Building Abstractions for Concurrent Object-Oriented Programming

Ole Lehrmann Madsen

Computer Science Department, Aarhus University

Ny Munkegade, DK-8000 Aarhus C, Denmark

Tlf.: +45 86 12 71 88 - E-mail: olmadsen@daimi.aau.dk

Draft — February 1993

Abstract

A large number of different models for concurrent object-oriented programming have been proposed. None of this models have received widespread acceptance. In this paper the Simula model of quasi-parallel execution is extended to include concurrency and synchronization. Active objects may execute concurrently and access shared objects without synchronization. Synchronization may be obtained by means of semaphores. In order to avoid explicit low-level manipulation of semaphores, it is possible to build high-level concurrency abstractions that encapsulates and hide the details of the semaphores. A class library may then include abstractions corresponding to commonly used mechanisms like monitor and rendezvous. When these mechanisms are not adequate, it is possible to define new concurrency abstractions. This approach to concurrent object-oriented programming makes it possible to use many different concurrency abstractions within the same language. It is also possible to some extent to inherit synchronization constraints.

Category: Research.

1 Introduction

Many models for concurrent object-oriented programming have been proposed. These models include monitors, CSP/Ada-like type of rendezvous, objects with more than one internal thread, objects with a communication manager controlling the execution of its operations [20], and many more [1, 2, 3, 7, 19, 21, 22, 23, 25, 26]. When concurrency is combined with inheritance there is also a large number of proposals for mechanisms for inheriting the protocol or synchronization constraints defined in the superclass [10, 8].

In one end of this large spectrum of concurrency mechanisms there are the high-level abstractions, like monitors and rendezvous, which are easy to use and provide a high degree of security. Their disadvantage is that they are only well suited to express solutions to the class of concurrency problems for which they have been designed. A monitor is ideal for providing mutual access to shared data where at most one operation may be executed at a time. The rendezvous mechanism is well suited to express direct communication between concurrent tasks. There are many concurrency problems where these abstractions are not adequate. (No examples are given here, since it is, not the purpose of this paper to discuss the advantages and disadvantages of monitors and rendezvous). In general, a high-level language abstraction is only useful for a certain class of problems. Consider the analogy with loop abstractions like for-imperatives, while-imperatives and repeat-imperatives. Each of these cover a certain class of loops, but none of them is well suited to express a loop with a test in the 'middle'. If the concurrency mechanisms of a given language are well suited for most purposes then they are in general acceptable. From the literature it seems that many authors often find high-level mechanisms too limited in their usefulness. A symptom on this is that none of the main stream OO languages have high-level abstractions for concurrency.

At the other end of the spectrum, we find the low-level mechanisms like semaphores and locks. These mechanisms are quite flexible, since they can be used in most situations. On the other hand it is well known that it may be quite complicated and a potential source of errors to use e.g. semaphores for synchronization. For this reason many people do not find semaphores to be an acceptable mechanism for synchronization in a high-level language. Smalltalk is an example of a language where synchronization has to be made using semaphores.

The flexibility of the low-level mechanisms and the security and expressiveness of the high-level mechanisms can be combined if it is possible to define the high-level mechanisms as abstractions based on the low-level mechanisms. The object-oriented language may then have low-level mechanisms as primitives and supply a class library of high-level concurrency abstractions corresponding to the most common class of problems. If the concurrency abstractions in the library are not suited for a given class of problems, the user has the possibility of defining his/her own abstractions. In this paper it will shown how it is possible to define high-level abstractions like monitor and rendezvous based on semaphores within a object-oriented language with powerful abstractions mechanisms.

The first object-oriented language Simula has support for quasi-parallel processes. A Simula object may be **active** in the sense that it has a body consisting of a sequence of imperatives that may be executed. A simula object may be executed as a coroutine, i.e. it works as the head of an execution thread. A Simula program is a collection of interacting quasi-parallel coroutines. Few modern object-oriented languages have active objects as in Simula.

One exception is the BETA language which we will use to illustrate the concurrency mechanisms in this paper. In main-stream object-oriented languages such as C++, Eiffel and Smalltalk, an object is a passive entity that does not have an active body as in Simula. A BETA object may be either passive or active. An active BETA object may be executed as a coroutine as in Simula, but it may also be executed concurrently with other active objects. An active BETA object is thus the head of a thread of procedure activations and such a thread may be executed as a coroutine (corresponding to pre-empty scheduling) or concurrently (corresponding to non-preemptive scheduling).

The original model for communication and synchronization in BETA was similar to the rendezvous mechanism of Ada. In this model two objects may communicate by means of synchronized execution of procedures. This model, however, appeared to be too restrictive for many classes of problems. A new simpler model for concurrency has therefore been developed. Instead of insisting on all communication and synchronization taking place using rendezvous, objects may execute concurrently and access shared objects without any synchronization. Objects may synchronize access to shared objects by means of *semaphores*. In order to avoid using semaphores directly, it is possible to define high-level concurrency abstractions that encapsulates and hides the manipulation of semaphores.

The main point of this paper is to show that in a language with powerful abstraction mechanism it is possible to build abstractions in the form of abstract super classes which can cover a large number of different communication models.

The basic idea is to make use of "prefixed procedures" as found in BETA and which has also been proposed by Jean Vaucher in [24]. He has also shown how the concept of "prefixed procedure" may be used to define abstract super classes for Hoare's monitors. In this paper we extend these ideas and show how more elaborate communication models can be abstracted. In addition examples of inheritance of synchronization constraints will be given.

The paper presents the basic model of concurrency and synchronization in BETA. It is then shown how to define high-level concurrency abstractions using BETA patterns. The examples of high-level concurrency abstractions are well know mechanisms like monitor and rendezvous, but it is also shown how to define control abstractions allowing more than one concurrent thread in an object. The point of this paper is **not** to propose a concurrent object-oriented language with monitors and rendezvous, but to show how such abstractions can be defined from simple primitives using a powerful object-oriented language. Monitors and rendezvous have been chosen to illustrate the ideas in this paper, since these abstractions are well-known and have demonstrated their usefulness for a large class of problems. The Mjølner BETA System includes a library of concurrency abstractions, but a user may easily define his/her own concurrency abstractions.

Although the language used for presentation of the concurrency constructs is BETA, the ideas may be incorporated into most other object-oriented languages. There are, however, some BETA constructs not found in other languages that are essential for the techniques presented in this paper. The most important one is the **inner** construct, but nested classes are also important.

2 Basic Language Constructs

In this section, the language elements of BETA used in this paper will be briefly introduced. It is an advantage if the reader has some previous knowledge about BETA. Especially familiarity with the **inner** construct is important. This may be obtained through some of the references [11, 12, 13, 15, 16, 17, 5]. It is also an advantage if the reader has some knowledge about coroutine sequencing in Simula and BETA. This can be obtained from [4, 6, 14].

The BETA program in figure 1 shows the basic elements of BETA including passive and active objects and the execution of active objects as coroutines. The program defines a number of patterns: `Account`, `Deposit`, `Withdraw` and `Person`. The syntax of a pattern is

```
P: (# ... #)
```

Patterns are a unification of abstraction mechanisms like class, procedure and process in the sense that a pattern may be used as a class, procedure, process, etc. `Account` is used as a class pattern, `Deposit` and `Withdraw` are procedure patterns and `Person` is a process pattern.

Variables may either be objects or references to objects. Object variables are declared like

```
MarysAcc,JoesAcc: @Account
```

or

```
Mary,Joe: @active Person
```

The first example is a declaration of a variable denoting a passive object. The second example declares variables denoting active objects. The above declarations specify part objects that are generated as part of the object containing the declarations.

This in contrast to the following declarations which specify dynamic references to objects

```
(# Account:
    (# Deposit:
        (# amount: @integer
        enter amount
        do amount+balance->balance
        #)
       Withdraw:
        (# amount: @integer
        enter amount
        do amount+balance->balance
        #)
        balance: @integer
    #);
   MarysAcc,JoesAcc: @Account;

   Person:
    (# myAccount: ref Account;
    do cycle(#
       do amount->myAccount.WithDraw;
          suspend
       #)
    #);
   Mary, Joe: @active Person;

   BankAgent: @active
    (#
    do cycle(#
       do amount1->MarysAcc.Deposit;
          amount2->JoesAcc.Deposit;
          suspend
       #)
     #)
do MarysAcc[] -> Mary.myAccount[];
   JoesAcc[] -> Joe.myAccount[];
   cycle(#
   do BankAgent; Mary; Joe;
   #)
#)
```

Figure 1: A program with active coroutine objects

```
myAccount: ref Account;
mySister: ref active Person
```

The variables `myAccount` and `mySister` (`mySister` is not used in the example) are references to objects. They may be assigned a reference by assignments like:

```
MarysAcc[] -> Mary.myAccount[]
```

or by generation of new objects:

```
new Account[] -> myAccount[]
```

It is also possible to specify objects directly without referring to a class pattern. The active object `BankAgent` is an example of such a *classless* (or *singular*) object. The whole program itself is also an example of a classless object.

As mentioned, some objects are passive (`MarysAcc`, and `JoesAcc`) and and some are active (`Mary`, `Joe`, `BankAgent` and the main program). The active objects may execute the imperatives described in their do-part. Such execution of actions may be performed by other objects. In the example, the main program is initially active. After initialization of the `myAccount` attribute of `Joe` and `Mary`, the main program executes a cycle operation which repeatedly executes:

```
BankAgent; Mary; Joe
```

Each of the imperatives `BankAgent`, `Mary`, and `Joe` is a coroutine call. A coroutine call implies that the do-part of the corresponding object is executed and this execution continues until the object executes a `suspend`, after which it returns to the caller. A subsequent call of the object will then resume execution after `suspend`.

The `BankAgent` repeatedly executes the imperatives:

```
amount1->MarysAcc.Deposit;
amount2->JoesAcc.Deposit;
suspend
```

This means that each times it is called from the main program, it will deposit some amounts on the two accounts and then suspend its execution and return to the caller.

The objects `Mary` and `Joe` will similarly repeatedly execute the imperatives

```
amount->myAccount.WithDraw;
suspend
```

These objects will thus withdraw some amounts from their account and suspend execution.

The main program is thus functioning as a scheduler that alternates between calling the three active objects. Each of the active objects executes some imperatives and returns to the scheduler. At most one object is executing at a time, i.e. no concurrency is going on.

```
(# Account: (# {as before} #)
   MarysAcc,JoesAcc: @Account;

   Person:
    (# myAccount: ref Account;
    do cycle(#
       do amount->myAccount.WithDraw;
          ...
       #)
    #);
   Mary, Joe: @active Person;

   BankAgent: @active
    (#
    do cycle(#
       do amount1->MarysAcc.Deposit;
          amount2->JoesAcc.Deposit;
          ...
       #)
     #)
do MarysAcc[] -> Mary.myAccount[];
   JoesAcc[] -> Joe.myAccount[];
   bankAgent.fork;
   Mary.fork; Joe.fork;
#)
```

Figure 2: A program with active concurrent objects

Concurrent execution. The example from figure 1 is rewritten in figure 2, such that the active object are executed concurrently instead of as coroutines. There have been changes in `Person` and `BankAgent` reflecting that they are now executed concurrently. The **suspend** has been replaced by ... indicating that these objects execute some other actions in between accessing the accounts. In this example, the active objects all access the (shared) `Account` objects without any synchronization. In the next section it will be shown how synchronization between the active objects can be obtained.

3 Semaphores and Critical Regions

For synchronization a basic pattern `Semaphore` is available. The semaphore is a generalized semaphore that can take any value. A semaphore has the standard `P` and `V` operations and a `count` operation returning the number of processes

```
Account:
  (# mutex: @semaphore;
     Deposit:
      (# amount: @integer
      enter amount
      do mutex.P;
         amount+balance->balance ;
         mutex.V
      #)
     Withdraw:
      (# amount: @integer
      enter amount
      do mutex.P;
         amount+balance->balance;
         mutex.V
      #)
      balance: @integer;
  #);
```

Figure 3: The Account pattern defined using a semaphore

currently waiting for the semaphore.

```
Semaphore: (# P: ...; V: ...; count: ... #)
```

The execution of a semaphore operation is indivisible.
In figure 3, the Account pattern has been rewritten using semaphores. A semaphore `mutex` has been added to control the access to `Account` objects. The do-part of the operations `Deposit` and `Withdraw` starts by executing a `mutex.P` call and ends by executing `mutex.V`. This implies that actions like `amount1->JoesAcc.Deposit` and `amount2->JoesAcc.Withdraw` cannot be executed concurrently.

The use of a semaphore for controlling the access to an object can be encapsulated in an abstract superpattern defined as in figure 4. The pattern `entry` is an abstract superpattern that may be used as superpatterns of operations (patterns) in subpatterns of `CriticalRegion`. `Entry` will take care of manipulation of the `mutex` semaphore. The pattern `init` must be called to initialize the semaphore. `Init` is a virtual pattern that may be specialized in subpatterns of `CriticalRegion`. Virtuals patterns in BETA are e.g. described in [13, 16]. The `Account` pattern may now be defined as in figure 5.

The class pattern `Account` will inherit the semaphore `mutex` and the abstract patterns `entry` and `init` from `CriticalRegion`. The operations `Deposit`, and `Withdraw` are defined as subpatterns of `entry`. Execution of `Deposit` will thus start with execution of the do-part of `entry`. Execution of `INNER` in this do-part will then imply that the do-part of `Deposit` is executed where after control

```
CriticalRegion:
  (# mutex: @ Semaphore;
     entry: (#do mutex.P; INNER; mutex.V #);
     init:< (#do INNER; mutex.V  #)
  #);
```

Figure 4: An abstract pattern defining a critical region

```
Account: CriticalRegion
  (# Deposit: entry
      (# amount: @integer
      enter amount
      do amount+balance->balance ;
      #)
     Withdraw: entry
      (# amount: @integer
      enter amount
      do amount+balance->balance;
      #)
      balance: @integer;
  #);
```

Figure 5: `Account` as subpattern of `CriticalRegion`

returns to the do-part of `entry`. The `entry` superpattern of the operations will thus take care of manipulating the `mutex` semaphore before and after execution of the do-part of the operation. As it can be seen, the pattern `CriticalRegion` is an abstract superpattern that hides the details of the semaphore.

The two versions of `Account` in figure 3 and 5 illustrate the difference between **simulation** and **abstraction** of a given language mechanism, in this case a critical region. Figure 3 is a simulation of a critical region. The behavior of a critical region can be simulated by explicitly using a semaphore as shown in figure 3. Each time a critical region is used, it may be simulated using this technique. Most language constructs may be simulated in a given language. The problems in simulating a language construct is that the programmer has to repeat a given code template and it is easy to make errors when doing this. This can be avoided if an **abstraction** can be made. The `Account` in figure 5 is defined using the abstraction of a critical region defined in figure 4. It is clearly an advantage to be able to use an abstraction instead of simulating a language construct since this reduces the possibility of making errors. The power of a language with respect to abstraction depends on how good it is to define abstractions corresponding to language constructs in other languages. The critical region is a simple example, but the examples below are more complicated making the use of abstractions much more convenient.

The technique for defining an abstraction corresponding to a critical region can be used to define a large number of different abstractions for concurrent object-oriented programming. In the rest of this paper more examples of such abstractions are given.

4 Monitors

A monitor is a simple an powerful mechanism for handling a large class of concurrency problems. A monitor abstraction is similar to the simple `CriticalRegion` defined in the previous section. The main difference is that a process executing a monitor procedure may temporarily wait on a condition variable. The pattern in figure 6 (adapted from [24]) describes a traditional monitor. The `monitor` pattern may be used to define a buffer as shown in figure 7.

Note that a process executing a signal is delayed if there are processes waiting on the corresponding `Condition`. As mentioned in [9] `Signal` is often the last imperative executed in an operation. This implies that the signaling process may be unnecessarily delayed.

The purpose of this example is not to introduce monitor as a concurrency mechanism for BETA. The purpose is to show that such abstractions can be defined in the BETA language. BETA provides a framework that makes it possible to define a large number of different abstractions for concurrent object-oriented programming.

```
Monitor:
  (# mutex,  {control entry to the Monitor}
     urgent: {delays a signaling process}
       @Semaphore;
     return:
       {Executed by processes leaving the monitor.
       {Reactivates possible processes waiting for
       {entry: delayed signaling processes (urgent)
       {have first priority}
       (#
       do (if urgent.count>0 // true then urgent.V
           else mutex.V if)
       #);
     Condition:
       (# q: @ Semaphore;
          Wait: (#do Return; q.P #);
          Signal:
           (#do (if q.Count>0 // true then
                    q.V; urgent.P
           if)#);
       #);
     entry:(#do mutex.P; INNER; return #);
     init:< (#do INNER; mutex.V #)
  #)
```

Figure 6: An abstract pattern defining a monitor

```
buffer: @ active Monitor
  (# {representation of a buffer}
     full,empty: @ Condition
     Put: entry
       (# e: @integer
       enter e
       do (if {full} then full.wait if);
          {put e into buffer}
          empty.signal;
       #);
     get: entry
       (# e: @integer
       do (if {empty} then empty.wait if);
          {take e out of buffer}
          full.signal;
       exit e
       #)
  #)
```

Figure 7: A `buffer` pattern

5 Ada-like Task Abstraction

The example in figure 8 shows how to simulate an Ada-like rendezvous mechanism. The object `R` (corresponding to a task in Ada) has a number of operations `f1`, `f2`, and `f3` (corresponding to entry-procedures in Ada). Each operation has an associated set of semaphores for controlling the operation. The do-part of `R` controls when these operations can be accepted. The objects `S1` and `S2` attempt to execute operations on `R`. When `S1` executes `R.f1`, then `m1.P` in the do-part of `f1` is executed. This implies that `S1` is delayed until `R` has executed `m1.V` following the comment {`accept f1:`}. `R` will continue to execute `mutex1.P` which means that it will be delayed until `S1` has executed `mutex1.V` at the end of the do-part of `f1`. Altogether, the protocol specifies that `R` accepts `R.f1` by `S1`, `R.f2` by `S2` and `R.f3` by `S1`.

This example shows how an Ada-like task mechanism with rendezvous can be **simulated**. The pattern `task` in figure 9 defines an abstraction for this. The concept of a `port` for encapsulating the semaphores is introduced. `port` is an example of a **nested pattern**. In addition, the patterns `entry` and `accept` are nested within `port`. Nested patterns are used quite heavily in BETA, see e.g. [18, 17].

The active object `R` may be defined using the `task` abstraction as shown in figure 10. The semaphores `m1`, `mutex1` ,`m2`, `mutex2`, `m3`, and `mutex3` are represented

```
R: @ active
  (# mutex1,m1,mutex2,m2,mutex3,m3 : @Semaphore;
     f1: (#do m1.P; .... mutex1.V #);
     f2: (#do m2.P; .... mutex2.V #);
     f3: (#do m3.P; .... mutex3.V #);
  do ...; {accept f1:} m1.V; mutex1.P;
     ...; {accept f2:} m2.V; mutex2.P;
     ...; {accept f3:} m3.V; mutex3.P;
     ...
  #);
S1: @ active (#do ...; R.f1; ...; R.f3; ... #);
S2: @ active (#do ...; R.f2; ... #)
```

Figure 8: Simulation of an Ada-like task

```
task:
  (# port:
       (# mutex, m: @semaphore;
          entry: (#do m.P; INNER; mutex.V #);
          accept: (#do m.V; mutex.P #)
       #)
   do INNER
   #)
```

Figure 9: Abstract pattern for Ada-like task

```
R: @ active task
  (# P1,P2,P3: @ port;
     f1: P1.entry(# ... #);
     f2: P2.entry(# ... #);
     f3: P3.entry(# ... #);
  do ...; {accept f1:} P1.accept;
     ...; {accept f2:} P2.accept;
     ...; {accept f3:} P3.accept;
     ...
  #)
```

Figure 10: Use of the `task` abstraction

by `P1.m`, `P1.mutex`, `P2.m`, `P2.mutex`, `P3.m`, and `P3.mutex` respectively.

Notice that `f1` is a subpattern of `P1.entry` which is the `port` pattern defined within the `P1` object. This means that when `f1` is executed, execution starts with executing the do-part of the `entry` pattern defined within `P1`. The semaphores `mutex` and `m` referred in this do-part are thus the ones defined within `P1`. The corresponding operation `P1.accept` also refers to the semaphores defined within `P1`.

The operations `f2` and `f2` are similarly subpatterns of the `entry` pattern defined within `P2` and `P3` respectively.

Compared to Ada, a port instance must be declared in addition to the operation. This is of course a bit more awkward. There are, however, also some advantages since it is possible to associate more than one operation with a given `port`. If `f2` is declared as `f2: P1.entry(# ... #)` then `P1.accept` will imply that either of `f1` or `f2` may be accepted.

It is possible to extend the definition of `task` such that a possible restriction of the calling object may be specified as in CSP. The definition of `accept` may be extended to be called in the following way:

```
S1[]-> P1.accept
```

which express that only `S1` is allowed to execute an operation associated with `P1`. The class library for BETA includes these extensions.

6 Inheritance of Synchronization Constraints

One of the open issues in concurrent object-oriented programming is how to inherit synchronization constraints from a superclass. The language mechanisms presented here supports some form of inheritance of synchronization constraints.

```
DrinkMachine: task
  (# activate: @ port;
     makeCoffee: activate.entry
       (# do ... coffeeReady[]drinkReady[] #);
     makeSoup: activate.entry
       (# do ... soupReady[]drinkReady[] #);
     coffeeReady, soupReady: @ port;
     getCoffee: coffeeReady.entry
        (# do ... exit someCoffee [] #);
     getSoup: soupReady.entry
        (#do ... exit someSoup [] #);
     drinkReady: ^ port
  do cycle(#do activate.accept; drinkReady.accept #)
  #);

ExtendedDrinkMachine: DrinkMachine
  (# makeTea: activate.entry
       (#do ... teaReady[]->drinkReady[] #);
     teaReady: @ port;
     getTea: teaReady.entry(# ... exit someTea[] #)
  #)
```

Figure 11: Inheritance of synchronization constraints

In figure 11 is shown a pattern `DrinkMachine` which models a drink machine that provides coffee and soup. A customer operates the machine by pushing either `makeCoffe` or `makeSoup`. If `makeCoffe` has been pushed, then the customer may obtain the coffee by means of `getCoffee`. Similarly if `makeSoup` has been pushed then the soup may be obtained by means of `getSoup`.

The `DrinkMachine` may be used in the following way:

```
aDrinkMachine: @ active DrinkMachine
...
aDrinkMachine.makeCoffee;
aDrinkMachine.getCoffee;
aDrinkMachine.makeSoup;
aDrinkMachine.getSoup;
```

An instance of `DrinkMachine` has three ports, `activate`, `coffeeReady`, and `soupReady`. The operations `makeCoffee` and `makeSoup` are associated with the `activate` port. The operations `getCoffee` and `getSoup` are associated with the ports `coffeeReady` and `soupReady` respectively.

Initially a `DrinkMachine` is ready to accept either `makeCoffee` or `makeSoup`. This is signaled by executing `activate.accept`.

If e.g. `makeCoffee` is executed, then when "the coffee has been made", the `DrinkMachine` is willing to accept the operation `getCoffee`. This is signaled by executing an accept on the port `coffeeReady`. Technically this is implemented by assigning a reference to `coffeeReady` to the port reference `drinkReady`. The do-part of `DrinkMachine`, then makes an accept on `drinkReady`.

When the operation `getCoffe` has been executed, the `DrinkMachine` is again ready to accept a new operation associated with the activate port.

It is possible to specialize the `DrinkMachine` into a machine that accepts further operations. The `ExtendedDrinkMachine` also shown in figure 11 inherits the operations and protocol from `DrinkMachine` and adds new operations to the protocol. The new operations are `makeTea` and `getTea`. `MakeTea` is associated with the `activate` port and the `teaReady` port is assigned to `the drinkReady` port if `MakeTea` gets executed.

The drink machine example shows one example of how it is possible to inherit and extend the protocol from a superclass. ...

7 Readers and Writers Problem

In the following sections examples of other abstractions for concurrent object-oriented programming will be described. It should be obvious from the above examples that its is possible to describe a whole range of such abstractions. The first example is the classical problem of readers and writers. A resource is shared by a number of processes that are divided into two classes:

```
Resource:
  (# W,{ensure exclusive access for writers}
     T:{ensure exclusive access to rCount}
       @Semaphore;
     rCount: @ Integer; {no. of concurrent readers}
     ReaderEntry:
       (#
       do T.P; rCount+1->rCount;
          (if rCount//1 then W.P if);
          T.V;
          INNER;
          T.P; rCount-1->rCount;
          (if rCount//0 then W.V if);
          T.V
       #);
     WriterEntry: (# do W.P; INNER; W.V #)
   #);
R: @ active Resource
  (# ...
     R1: ReaderEntry(# ... #);
     R2: ReaderEntry(# ... #);
     R3: ReaderEntry(# ... #);
     W1: WriterEntry(# ... #);
     W2: WriterEntry(# ... #)
  #)
```

Figure 12: Readers and writers abstraction

1. **Writers** — which must have exclusive access to the resource
2. **Readers** — which can share the resource

The resource can be accessed by several readers whereas a writer must have exclusive access. The abstract pattern `Resource` in figure 12 describes one possible solution. Readers will access the resource though `ReaderEntry`-operations and writers through `WriterEntry`-operations. In this solution a reader will never wait unless the resource is occupied by a writer. A writer will have to wait until no more readers are accessing the resource.

8 Concurrent Put and Get Operations of Buffer

In the buffer example in figure 7, it would be possible to execute the operations `Put` and `Get` concurrently, since they operate on disjoint parts of the shared

```
Buffer: @ active
  (# {representation of a buffer}
     putMonitor: @ Monitor(# full: @Condition #);
     getMonitor: @ Monitor(# empty: @Condition #);
     Put: putMonitor.entry
       (# e: @integer
       enter e
       do (if {full} then putMonitor.full.wait if);
          {put e into buffer}
          getMonitor.empty.signal;
       #);
     get: getMonitor.entry
       (# e: @integer
       do (if {empty} then getMonitor.empty.wait if);
          {take e out of buffer}
          putMonitor.full.signal;
       exit e
       #)
  #);
```

Figure 13: Buffer with concurrent `put` and `get` operations

data. This is expressed in the example in figure 13, where the buffer has two local `Monitor` objects for excluding concurrent `Put` operations and concurrent `Get` operations.

9 Conclusion

In this paper it has been demonstrated that by means of semaphores and powerful abstraction mechanisms, it is possible to build powerful abstractions for concurrent object-oriented programming. Despite of intensive research in the area of concurrent object-oriented programming, there are no proposals for this that have received widespread acceptance. This indicates that there is still a need for further experiments with concurrency mechanisms in object-oriented languages. The mechanisms and techniques presented in this paper make it possible to experiments with different concurrency abstractions.

The mechanisms have been implemented for the BETA language. The semaphore has been selected as the low-level mechanisms since it is very general and is well understood through many years of practice. As mentioned several times, it is difficult to program using semaphores directly. However, most people using BETA will not be doing this. The basic BETA library provides abstractions for the most common type of concurrency problems.

The **inner** mechanism of Simula and BETA is central for the technique used defining the concurrency abstractions in this paper. In addition the possibility to define subprocedures (called "prefixed procedures" in [24]). Neither inner, subprocedures or nested patterns (classes)[1] are available in main stream object-oriented languages. We think that the examples in this paper has demonstrated the usefulness of these mechanisms.

Finally concurrency must of course be supported by the object-oriented language in order to support concurrent object.oriented programming. BETA is based on the concept of active objects in the form of coroutines as found in Simula. Within the Simula community, this model of quasi-parallel execution has demonstrated its usefulness through more than 25 years. BETA extends this model by allowing active objects to be executed concurrently.

References

[1] G. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Mass., 1986.

[2] P. America, *Inheritance and Subtyping in a Parallel Object-Oriented Language*, Proc. ECOOP'87, European Conference on Object-Oriented Programming (ed. P Cointe), Paris, Springer Lecture Notes in Computer Science Vol. ??, 1987.

[3] P. America, F. van der Linden: *A Parallel Object-Oriented Language with Inheritance and Subtyping*, OOPSLA/ECOOP'90, Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming, SIGPLAN Notices, vol. 25, no. 10, Oct. 1990.

[4] G.M. Birtwistle, O.J. Dahl, B. Myhrhaug, K. Nygaard: *SIMULA BEGIN*, Lund, Sweden: Studentlitteratur, 1973.

[5] G. Bracha, W. Cook: *Mixin-based Inheritance*, Proc. OOPSLA/ECOOP'90, Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming, SIGPLAN Notices, vol. 25, no. 10, Oct. 1990.

[6] O.-J. Dahl, C.A.R Hoare *Hierarchical Program Structures* In: Structured Programming (Dahl, Dijkstra, Hoare), Academic Press, 1972

[7] J.-P. Briot, A. Yonezawa: *Inheritance and Synchronization in Concurrent OOP*, Proc. ECOOP'87, European Conference on Object-Oriented Programming (ed. P Cointe), Paris, Springer Lecture Notes in Computer Science Vol. ??, 1987.

[1] C++ has a form of nested classes, but a nested class cannot refer to variables in the enclosing class. Only local variables and global variables can be referred within a class.

[8] S. Frølund: *Inheritance of Synchronization Constraints in Concurrent Object-oriented Programming Languages*, Proc. ECOOP'92, European Conference on Object-Oriented Programming (ed. O.L Madsen), Utrecht, Springer Lecture Notes in Computer Science Vol. 615, 1992.

[9] C.A.R. Hoare: *Monitors: An Operating System Structuring Concept*, Comm. ACM, Oct. 74, Vol. 17, No. 10, pp. 549-557.

[10] D.G. Kafura, K.H. Lee: *Inheritance in Actor Based Concurrent Object-Oriented Languages*, Proc. ECOOP'89, European Conference on Object-Oriented Programming (ed. S. Cook), British Computer Society Workshop Series, Cambridge University Press, 1989.

[11] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard*Abstraction Mechanisms in the BETA Programming Language* Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, January 24-26 1983, Austin, Texas

[12] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard *Classification of Actions or Inheritance also for Methods* Proc. ECOOP'87, European Conference on Object-Oriented Programming (ed. P.), Paris, Springer Lecture Notes in Computer Science Vol. ,? 1987.

[13] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard*The BETA Programming Language* In: B.D. Shriver, P.Wegner (ed.), *Research Directions in Object Oriented Programming*, MIT Press, 1987

[14] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard*Coroutine Sequencing in BETA* Hawaii International Conference on System Sciences - 21, January 5-8, 1988

[15] O.L. Madsen, B. Møller-Pedersen: *What Object-Oriented Programming may be, and what it does not have to be*, Proc. ECOOP'88, European Conference on Object Oriented Programming (eds. S. Gjessing, K Nygaard), Oslo, August 1988

[16] O.L. Madsen, B. Møller-Pedersen: *Virtual Classes — A Powerful Mechanism in Object-Oriented Programming*, OOPSLA'89, Object-Oriented Programming, Systems, Languages, and Systems, ...

[17] O.L Madsen, B. Møller-Pedersen: *Part Objects and their Location* Technology of Object-Oriented Languages and Systems, TOOLS 7, Dortmund, Prentice Hall 1992

[18] O.L. Madsen *Block Structure and Object Oriented Languages* In: B.D. Shriver, P. Wegner (ed.): *Research Directions in Object Oriented Programming*, MIT Press, 1987

[19] J.E.B. Moss, W.H. Kohler: *Concurrency Features for the Trellis/Owl Language*, Proc. ECOOP'87, European Conference on Object-Oriented Programming (ed. P. Cointe), Paris, Springer Lecture Notes in Computer Science Vol. ??, 1987.

[20] C. Neusius: *Synchronizing Actions*, In Proceedings ECOOP'91, European Conference on Object-Oriented Programming (ed. P. America), Geneva, Springer Lecture Notes in Computer Science Vol. 512, 1991.

[21] O. Nierstrasz: *Active Objects in Hybrid*, Proc. OOPSLA'87, SIGPLAN Notices, vol. 22, no. 12, Dec. 1987.

[22] M. Papathomas: *Language Design Rational and Semantic Framework for Concurrent Object-Oriented Programming*, University of Geneve, Ph.D. Thesis, Jan. 1992.

[23] J. Van Den Bos, C. Laffra: *PROCOL: A Parallel Object Language with Protocols*, Proc. OOPSLA'89, SIGPLAN Notices, vol. 24, no. 10, Oct. 1989.

[24] J. Vaucher: *Prefixed Procedures: A Structuring Concept for Operations*, INFOR, Vol. 13, No. 3, October 1975.

[25] Y. Yokote, M. Tokoro: *Concurrent Programming in ConcurrentSmalltalk*, In *Object-Oriented Concurrent Programming* (ed. M. Tokoro), The MIT Press, Cambridge, Mass., 1987.

[26] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda: *Modelling and Programming in an Object-Oriented Concurrent Language ABC/1*, In *Object-Oriented Concurrent Programming* (ed. M. Tokoro), The MIT Press, Cambridge, Mass., 1987.