Problem of the Sleeping Barbers

Analysis of the concurrent solution – 2

António Rui Borges

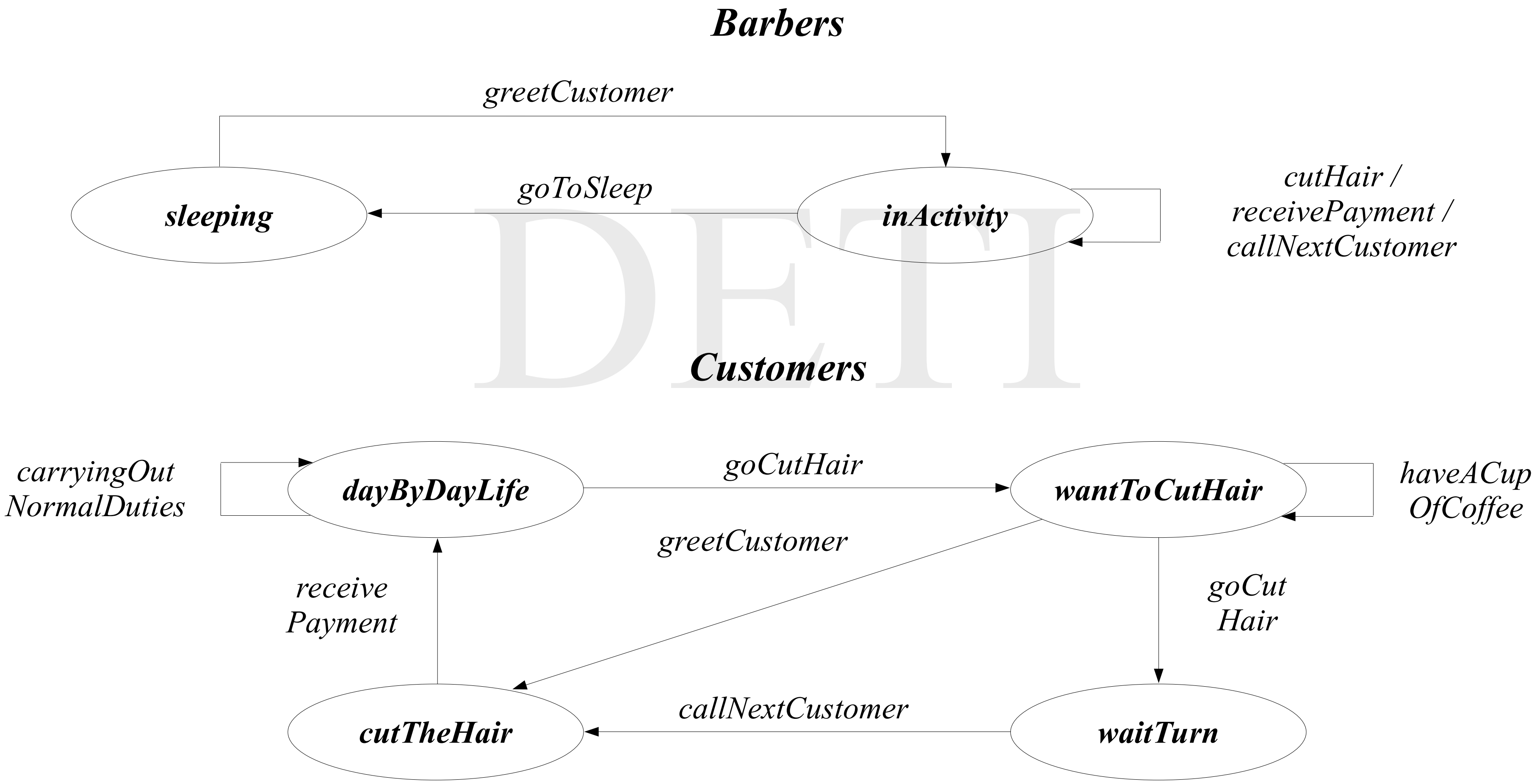
Departamento de Electrónica, Telecomunicações e Informática

Summary

Dynamic solution

- *State diagrams*
- *Life cycles*
- *General characterization of the* `BarberShop` *data type*
- *Implementation* (*monitors / semaphores*)
- *Self-referencing implementation*

Dynamic solution: state diagrams



Dynamic solution: life cycle of the involved entities

Barbers (a maximum of *M* barbers may coexist)

parameters *b* and *c* are passed upon instantiation

```
greetCustomer (c);              // upon waking up, the barber greets the customer
do
{ cutHair ();                                   // the barber cuts the customer hair
  receivePayment (c);                             // the barber finishes his service
                                                      // and receives payment for it
} while ((c = callNextCustomer ()) != -1);      // the barber checks the waiting
                         // chairs, if there are a customer waiting, he calls him
goToSleep ();                                           // the barber goes to sleep
```

Customers (c = 0,1,...,N-1)

```
forever
{ carryingOutNormalDuties ();           // the customer carries out normal duties
  while (!goCutHair ())              // the customer checks if he can cut his hair
    haveACupOfCoffee ();             // if the barber shop is full, he tries later
}
```

Dynamic solution: barber shop

Solution decomposition supposes the existence of a shared data type, called `BarberShop`, with the following organization

Internal data structure

waiting turn queue (FIFO – stores customer id and has a size K)
nOcCutChair – number of occupied cutting chairs
stateCutChair (array of boolean – signals whether the matching chair is occupied)

Synchronization

access with mutual exclusion to the internal data structure
customers – one blocking point per customer where each customer both waits his turn to cut the hair and sits on the cutting chair while having his hair cut

Operations called on it

goToSleep – called by the barber
goCutHair – called by the customer
greetCustomer – called the barber
receivePayment – called the barber
callNextCustomer – called the barber

Implementation of the dynamic solution with monitors

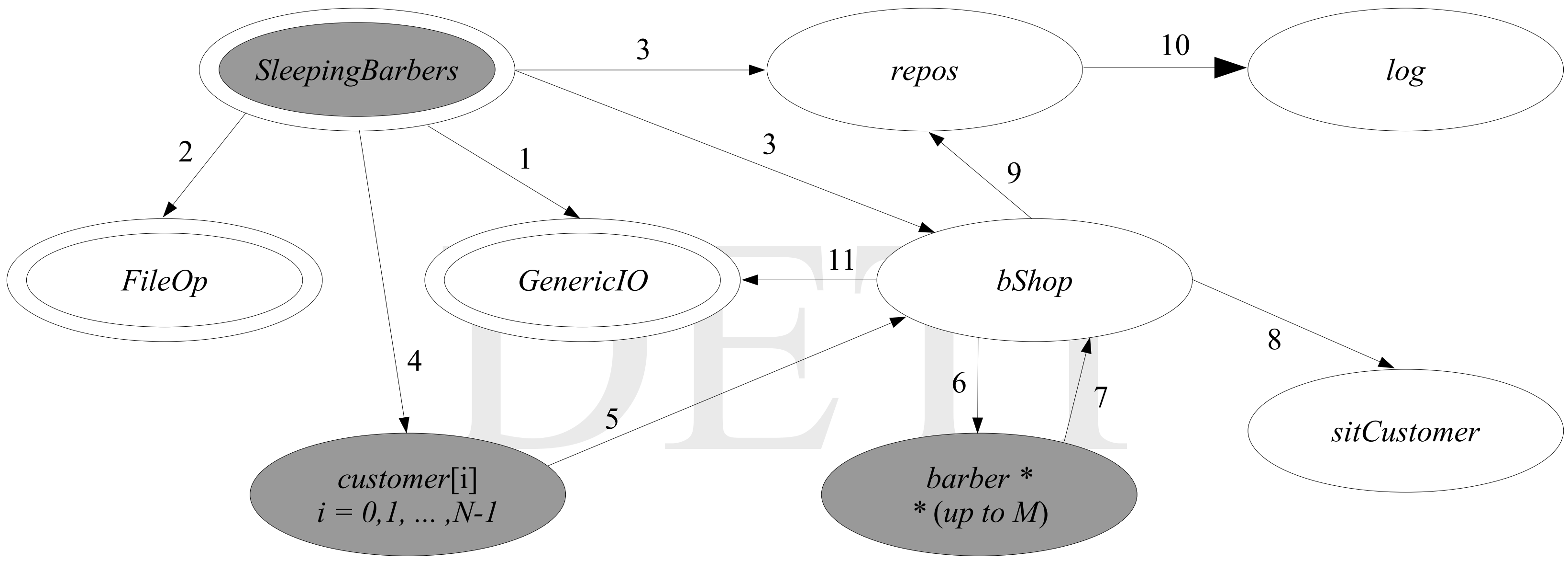
The `BarberShop` data type becomes a monitor.

Since *condition variables* are restricted to a single one, associated to the monitor object, by Java concurrency model, it is necessary to turn the blocking conditions explicit by repetitive testing of normal variables.

Synchronization

- *access with mutual exclusion to the internal data structure* – monitor *locking* (all public methods are synchronized)
- *customers* – one blocking point per customer where each customer both waits his turn to cut the hair and sits on the cutting chair while having his hair cut (use is made of his state value).

Interaction diagram: dynamic solution with monitors



1 – readlnInt, readlnChar, readlnString, writeString, writelnString
2 – exists
3 – instantiate
4 – instantiate, start, join
5 – goCutHair
6 – instantiate, start
7 – goToSleep, greetCustomer, receivePayment, callNextCustomer
8 – instantiate, empty, full, write, read
9 – setBarberState, setCustomerState, setBarberCustomerState
10 – instantiate, openForWriting, openForAppending, close, writelnString
11 – writelnString

Implementation of the dynamic solution with semaphores

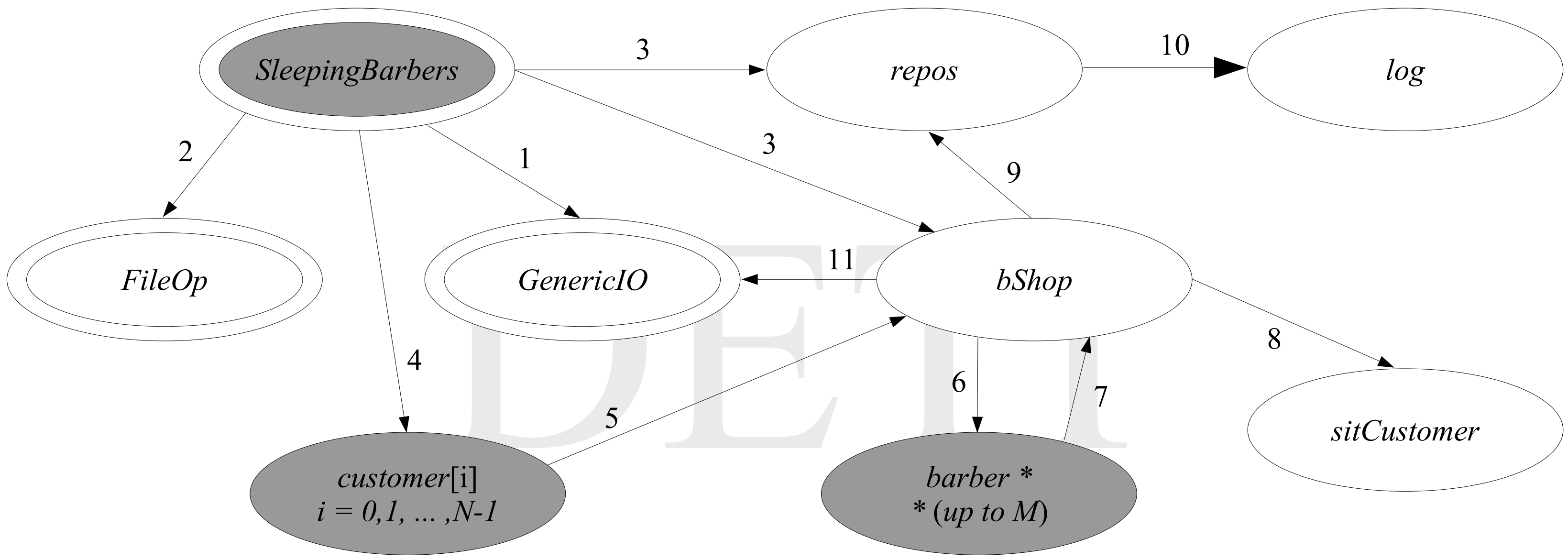
The `BarberShop` data type is a conventional data type.

Synchronization

access with mutual exclusion to the internal data structure – binary semaphore, initialized to *green*

customers – one blocking point per customer where each customer both waits his turn to cut the hair and sits on the cutting chair while having his hair cut (binary semaphore array, each element initialized to *red*).

Interaction diagram: dynamic solution with semaphores



1 – readlnInt, readlnChar, readlnString, writeString, writelnString
2 – exists
3 – instantiate
4 – instantiate, start, join
5 – goCutHair
6 – instantiate, start
7 – goToSleep, greetCustomer, receivePayment, callNextCustomer
8 – instantiate, empty, full, write, read
9 – setBarberState, setCustomerState, setBarberCustomerState
10 – instantiate, openForWriting, openForAppending, close, writelnString
11 – writelnString

Dynamic solution: end of operations

Although the Problem of the Sleeping Barbers assumes infinite life cycles for both the barbers and the customers, any simulation must make them finite. It is obviously trivial to make the customers life cycle finite and, in this case, the same is true for the barbers life, which is by definition always finite.

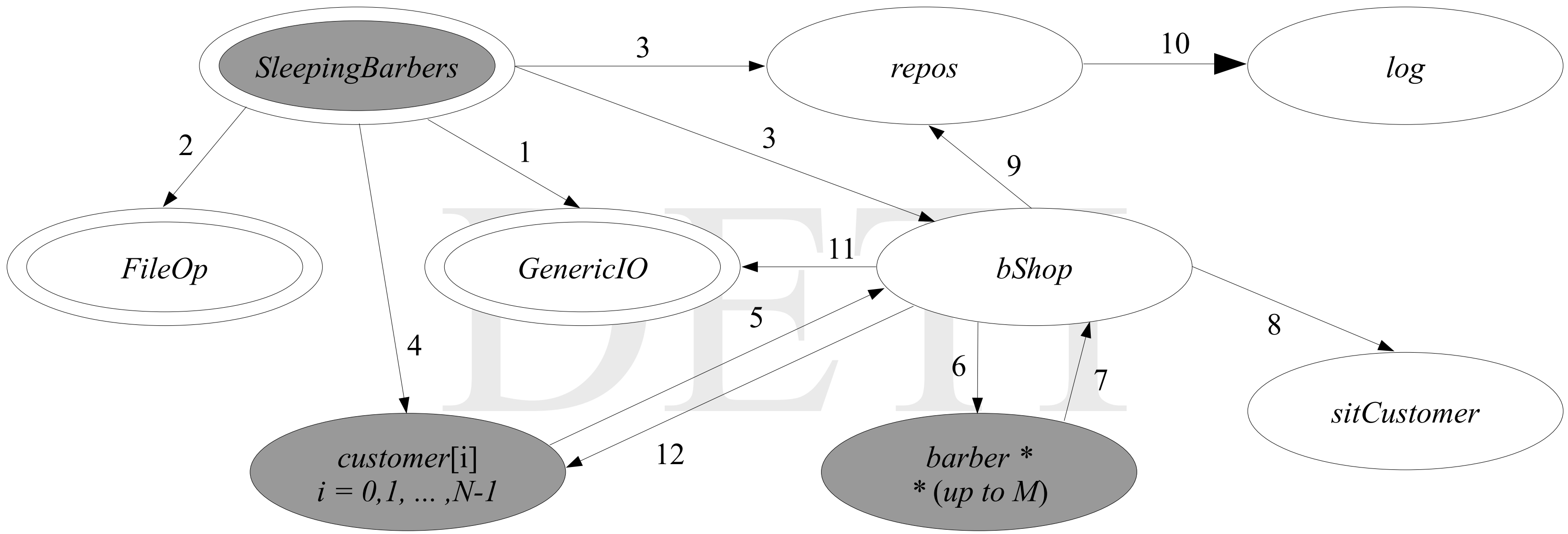
***Implementation of an alternative dynamic solution with monitors* (*valid in Java*)**

Instead of blocking the customer threads in the condition variable associated with the monitor object resulting from the instantiation of the reference data type `BarberShop`, it is possible to block each customer thread in the condition variable associated with its own object.

This approach means that resource is made to a self-reference mechanism, which is not in general available in concurrent programming languages, but that is valid in Java.

Thus, the operations *goCutHair* and *receivePayment* can not be wholy synchronized because they require access in succession to two different monitors!

Interaction diagram: dynamic solution with monitors (*valid in Java*)



1 – readlnInt, readlnChar, readlnString, writeString, writelnString
2 – exists
3 – instantiate
4 – instantiate, start, join
5 – goCutHair
6 – instantiate, start
7 – goToSleep, greetCustomer, receivePayment, callNextCustomer
8 – instantiate, empty, full, write, read
9 – setBarberState, setCustomerState, setBarberCustomerState
10 – instantiate, openForWriting, openForAppending, close, writelnString
11 – writelnString
12 – wait, notifyAll

Tricky question - 1

```
public boolean goCutHair ()
{
  int barberId, customerId;

  synchronized (this)                                   // monitor bshop
  { customerId = ((Customer) Thread.currentThread ()).getCustomerId ();
    cust[customerId] = (Customer) Thread.currentThread ();
    cust[customerId].setCustomerState (CustomerStates.WANTTOCUTHAIR);
    repos.setCustomerState (customerId, cust[customerId].getCustomerState ());
    if (sitCustomer.full ()) return (false);
    if (nOcCutChair < SimulPar.M)
       { barberId = allocCuttingChair ();
         new Barber ("Barb_" + (barberId+1), barberId, customerId, this).start ();
       }
       else { cust[customerId].setCustomerState (CustomerStates.WAITTURN);
              repos.setCustomerState (customerId, cust[customerId].getCustomerState ());
              try
              { sitCustomer.write (customerId);
              }
              catch (MemException e)
              { ... }
            }
  }
  synchronized (cust[customerId])                        // monitor customer[customerId]
  { try
    { cust[customerId].wait ();
    }
    catch (InterruptedException e) {}
  }
  return (true);
}
```

thread customer blocks in the implicit condition variable of its own monitor

Tricky question - 2

```
public void receivePayment (int customerId)
{
  int barberId;

  synchronized (this)                                    // monitor bshop
  { cust[customerId].setCustomerState (CustomerStates.DAYBYDAYLIFE);
    repos.setCustomerState (customerId, cust[customerId].getCustomerState ());
  }
  synchronized (cust[customerId])                        // monitor customer[customerId]
  { cust[customerId].notifyAll ();
  }
}
```

thread barber wakes up the thread customer blocked in the implicit condition variable of its own monitor

Notice that, although each customer thread is apparently the only thread blocked in the condition variable of its own monitor, the barber thread when it wakes it up, performs a *notifyAll* operation, instead of a simple *notify* operation.

In fact, if the *notifyAll* operation were replaced by the simple *notify* operation, the program would enter a deadlock state (*try it*)!

Why it is so?