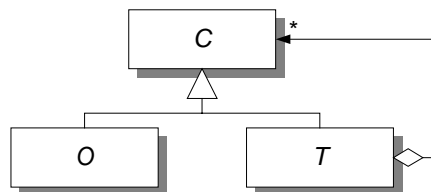


***A C++ Framework for Active Objects in
Embedded Real-Time Systems***
– *bridging the gap between modeling and implementation*

COT/2-16-V2.0



Centre for Object Technology

A C++ Framework for Active Objects in Embedded Real-Time Systems – bridging the gap between modeling and implementation

Michael E. Caspersen
Centre for Object Technology
Department of Computer Science, University of Aarhus
Aabogade 34, DK-8200 Aarhus N, Denmark
mec@daimi.au.dk

Abstract

In research communities it is now well-accepted that the object paradigm provides a good foundation for the challenges of concurrent and distributed computing. For practitioners, however, it is not without problems to combine the concepts of processes and objects. A major reason for this is that the predominant object-oriented programming language in industry, C++, does not support concurrency. In this paper we present a simple and powerful approach to extending C++ with constructs for concurrent programming. We discuss the design, application, and implementation of a framework that supports standard concurrency constructs and, contrary to what is suggested in several books on object-oriented modeling techniques for real-time systems, we demonstrate that it is possible to integrate the notions of object and process and maintain a smooth –virtually non-existing– transition from modeling to implementation. The framework has been used at Bang & Olufsen, a major Danish manufacturer of audio and video equipment, to develop software for a CD player; we illustrate the strength of the approach with examples from this project. As a pleasant side benefit, the framework approach results in a considerable reduction of the code size of more than 50% compared to earlier versions of the system.

Keywords: Object technology, concurrency, active objects, embedded systems, real-time methods, frameworks, C++.

1. Introduction

In the design of embedded real-time systems it is often appropriate to employ concurrent processes, among other things to handle the system's interaction with external devices, and industrial companies have many years of experience in doing this. In some cases a dedicated concurrent programming language is applied (e.g. Concurrent Pascal, Modula-2 or Ada), but most often a traditional sequential programming language is used (e.g. C) where a function library provides access to relevant primitives in an underlying real-time operating system.

The object-oriented paradigm is becoming more and more prevailing in industry, also within the design and construction of embedded real-time systems. Companies at large have come to realize that comprehensive solutions require combining the flexibility of concurrency (and distribution) with the power of object-oriented technology. This calls for a combination of object and process; this combination is captured in the notion of an *active object*.

In research communities it is now well-accepted that the object-oriented paradigm provides a good foundation for the challenges of concurrent and distributed computing [10, 16, 19, 20, 21, 23, 25, 29], and a number of widespread object-oriented programming

languages are supporting concurrency [4, 7, 28]. As such it is –at least in theory– fairly straightforward to combine the two programming paradigms.

1.1. Transformational Approaches to Combining OO and Concurrency

For practitioners, however, it is not without problems to combine the concepts of processes and objects. The literature provides constructive and helpful techniques on modeling [5, 13, 17, 26], but after all –when it comes to the refinement of domain models to a technical design and code for a specific target environment– there is not much to come for. Often, the target environment (processor, programming language, and real-time operating system) is given a priori due to organizational and business related constraints. This preselection gives rise to the problem because it excludes the “easy” and obvious solution: an object-oriented programming language that supports concurrency. The question is how to transform an object-oriented design with implicit concurrency into an implementation with explicit concurrency on a given target environment. Unfortunately, it is hard to find constructive, specific, and satisfactory answers to this question in the literature.

- [17] recognizes the problem but does not provide a solution: “it is essential to consider how [the target environment] can be adapted to fit the object structure, so that the structure is minimally distorted”.
- [5 and 26] prescribe a mapping from object model to task model, i.e. the concepts are not integrated.
- [13] assumes class and object models to be fundamentally concurrent and notes very briefly that active objects are normally implemented as “the root composite object of a thread”; otherwise he does not address the problem.

The objective in [17] is attractive because it aims at preserving the object structure in the implementation; the objective is to adapt the target environment to the object structure and not vice versa. However, there is no indication at all of how to achieve this.

In [26] the authors present a modeling language, ROOM (Real-Time Object-Oriented Modeling), but when it comes to implementation and integration of the ROOM model into an existing run-time environment they resort to transforming the ROOM model specification into an implementation based on concepts supported by more traditional real-time environments (e.g. processes, semaphores, and event queues) – a mapping into low-level concepts that introduce an undesirable distortion to the object structure.

OCTOPUS [5] is a method that provides a systematic approach for developing object-oriented software for embedded real-time systems. The method is based on OMT [24] and Fusion [12] but also embodies common practice found in real-time systems. OCTOPUS makes a distinction between the notion of an implicit and an explicit concurrency model; the main difference between the two is the timing of combining the objects and processes. The OCTOPUS method uses the implicit concurrency model in the analysis phase. In the design phase, the concurrency is gradually made more explicit, and a stepwise procedure for transforming the implicit concurrency model into an explicit is presented.

The suggestions in [26] and [5] of transforming the object model into a task model are rooted in necessity more than desire. In [5] the authors are very honest and explicit about the reasons for their recommendations concerning the transformation from an implicit to an explicit concurrency model and they acknowledge the missing link.

In the endeavour of being a practical approach, the OCTOPUS method presumes that a standard object-oriented programming language and a conventional real-time operating

system are used, but this introduces the key challenge of combining object-orientation and concurrency [5, pp. 115-116]:

Commonly used object-oriented languages, such as C++, are sequential by nature. They do not support implicit concurrency of the objects [...]. Because of this deficiency, the required feature is made available through the operating system. [...] Unfortunately, this preselection requires a combination of two distinct concepts, processes and objects, which in itself is not without problems. [...]

It may be that the relationship between processes and objects is only a difficulty for practitioners, because the existing literature does not give a clear answer to this question. Some authors, such as [Jacobson 92], recognize the problem and give a few guidelines, but these are not sufficient for practical use. For example, the statement “it is essential to consider how the target environment can be adapted to fit the object structure, so that the object structure is minimally distorted” indicates a difficulty without giving any solution.

As already noted, the solution of the OCTOPUS method is a transformational approach where concurrency aspects is introduced rather late.

1.2. Hand-in-hand modeling of objects and processes

In our opinion, the transformational approaches suggested in [5, 26] are unfortunate. We believe that it is undesirable and problematic to transform the object model and this should be avoided as far as possible.

The problem with the transformational approach is that it violates one of the corner stones of object-orientation: maintaining the same fundamental structure at all levels of description – from the most abstract architectural descriptions and down to the code that actually implements the artifact being constructed. Not only is it nice to stick to a fundamental structure, it is vital. Without a common fundamental structure, it becomes virtually impossible to switch forth and back between modeling and implementation and to keep the models and documentation up-to-date.

Furthermore, we firmly believe that the modeling of domain level concurrency (as opposed to design level concurrency introduced for efficiency or due to hardware constraints) is as important as traditional class modeling. In our opinion, any analysis model that claims to cover all essential aspects of a domain must encompass a model of the concurrency inherent to the domain. We therefore dislike that the OCTOPUS method postpones concurrency modeling to a late stage of the design phase. In short, we prefer early modeling of concurrency because we believe it to be essential and because we want to avoid later transformations that might distort the object model.

Regarding the practical aspects of combining an object structure with the target environment, we suggest a switching of the mind-set:

It is not the object structure that must be transformed to match the target environment; it is the target environment that must be adapted to match the object structure.

There are two reasons for this objective. First of all, we want to eliminate the need to transform an object model into a task model as described above. Secondly, we want to lay the ground for design level re-use by encapsulating the target environment in general abstractions that facilitates a smooth integration with an object (and concurrency) model.

To bridge the gap between modeling and implementation we have constructed a C++ framework for active objects in embedded real-time systems; the paper reports on the design, application and implementation of this framework.

1.3. Structure of the paper

In section 2 we recall fundamental concepts and notations for concurrent programming that plays a role in our approach to integrating object-orientation and concurrency. In section 3 we present our solution to the question of how to integrate object-orientation and concurrency: a C++ framework for active objects. In section 4 we illustrate the applicability of our approach with an example from an industrial project with Bang & Olufsen, a major Danish manufacturer of audio and video equipment: software for a CD player. In section 5 we discuss a couple of nice aspects of the implementation of the framework, and section 6 is a wrap-up with conclusions, references to related work, and indications of possible future work.

The ideas and the approach presented is independent of any particular programming language; all that is required is an object-oriented programming language and a real-time operating system. However, for practical reasons and to avoid unnecessary generality, the presentation will be in terms of C++, the programming language that was used in the project with Bang & Olufsen.

2. Classical Language Constructs for Concurrency

Fundamental concepts and notations for concurrent programming were invented a quarter of a century ago, and these are well-documented in the literature [2, 3, 15]. In particular, the concepts of *process* (a sequential program that may be executed concurrently with other processes) and *monitor* (a means of synchronizing access of processes to shared variables) were invented. Both concepts were supported in the programming language Concurrent Pascal [8], and this language has had a major impact on the design of our C++ framework for active objects.

2.1. Producer/Consumer in Concurrent Pascal

A classical example of a concurrent program is the producer/consumer example. One (type of) process, the *producer*, produces items to be consumed by another (type of) process, the *consumer*. To decouple the two processes, a buffer is placed between them; this arrangement allows the producer to produce new items even if the consumer temporarily is incapable of keeping pace with the producer. To synchronize access to the buffer, we make it a monitor; this ensures exclusive access to the data structure defining the internal representation of the buffer. A class model of the system is described in figure 1; we use the stereotypes *Active object* and *Monitor* to decorate the class diagram.

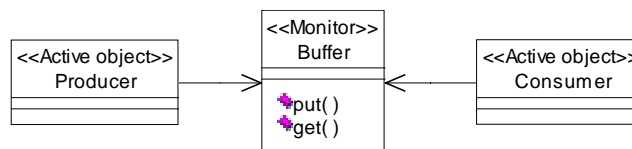


Figure 1: Class diagram for producer/consumer

An implementation in Concurrent Pascal is straightforward and is shown in figure 2.

```
Buffer = monitor
  var t: item;
      count: integer;

  procedure entry put(x: item);
  begin
    await not full;
    t:= x; count:= count+1
  end;

  function entry get(): item;
  begin
    await not empty;
    count:= count-1; get:= t
  end;

  function empty: boolean;
  begin empty:= count=0 end;

  function full: boolean;
  begin full:= count=1 end;

begin count:= 0 end;

Producer = process (b: Buffer);
  var t: item;
begin
  while true do begin
    produce(t);
    b.put(t)
  end end;

Consumer = process (b: Buffer);
  var t: item;
begin
  while true do
    consume(b.get)
  end;
```

Figure 2: Concurrent Pascal version of producer/consumer

The await-statement used in the monitor operations `put` and `get` is not an ordinary Concurrent Pascal construct. It is used here to synchronize the timing of the producer and the consumer (the producer must wait if the buffer is full, and the consumer must wait if the buffer is empty). In Concurrent Pascal, Brinch Hansen decided to use queues instead, but this was merely a matter of taste [9]; we apply the await-statement for simplicity and because this is the approach we have adopted in the C++ framework.

3. A Framework for C++

There are numerous ways to combine object-oriented programming languages with concurrency constructs: [10] identifies a coarse classification in three approaches: the library

approach, the integrative approach, and the reflective approach; and [14] discusses specific issues related to making concurrent extensions of C++.

Our approach to extending C++ with concurrency constructs is to integrate object and process in the notion of an active object; another approach that refrains from introducing the notion of active object is reported in [22, 23]. Our approach is a combination of the library approach (extend the library rather than the language) and the integrative approach (extend the language) [10]. Our approach resembles the integrative approach by defining new keywords (**entry** and **await**), and it resembles the library approach in the way these keywords (and the classes that realizes them) are implemented: as a class library that constitutes a framework where calls of the library methods are implicit in the application code.

Other C++ extensions exists. The widespread use of C++ has resulted in a proliferation of parallel programming systems for this language [28]. However, most of these systems are experimental or implemented only at general platforms (e.g. Unix) and therefore not immediately applicable in a specific target environment for real-time systems.

The C++ framework we have constructed is heavily influenced by Concurrent Pascal. Simplicity was one of the overall design goals, and as we shall see the framework is simple and easy to use; it does not require a special compiler or pre-processor, and it is straightforward to apply for practitioners.

The framework consists of a few essential abstractions: two new “keywords”: **entry** and **await**; five abstract classes: `Monitor`, `ConditionMonitor`, `Process`, `TimerProcess`, and `Timer`; and four concrete classes: `Semaphore`, `Entry`, `RelativeTimer` and `AbsoluteTimer`. Figure 3 presents the framework as a class diagram.

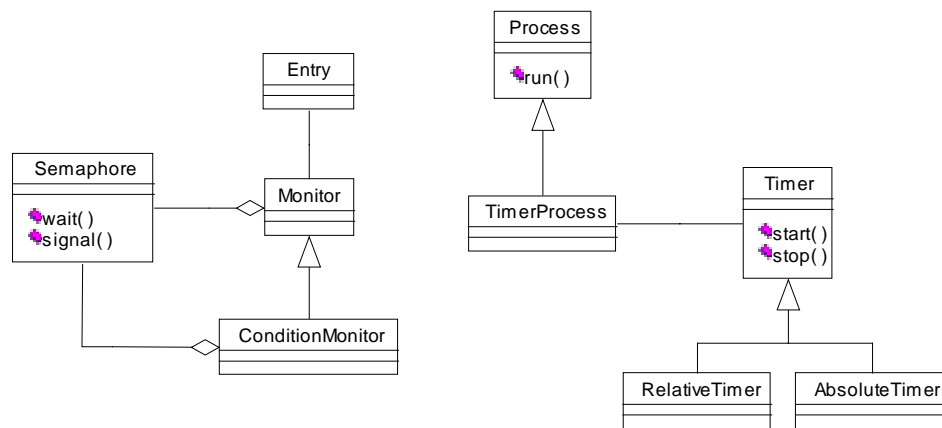


Figure 3: The framework (user perspective)

A specific application with active objects is constructed through extensions of `Process` (or `TimerProcess`, if the active object must own a timer). Subclasses to `Process` (and `TimerProcess`) can redefine the virtual method `run` and thereby define the specific behaviour of active objects. Along the same lines classes that are shared between several processes, must be made subclasses of `Monitor` (or `ConditionMonitor` if a synchronization through an `await`-statement is needed in the monitor).

An active object, in contrast with a normal “passive” object which simply provides services, has its own control agenda; this immediately raises the question of synchronizing

an active object when one of its clients calls one of its services. Many researchers have addressed this problem [1, 11]. Since our notions of monitor and process are totally independent, our approach is simply to make an active object a monitor too, if necessary.

The separation of `Monitor` and `ConditionMonitor`, and `Process` and `TimerProcess` is not crucial but more a matter of taste. `Monitor` and `ConditionMonitor` support two distinct forms of synchronization: mutual exclusion and conditional wait. A `TimerProcess` is simply a `Process` that reacts to time-outs.

Application of the framework represents a smooth transition from domain level modeling to technical design; the transition can be expressed as a refinement of the class model. In figure 4 we illustrate this with the producer/consumer example from the previous section.

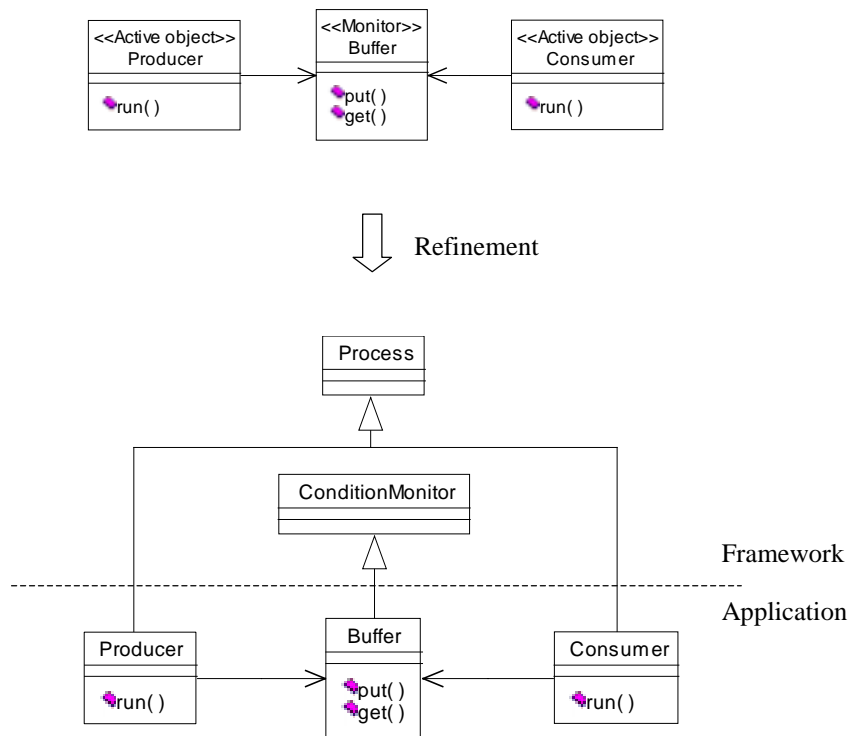


Figure 4: Refinement from design level modeling to technical design

`Producer` and `Consumer` are both realized as subclasses of `Process`, and the virtual method `Process::run` is redefined to reflect the specific behaviour of producers and consumers respectively. `Buffer` is realized as a subclass of `ConditionMonitor` because conditional waits (`await`) are needed, and the methods `put` and `get` are both decorated with the “keyword” `entry` to make them monitor operations and thereby ensure mutual exclusion (see section 5 for details about the implementation of the framework). The “keyword” `forever` is simply shorthand for `for (;;)` . The ultimate C++ version of producer/consumer is presented in figure 5.

```
class Producer : public Process {
public:
    Producer(Buffer& bb) : b(bb) {}
}
```

```

        virtual void run() {
            forever {
                item t;
                // produce t ...
                b.put(t);
            }
        }
private:
    Buffer& b;
};

class Consumer : public Process {
public:
    Consumer(Buffer& bb) : b(bb) {}
    virtual void run() {
        forever {
            item t;
            t = b.get();
            // consume t ...
        }
    }
private:
    Buffer& b;
};

class Buffer : public ConditionMonitor {
public:
    void put(item x) { entry
        await( !full() );
        t = x;
        count++;
    }
    item get() { entry
        await( !empty() );
        count--;
        return t;
    }
private:
    bool empty() { return count == 0; }
    bool full() { return count == 1; }
    item t;
    int count = 0;
};

```

Figure 5: C++ version of producer/consumer

Of course, the simplicity of the C++ code and the structural 1-1 correspondence between the abstract model and the implementation is the key issue.

4. Application for a CD Player

In this section we will demonstrate that the framework presented in the previous section has fulfilled its goal: to make possible a smooth transition from modeling to implementation and to obtain a structural 1-1 correspondence between the model and the ultimate C++ code running in the target environment.

In figure 6 we present a class model of central parts of a CD player. In the model, four classes are decorated as active objects, and four other classes are decorated as monitors. The choice of active classes was guided by one of our design heuristics: choose as active objects those classes that handles the asynchronous events that occurs in the system; the choice of

monitor classes is straightforward: if a class has more than one active object as client, make that class a monitor.

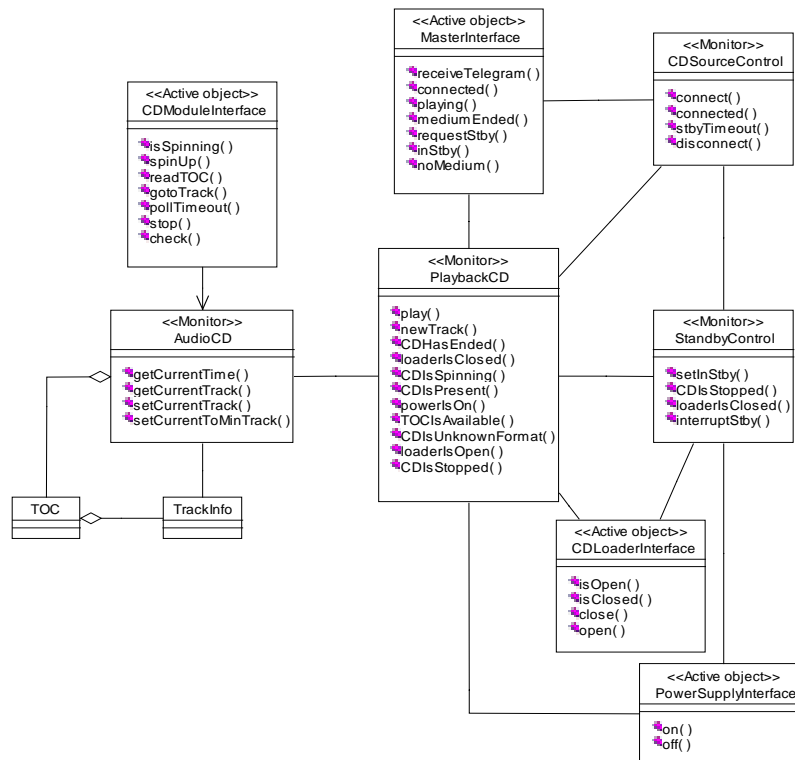


Figure 6: Class diagram for central parts of a CD player

As an example, the ultimate C++ code for the class `PowerSupplyInterface` is shown in figure 7. Notice that there is no explicit run method; the entire task loop is programmed once and for all in `TimerProcess::run` in the framework.

```

class PowerSupplyInterface : public TimerProcess {
public:
    PowerSupplyInterface(PlaybackCD* aPlayBackCD);
    void on();
    void off();
private:
    void freeOfNoise();
    void powerIsOn();
    Timer *freeOfNoiseTimer, *powerIsOnTimer;
    PlaybackCD* thePlaybackCD;
    PowerState state;
};

PowerSupplyInterface(PlaybackCD* aPlayBackCD)
: thePlaybackCD(aPlayBackCD) {
    freeOfNoiseTimer = new RelativeTimer(200, ONE_SHOT,
        (PTimeoutF)&PowerSupplyInterface::freeOfNoise);
    powerIsOnTimer = new RelativeTimer(500, ONE_SHOT,
        (PTimeoutF) &PowerSupplyInterface::powerIsOn);
}

```

```

PowerSupplyInterface::on() {
    turn_on_main_supply();
    freeOfNoiseTimer->start(); // 200 ms
}

PowerSupplyInterface::freeOfNoise() {
    turn_on_serial_port();
    powerIsOnTimer->start(); // 500 ms
}

PowerSupplyInterface::powerIsOn()
{ state = SOn; thePlaybackCD->powerIsOn(); }

PowerSupplyInterface::off() { ... }

```

Figure 7: C++ code for PowerSupplyInterface

From the original specification we know that after the main supply has been turned on, one has to wait for 200 ms to avoid noise on the serial port; after that, the serial port can be turned on. Then one has to wait another 500 ms to be sure that the power is on. This specification (and a similar one for turning off the power supply) is exactly what constitutes the class PowerSupplyInterface. In the constructor a 200 ms relative timer, freeOfNoiseTimer, is created. This timer is started in PowerSupplyInterface::on(), and when the freeOfNoiseTimer times out, the private method freeOfNoise will be executed (the details that make this happen are encapsulated in the framework). Similar things happen with the powerIsOnTimer. Eventually the active object thePowerSupplyInterface calls back to PlaybackCD.powerIsOn, and the music can start playing. Figure 8 gives an overview of this object interaction.

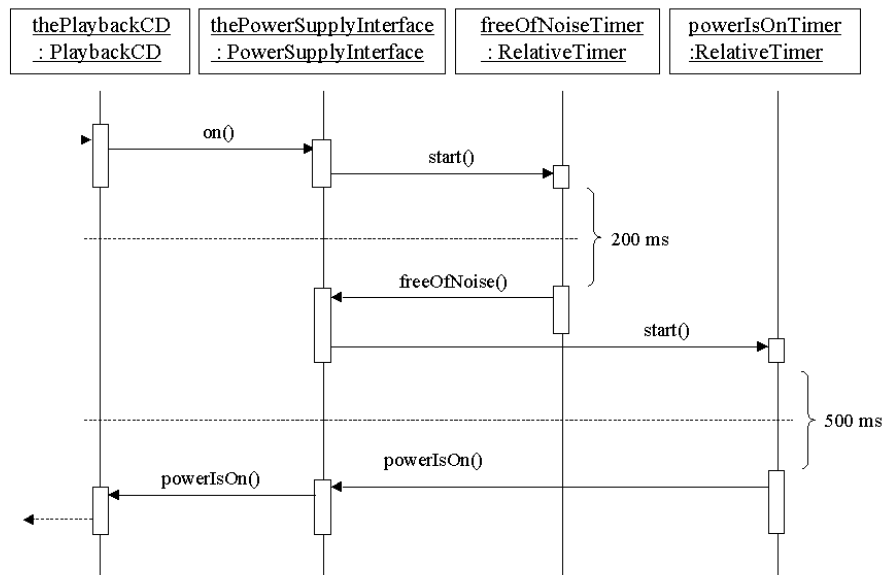


Figure 8: Interaction diagram for PowerSupplyInterface::on

The function calls `turn_on_main_supply` and `turn_on_serial_port` are calls to C functions in the surrounding legacy code. This is typical; at the same time the interface classes are active objects and wrapper classes to existing legacy code.

Admittedly, `PowerSupplyInterface` is a very simple class, but the rest of the classes are implemented in exactly the same fashion, smoothly and simple, resulting again in a structural 1-1 correspondence between the object model and the C++ code. The implementation is straightforward and no distortion at all is added to the object structure.

5. Aspects of the Implementation

In this section we will discuss a couple of aspects of the implementation of the framework; in particular we will present the implementation of the `Monitor` abstraction and the new keyword `entry` as well as the `ConditionMonitor` abstraction and the keyword `await`. The discussion will uncover some of the crucial design decisions and stress our efforts to make the framework as simple as possible to use for application programmers.

The monitor abstraction is implemented as an abstract class that in a standard way uses a binary semaphore to implement two protected methods, `enter` and `exit`. Calls to these methods must surround the code in application-specific monitor methods (in subclasses to the abstract `Monitor` class) to ensure mutual exclusion. As an example, the method `Buffer::put` should look like this:

```
void Buffer::put(item x) {
    enter();

    await( !full() );
    t = x; count++;

    exit();
}
```

However, to avoid explicit calls of `enter` and `exit`, we define a macro, `entry`, which is a declaration of an instance of the class `Entry`. An `Entry` object encapsulates a reference to the monitor in which it is declared, and the constructor of the `Entry` object calls `enter` on its monitor, and the destructor calls `exit` (and `notifyAll`, which has no effect at this level). Consequently, all the application programmer has to do is –like in Concurrent Pascal– to write `entry` at the beginning of a monitor method, and then the runtime system will take care of calling `enter` and `exit` at the right time regardless of how the control leaves the monitor method (e.g. multiple exits, exceptions, etc.). Hence, the method `Buffer::put` ends up being coded like this:

```
void Buffer::put(item x) { entry
    await( !full() );
    t = x;
    count++;
}
```

The implementation of the monitor abstraction does not at all depend on the underlying real-time operating system; it is implemented solely in terms of the class `Semaphore` which is part of the framework. A skeleton of the C++ code for `Monitor` and `Entry` is presented in figure 9.

```

class Monitor {
    friend class Entry;
protected:
    Monitor();
    void enter() { ... s.wait(); ... }
    void exit() { ... s.signal(); ... }
    virtual void notifyAll() {}
private:
    Semaphore s;
    Process* p;
    // used for nested calls
protected:
    int entryCount;
    // used for nested calls
};

class Entry {
public:
    Entry(Monitor& mm) : m(mm) { m.enter(); }
    ~Entry() { m.notifyAll(); m.exit(); }
private:
    Monitor& m;
};

#define entry Entry e(*this);

```

Figure 9: The classes `Monitor` and `Entry`

Along the same lines, the `ConditionMonitor` is implemented as an abstract class that uses a semaphore to implement the conditional wait (see figure 10). The implementation of `await` is to call `wait` which again calls `exit` (inherited from `Monitor`). The call to `exit` releases the monitor for other processes to enter and establish the desired condition. Whenever a process exits the monitor, the refined `notifyAll` will be called (via the destructor of the `Entry` object) and in turn let all waiting processes test their condition. This notification policy is quite coarse-grained; all waiting processes are being notified whenever a process leaves the monitor. However, the current implementation was chosen for simplicity, not for efficiency. If performance turns out to be a problem, we know where to tune the code.

```

class ConditionMonitor : public Monitor {
protected:
    ConditionMonitor();
    void wait() { ... exit(); ... c.wait(); ... enter(); }
    virtual void notifyAll() { ... c.signal(); ... }
private:
    Semaphore c;
    int conditionCount;
};

#define await(cond) while (!(cond)) wait()

```

Figure 10: The class `ConditionMonitor`

We will not get into details about the implementation of the `Process`, `TimerProcess`, and `Timer` abstractions, but only mention one nice thing about the interaction between timers and timer processes supported by our design. When a `Timer` is being created, its constructor takes as argument a pointer to a member function in the `TimerProcess` that creates the `Timer` – the member function that must be executed on time-out. Because of this (and a few similar things), the traditional task loop vanishes completely from the application code and resides only in the framework. This results in a considerable reduction of the application code for active objects, typically more than 50%. For the active object `PowerSupplyInterface` from the previous section, the reduction was more than 80%.

6. Concluding remarks

We have developed a C++ framework for active objects in embedded real-time systems, and the framework has been used to develop object-oriented software for a CD player. The framework integrates the notions of object and process and adapts a target environment to match any object model expressed in terms of the stereotypes *Active object* and *Monitor*.

Due to extensive re-use of code implemented once and for all in the framework –in particular the traditional task loop shared by all active objects in the CD player project– application of the framework gives a surprisingly large reduction of the application code for active objects. However, it is more important that the remaining code is essential code (i.e. problem-specific code); it is the technical, low-level details that are swept under the rug.

The version of the framework that was used in the CD player project employs non-preemptive, priority-based scheduling; this means that all processes at regular intervals must call `Process::pause` – a class method that implements the scheduler. However, the design of the framework is independent of the scheduling policy, and it works as well with preemptive scheduling.

The current version of the framework supports nested monitor calls on the same monitor but not on different monitors although this could easily be implemented.

The framework is implemented directly on top of APOS, the real-time operating system used at Bang & Olufsen [6]; however, in order to generalize the framework and facilitate an easy adaption to other target environments, a re-design in terms of an abstract real-time operating system is planned.

[18] describes the Active Object pattern which decouples method execution from method invocation in order to simplify synchronized access to a shared resource by methods invoked in different threads of control. It would be interesting to investigate the applicability of the Active Object pattern with respect to our framework.

7. Acknowledgements

It is a pleasure to acknowledge Kresten Krab Thorup and Carsten Bjerring for stimulating discussions and collaboration during design and implementation of the framework and Carsten Juel Andersen, Mogens Lauridsen, Kristian Lippert, and Elmer Sandvad for providing valuable comments on a draft version of the paper.

This work was supported by the Danish National Centre for IT Research, research grant COT 74.1.

8. References

- [1] America, P. H. M. "POOL-T: A Parallel Object-Oriented Language", in [29].
- [2] Andrews, G. R. and Schneider, F. B. "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, Vol. 15, No. 1, January 1983, pp. 3-43. Reprinted in [Gehani 88].
- [3] Andrews, G. R. *Concurrent Programming – Principles and Practice*, The Benjamin/Cummings Publishing Company, 1991.
- [4] Arnold, K. and Gosling, J. *The Java Programming Language – Second Edition*, Addison-Wesley, 1998.
- [5] Awad, M., Kuusela, J. and Ziegler, J. *Object-Oriented Technology for Real-Time Systems – A Practical Approach Using OMT and Fusion*, Prentice-Hall, 1996.
- [6] Bang & Olufsen, The APOS Project Group (NIB) *APOS+ (APOS_OS.SPC)*, January 6, 1993.
- [7] Barnes, J. *Ada 95 Rationale: The Language, The Standard Libraries*, Springer-Verlag, 1997.
- [8] Brinch Hansen, P. *The Architecture of Concurrent Programs*, Prentice-Hall, 1977.
- [9] Brinch Hansen, P. *The Search for Simplicity – Essays in Parallel Computing*, IEEE Computer Society Press, 1996.
- [10] Briot J.-P., Guerraoui, R. and Löhr, K.-P. "Concurrency and Distribution in Object-Oriented Programming", *ACM Computing Surveys*, Vol. 30, No. 3, September 1998, pp. 291-329.
- [11] Caromel, D. "Toward a Method of Object-Oriented Concurrent Programming", *Communications of the ACM*, Special Issue on Concurrent Object-Oriented Programming, Vol. 36, No. 9 (September 1993), pp. 90-102.
- [12] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremes, P. *Object-Oriented Development – The Fusion Method*, Prentice-Hall, 1994.
- [13] Douglas, B. P. *Real Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley Longman, 1998.
- [14] Eliëns, A. *Principles of Object-Oriented Software Development*, Addison-Wesley, 1995.
- [15] Gehani, N. H. and McGettrick, A. D. (Eds.) *Concurrent Programming*, Addison-Wesley, 1988.
- [16] Hutchinson, N., Raj, R. K., Black, A. P., Levy, H. M. and Jul, E. *The Emerald Programming Language Report*, Technical Report 87-10-07, University of Washington, Seattle, October 1987.
- [17] Jacobson, I. *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison-Wesley, 1992.
- [18] Lavender, R. G. and Schmidt, D. C. *Active Object: An Object Behavioral Pattern for Concurrent programming*, chapter 30 in [27].
- [19] Lea, D. *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley Longman, 1997.
- [20] Löhr, K.-P. "Concurrency Annotations for Reusable Software", *Communications of the ACM*, Special Issue on Concurrent Object-Oriented Programming, Vol. 36, No. 9, September 1993, pp. 81-89.
- [21] Madsen, O. L., Petersen, B. M. and Nygaard, K. *The BETA Programming Language*, Addison-Wesley, 1993.
- [22] Meyer, B. "Systematic Concurrent Object-Oriented Programming", *Communications of the ACM*, Special Issue on Concurrent Object-Oriented Programming, Vol. 36, No. 9 (September 1993), pp. 56-80.
- [23] Meyer, B. *Object-Oriented Software Construction*, Prentice-Hall, 1997 (2nd edition).
- [24] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [25] Schmidt, D. C. and Fayad, M. E. "Lessons Learned Building Reusable OO Frameworks for Distributed Software", *Communications of the ACM*, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997, pp. 85-87.
- [26] Selic, B., Gullekson, G. and Ward, P.T. *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.
- [27] Vlissides, J. M., Coplien, J. O. and Kerth, N. L. (Eds.) *Pattern Languages of Program Design 2*, Addison-Wesley, 1997.
- [28] Wilson, G. V. and Lu, P. (Eds.) *Parallel Programming Using C++*, MIT Press, 1996.
- [29] Yonezawa, A. and Tokoro, M. (Eds.) *Object-Oriented Concurrent Programming*, MIT Press, 1987.