c1, loaded_c2;
interface_type1 i;
recept_type1 r;
loaded_c1 = loader_cf.load(component_type1);
loaded_c2 = loader_cf.load(component_type2);
<c1, i> = loader_cf.instantiate(loaded_c1,
                         interface_type1, <>);
<c2, r> = loader_cf.instantiate(loaded_c2,
                         recept_type1, <>);
binder_cf.bind(i, r);
```

This uses the least-specific, most transparent, *load()* signature to load the two components, which results in the loader policy dispatching to appropriate per-caplet loaders (L_1 and L_2 respectively). Having instantiated the components in their respective host caplets, the least-specific *bind()* signature is then used to bind the two components. As in the loading case, the binder policy selects a 'real' competent binder—i.e. B_2—on the basis of meta-data attached to the to-be-bound interface and receptacle (here, the meta-data presumably relates to their hosting caplet types).

The abstraction power of third-party loading and binding is clearly seen in this example. It employs a very simple 'create and connect' pattern that abstracts over the presence of multiple heterogeneous caplets, and multiple loading and binding mechanisms. The above code could be executed unchanged with identical semantics from within any component running in any caplet in the capsule—including components running in very primitive caplet environments in embedded systems.

3. Deploying OpenCOM-based Systems

As mentioned, OpenCOM offers more than merely a runtime programming model for building systems in terms of components. It also explicitly addresses the *deployment* of this model in a diverse and heterogeneous range of deployment environments and the *runtime reconfiguration* of target systems.

This is facilitated by introducing a separation of concerns between the roles of: *deployment programmers*, *systems programmers*, and *meta-systems programmers*. These are illustrated in figure 2.

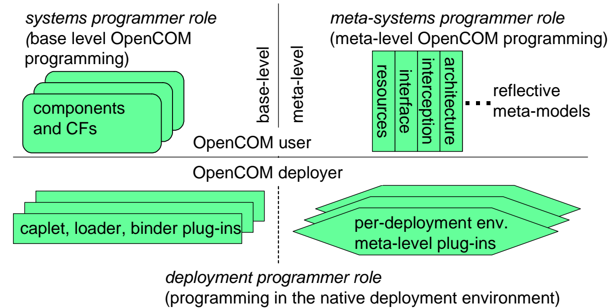

Fig. 2. OpenCOM Programmer Roles

Deployment programmers create a viable OpenCOM programming model in the target deployment environment (lower quadrants of figure 2). They bridge the "implementation gap" between the concrete deployment environment (hardware or software) and the abstract component-based programming model. They do this by working with a small set of low-level abstractions that serve to structure and organise the task of deploying OpenCOM. These are (primarily) implementations of the above-mentioned caplet, loader and binder plug-ins, which form a suitably canonical set of abstractions over the key services offered by heterogeneous deployment environments; but these are augmented with a number of additional lower-level abstractions (not discussed in this paper) that are not visible to systems or meta-systems programmers.

Systems programmers then use the standard OpenCOM programming model to develop a target system or application (upper left quadrant in figure 2)—they are (selectively) isolated from the particularities of different deployment environments. Systems programming can be loosely divided into two sub-roles: the development of more-or-less generic CFs that are targeted at specific areas of functionality within the target application domain; and the development of high-level code that builds on or uses these CFs. This is not a hard differentiation, however. The essence of systems programming is that it employs 'pure' OpenCOM programming: all deployment-environment-specific issues are assumed to have been addressed by the deployment programmer role.

Finally, *meta-systems programmers* operate in the upper-right quadrant of figure 2. They employ appropriate reflective meta-models to structure the task of reconfiguring the target system at runtime. Some meta-models could be generic and independent of any particular deployment environment and can be produced by the systems programmer role; but others might rely on deployment-environment specific mechanisms that must be implemented by deployment programmers. An example of the latter is an operation invocation interception meta-model that relies on detailed knowledge of CPU and language-dependent calling conventions [15].

4. Related Work

OpenCOM v2 can usefully be positioned against two separate categories component model: application-level models; and specifically systems-oriented models. Regarding the former, OpenCOM differs from designs such as EJB [4], and the CORBA Component Model [5] in being considerably more lightweight. OpenCOM's capsule concept is superficially related to the 'container' concept espoused by these models; but OpenCOM capsules are policy free and only contain minimal, low-level, functionality (e.g., loading and binding related). OpenCOM shares with these models an emphasis on third-party deployability of components; but for OpenCOM, third-party deployability is important not only for reasons of software re-use, but also to facilitate system (re)configurability, and to enable primitive slave caplets (e.g. representing a microcontroller) to function as first-class players in the programming model. OpenCOM also differs from EJB and ICENI in being language independent. Being a systems-building technology, it is an aspiration for OpenCOM to serve as a basis for the implementation of application-level components models: our approach here would be to implement containers and policies etc. in terms of CFs.

Regarding specifically systems-oriented component models, the following are some major players: Knit [8], Koala [16], MMLite [10] and THINK [9]. Of these, Knit and Koala are build-time component models: i.e. components are not visible at runtime, so there is no systematic support for dynamic component loading, still less managed reconfiguration. Knit has been targeted primarily at operating systems, but has additionally been used build software routers (e.g. a router called 'Clack'), although again only on PCs. Koala is a proprietary system from Philips and is representative of several similar efforts from the embedded systems community. Like OpenCOM, Koala components support both 'provided' and 'required interfaces' (cf. interfaces and receptacles); also, the model is inherently programming language independent. MMLite and THINK, on the other hand, have in common with OpenCOM the property of being run-time component models. MMLITE was an attempt to adapt and apply Microsoft's COM as a vehicle for building operating systems. This early work demonstrated the feasibility and flexibility of the component approach (including primitive support for reconfiguration), but was confined to building operating systems in conventional PC environments. THINK is much closer to OpenCOM in its goals and approach. Like OpenCOM, it addresses dynamic reconfiguration and supports multiple implementations of binding (although not loading and scoping). It also demonstrates the possibility and benefits of abstracting hardware (e.g. I/O devices, paging hardware) as components. However, its programming model is at a lower level of abstraction than OpenCOM's and it has no equivalent of OpenCOM's meta models to manage reconfiguration. In addition, it has so far only been used in conventional, PC-based, deployment environments.

5. Conclusions

To conclude, we briefly sum up the benefits of OpenCOM by revisiting the requirements set out in section 1 and showing how these are realised. *Wide applicability* is achieved through the horizontal extensibility offered by the caplet, loader and binder plug-in capability (plus the small and easy-to-port microkernel runtime). *Policy independence* is achieved by providing a clear separation between the foundational component model (mechanism only) and higher-level CFs that embody policy in specific areas of concern. *Support for runtime reconfiguration* is achieved at the mechanism level by means of OpenCOM's ability to load, bind and rebind components at runtime; and at the management level by means of reflective meta-models which an provide inspection and adaptation capability, and CFs which provide constraint on reconfiguration. *Selective transparency of deployment-environment-specific features* is achieved by wrapping non-standard deployment environment specific features in terms of plug-in caplets, loaders and binders and then offering the OpenCOM programmer selective transparency in the selection of these plug-ins. *Separation of concerns* is achieved i) through the separation of mechanism and policy referred to above, and ii) through the differentiated programming roles discussed in section 3. Finally, *high performance* is achieved by having the runtime focus on loading and binding, both of which are 'out-of-band' with respect to the execution of the

application. In addition, performance-enhancing mechanisms available in the deployment environment (e.g. code morphing, or the availability of special hardware message channels) can be used directly as a side-effect of instantiating a corresponding programming-model-level binding. Additionally, the (apparent) overhead of IDL-specified interfaces and receptacles can be nullified in primitive environments: loaders and/ or CFs associated with the primitive environment can arbitrarily constrain the number and form of interfaces (e.g. to allow operations to take only integer arguments) and can forgo the use of stubs and skeletons where these are not required.

To date we have primarily validated the use of OpenCOM v2 in a programmable networking based systems environment using the Intel IXP1200 network processor [17]. Network processor hardware is a particularly interesting deployment environment because it is both heterogeneous (e.g. it employs a number of processor types including processors that are specialised for packet forwarding), resource poor (e.g. packet forwarding processors are typically very primitive and have only a small amount of memory), and performance constrained (i.e. packets must be forwarded at line speeds). Our work in this area is reported in [18] and [19].

In our ongoing research we are looking at further validating our systems-building approach by deploying OpenCOM v2 in a range of application areas/ deployment environments. In particular, we are investigating the use of OpenCOM in mobile Grid computing environments [20], and in building a generic OS/ communications layer for miniaturised devices in wireless sensor network environments.

References

[1] Szyperski, C., “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, 1998.

[2] Wolfgang Emmerich, W., “Distributed Component Technologies and their Software Engineering Implications”, Proc. of the 24th Intl Conf on Software Engineering, Orlando, Florida, pp. 537-546, 2002.

[3] Mozilla Organization, XPCOM project, 2001, http://www.mozilla.org/projects/xpcom.

[4] Sun Microsystems, Enterprise JavaBeans Specification Version 1.1, http://java.sun.com/products/ejb/index.html.

[5] Object Management Group, CORBA Components Final Submission, OMG Document orbos/99-02-05.

[6] Dowling, J., Cahill, V., “The K-Component Architecture Meta-Model for Self-Adaptive Software”, Proc. Reflection 2001, LNCS 2192, 2001.

[7] Bruneton, E., Riveill, M., “JavaPod: an Adaptable and Extensible Component Platform”, Proc. Reflective Middleware 2000, New York, 2000.

[8] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E., “Knit: Component Composition for Systems Software”, Proc. OSDI 2000, pp 347-360, Oct 2000.

[9] Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G., “THINK: A Software Framework for Component-based Operating System Kernels”, Usenix Annual Technical Conf., Monterey (USA), June 10th-15th, 2002.

[10] Helander, J., Forin, A., “MMLite: A Highly Componentized System Architecture”, 8th ACM SIGOPS E Workshop, pp 96-103, Sintra, Portugal, Sept 1998.

[11] Clarke, M., Coulson, G., “An Architecture for Dynamically Extensible Operating Systems”. Proc. 4th Intl. Conf. on Configurable Distributed Systems (ICCDS'98), Annapolis MD, USA, May 1998.

[12] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F., “The Click Modular Router”, Proc. ACM SOSP 1999, pp 217-231, Dec 1999.

[13] Campbell, A.T., Chou, S., Kounavis, M.E., Stachtos, V.D., and Vicente, J.B., “NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers”, 5th IEEE Intl. Conf. on Open Architectures and Network Programming (OPENARCH' 02), June 2002.

[14] Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N., “The Design of a Highly Configurable and Reconfigurable Middleware Platform”, ACM Distributed Computing Journal, Vol 15, No 2, pp 109-126, Apr. 2002.

[15] Coulson, G., Blair, G.S., Grace, P., “On the Performance of Reflective Systems Software”, Proc. Intl. Workshop on Middleware Performance (MP2004), Phoenix, Arizona; Satellite workshop of the IEEE Intl. Performance, Computing and Communications Conf. (IPCCC 2004), Apr, 2004.

[16] Fioukov, A.V., Eskenazi, E.M., Hammer, D.K., Chaudron, M.R.V., “Evaluation of static properties for component-based architectures”, Proc. 28th Euromicro Conf., Dortmund, Germany, pp 33–39, IEEE Computer Society Press, Sept. 2002.

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

[18] Coulson, G., Blair, G.S., , A.T., Joolia, A., Lee, K., Ueyama, J., Ye, Y., “NETKIT: A Software Component-Based Approach to Programmable Networking”, ACM SIGCOMM Computer Communications Review (CCR), Vol 33, No 5, October 2003.

[19] Ueyama, J., Schmid, S., Coulson, G., Blair, G.S., Gomes, A.T., Joolia A., Lee, K, “A Globally-Applied Component Model for Programmable Networking”, Proc. International Workshop on Active Networks (IWAN 2003), Kyoto Japan, 10-12 Dec 2003.

[20] Grace, P., Coulson, G., Blair, G., Mathy, L., Yeung, W.K., Cai, W, Duce, D., Cooper, C., “GRIDKIT: Pluggable Overlay Networks for Grid Computing”, to appear in Proc. Distributed Objects and Applications (DOA’04), Cyprus, Oct 2004.