5 Objects with Coordinated Access

Introduction

The problem of sharing data between several actors was briefly discussed in Chapter 0. To avoid inconsistencies during accesses to shared data, the three languages provide several high level and low level constructs that allow to coordinate the access to shared data. Those constructs can be used for indirect actor communication. The intention of this chapter is to examine and compare these constructs.

5.1 Objects with Coordinated Access in Ada

5.1.1 Protected Objects

Protected objects provide coordinated access to shared data through calls to their visible protected operations. Such protected operations can be protected subprograms or protected entries.

5.1.1.1 Definition

Like a task, a protected unit may be declared as a type or as a single instance. It has a specification and a body.

The specification of a protected unit has the following syntax:

```
protected_type_declaration ::=
   PROTECTED TYPE defining_identifier [known_discriminant_part]
        IS protected_definition;

single_protected_declaration ::=
   PROTECTED defining_identifier IS protected_definition;

protected_definition ::=
     { protected_operation_declaration }
[ PRIVATE
     { protected_element_declaration } ]
END [protected_identifier]

protected_operation_declaration ::=
   subprogram_declaration | entry_declaration

protected_element_declaration ::=
   protected_operation_declaration | component_declaration
```

The body of a protected unit is declared using the following syntax:

```
protected_body ::=
    PROTECTED BODY defining_identifier IS
      { protected_operation_item }
    END [protected_identifier];
```

```
protected_operation_identifier ::=
    subprogram_declaration | subprogram_body | entry_body

entry_body ::=
   ENTRY  defining_identifier  entry_body_formal_part  entry_barrier  IS
      declarative_part
   BEGIN
      handled_sequence_of_statements
   END [entry_identifier];

entry_body_formal_part ::=
    [ (entry_index_specification) ] parameter_profile

entry_barrier ::= WHEN condition

entry_index_specification ::=
   FOR defining_identifier IN discrete_subtype_definition
```

If a protected_identifier appears at the end of a protected_definition or protected_body, it shall repeat the defining_identifier. A protected declaration requires a completion, which shall be a protected_body, and every protected_body shall be the completion of some protected declaration.

A protected unit has a visible part and a private part. The visible part consists of the list of protected_operation_declarations in a protected_definition together with the known_discriminant_part, if any. The optional list of protected_element_declarations after the reserved word PRIVATE is called the private part of the protected unit.

A protected type is a limited type, it has neither an assignment operation nor predefined equality operators. The declarations in the private part are only visible within the private part and the body of the protected unit.

5.1.1.2 Execution Resource

Each protected object is associated with an execution resource that is essentially a lock variable of the object. Furthermore, the object includes a representation of the state of the execution resource.

To read or update any components of a protected object, the execution resource associated with the protected object has to be acquired. It can be acquired either for concurrent read-only access, or for exclusive read-write access.

5.1.1.3 Protected Functions, Protected Procedures, and Protected Entries

Protected functions provide concurrent read-only access to the data of the protected object. Within the body of a protected function, the current instance of the enclosing protected unit is defined to be a constant, i.e., a protected function may only read but not update the subcomponents of the protected unit.

Protected procedures provide exclusive read-write access to the data of the protected object. Within the body of a protected procedure, the current instance is defined to be a variable. That means, all its subcomponents that are variables may be read and written (constants may only be read).

For better understanding, we give an example of a protected type. Assume that we want to implement an account. Such an account contains a data component, the balance, and three operations. We need one operation for depositing money, one for withdrawing money and one for providing the current balance of the account. Since the first two operations write the balance, they have to be implemented as protected procedures or protected entries. The operation that provides the current balance only reads the balance's value and can therefore be implemented by using a protected function. The data component—the balance—that is read and written has to be put into the private part. According to these considerations, the code of the account is given by Example 5.1.

```
protected type Account is
   procedure Deposit (Amount : in Positive);
   procedure Withdraw (Amount : in Positive);
   function Provide_Balance return Integer;
private
   Balance : Integer := 1000;
end Account;

protected body Account is
   procedure Deposit (Amount : in Positive) is
   begin
      Balance := Balance + Amount;
   end Deposit;
   procedure Withdraw (Amount : in Positive) is
   begin
      Balance := Balance - Amount;
   end Withdraw;
   function Provide_Balance return Integer is
   begin
      return Balance;
   end Provide_Balance;
end Account;
```

Example 5.1: A simple account

Protected entries provide exclusive read-write access to the data of the protected object. Within an entry_body, the current instance is defined to be a variable. That means, all its subcomponents that are variables may be read and written (constants may only be read). In contrast to protected procedures, protected entries have two special properties:

1. entry_barrier
2. entry queue

A protected entry is always guarded by a boolean expression, the entry_barrier. The condition in the entry_barrier must not reference the formal parameters of the entry. This restriction ensures that all calls to the same entry see the same barrier value and the barrier has to be evaluated only once. If there was no such restriction, the entry_barrier of a given entry had to be reevaluated for each task inside the queue of that entry (since each task can have different parameter values). An entry_body can only be executed if the condition of its entry_barrier evaluates to true. In that case, the entry is said to be open; otherwise, it is closed. Hence, protected entries are suitable for implementing protected operations that shall only be executed if some condition holds.

Each entry of a given protected object has its own entry queue where calls to it are queued. If a call to an entry is made and the entry is closed, the entry call is added to the associated entry queue. For an entry call that is added to a queue, the calling task is blocked until the call is cancelled or the call is selected, i.e., the corresponding `entry_body` is executed. When a queued call is selected, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called servicing the entry queue.

Queued tasks take always precedence over tasks that are outside the protected object and want to aquire the execution resource. Hence, as stated in (Barnes, 1996, [p. 409]), there are two distinct levels of protection:

1. the mutual exclusion guaranteed by the execution resource

2. the barrier protection mechanism that is superimposed on the mutual exclusion mechanism

This two level model is called the eggshell model. As depicted in Figure 5.1, the protected object with its entry queues is surrounded by a shell (that illustrates the first level of protection). To penetrate the shell and enter the object, a task must aquire the execution resource. Tasks can wait on two levels: outside the object (contending for the execution resource) and inside the object (in an entry queue). The tasks waiting inside take always precedence over the tasks waiting outside.

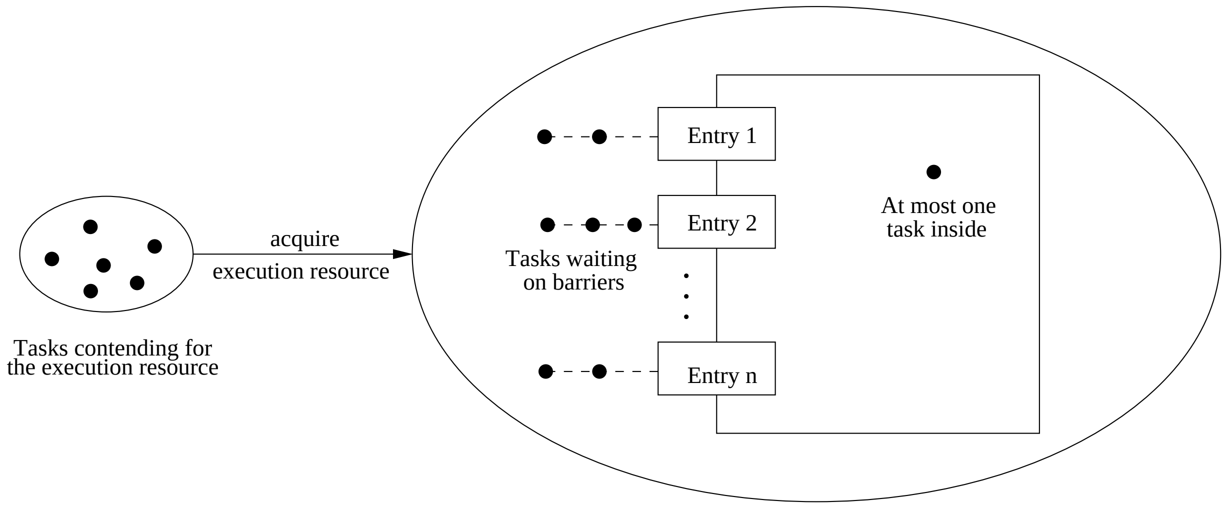

Figure 5.1: The eggshell model

The eggshell model ensures the so called "immediate resumption". We define immediate resumption as follows: Assume that there is an actor A calling a conditional operation O of a monitor M and A is delayed since the condition of O is not satisfied. If—after the condition of O became true—A is activated and may execute O in preference to any new actors attempting to enter M, then this is called "immediate resumption".

We conclude this section with a simple example illustrating the use of a protected entry. This example is taken from (ARM, 1995, [9.4(26)]).

```
protected type Resource is
   entry Seize;
   procedure Release;
private
   Busy : Boolean := False;
end Resource;
```

```
protected body Resource is
   entry Seize when not Busy is
   begin
      Busy := True;
   end Seize;
   procedure Release is
   begin
      Busy := False;
   end Release;
end Resource;
```

Example 5.2: A simple resource

5.1.1.4 Protected Actions

If a protected function, protected procedure, or protected entry is called and the call is an external call, a new protected action is started on the object. (An external call uses the full name and therefore has the following form: `protected_object.protected_subprogram`. This holds even if the call is made from within `protected_object`.)

When a protected action is started on a protected object, it acquires the execution resource associated with the protected object. If the protected action is for a call on a protected function the execution resource is acquired for concurrent read-only access. Otherwise it is acquired for exclusive read-write access. When a protected action is completed, it releases the associated execution resource.

A new protected action is not started on a protected object while another protected action on the same protected object is underway unless both actions are due to a call to a protected function.

If two or more tasks try to start a protected action on a protected object, and at most one is calling a protected function, then only one task can proceed. The other task that cannot proceed is not considered blocked and it might be consuming processing resources while it awaits its turn. The language does not define the order in which tasks competing for a protected action are executed.

Now we have to examine protected actions in more detail. There are the following kinds of protected actions:

1. protected action that is caused by a call to a protected function

2. protected action that is caused by a call to a protected procedure

3. protected action that is caused by a call to a protected entry

Protected Action caused by to Protected Function

If the protected action is caused by a call on a protected function and no other task has already acquired the execution resource for exclusive read-write access, the protected action is started. Then the body of the protected function is executed and after that the protected action completes.

In Figure 5.2 which summarizes these steps *cro_lock* corresponds to acquiring the execution resource for concurrent read-only access, *cro_release* corresponds to releasing the execution resource for concurrent read-only access.

```
cro_lock
execute function body
cro_release
```

Figure 5.2: Executing a protected action that is caused by a protected function

Protected Action caused by Protected Procedure

Let T be the task calling a protected procedure. If no other task has already acquired the execution resource for concurrent read-only access or the execution resource for exclusive read-write access, the protected action is started. Then the body of the protected procedure is executed by T. The rest of the protected action can be executed by any task of the system. All entries that have queued calls and whose barriers reference variables or attributes which might have changed since they were last evaluated are reevaluated. If any entries with queued calls are open, one open entry is chosen and the associated entry_body is executed. This will be repeated as long as there are open entries with queued calls. If there are no open entries, the protected action completes.

In Figure 5.3 which summarizes these steps *erw_lock* corresponds to acquiring the execution resource for exclusive read-write access, *erw_release* corresponds to releasing the execution resource for exclusive read-write access. The Roman font indicates that an action is executed by T and Sans Serife indicates that an action can be executed by any task of the system.

```
erw_lock
execute procedure body
loop
   compute all barriers that might have changed
   if there is no open entry with queued calls
      exit
   else
      choose one open entry and execute the associated entry_body
   end if
end loop
erw_release
```

Figure 5.3: Executing a protected action that is caused by a protected procedure

Protected Action caused by Protected Entry

Let T be the task calling a protected entry. If no other task has already acquired the execution resource for concurrent read-only access or the execution resource for exclusive read-write access, the protected action is started. Then the associated entry_barrier is evaluated by T if this is necessary, i.e., if it may have changed since the last evaluation or if it has not yet been evaluated.

If the entry_barrier evaluates to true, the entry_body is executed by T. Then, as in the case of a protected procedure, all entries that have queued calls and whose barriers reference variables or attributes which might have changed since they were last evaluated are reevaluated. If any entries with queued calls are open, one open entry is chosen and the associated entry_body is executed. This is done by using any task of the system and will be repeated until there are no open entries with queued calls. At this point, the protected action is completed.

If the entry_barrier evaluates to false, the entry call is added to the queue of the entry. Then again, all entries that have queued calls and whose barriers reference variables or attributes which might have changed since they were last evaluated are reevaluated. If any entries with queued calls are open, one open entry is chosen and the associated entry_body is executed. This will be repeated until there are no open entries with queued calls. At this point, the protected action is completed. In Figure 5.4 which gives a summary of these steps a Roman font indicates that an action is done by T and Sans Serife indicates that an action can be done by any task of the system.

```
erw_lock
compute barrier if it might have changed or has not yet been evaluated
if (barrier) then
   execute entry_body
   loop
      compute all barriers that might have changed
      if there is no open entry with queued calls
         exit
      else
         choose one open entry and execute the associated entry_body
      end if
   end loop
else
   add entry call to corresponding queue
   loop
      compute all barriers that might have changed
      if there is no open entry with queued calls
         exit
      else
         choose one open entry and execute the associated entry_body
      end if
   end loop
end if
erw_release;
```

Figure 5.4: Executing a protected action that is caused by a protected entry

When it is attempted to cancel an entry call, this can only take place if the call is in some entry queue. At such a situation (when the call is in some entry queue), the call is removed from the entry queue and the call completes due to cancellation. Such cancellation is a protected action and it cannot take place while any other protected action of the same protected object is in progress. Figure 5.5 gives a short formalization of that.

```
erw_lock
if (call is in some entry queue) then
   remove call
   loop
      compute all barriers that might have changed
      if there is no open entry with queued calls
         exit
      else
         choose one open entry and execute the associated entry_body
      end if
    end loop
  end if
  erw_release
```

Figure 5.5: Attempting to cancel an entry call

5.1.1.5 Bounded Errors and Protected Objects

First we should explain what a bounded error is. In Ada, there are certain kinds of errors that need not be detected either prior to or at runtime; but if not detected, the range of possible effects shall be bounded. The errors of this category are called bounded errors. The possible effects of a given bounded error are specified for each such error. But in any case the exception `Program_Error` might be raised.

If a protected operation is executed by a task, other tasks are blocked when they try to gain access to that protected object. So it is clear that the code inside a protected operation should be as short as possible. To prevent such blocked tasks from waiting indefinitely, Ada defines it to be a bounded error if a potentially blocking operation is invoked during a protected action. The following operations are defined to be potentially blocking:

- select statement
- accept statement
- delay statement
- entry call statement
- abort statement
- task creation and activation
- external call to a protected subprogram with the same target object as of the protected operation (This is a bounded error since the calling task has already acquired the execution resource and tries to get it again.)
- call on a subprogram whose body contains a potentially blocking operation

If a bounded error is detected, `Program_Error` is raised. If the bounded error is not detected, deadlock might result. There are some language-defined subprograms that are potentially blocking. For instance, the subprograms of the input-output packages that manipulate files are potentially blocking. Since it is important to know whether a subprogram is potentially blocking or not, potentially blocking operations are identified where they are defined. If a language-defined subprogram is not specified as potentially blocking, then it is nonblocking.

5.1.2 Control of Shared Variables

The Systems Programming Annex provides two pragmas to support safe reading and writing of shared variables—another form of indirect communication—between unsynchronized tasks. These two pragmas have the following syntax:

```
pragma Atomic (local_name);
pragma Atomic_Components (array_local_name);
```

They can be applied to certain objects or to types. In the second case, all objects of such atomic types are atomic. The pragma `Atomic_Components` is applied to array types or array objects and ensures that all components of the array are atomic.

For an atomic object, all reads and writes of the object as a whole are indivisible. That is, all reads and writes are atomic to each other.

By using `pragma Atomic` we can implement our account as a simple package in which the balance is made atomic. The code for that package is given in Example 5.3.

```
package Account is
   procedure Deposit (Amount : in Positive);
   procedure Withdraw (Amount : in Positive);
   function Provide_Balance return Integer;
end Account;

package body Account is
   Balance : Integer := 1000;
   pragma Atomic(Balance);
   procedure Deposit (Amount : in Positive) is
   begin
      Balance := Balance + Amount;
   end Deposit;
   procedure Withdraw (Amount : in Positive) is
   begin
      Balance := Balance - Amount;
   end Withdraw;
   function Provide_Balance return Integer is
   begin
      return Balance;
   end Provide_Balance;
end Account;
```

Example 5.3: Mutual exclusion by using pragma Atomic

This solution has the disadvantage that it is not possible to execute an operation only if some condition holds and to queue the call to that operation otherwise.

An implementation is not required to support these pragmas for all types. If the implementation cannot support indivisible reads and writes, the pragma has to be rejected by the compiler.

5.1.3 Synchronous Task Control

The package `Ada.Synchronous_Task_Control` is part of the Annex D of the Ada Reference Manual and provides a binary semaphore. This package has the following specification:

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;
   procedure Set_True (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);
   function Current_State (S : Suspension_Object) return Boolean;
   procedure Suspend_Until_True (S : in out Suspension_Object);
```

```
private
   ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The semaphore is represented by an object of type `Suspension_Object`. Such an object has two visible states: true and false. Its value is initially set to false.

The operations `Set_True` and `Set_False` are used to set the state to true and false, respectively. The procedure `Suspend_Until_True` blocks the calling task until the state of the object is true. When the state of the object becomes true, the task becomes ready and the state of the object becomes false. A call to the function `Current_State` returns the current state of the object. If there is a task waiting on a suspension object S and another task invokes `Suspend_Until_True` for S, then Program_Error is raised.

The operations `Set_True`, `Set_False` and `Suspend_Until_True` are atomic with respect to each other.

In essence, tasks can synchronize themselves on `Suspension_Objects`; hence these provide a sort of (low-level) communication.

By using `Ada.Synchronous_Task_Control`, we can implement our account as a package whose body contains the balance and the semaphore variable. The code for that is given in Example 5.4.

```
package Account_With_Semaphore is
   procedure Deposit (Amount : in Positive);
   procedure Withdraw (Amount : in Positive);
end Account_With_Semaphore;

with Ada.Synchronous_Task_Control;
use Ada.Synchronous_Task_Control;

package body Account_With_Semaphore is
   S : Suspension_Object;
   Balance : Integer := 0;
   procedure Deposit (Amount : in Positive) is
   begin
      loop
         begin
            Suspend_Until_True (S);
            exit;
         exception
            when Program_Error => null;
         end;
      end loop;
      Balance := Balance + Amount;
      Set_True (S);
   end Deposit;

   procedure Withdraw (Amount : in Positive) is
   begin
      loop
         begin
            Suspend_Until_True (S);
            exit;
         exception
            when Program_Error => null;
         end;
      end loop;
      Balance := Balance - Amount;
      Set_True (S);
   end Withdraw;

   begin
      Set_True (S);
   end Account_With_Semaphore;
```

Example 5.4: Mutual exclusion by using the predefined semaphore

5.2 Objects with Coordinated Access in Java

To realize coordinated access to objects, Java provides the concept of synchronized methods and synchronized statements. These mechanisms are used for allowing only one thread at a time to execute a certain block of code. With the help of synchronized methods, monitors can be realized.

5.2.1 Locks on Objects

Every object in Java is associated with a lock. There is no way to perform separate lock and unlock actions in an explicit manner. These actions are implicitely performed by synchronized methods or statements.

5.2.2 Synchronized Methods

A method is synchronized if the method declaration contains the keyword `synchronized`.

If a call to a synchronized method was made, the synchronized method automatically performs a lock action. The body of the synchronized method is only executed when the lock action has been successfully completed. If the synchronized method is a class method, the lock associated with the class object representing the method's class is used. If the synchronized method is an instance method, then the lock associated with the instance for which the method was invoked is used.

After the lock action has been successfully completed, the body of the synchronized method is executed. When the execution of the body of the synchronized method completes, an unlock action is automatically performed on that same lock.

This mechanism guarantees that at most one synchronized method of a given object may be executed at a given time. That is, calls to synchronized methods of the same object are mutuallay exclusive to each other.

Figure 5.6 depicts the process of calling a synchronized method.

Now we are ready to give a first solution of our earlier example of an account that should contain a data component (balance) and three operations for withdrawing and depositing money and for providing the actual balance. These operations are implemented as synchronized methods to enforce mutual exclusion. Example 5.5 shows this first solution.

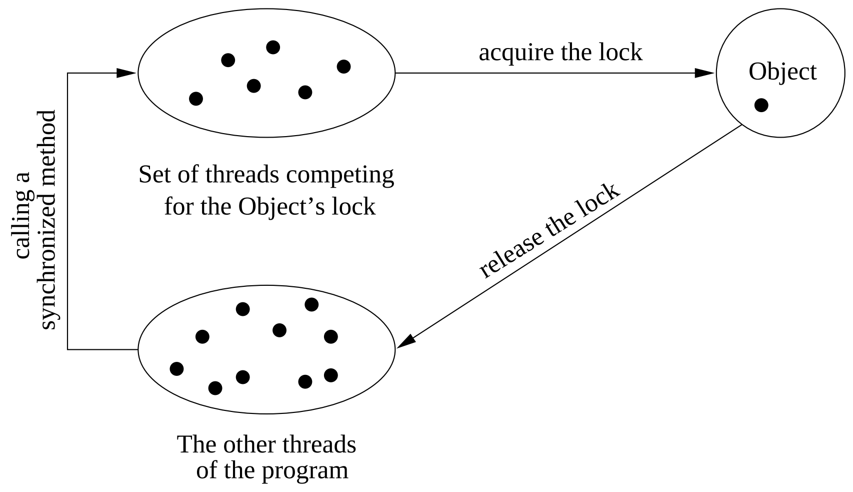
acquire the lock
Object
calling a synchronized method
Set of threads competing for the Object's lock
release the lock
The other threads of the program

Figure 5.6: Calling a synchronized method

```
class account {
  private int balance;
  account (int balance) {
    this.balance = balance;
  }
  public synchronized void deposit (int amount) {
    balance = balance + amount;
  }
  public synchronized void withdraw (int amount) {
    balance = balance - amount;
  }
  public synchronized int provide_balance () {
    return balance;
  }
}
```

Example 5.5: Use of synchronized methods

5.2.3 Synchronized Statements

The syntax of a synchronized statement reads as follows:

```
synchronized (Expression) Block
```

The type of Expression must be a reference type; otherwise a compile-time error occurs. The first step of executing a synchronized statement is the evaluation of the Expression. If this evaluation completes abruptly, the synchronized statement completes abruptly. If the value of Expression is null, a `NullPointerException` is thrown.

Otherwise, let V be the value of Expression. The thread that executes the synchronized statement locks the lock associated with V. After that the Block is executed. After normal execution of the Block the lock is unlocked and the synchronized statement completes normally. If the execution of the Block completes abruptly, the lock is unlocked and the synchronized statement completes abruptly.

The example of the account can be implemented using synchronized statements instead of synchronized methods. This approach is given in Example 5.6.

```
class account {
  int balance;
  account (int balance) {
    this.balance = balance;
  }
  public void deposit (int amount) {
    synchronized (this) {
      balance = balance + amount;
    }
  }
  public void withdraw (int amount) {
    synchronized (this) {
      balance = balance - amount;
    }
  }
  public int provide_balance () {
    synchronized (this) {
      return balance;
    }
  }
}
```

Example 5.6: Use of synchronized statements

5.2.4 Drawback of Synchronized Methods / Statements

As mentioned earlier, synchronized methods and statements provide mutually exclusive access to objects. But mutual exclusion is only guaranteed if all components of a given object are manipulated via synchronized methods/statements. If a thread has acquired the lock of a given object, then this does not prevent other threads from directly manipulating public components of that object or invoking unsynchronized methods of the object.

Let us consider our above example. Additionally, the following declaration shall appear in a program:

```
account my_account = new account (100);
```

Threads that manipulate the value of `my_account.balance` via the synchronized methods `withdraw` and `deposit` do so under mutual exclusion. But if `balance` did not have the attribute `private`, it would be possible for a thread to execute the following statement:

```
my_account.balance = 0;
```

Such direct manipulations are not mutually exclusive to each other and data inconsistencies are possible.

To overcome this problem all data components that are accessed by more than one thread should get the attribute `private` and all methods that manipulate such components should be made synchronized. According to these rules, the code for our account is given in Example 5.7.

```
class account {
  private int balance;
  account (int balance) {
    this.balance = balance;
  }
  public synchronized void deposit (int amount) {
    balance = balance + amount;
  }
  public synchronized void withdraw (int amount) {
    balance = balance - amount;
  }
  public synchronized int provide_balance () {
    return balance;
  }
}
```

Example 5.7: The correct solution of the account

5.2.5 Wait Sets and Notification

When using monitors, it is often useful to have the possibility to execute an operation of that monitor only if some condition is satisfied that is evaluated inside the monitor. For that purpose, Ada provides protected entries where each protected entry has a barrier and might only be executed if that barrier evaluates to true. Java uses another concept that will be explained now.

As mentioned earlier, every object has an associated lock. In addition to that, each object has an associated wait set. This wait set is a set of threads. Initially, when the object is created, the wait set is empty.

Wait sets are used together with the methods `wait`, `notify` and `notifyAll` that are defined in class `java.lang.Object`. We now give a detailed description of these methods.

```
public final void wait (long millis) throws
  IllegalMonitorStateException, InterruptedException
```

This method may only be called when the thread calling it is already synchronized on the object containing the call to `wait`. In other words, this method must be called from within a synchronized method or synchronized block. When it is attempted to call the `wait` method from within a method or block that is not synchronized, an `IllegalMonitorStateException` is thrown.

Let T be a thread executing a synchronized method S of an object O. S shall contain a call to `wait`. After `wait` was called from within S, T is placed into the wait set for O and T releases the lock on O. T becomes disabled for thread scheduling purposes and is suspended until one of the following things happens:

- another thread invokes the `notify` method for O and T is the one thread to be awakened
- another thread invokes the `notifyAll` method for O
- the specified amount of time that is given by `millis` (and measured in milliseconds) has expired

Then T is removed from the wait set and reenabled for thread scheduling. That means, it has to compete with the other threads to lock the object O. Once it has locked O again, the synchronization state of O is restored to the situation when the `wait` method was invoked.

There are two other forms of the `wait` method:

```
public final void wait() throws
   IllegalMonitorStateException, InterruptedException

public final void wait (long millis, int nanos) throws
   IllegalMonitorStateException, InterruptedException
```

The first behaves as a call to `wait(0)`. The second measures the amount of time in nanoseconds; amount is given by: 1000000∗millis+nanos.

```
public final void notify() throws IllegalMonitorStateException
```

This method may only be called from within a synchronized method or synchronized block. Otherwise, an `IllegalMonitorStateException` is thrown.

Let O be the object containing the synchronized method or synchronized block that includes the call to `notify()`. If there are any threads in the wait set of O, then one of these threads is chosen in an arbitrary manner and is awakened. That means, the chosen thread is taken from the wait set and put into the set of threads that compete for the lock on O. During the competition the awakened thread is treated exactly in the same manner as the other threads that want to lock O.

```
public final void notifyAll() throws IllegalMonitorStateException
```

Similarly to the other methods, this method may only be called from within a synchronized method or synchronized block. Otherwise, an `IllegalMonitorStateException` is thrown.

Let O be the object containing the synchronized method or synchronized block that includes the call to `notifyAll()`.

All threads that are currently in the wait set of O are put into the set of threads that compete for the lock on O. During this competition the awakened threads are treated in the same manner as the other threads that want to lock O.

Now, we can extend Figure 5.6 to give an illustration of the interaction between an object's lock, its wait set and the set of threads competing for the lock. This is shown in Figure 5.7.

The dashed line in the illustration shall emphasize that the thread T calling `notify()` is not the one to be awakened. T must have acquired the lock of the object to execute the `notify()` method.

With the help of the methods `wait` and `notify()`, we can extend our example so that withdrawing is only possible when the account's balance is not less than the amount to be withdrawn. This approach is given in Example 5.8.

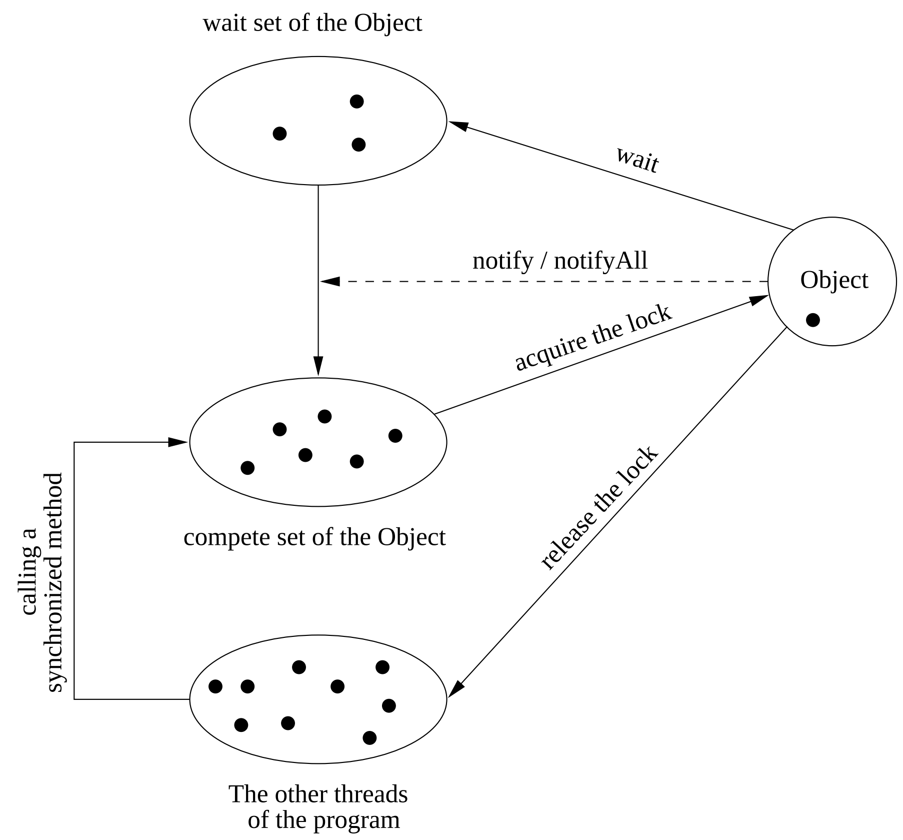
wait set of the Object
wait
notify / notifyAll
Object
acquire the lock
calling a synchronized method
compete set of the Object
release the lock
The other threads of the program

Figure 5.7: State diagram for the wait/notify mechanism

```
class account {
  private int balance;
  account (int balance) {
    this.balance = balance;
  }
  public synchronized void deposit (int amount) {
    balance = balance + amount;
    notify ();
  }
  public synchronized void withdraw (int amount)
   throws InterruptedException {
    while (balance < amount) {
      wait ();
    }
    balance = balance - amount;
  }
  public synchronized int provide_balance () {
    return balance;
  }
}
```

Example 5.8: Improved account with a conditional call

5.3 Objects with Coordinated Access in CHILL

5.3.1 Regions

CHILL regions provide mutually exclusive access to data objects declared inside the region. Regions are monitors.

Syntax:

```
<region> ::=
            [ <context list> ] [ <defining occurrence> : ]
             REGION [ BODY ] <region body> END
            [ <handler> ] [ <simple name string> ];
        | <remote modulion>
        | <generic region instantiation>
```

A region must not be surrounded by a block other than the main program. Accesses to the data objects declared inside the region can only be made by calling procedures that are defined inside the region and that are exported by the region (exporting is done with the grant action). Such exported procedures are called critical procedures.

A region can be locked and released. A region will be locked to prevent concurrent execution if

- a critical procedure of that region is called
- a process that was delayed in the region is reactivated

The region will be released if

- a critical procedure returns
- a critical procedure P executes an action that causes the process that executes P to become delayed
- a process that executes a critical procedure terminates

While a region is locked, all threads that attempt to call a critical procedure of that region are suspended until the region is released. The same applies to threads that were delayed in a region and become reactivated while the region is locked. A thread that is suspended in such a way remains active in the CHILL sense.

When a region is released and more than one thread has been suspended on it, only one thread will be selected to lock the region. It is implementation defined which thread will be selected.

If a thread becomes delayed while executing a critical procedure, then the associated region is released and the dynamic context of the thread is retained until it is reactivated. When the thread becomes reactivated, it attempts to lock the region again which may cause it to be suspended.

Example 5.9 gives a first CHILL solution for our example of an account.

```
account:
REGION
   GRANT deposit, withdraw, provide_balance;
   DCL balance INT := 100;
   NEWMODE positive = INT(1:1000);

   deposit:
   PROC (amount positive);
      balance := balance + amount;
   END deposit;

   withdraw:
   PROC (amount positive);
      balance := balance - amount;
   END withdraw;

   provide_balance:
   PROC () RETURNS (INT);
      RETURN balance;
   END provide_balance;
END account;
```

Example 5.9: Simple account in CHILL

5.3.2 Events

As we have seen so far, monitors can be implemented by using CHILL regions. But it is often necessary to execute a critical procedure only if some condition is satisfied. For this purpose, CHILL provides events, the syntax of which is given by:

```
<event mode> ::=
              EVENT  [ ( <event length> ) ]
            | <event mode name>
<event length> ::=
              <integer literal expression>
```

Event mode locations provide a means for synchronization between processes. The maximum number of processes that may become delayed on an event location is given by the event length. If it is specified, it must deliver a positive value. If no event length is specified, the number of processes that may become delayed is unlimited.

The following operations are defined on event mode locations: continue action, delay action and delay case action. Figure 5.8 shows the use of these actions.

If a process executes a delay action or a delay case action, it becomes delayed on the specified event. That means, it becomes a member of the set of processes delayed on that event. A process that is delayed on an event can only be reactivated (i.e., taken from the set of processes delayed on the event and put into the set of active processes accessing the event) if another active process executes a continue action.

A more detailed description of the operations defined on events will follow.

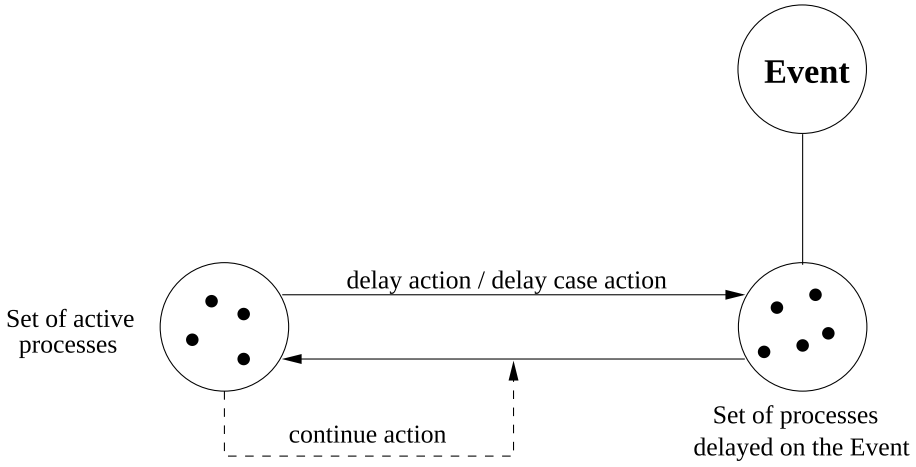
Event
delay action / delay case action
Set of active processes
continue action
Set of processes delayed on the Event

Figure 5.8: State Diagram of CHILL Events

Delay Action

Syntax:

```
<delay action> ::=
            DELAY  <event location> [priority]
<priority> ::=
            PRIORITY  <integer literal expression>
```

If the event location has a mode with an event length that is equal to the number of processes already delayed on the event location, then a DELAYFAIL exception occurs. Otherwise, the executing process becomes delayed. This entails that it becomes a member with a priority of the set of delayed processes of the specified event location. If no priority is specified, the lowest priority (zero) is taken.

Delay Case Action

Syntax:

```
<delay case action> ::=
            DELAY CASE  [ SET  <instance location> [<priority>] ;  |
                                          <priority> ; ]
            {<delay alternative>}+
            ESAC
<delay alternative> ::=
            (<event list>):<action statement list>
<event list> ::=
            <event location> {,<event location>}*
```

If any event location has a mode with an event length that is equal to the number of processes already delayed on the current location, a DELAYFAIL exception occurs. Otherwise, the executing process becomes delayed. This entails that it becomes a member with a priority of the set of delayed processes of each of the specified event locations. If priority is not specified, the lowest (zero) is taken. If an instance location is specified, then the instance value identifying the process instance that has executed the continue action is stored in it.

Continue Action

Syntax:

```
<continue action> ::=
        CONTINUE <event location>
```

If the set of delayed processes of the event location is non-empty, the process with the highest priority is reactivated. If there are several such processes (with the same highest priority), one of them is selected in an implementation defined way. The reactivated process is removed from all sets of delayed processes in which it was contained.

If the set of delayed processes is empty, the continue action has no effect.

With the help of events and the their associated operations, we are able to improve our account in such a way that withdrawing is only possible if the account's balance is greater than zero. This solution is given in Example 5.10.

```
account:
REGION
   GRANT deposit, withdraw, provide_balance;
   DCL balance INT := 100;
   DCL low_balance EVENT;

   deposit:
   PROC (amount INT);
      balance := balance + amount;
      CONTINUE low_balance;
   END deposit;

   withdraw:
   PROC (amount INT);
      DO WHILE balance <= 0;
         DELAY low_balance;
      OD;
      balance := balance - amount;
   END withdraw;

   provide_balance:
   PROC () RETURNS (INT);
      RETURN balance;
   END provide_balance;
END account;
```

Example 5.10: Improved account with conditional call

5.3.3 Buffers

A CHIILL Buffer provides a means for synchronization and communication between processes. Several processes may put values in and get values from a buffer. This is done by using the two operations defined on buffers, the send buffer action and the receive buffer case action.

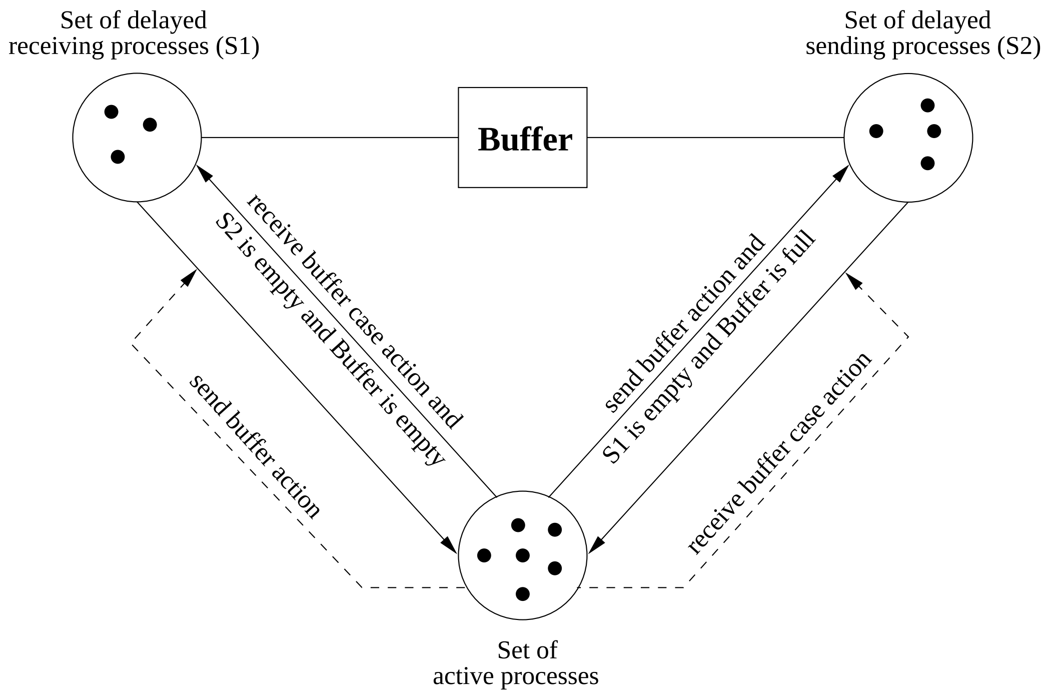
Set of delayed receiving processes (S1)
Set of delayed sending processes (S2)
Buffer
receive buffer case action and S2 is empty and Buffer is empty
send buffer action and S1 is empty and Buffer is full
send buffer action
receive buffer case action
Set of active processes

Figure 5.9: State Diagram of a CHILL Buffer

Syntax of a buffer mode:

```
<buffer mode> ::=
                BUFFER [(<buffer length>)] <buffer element mode>
           |    <buffer mode name>
<buffer length> ::=
                <integer literal expression>
<buffer element mode> ::=
                <mode>
```

The optional buffer length specifies the maximum number of values that can be stored in a buffer location. It must deliver a non-negative value. If no buffer length is specified, the number of values is unlimited. The buffer is full if the buffer length is equal to the number of values already stored in the buffer. Each buffer has two sets attached, a set of delayed receiving processes and a set of delayed sending processes. The set of delayed receiving processes contains all processes that were delayed while executing a receive buffer case action and the set of delayed sending processes contains all processes that became delayed during a send buffer action on that buffer. Finally, there are the active processes that access the buffer via the receive buffer case action and the send buffer action. Figure 5.9 illustrates that situation.

Now, we describe the send buffer action and the receive buffer case action in more detail.

Send Buffer Action

Syntax:

```
<send buffer action> ::=
                SEND <buffer location> (<value>) [<priority>]
```

If the set of delayed receiving processes of the buffer location is non-empty, then one of the delayed processes is selected in an implementation defined way and then reactivated. This entails that it is removed from all sets of delayed receiving processes in which it was contained.

If the set of delayed receiving processes is empty and the buffer is full, then the executing process becomes delayed with a priority. If priority is not specified, the lowest priority (zero) is taken. That means, it is put into the set of delayed sending processes attached to the buffer location.

If the set of delayed receiving processes is empty and the capacity of the buffer location is not exceeded, then the value given by value is stored with a priority. If priority is not specified, then the lowest priority (zero) is taken.

The value that is put into the buffer is specified by value. The class of value must be compatible with the buffer element mode of the mode of the buffer location.

Receive Buffer Case Action

Syntax:

```
<receive buffer case action> ::=
            RECEIVE CASE [ SET <instance location> ; ]
            { <buffer receive alternative> }+
            [ ELSE <action statement list> ]
            ESAC
          | RECEIVE [ SET <instance location> ]
           ( <buffer location> IN <location> )
<buffer receive alternative> ::=
           ( <buffer location> IN <defining occurrence> ) : <action statement
list>
```

If a process executes a receive buffer case action, either of the following is possible:

- the process receives a value
- the process enters the action statement list of the ELSE
- the process becomes delayed

The executing process receives a value if a value is stored in one of the specified buffer locations or if a sending process is delayed on one of the specified buffer locations. If there are more than one such values, one with the highest priority is selected in an implementation defined way. If the executing process receives a value from a buffer location which has a non-empty set of delayed sending processes, then one of the delayed sending processes is reactivated. If the received value was stored in the buffer location, the reactivated process is one with the highest priority and its value is stored in the buffer location. If the received value was not stored in the buffer location, the reactivated process is the one sending the received value.

If the executing process cannot receive a value and ELSE is specified, the corresponding action statement list is entered.

If the executing process cannot receive a value and ELSE is not specified, then the executing process becomes delayed. This entails that it becomes a member of the set of delayed processes of each of the specified buffer locations. If this delayed process becomes reactivated by another process that executes a send buffer action, it receives a value.

A simple example of two processes sending data to and receiving data from a buffer is given in Example 5.11.

```
simple_buffer:
MODULE
   NEWMODE int_buffer = BUFFER (10) INT;
   DCL my_buffer int_buffer;

   write_to_buffer:
   PROCESS (value INT);
      DO FOR EVER;
         SEND my_buffer (value);
      OD;
   END write_to_buffer;

   read_from_buffer:
   PROCESS ();
      DCL value INT;
      DO FOR EVER;
         RECEIVE (my_buffer IN value);
      OD;
   END read_from_buffer;

   START write_to_buffer (10);
   START read_from_buffer ();
END simple_buffer;
```

Example 5.11: A simple buffer in CHILL

Summary and Comparison

All three languages provide a kind of monitor. In Ada, this can be done by using protected types, in Java synchronized methods are used and CHILL provides regions. All these constructs are similar in that they guarantee mutual exclusion. But they differ in

- the kind of operation that is supported
- the concepts for avoiding deadlocks
- the support of "immediate resumption"

Ada's protected objects provide three kinds of accessing the shared data encapsulated in the protected object: protected functions, protected procedures and protected entries. Protected functions allow concurrent read-only access whereas protected procedures and protected entries are mutually exclusive to each other and to protected functions. Additionally, protected entries are used if a protected operation may only be executed if some condition that is represented by the `entry_barrier` holds. If such a condition is not satisfied, the entry call is queued in the corresponding entry queue. It is possible to queue entry calls according to their order of arrival (FIFO) or according to their priorities (see Chapter 6).

To help avoiding the occurrence of deadlocks, Ada provides the concept of potentially blocking operations. This is a means to bound the period of time a task can lock a protected object (and so blocks other tasks accessing the protected object).

To achieve "immediate resumption", Ada provides the concept of servicing the entry queues. Since only protected procedures and protected entries can change the state of the protected object, each time a protected procedure or protected entry has completed (but before the protected action completes), all entries with queued calls are checked if their barrier has become true. If there are such entries, the barrier of which has changed to true, then all queued calls of these entries are serviced before the protected action completes.

Java's approach to monitors is based on the synchronized methods. Calls to synchronized methods are mutually exclusive to each other; concurrent read-only access is not possible.

Conditional execution of synchronized statements can be done by the use of the methods `wait()` and `notify()`. The `wait()` method causes the currently executing thread to be put into the wait set associated with the monitor object until another thread invokes the `notify()` method. As a result, one thread is selected from the wait set and is reenabled for competing for the object's lock (there is no guarantee that it will be the next to lock the object). Obviously, this mechanism does not allow any queuing of threads and, therefore, the order in which the threads arrived is lost. Although each synchronized method may have its own condition, there is only one wait set per monitor. This might in awaking threads (via calls to `notify()`) that were delayed on a condition that has not changed at all and a deadlock could occur. The only way to overcome this problem is using the `notifyAll()` method instead of `notify()`. But this solution suffers from a big overhead since all threads in the wait set are awakened and have to reevaluate the condition they were delayed on.

Java does not provide any concept to avoid deadlocks. But this does not seem to be a real disadvantage since a Java thread may hold a lock more than once. There is no support of "immediate resumption" in Java.

As mentioned earlier, the fact that a Java thread has acquired the object's lock does not necessarily prevent other threads from accessing the monitor's data via methods that are not synchronized or via directly manipulating (if the data is not explicitely made `private`). This seems to be a real disadvantage.

The CHILL solution of a monitor is based on the region. Regions provide mutually exclusive access to the data declared inside the region via critical procedures. Since critical procedures a mutually exclusive to each other, concurrent read-only access is not possible.

Conditional execution of critical procedures is achieved by events and the operations DELAY and CONTINUE that are defined on events. This mechanism is quite similar to Java's concept of `wait()` and `notify()`. A DELAY causes the current process to be delayed on the specified event until another process invokes CONTINUE and one process with the highest priority is reactivated. If no priority was specified, the choice is implementation defined. This means, that priority queuing is the only possible queuing policy within the set of processes delayed on an event. Since each critical procedure may use a separate event, we can be assured that we awake only those processes (by using CONTINUE) the condition on which they were delayed has changed to true.

CHILL does not provide any concept to bound the time a process can lock a region. There is no support of "immediate resumption".

Additionally to the high level construct of protected objects, Ada provides two low level constructs: a semaphore and a pragma for making accesses to a shared variable atomic. Besides the regions, CHILL provides another means for synchronization and communication between processes: the buffers.

All the constructs that are described in this chapter can be used for indirect actor communication.

Table 5.1 gives a compact overview of the several concepts that are provided by the languages.

Concept	**Ada**	**CHILL**	**Java**
exclusive read-write access	Yes	Yes	Yes
concurrent read-only access	Yes	No	No
conditional operations	Yes	Yes	Yes
queuing of calls according FIFO	Yes	No	No
queuing of calls according priorities	Yes	No	No
support of immediate resumption	Yes	No	No
language predefined Buffers	No	Yes	No
language predefined Events	No	Yes	No
language predefined atomic variables	Yes	No	No
language predefined Semaphores	Yes	No	No

Table 5.1: Summary of the concepts provided by the three languages