Patterns to Ease the Port of Micro-kernels in Embedded Systems

Michel de Champlain

Department of Computer Science
University of Canterbury, Christchurch, New Zealand
`michel@cosc.canterbury.ac.nz`

July 29, 1996

Abstract

Micro-kernels are difficult to port to a new hardware platform. During the initial phases of a port, much time and effort is lost on debugging critical machine-dependent subsystems. These subsystems are generally very tightly coupled and cannot be tested in an incremental fashion. Tight coupling occurs because the subsystems share many global variables forcing them to be debugged with the complete micro-kernel's code. The problem of porting and testing new micro-kernel ports has so far received little attention, and the work described in this paper is an attempt to fill this gap.

A typical design pattern captures and communicates the development and reuse of objects and their interaction in a specific context. In the embedded domain, we have found design patterns very useful to encapsulate requests, to decouple subsystems, and to make policies interchangeable. Our patterns are more than objects that work together to solve a design problem, they are also patterns for organizing other patterns to address the testing and porting issues.

We describe the design and implementation of various object-based design patterns which address the problem of the initial porting of micro-kernels for embedded systems in a systematic and incremental fashion using bottom-up integration testing. The motivation behind our design patterns is presented along with an outline of their consequences, constraints and applicability to the port. In the course of several iterations, these patterns have been tested and refined through their use in porting of an existing embedded micro-kernel to different hardware platforms.

Keywords: Design patterns, embedded systems, micro-kernel initial port, bottom-up testing, object-based programming.

1 Introduction

An important concern in the development of micro-kernels is their portability across different hardware platforms. Most micro-kernel subsystems, including the machine-dependent ones, are written in high level languages such as C [1, 2], C++ [3], and Modula-3 [4]. As a result, very little machine-dependent assembly code needs to be rewritten for each new port.

Even so, the design and implementation of portable micro-kernels to run on different computer architectures, such as Mach [5], V [6] and Chorus [7], are serious challenges [8, 2]. In particular the debugging phase of a micro-kernel port is still considered black magic and an art, especially for machine-dependent subsystems that are time consuming and require exhaustive testing.

Even if the micro-kernel's code is readable and easy to maintain, the initial port to a new machine demands a significant effort since this phase is generally tested monolithically. In spite of the careful modularisation some critical machine-dependent subsystems, such as the real-time clock, the context switch and the scheduler, are very hard to debug and test.

A micro-kernel port needs to be tested in a more systematic and incremental fashion, we have used these patterns during several micro-kernel design and implementation allowing each critical component to be fully tested. This paper describes the *object manager*, *timer*, and *scheduler* patterns we have adopted and their applicability. We have implemented and used these patterns for several ports. While our work occurs in the context of embedded systems, we believe our approach will be also useful to port general purpose micro-kernels across different processors in a more systematic and rapid manner.

This paper is organised as follows. Section 2 briefly describes the overall development life cycle and architecture of our patterns, and the context in which this work has been done. Section 3 introduces each individual pattern and their respective objects. Section 4 examines and comments on related issues and finally, section 5 presents a brief evaluation, future plans, and some conclusions.

2 The Overall Approach

This section describes the overall development life cycle and architecture of our machine-dependent patterns for the port of micro-kernels in embedded systems.

Context The long term objective of our projects [9, 10, 11, 12] was to simplify the porting of micro-kernels for embedded applications. By this we mean that patterns will encapsulate hardware features and will be part of an effective testing process.

Problem Micro-kernels are difficult to port to a new hardware platform. Good encapsulation is not enough to eliminate all problems during an initial port. Without methodical and incremental tests some non-trivial machine-dependent objects such as the real-time clock, the context switch and the scheduler can be extremely time consuming to debug.

Motivation Our main motivation was not to discover objects for reuse but to develop patterns which promote portability (with greater ease) across different platforms. These problems force us to design, implement and further enhance frameworks to provide a more systematic porting procedure. After several iterations we found that our design requirements were very compatible with the object-oriented (OO) paradigm. It is worthwhile to note that previous work has already demonstrated the advantage of OO design for micro-kernels [2]. But our work refines on a missing portability issue: capturing and describing patterns which contribute to the initial phases of a micro-kernel port.

Embedded soft real-time[1] systems must satisfy timing constraints [13] in which a micro-controller is usually interfaced directly to some physical equipment and is dedicated to monitoring the operation of that equipment [14]. Most of these embedded systems have restricted (volatile and non-volatile) memory space requirements. Fuel injection systems, washing machines and mobile radio systems for remote site monitoring and control are typical examples. In this domain of applications the use of the full spectrum of OO techniques is sometimes difficult. The reasons induct overhead in code and memory space; and possible non-deterministic behavior of unexpected creation and deletion of objects, dynamic binding, etc. Instead we are using object-based techniques as a trade-off to be

---

[1] A soft real-time requirement is characterized by an average response time, e.g. a request that must be satisfied on average.

both predictable and still benefit from object-orientation. Consequently, another goal of our projects is to explore the potential and impact of object-orientation for embedded systems where timing constraints are relevant.

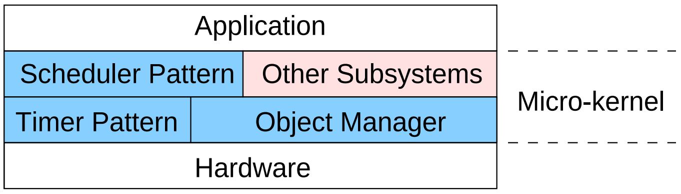


Figure 1: Typical Micro-kernel Architecture

Architecture

A typical micro-kernel architecture provides a minimal number of subsystems (see Figure 1). We have divided a micro-kernel architecture in two major structures: *machine-dependent patterns* (dark shaded) and *machine-independent subsystems* (light shaded). Machine-dependent patterns are object manager, timer, and scheduler. Machine-independent subsystems (Other Subsystems in Figure 1, such as memory management, thread management, inter-thread communication, etc.) are not part of the patterns and will not be discussed further.

Forces

By actively developing and experimenting with embedded micro-kernels [9, 10, 11] we have found patterns that were useful in porting and testing micro-kernels quickly and reliably. Operating system developers have a need to organize the port of micro-kernels in a suitable architecture to simplify their implementations.

The forces organizing and affecting these patterns are:

1. **Bottom-up integration testing.** The testing process progresses in steps where testing is carried out incrementally in conjunction with micro-kernel integration. The model of micro-kernel testing shown in Figure 2 is based on the notion of incremental bottom-up integration where a group of objects are combined to form patterns. Each pattern represents a set of services that are tested individually and together using a test driver. At higher levels of integration, these patterns are used and tested by subsystems and finally the subsystems are integrated into a complete micro-kernel.

Our *incremental testing strategy* is bottom-up involving test drivers to exercise the critical lower-level system objects. Incremental bottom-up testing is not a new approach, but seems very effective and appropriate for micro-kernels designed in an object-oriented fashion. In this way, individual objects can be tested using their own test drivers. They are then integrated and the pattern is tested. Patterns encapsulate requests and decouple subsystems. They are not only useful frameworks but they simplify and ease the test and validation of each individual subsystem contributing to a straightforward step-by-step initial micro-kernel's port.

Our *integration testing strategy* is a systematic technique for constructing patterns, subsystems, and micro-kernels while at the same time conducting tests in small steps, where errors are easier to isolate and correct.

2. **Dynamicity.** These patterns make policies interchangeable not only at compile-time, but also at run-time using reflective operations. These operations `Info` (reify) and `UnBind/Bind` (reflection) give to an application the ability to replace a subsystem dynamically.

Pattern	**Object**	**Test**
Scheduler	Scheduler	8
	Context Switch	7
Timer	Real-Time Clock	6
	Critical Section	5
Object Manager	Delta List	4
	Priority Queue	3
	Descriptor Stack	2
	Kernel Input/Output	1

Table 1: Bottom-up integration tests

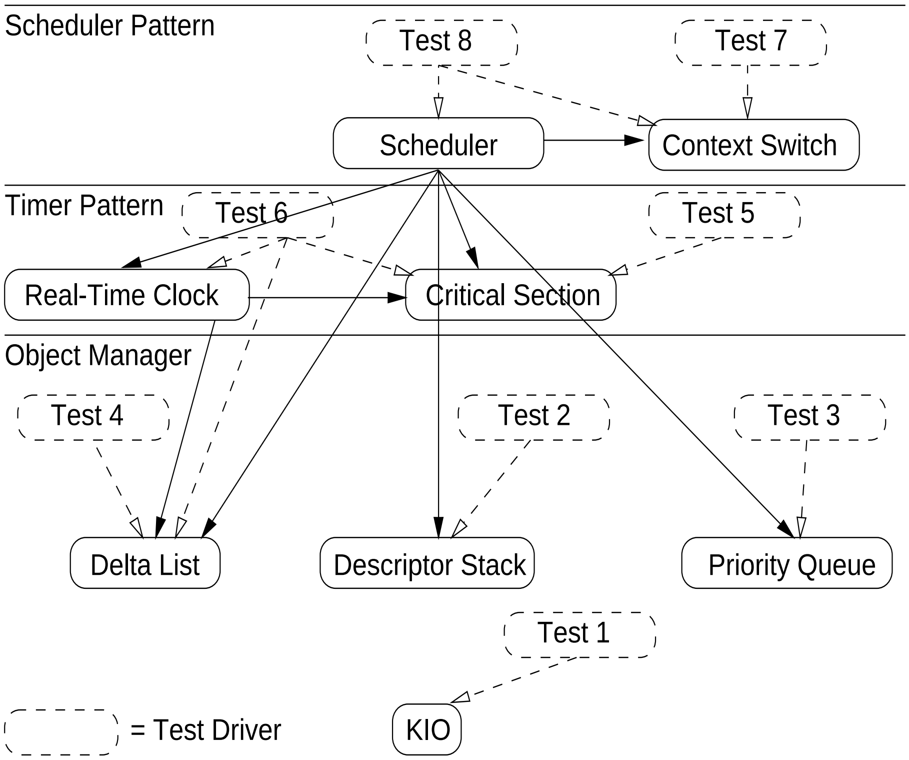


Figure 2: Testing View of Patterns in Layers

Applicability

Table 1 shows how the patterns actually simplify porting. This simplification is denoted by the systematic order of the tests involved in verifying the functionality of all design patterns before linking them with the rest of the micro-kernel, that is all the other subsystems.

The first step in testing is for the kernel input/output (KIO) object on the target system. This basic object is mainly used by all test drivers (1-8), other objects, and patterns for porting and debugging purposes. KIO is transparent to whatever communication link (such as serial and memory mapped) used to reach the target system. When the port is done, KIO can be replaced by a more sophisticated interface that is more suitable for applications.

Our bottom-up integration testing strategy is achieved on the machine-dependent patterns with the following layers:

1. Foundation embedded objects (tests 1-4) are tested individually by their own test drivers. These objects are then combined into the *Object manager* pattern.

2. The real-time clock and critical section objects (tests 5-6) are tested individually by their own test drivers. These objects are then combined into the *Timer* pattern.

3. The scheduler and context switch objects (tests 7-8) are tested individually by their own test drivers. These objects are then combined into the *Scheduler* pattern.

After each of the above layers where each individual object has been tested, the pattern is tested (top-down testing of the layer), the test drivers (in dotted lines, see Figure 2) are removed and the pattern is combined and moved upward in the micro-kernel structure. It is worth mentioning that the upper steps validate also the descriptor stack, the priority queue, and the delta list.[2]

Consequences

These patterns have the following benefits and constraints:

1. **Portability.** Less effort is required to achieve the initial port of a micro-kernel to a new architecture by solving different portability problems [15]. Mainly our patterns do not depend on particular run-time features or libraries which are not shared by all implementations. For example, the *Scheduler* pattern isolates the context switch mechanism in a portable interface.

Another problem that we are currently addressing is the scalability of basic types for different machine architectures (from 8-bit to 64-bit). We are working on a scalable interface using objects to avoid coding any representation dependencies into the patterns.

2. **Predictability.** The execution time of these patterns are bounded offering a deterministic behavior more suitable for embedded soft real-time applications. For example, all individual objects in our *object manager pattern* have been measured to evaluate their linearity $O(n)$ when they manipulate n objects (where n is bounded from 1 to 255). Only some operations require searching a position in a list or a queue and are $O(lg\ n)$. Since the number of objects is bounded to 255, this gives predictability to our operations. Some more performance measurements of these patterns could be presented if relevant. See [12] for some details.

Known Uses

OpenKernel is our latest embedded micro-kernel [11]. The main principle of OpenKernel is to provide a simple and extensible micro-kernel with a minimal

---

[2] A delta list simulates multiple real-time clocks by sorting and linking all timeout requests together as delta-times. Each tick decrements only the first request on the list. This request is removed when it gets to zero (time expiration).

number of subsystems. The overall architecture is composed of two major structures (see Figure 1): *machine-dependent patterns* (dark shaded) and *machine-independent subsystems* of OpenKernel (light shaded). Machine-dependent patterns are object manager, timer, and scheduler. The OpenKernel architecture also provides reflective operations to allow replacement of the majority of its subsystems (Other Subsystems in Figure 1) with different policies at run-time.

An application can modify the micro-kernel's behavior during runtime. Each subsystem specifies a uniform interface supporting the necessary reflective operations that replaces policies in response to the needs of the application. For example, an application designer can dynamically replace the current scheduling algorithm by a more appropriate one according to the system's run-time behavior [16, 12]. Our micro-kernel is limited to soft real-time and focuses on the objectives of flexibility and reuse.

3 Patterns in Embedded Context

Having introduced the overall approach of our patterns, we will now explain them in detail. The patterns presented in this section are based on the diagrammatic notations of Gamma et al. [17].

3.1 Pattern 1: Object Manager

Intent Implement and manage three basic abstractions, called *foundation embedded objects*. The objects, including a stack for descriptors, a priority queue, and a delta list are needed by all the other patterns in OpenKernel.

Motivation Provide useful and mandatory abstractions needed to design and implement subsystems in micro-kernel. The stack (DescriptorStack) is a container of generic descriptors for all types of objects in the micro-kernel. The queue (PriorityQueue) provides possible instances of queues to manage the order of resources by priority or deadline. Finally the delta list (DeltaList) manipulates objects with time requests.

Applicability Use this pattern whenever you need a deterministic behavior or bounded execution time for embedded soft real-time applications. These foundation embedded objects are responsible for managing pre-allocated descriptors.

Structure Table 2 summarises the operations of the foundation embedded objects:

DescriptorStack	**PriorityQueue**	**DeltaList**
Init()	Init()	Init()
Clear()	Clear()	Clear()
Count()	Count()	Count()
IsNotEmpty()	IsNotEmpty()	IsNotEmpty()
Get()	Get()	Get()
Put()	—	Put()
—	PutFirst()	—
—	PutLast()	—
—	—	Tick()
—	—	Check()
—	—	MakeUp()

Table 2: Foundation embedded objects interface.

Participants

DescriptorStack stores pre-allocated and available objects.

PriorityQueue manipulates objects by priority or by dead-line.

DeltaList manipulates objects with timeout requests.

KIO provides a portable interface mainly for debugging purposes.

Collaborations

These objects are private to OpenKernel in that both the representation and the operations are managed only by the patterns of OpenKernel. The *timer* pattern uses DeltaList, whereas the *scheduler* pattern uses DescriptorStack and PriorityQueue.

Consequences

Some of the benefits of the Object manager are as follows:

1. **Predictability.** Their operations are deterministic, that is their execution times are measurable and predictable (again here, measurements can be provided [12] if relevant).

2. **Simplicity, uniformity, and minimality.** These objects were kept simple and minimal. We wanted uniform and consistent interfaces to ease their understandability and usage (see Table 2).

3.2 Pattern 2: Timer Pattern

Intent

Decouple a scheduler from a real-time clock handler.

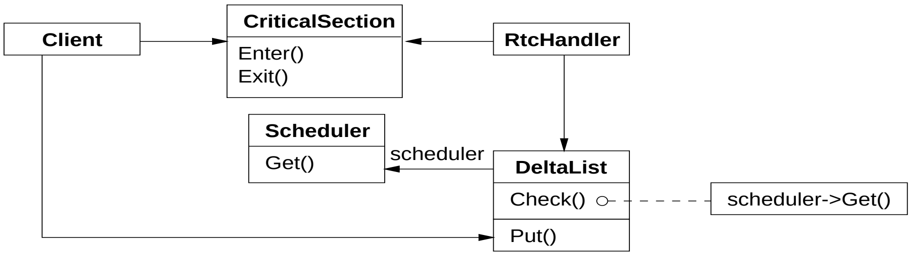
Motivation

Allow decoupling of a scheduler from a real-time clock handler (RtcHandler) by using a DeltaList object to encapsulate the interrupt from the real-time clock device. This is achieved by parameterising an object (DeltaList) with a procedure to call (Scheduler).

Applicability

This pattern may be used when you have objects that need to be scheduled or you have generation of timing signals. Test 6 (see Figure 2) illustrates an application of this pattern as a top-down test that emulates the behavior of an application (client) and a scheduler.

Structure



Participants

DeltaList defines a call-back between a Scheduler object (scheduler) and a call (Get). It implements Check by invoking the corresponding operation(s) (in this case scheduler->Get) on Scheduler to return possible expired object(s). It also implements Put that allows a Client to perform a time request for a specific object.

Client performs a time request (using Put) via a micro-kernel call.

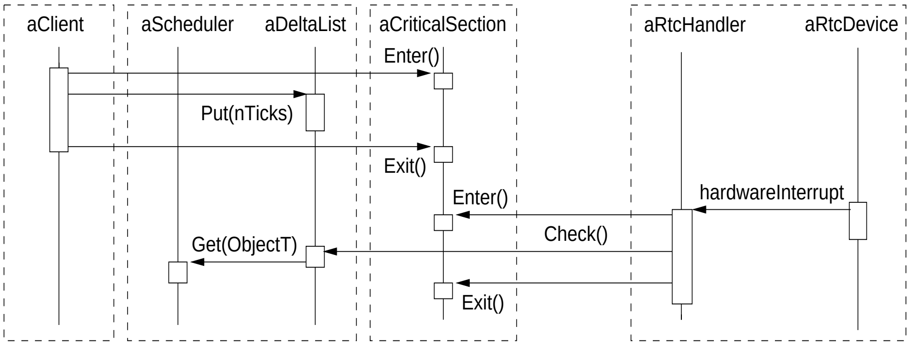
RtcHandler periodically updates the DeltaList object and calls Check to verify the presence of expired object(s).

Scheduler gets a time request from a specific active object and decides how to coordinate active objects based on its current scheduling policy.

CriticalSection ensures a mutual exclusion access by the client to the delta list which is periodically updated by the real-time clock handler.

Collaborations

A Client executes a time request by specifying a number of ticks (using Put). Concurrently, a real-time clock device RtcDevice generates periodic interrupts that are serviced by a RtcHandler. Each handler's service issues a Check to a DeltaList that calls a Scheduler using Get to return possible expired objects (via the parameter ObjectT) to be eventually rescheduled. The following diagram shows the interactions between these objects. It illustrates how the delta list decouples the real-time clock handler from the scheduler.



Interfaces

The following interfaces illustrate the critical section and real-time clock objects. These objects are encapsulating the real-time device and the enable/disable hardware interrupt mechanism on a specific platform. In this way, they isolate the timer pattern from the hardware. For the critical section, the Init operation installs the critical section object. The Enter and Exit operations ensure an exclusive access of a code's section shared by patterns. A critical section can be nested inside another. For the real-time clock, the Init operation installs the RtcHandler (see Structure). The Start operation enables the RtcDevice to generate hardware interrupts. The Stop operation disables the RtcDevice. All of these operations are invoked exclusively by the Scheduler.

CriticalSection
void Init()
void Enter()
void Exit()

RealTimeClock
void Init()
void Start()
void Stop()

Consequences

Timer pattern has the following benefits:

1. **Isolation of critical testing.** Isolate the testing of critical machine-dependent objects such as the real-time clock handler and the critical section (see Figure 2).

2. **Decoupling.** Completely decouple the scheduler from the timer pattern.

3. **Predictability.** Operations at the level of this pattern (real-time handler and critical section) and at object manager level (delta list) are bounded [12]. The cost of removing expired objects is linear ($O(n)$), since there is no list traversal. The worst cost of inserting objects is $O(lg\ n)$ since the list has a fixed size.

See also Command pattern [17], also known as call-back.

3.3 Pattern 3: Scheduler Pattern

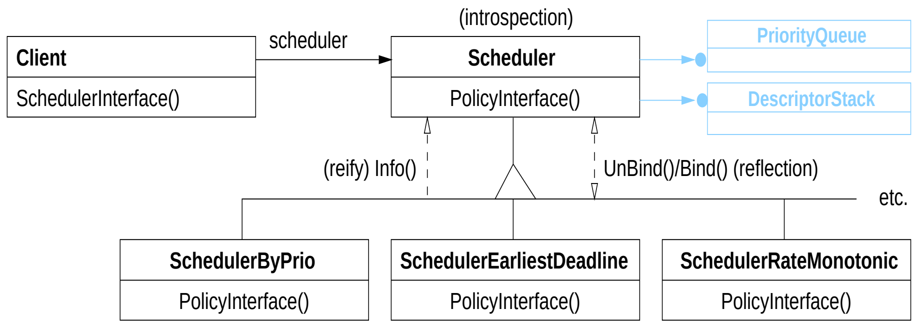
Intent Define a family of scheduler policies.

Motivation Provide reflective operations that replace policies in response to the needs of the application. A basic protocol for reflection programming (also called metaobjets protocol [18]) suggests at least three operations:

- *reify*—inspect the internals of an object,
- *introspection*—take a decision based on the information, and
- *reflection*—change a policy of object management.

Applicability As an alternative to inheritance, this pattern may be used to build system on object/metaobject separation and metahierarchy increasing the *dynamicity* of the system. These policies are interchangeable at run-time. This scheduler pattern lets the current policy be replaced by a more appropriate one according to the system's run-time behavior.

Structure



Participants Scheduler declares an interface common to all supported policies.

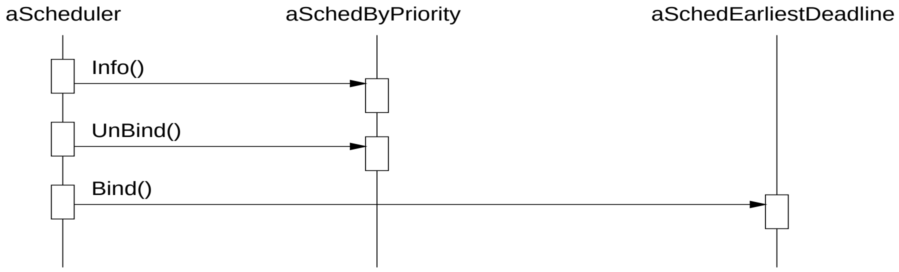
SchedulerByPrio, SchedulerEarliestDeadline, SchedulerRateMonotonic, etc. implement different policies using the Scheduler interface.

Client uses this interface to unbind the current scheduling policy and bind another one.

Each concrete strategy uses two related objects part of the *object manager* pattern: PriorityQueue and DescriptorStack. These objects are also participants (light shaded) and are used internally by the *scheduler* pattern. The DescriptorStack makes instances of objects and the PriorityQueue manipulates objects by priority or by deadline.

Collaborations

The following diagram shows the interaction between these objects. It illustrates how a client reifies (Info) the current scheduling policy, takes a decision based on that information (introspection), and replaces the current policy (Unbind) by a new one (Bind).



Interfaces

The following object illustrates our scheduler interface:

Scheduler
void Init()
InfoT Info()
char* Policy()
void Bind(SchedT)
void UnBind(SchedT)
void Start()
void Stop()
void Lock()
void UnLock()
Bool IsLocked()
void Defer()
void Schedule()
void PutLast(ObjectT)
void PutFirst(ObjectT)
void Get(ObjectT)

Consequences

The Scheduler pattern has the following benefits and constraints:

1. **Isolation of critical testing.** Isolate the testing of a critical machine-dependent object: context switch (see Figure 2).

2. **Dynamicity.** Make scheduling policies interchangeable at run-time. The Strategy pattern [17] also make policies interchangeable, but rather at compile-time, since inheritance is static and implies creation time sharing. The Scheduler pattern refines Strategy by supporting lifetime (run-time) sharing of policies (via reflective operations). Reflective operations are a flexible alternative to inheritance when the programming language used to implement the target embedded system is object-based.

See also

Strategy pattern [17] used inheritance to support a variety of policies, algorithms or behaviors. In the Scheduler pattern, reflective operations are used instead as an object-based alternative.

4 Related Work

Our architecture is similar to the recent redesign of the Choices [3] object-oriented operating system in μChoices. μChoices breaks the original micro-kernel into a nano-kernel that encapsulates the hardware through a single interface. Here the main goal was to provide a close approximation to real hardware in an idealised retargeable UNIX virtual machine. Several ports have been done on top of UNIX variants (SunOS, Solaris and Linux).

OpenKernel is not a general purpose operating system (OS) like Unix. OpenKernel is different in the way that it provides incremental patterns that simplify testing and debugging of all machine-dependent objects. These objects, even those private to other subsystems, offer interfaces that can be tested independently.

Another important difference is that our patterns were designed to be used in embedded systems and are not necessarily for Unix-like systems. We are currently porting OpenKernel to Solaris strictly as an instructional embedded micro-kernel environment for our students.

Several OS researchers [19, 20, 21] mention the importance of matching OSs to application needs in the next generation of OS. Other aspects that are beyond the scope of this paper, but need further investigation, are features that are not supported by OpenKernel (such as virtual memory, file systems, caching, etc.) and which are certainly essential for a full scale multiuser distributed time-sharing system (like Solaris) but are an overkill for an embedded OS.

5 Conclusions

Each domain can have patterns outlining how to design certain applications. Currently our main concern is the embedded systems domain where more usable patterns need to be derived as vehicle for communicating mature designs in an efficient way. If we classify our patterns based on the Gamma catalog, the scope of our patterns are object-based. Our *object manager* is a mixed of creational and structural objects managing the availability and the collection of pre-allocated objects. Our *timer* and *scheduler* patterns are *behavioral object patterns* in which objects decouple, interact, and define policies. The combination of our patterns in the micro-kernel facilitate testing and can be classified as *structural object patterns*. While continuing to improve our patterns we intend to apply this experience to other subsystems of OpenKernel, such as memory management, inter-task communication, etc.; and also to express the reflective operations `Info` (reify) and `UnBind/Bind` (reflection) which give to an application the ability to replace a subsystem dynamically.

Using patterns has proved to be a valuable tool in reducing the complexity of porting OpenKernel. We believe that our patterns are more than objects that work together to solve a design problem, they are also a way of solving the testing problem, i.e., a way of organizing our (human) thought around this/these problem(s). In fact, our patterns put together other patterns to address the porting and testing issues. They are a useful pattern language for rapid porting of micro-kernels to several hardware platforms. We hope that the design patterns described in this paper will help to communicate a reusable design to the operating system community by motivating other designers to describe the patterns they use to ensure portability. Such patterns are usually learned only by experience.

While our work occurs in the context of embedded systems, we believe our approach will also be useful to port general purpose micro-kernels across different processors in a more systematic and rapid manner.

Acknowledgement

I wish to acknowledge my shepherd, Bob Hanmer, for his insightful suggestions and careful review that much clarified and improved this paper. I would like to thank Paul Ashton, Wolfgang Kreutzer, Paddy Krishnan, and Bruce McKenzie for their many constructive comments to earlier iterations of this paper.

References

[1] K. Thompson, "Unix Implementation," *Bell System Technical Journal*, vol. 56, no. 6, 1978.

[2] "Workshop proc. on micro-kernels and other kernel architectures," *USENIX Association*, Apr. 1992.

[3] S. Tan et al., "An Object-Oriented Nano-Kernel for Operating System Hardware Support," in *Proceedings of the Fourth IWOOOS*, (Lund, Sweden), IEEE Computer Society, Aug 1995.

[4] B. N. Bershad et al., "SPIN—an extensible microkernel for application-specific operating system services," *SIGPlan Notices*, 1995.

[5] M. Acceta et al., "Mach: a new kernel foundation for unix development," *Proceedings of the Summer 1986 USENIX Conference*, July 1986.

[6] D. Cheriton, "The V distributed system," *CACM*, vol. 31, pp. 314–333, Mar. 1988.

[7] A. Bricker et al., "A new look at microkernel based unix operating systems: Lessons in performance and compatibility," *Chorus Systèmes, CS/TR-91-7*, 1991.

[8] C. Pu, "Evolving Operating Systems and Architectures: How Do Kernel Implementors Catch Up?," in *Proceedings of the Second IWOOOS*, pp. 143–145, IEEE Computer Society, Sept. 1992.

[9] M. de Champlain, "Synapse: A real-time programming language," Master's Thesis, Computer Science Department, Concordia University, Montréal, QC, Canada, September 1989.

[10] M. de Champlain, "An Object-Based Real-Time Executive," *The C Users Journal*, vol. 8(12), pp. 73–88, December 1990.

[11] M. de Champlain, "OpenKernel Micro-kernel: Reference Manual," *Technical Report, Collège militaire royal de Saint-Jean, Computer Science and Engineering Department, Richelain, QC, Canada*, January 1994.

[12] M. de Champlain, "Modèle réactif and réflexif pour méta machines à états," Thèse Ph.D., Département de génie électrique and de génie informatique, Ecole Polytechnique de Montréal, Université de Montréal, Montréal, QC, Canada, 1995.

[13] J. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation-systems," *Computer*, vol. 21, pp. 10–19, Oct. 1988.

[14] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages.* Reading, MA: Addison-Wesley, 1990.

[15] I. Sommerville, *Software Engineering, 5th Edition.* Reading, MA: Addison-Wesley, 1995.

[16] M. de Champlain and J. Houle, "A reactive and reflective model for meta-state machines," *Proceedings of the Tenth IEEE Workshop on Real-Time Operating Systems and Software, New York, NY*, 1993.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[18] G. Kiczales et al., *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[19] M. Wilkes, "Operating systems in a changing world," *SOSP14*, Dec. 1993.

[20] A. P. Black and J. Walpole, "Objects to the rescue! or httpd: the next generation operating systems," *OSR*, vol. 29, Jan. 1995.

[21] P. Paradinas and J. J. Vandewalle, "New directions for integrated circuit cards operating systems," *OSR*, vol. 29, Jan. 1995.