Making Meta-Object Protocols Practical for Operating Systems

Brendan Gowing* Vinny Cahill
Distributed Systems Group, Computer Science Department,
Trinity College, Dublin 2, Ireland.

Abstract

This position paper considers how Meta-Object Protocol (MOP) technology can be used to support operating system flexibility including the dynamic adaption and extension of the system. We are interested in applying MOPs to a domain (that of operating systems) where they have had little acceptance. We believe that this is largely due to the complexity of operating system software and the possible security loophole(s) that MOPs can introduce. We address these problems by the novel use of multiple, fine-grained MOPs *and a category of MOPs called* Extension Protocols *to provide controlled, secure extension without the limitations of predefined "hooks" into the operating system.*

1 Introduction

Operating system designers are catering for an increasingly diverse range of application requirements by designing flexible systems. The noticeable increase in designs that include micro-kernel technology to support user-space replaceable system services is one effect of this change. However, the user-space server is a very coarse-grained structure, not easily subjected to incremental modification [DPH92][MA90]. It is also difficult to dynamically extend the services provided by these systems, be they kernel or user space based.

The solution then is to have a dynamic system structure, which in our case is object-oriented, that can cater for the differing requirements that applications place on system services. As applications are activated dynamically, the object-space in which these applications execute is continuously growing, shrinking, and altering to accommodate new application objects. It then follows that the system services which support such an application object space must also be dynamically able to grow, shrink and alter themselves to accommodate the requirements of application objects while avoiding system shutdowns for updates. Certainly a difficult task in a standalone system, but one whose difficulty can be greatly increased by distribution and heterogeneity.

In this position paper, we consider the use of Meta-Object Protocols (MOPs) as a mechanism for specifying the structure of flexible, open, object-oriented, system software that can be dynamically adapted and extended in a *non-predetermined* manner. We outline some of the problems faced when using MOPs in operating systems, and how they may be overcome by using multiple, fine-grained MOPs and a category of MOPs called Extension Protocols.

*bgowing@dsg.cs.tcd.ie

2 Related Work

There are currently a number of operating systems supporting extensibility. A common feature of many of these systems is the use of "hooks" into the system system to cater for dynamic extension. This means that the set of possible extensions is limited by the number of hooks that the designers have included. It also requires a certain amount of precognition on the part of the designer to include the appropriate hooks for the intended application base, if the latter can indeed be predicted. Our proposal differs from these approaches by including extension protocols which are used to suitably modify an object system's implementation to incorporate (possibly unpredictable) new behaviour.

Examples of systems utilising hooks include SPIN and the *caching kernel*. In SPIN [B+94] "spindles" can be dynamically hooked into the kernel at runtime. However, the predefined set of points at which extensions can be linked into the kernel are fixed. Similarly, the *Caching kernel* [CD94] can load specific kernel modules into a "cache" in kernel space, but the type of services are pre-defined and are limited to being either kernels, address spaces, memory spaces, and processes, i.e. there are no user-definable services allowed.

A related system, the Kernel Toolkit (KTK) [GMSS94], supports configurable object-oriented applications where *attributes* are manipulated by *policies*. However, KTK does not support run-time extensibility, as attributes and policies are fixed at compile time.

Apertos [Yok92] was the first OS developed using reflection, meta-objects and object/meta-object separation. It offers a scheme of "extensible hooks" through *reflectors*. The reflector class hierarchy only supports a single MOP which, though it is expandable through compile-time inheritance, only supports Actor-like, single-threaded, active objects as the system's executable entity. However, dynamic extensibility of the kind in which we are interested is not supported in the current release as neither alteration of reflector classes nor addition to the hierarchy of reflector classes is allowed at runtime.

3 Applying MOPs to OSs

Operating systems are large, complex pieces of software which must be secure against tampering. The

1063-5351/95 $04.00 © 1995 IEEE

history of OS design includes tight, inter-component coupling in an attempt to minimise execution time on processors with low computational power. Moreover, it cannot be denied that as computers develop, their operating systems are experiencing a similar growth and are becoming unwieldy. The need to support new software features and hardware devices is increasingly bloating the system software [PW94].

In order to tackle this problem, we are building on a body of research into object-oriented operating systems [Rus91][C+94] and the success of Meta-Object Protocols in the area of flexible language runtimes [KRB92][SKT94]. A Meta-Object Protocol is the *specification* of an object system's open *implementation* in terms of the meta-objects and classes needed to realise the system's intended behaviour [KRB92]. In this context, the protocol documents the objects which implement a system's objects; i.e., it is not concerned with the specification of an object's interface, but the specification of the implementation of the objects. An operating system can be effectively constructed by "instantiating" the hierarchy of MOPs specifying system services. By this we mean that the classes specified in the MOP are collectively instantiated. Just as an object is an instance of a class, we use the term *MOP instance* to describe the set of objects instantiated from the classes specified in the MOP.

Despite arguments in favour of the use of MOPs in open system software [KLM+93], the MOP has had little if no impact on the operating system community. We believe that this is largely due to the complexity of an operating system's intricate structure[1] and the possible security loopholes. Whereas the workings of a programming language's run-time can be adequately described by a MOP, an entire operating system is a far more daunting proposition. The issues to be solved include (a) deriving an appropriate "OS MOP", (b) altering (dynamic adaption or extension) the components of such a large compiled system, and (c) the seemingly large security loophole of allowing an application to manipulate system services. We address (a) by using *multiple, fine-grained MOPs*, where system software is decomposed [Mae94] into constituent services and represented by individual MOPs[2]; (b) by introducing a system of *controlled* extension through a category of MOPs that we call *Extension Protocols*; and (c) by separating the MOP from the mechanism of extension. These topics are covered in the following subsections.

3.1 Developing a MOP for an OS

Essentially, the kernel, user-space servers and run-time support systems all provide *services*. Combining all of these services into a single MOP would be a highly complex task, so we are isolating the decomposed services [Mae94] and specifying each one with its own MOP. In this fashion, the operating system can be described by a collection of MOPs offering disparate or similar services to clients.

[1] Especially in the case of legacy system software.

[2] To our knowledge, no other system attempts to use more than a single MOP.

The following list gives an indication of the nature of the MOPs that we are working with: Bootstrap Management, Thread Management, Storage Management, Memory Management, Device Management, Object Name Management, Distribution Management, User Management, Time Management, various Language Run-time MOPs, and of course, Extension Protocols for specifying various extension policies.

Our proposed method of bootstrapping an operating system based on MOPs involves instantiating the set of fundamental MOPs that provide minimal system services. The contents of the set are necessarily dependent upon the intended use of the system, so that, for example, an embedded OS might not need user management functions and have only limited file system functions. After instantiating MOPs for thread and file system management, a user management MOP can be instantiated to solicit for users. At this point the number and type of MOPs that are then loaded depends ultimately on what the users do. For example, if a user has files stored on a separate file system from the boot file system, then the second file system has to have a MOP instantiated for it by the minimal system that is available. As applications are chosen for execution by the various users, they and the MOPs that they require services from are also instantiated. In this fashion, the system is built up in response to user requirements to accommodate the particular applications being executed.

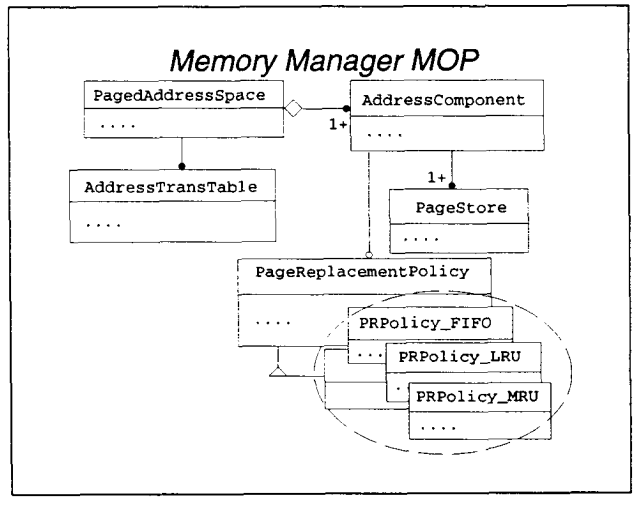

Figure 1: **A paged memory manager MOP showing its constituent classes, including various page replacement policies (`PRPolicy_XXX`).**

MOPs can benefit from the use of inheritance. A MOP as a set of classes offers extension through subclassing [KRB92]. A similar feature has already been exploited by other flexible systems [Rus91][C+94]. As an example, consider a Thread System which caters for different processors via alternate `Processor` and `Context` classes, which inherit from base processor and context classes, and which offer alternate scheduling policies in a `SchedulerPolicy` class hierarchy.

Now, using MOPs and without any prescience on the part of the system's designer, the thread system can be extended even though the existing object interface does not support extension. For example, dynamically extending the thread package by adding a new class to the `SchedulerPolicy` hierarchy may have been an obvious candidate for inclusion as part of the package's interface, but replacing the `Context` with one that maintains application dependent debugging information is less predictable as a requirement.

Figure 1 shows a simplified MOP for memory management. Given that applications tend to differ in their page replacement requirements, this MOP includes a small hierarchy of page replacement policy classes. The `PRPolicy_XXX` classes contain alternative algorithms, such as "most recently used" in `PRPolicy_MRU`. Applications can then choose the most appropriate policy through an Extension Protocol. Figure 2 shows application-level objects which have a Memory Manager MOP instance as part of their meta-level. In turn, the memory manager has an extension protocol as part of its meta-level. In this example, although the memory manager already has a set of scheduler policies, new ones can be added by using an specialised extension protocol. Figure 2 simply shows how an extension protocol can support adaption by allowing application software to select an appropriate page replacement policy by calling the *selectPRPolicy* method, despite the fact that the original memory manager design did not have to include such a facility.

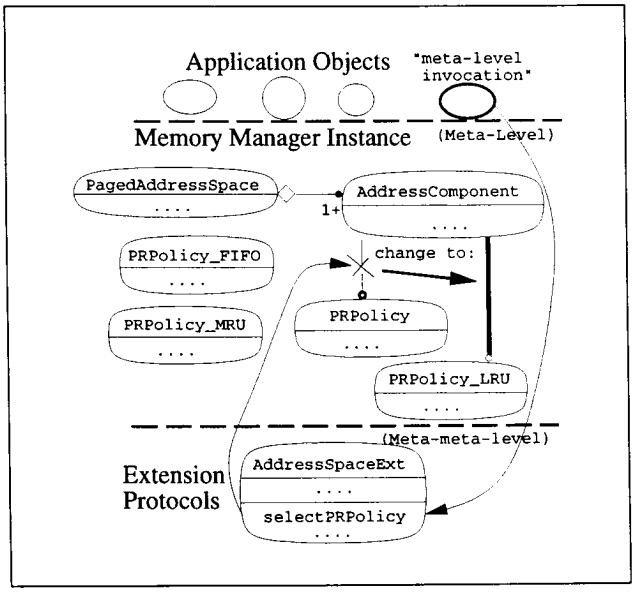

Figure 2: **Adapting the memory manager by dynamically (i.e., at runtime) changing the page replacement policy from a generic version to one that implements the least recently used algorithm.**

3.2 Dynamic Adaption and Extension

We have identified four categories of adaption and extension that must be supported:

1. **Adaption**: where the components, algorithms or policies are selected or tuned by a client object.
2. **Extension by Replacement**: of either a MOP instance in its entirety or an object within that MOP (such as one that implements a certain policy).
3. **Extension by Modification**: an existing object is extended in some way by either adding to its representation or interface.
4. **Extension by Introduction (of new phenomena)**: covers either the introduction of a new service implemented as a new MOP instance, or the introduction of a new class into a MOP instance.

Figure 3 shows an example of Extension by Introduction. In this case, an application object wants to plug its own page replacement policy object into the memory manager of its meta-level. Here, the extension protocol has an `addPRPolicy` method to cater for just such an eventuality. The method should be able to suitably modify the MOPs instance to accommodate the new object as well as ensure the integrity of the memory manager by validating that the operation does not create any security infringements.

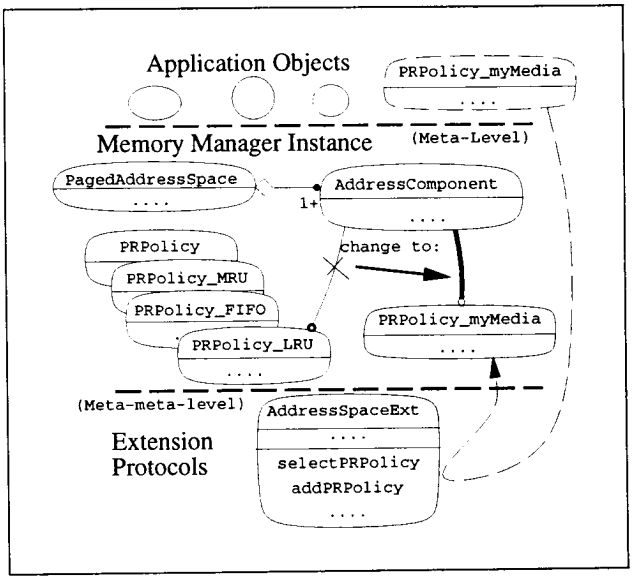

Figure 3: **A new page replacement policy being plugged in using `addPRPolicy` method of an Extension Protocol class.**

3.3 Security

Typically, language runtimes specified by a MOP offer client objects the opportunity of using a second or *meta-* interface [KRB92] which supports access to the

internals of the system's implementation. The components of the MOP can usually then be replaced at runtime as required. Obviously, a similar scheme is *not* suitable for an operating system, where arbitrary modifications of shared services, whether malicious or not, can have hazardous affects on system integrity. We thus consider that services and the objects that implement them are resources that can be either local, shared or global.

The intention is that while global and shared resources have to be protected against tampering, local resources can be less restricted; destructive modifications will only be possible for local resources where the harm can be localised using either hardware address space boundaries or a software scheme such as in [WLAG93]. Also, by marking the memory pages storing a MOP instance as read only, the meta-level objects can be protected against tampering while still in the same address space as insecure application-level code. The Extension Protocol can then control the extension process by maintaining stringent criteria for MOP instance extension. For example, an extension protocol for a file system MOP may only allow buffer and device driver extensions to be made, such that new buffers of the appropriate size can be added to the file system and local device drivers may be loaded for devices, but not unloaded. In this way, the extension protocol acts as a "firewall" between the MOP instance's objects and all other objects.

Note, however, we do not want to rule out the possibility that a MOP might include either an unsafe local resource which executes in a privileged mode of the processor or a shared resource maintained in an address space with local resources. Although dangerous for mainstream operating systems, it may be a requirement for a particular embedded system design. In the case of the former, a similar system to [WLAG93] can be utilised to maintain protection while the latter can use protected memory pages or virtual resources.

However, there are a number of open issues that we are still examining. They include: Which policies/MOPs, if any, should *never* be extensible? What extension protocols are generic (can be applied to objects or MOPs at any level of the software hierarchy; i.e., from the system service level at the bottom to the application software at the top)? Minimising cross address-space utilisation (including examining cross address space information replication and part page sharing for quick access). The use and efficiency of virtual resources; i.e., having a globally shared resource locally visible to an application's object but represented securely by being a *virtual resource*. The interesting area of extending Extension Protocols.

4 Summary

The goal of this position paper was to explain how Meta-Object Protocols, a feature of many flexible language run-time systems, can be made applicable to operating systems. To do this we have extended the MOP paradigm to include co-existing multiple, fine-grained MOPs which use Extension Protocols to support dynamic (i.e., at runtime) adaption and extension of application and system software securely.

References

[B+94] B.N. Bershad et al. *SPIN* – An Extensible Microkernel for Application-specific Operating System Services. In *6th European SIGOPS Workshop*, 1994.

[C+94] V. Cahill et al. Extensible Systems - The Tigger Approach. In *6th European SIGOPS Workshop*, 1994.

[CD94] D.R. Cheriton and K.J. Duda. A Caching Model of Operating System Kernel Functionality. In *6th European SIGOPS Workshop*, 1994.

[DPH92] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Modularity and Protection Should be Decoupled. In *The 3rd Workshop on Workstation Operating Systems*, pages 95–97. IEEE, April 1992.

[GMSS94] A. Gheith, B. Mukherjee, D. Silva, and K. Schwan. KTK: Kernel Support for Configurable Objects and Invocations. In *2nd International Workshop on Configurable Distributed Systems*. IEEE, ACM, March 1994.

[KLM+93] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The Need for Customizable Operating Systems. In *The 4th Workshop on Workstation Operating Systems*, pages 165–169. IEEE Computer Society Press, October 1993.

[KRB92] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1992.

[MA90] D. McNamee and K. Armstrong. Extending The Mach External Pager Interface To Accomodate User-Level Page Replacement Policies. In *Usenix Mach Workshop*, October 1990.

[Mae94] C. Maeda. Flexible System Software Through Service Decomposition. In *OOPSLA '94 Workshop on Flexibility in System Software*, 1994.

[PW94] C. Pu and J. Walpole. A Case for Adaptive OS Kernels. In *OOPSLA '94 Workshop on Flexibility in System Software*. Oregon Graduate Institute, 1994.

[Rus91] V.F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[SKT94] N. Saji, T. Kageyama, and M. Tajiri. C++ Metaobject on CLOS MOP. In *OOPSLA '94 Workshop on Multi-Language Object Models*, 1994.

[WLAG93] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient Software-Based Fault Isolation. In *14th ACM Symposium on Operating System Principles*, December 1993.

[Yok92] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Object-Oriented Programming Systems, Languages, and Applications '92*, volume 28 of *ACM SIGPLAN Notices*, pages 414–434. ACM Press, October 1992.