Chapter 7
Atomic actions, concurrent tasks and reliability

Chapter 2 considered how reliable software could be produced in the presence of a variety of errors. Modular decomposition and atomic actions were identified as two techniques essential for damage confinement and assessment. Also, the notions of forward and backward error recovery were introduced as approaches to dynamic error recovery. It was shown that where tasks communicate and synchronize their activities, backward error recovery may lead to the domino effect. In Chapter 3, exception handling was discussed as a mechanism for providing both forward and backward error recovery in sequential tasks. Chapters 4, 5 and 6 then considered the facilities provided by operating systems and real-time languages for concurrent programming. This chapter brings together exception handling and concurrency in order to show how tasks can interact reliably in the presence of other tasks and in the presence of faults. The notion of an atomic action is explored in more detail and the concept of asynchronous notification is introduced.

In Chapter 4, the interaction of tasks was described in terms of three types of behaviour:

- independent
- cooperation
- competition.

Independent tasks do not communicate or synchronize with each other. Consequently, if an error occurs within one task, then recovery procedures can be initiated by that task in isolation from the rest of the system. Recovery blocks and exception handling can be used as described in Chapters 2 and 3.

Cooperating tasks, by comparison, regularly communicate and synchronize their activities in order to perform some common operation. If any error condition occurs, it is necessary for all tasks involved to perform error recovery. The programming of such error recovery is the topic of this chapter.

Competing tasks communicate and synchronize in order to obtain resources; they are, however, essentially, independent. An error in one should have no effect on the others. Unfortunately, this is not always the case, particularly if the error occurred while a task was in the act of being allocated a resource. Reliable resource allocation is considered in Chapter 8.

Where cooperating tasks communicate and synchronize through shared resources, recovery may involve the resource itself. This aspect of resource allocation will also be considered in Chapter 8.

7.1 Atomic actions

One of the main motivations for introducing concurrent tasks into a language is that they enable parallelism in the real world to be reflected in application programs. This enables such programs to be expressed in a more natural way and leads to the production of more reliable and maintainable systems. Disappointingly, however, concurrent tasks create many new problems which did not exist in the purely sequential program. Consequently, the last few chapters have been dedicated to discussing some of the solutions to these problems: in particular, communication and synchronization between tasks using shared variables (correctly) and message passing. This was undertaken in a fairly isolated manner and no consideration has yet been given to the way in which groups of concurrent tasks should be structured in order to coordinate their activities.

The interaction between two tasks has, so far, been expressed in terms of a single communication. In reality, this is not always the case. For example, withdrawal from a bank account may involve a ledger task and a payment task in a sequence of communications to authenticate the drawer, check the balance and pay the money. Furthermore, it may be necessary for more than two tasks to interact in this way to perform the required action. In all such situations, it is imperative that the tasks involved see a consistent system state. With concurrent tasks, it is all too easy for groups of tasks to interfere with one other.

What is required is for each group of tasks to execute their joint activity as an **indivisible** or **atomic action**. Of course, a single task may also want to protect itself from the interference of other tasks (for example, during resource allocation). It follows that an atomic action may involve one or more tasks. Atomic actions have also been called *multiparty* interactions (Evangelist et al., 1989; Yuh-Jzer and Smolka, 1996).

There are several almost equivalent ways of expressing the properties of an atomic action (Lomet, 1977; Randell et al., 1978).

(1) An action is atomic if the tasks performing it are not aware of the existence of any other active task, and no other active task is aware of the activity of the tasks during the time the tasks are performing the action.

(2) An action is atomic if the tasks performing it do not communicate with other tasks while the action is being performed.

(3) An action is atomic if the tasks performing it can detect no state change except those performed by themselves and if they do not reveal their state changes until the action is complete.

(4) Actions are atomic if they can be considered, so far as other tasks are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

These are not quite all equivalent. For example, consider the second expression: an action is atomic if the tasks performing it communicate only among themselves and not with other tasks in the system. Unlike the other three, this does not really define the true nature of an atomic action. While it will guarantee that the action is indivisible, it is too strong a constraint on the tasks. Interactions between an atomic action and the rest of the system can be allowed as long as they have no impact on the activity of the atomic action and do not provide the rest of the system with any information concerning the progress of the action (Anderson and Lee, 1990). In general, in order to allow such interactions requires detailed knowledge of the atomic action's function and its interface to the rest of the system. As this cannot be supported by a general language implementation, it is tempting, following Anderson and Lee (1990), to adopt the more restrictive (second) definition. This can only be done, however, if the resources necessary to complete an atomic action are acquired by the underlying implementation, not by instructions given in the program. If resources are to be acquired and released when the programmer desires, tasks within atomic actions will have to communicate with general-purpose resource managers.

Although an atomic action is viewed as being indivisible, it can have an internal structure. To allow modular decomposition of atomic actions, the notion of a **nested atomic action** is introduced. The tasks involved in a nested action must be a subset of those involved in the outer level of the action. If this were not the case, a nested action could smuggle information concerning the outer level action to an external task. The outer level action would then no longer be indivisible.

7.1.1 Two-phase atomic actions

Ideally, all tasks involved in an atomic action should obtain the resources they require (for the duration of the action) prior to its commencement. These resources could then be released after the atomic action had terminated. If these rules were followed, there would be no need for an atomic action to interact with any external entity and the stricter definition of atomic action could be adopted.

Unfortunately, this ideal can lead to poor resource utilization, and hence a more pragmatic approach is needed. Firstly, it is necessary to allow an atomic action to start without its full complement of resources. At some point, a task within the action will request a resource allocation; the atomic action must then communicate with the resource manager. This manager may be a server task. If a strict definition of atomic action is adhered to, this server would have to form part of the atomic action, with the effect of serializing all actions involving the server. Clearly, this is undesirable, and hence an atomic action is allowed to communicate externally with resource servers.

Within this context, a resource server is defined to be a custodian of non-sharable system utilities. It protects these utilities against inappropriate access, but does not, itself, perform any actions upon them.

A further improvement in resource allocation can be made if a task is allowed to release a resource prior to completion of the associated atomic action. In order for this premature release to make sense, the state of the resource must be identical to that which would appertain if the resource was retained until completion of the atomic action. Its early release will, however, enhance the concurrency of the whole system.

If resources are to be obtained late and released early it could be possible for an external state change to be affected by a released resource and observed by the acquisition of a new resource. This would break the definition of atomic action. It follows that the only safe policy for resource usage is one that has two distinct phases. In the first 'growing' phase, resources can be requested (only); in the following 'shrinking' phase, resources can be released (but no new allocations can be made). With such a structure, the integrity of the atomic action is assured. However, it should be noted that if resources are released early then it will be more difficult to provide recovery if the atomic action fails. This is because the resource has been updated and another task may have observed the new state of the resource. Any attempt to invoke recovery in the other task may lead to the domino effect (see Section 2.5.3).

In all the following discussions, atomic actions are assumed to be two-phased; recoverable actions do not release any resources until the action successfully completes.

7.1.2 Atomic transactions

Within the theories of operating systems and databases, the term **atomic transaction** is often used. An atomic transaction has all the properties of an atomic action plus the added feature that its execution is allowed either to succeed or to fail. By failure, it is meant that an error has occurred from which the transaction cannot recover; for example, a processor failure. If an atomic action fails then the components of the system, which are being manipulated by the action, may be left in an inconsistent state. With an atomic transaction, this cannot happen because the components are returned to their original state (that is, the state they were *before* the transaction commenced). Atomic transactions are sometimes called **recoverable actions** and, unfortunately, the terms **atomic action** and **atomic transaction** are often interchanged.

The two distinctive properties of atomic transactions are:

- **failure atomicity** – meaning that the transaction must either complete successfully or (in the case of failure) have no effect;
- **synchronization atomicity** (or isolation) – meaning that the transaction is indivisible in the sense that its partial execution cannot be observed by any concurrently executing transaction.

Although atomic transactions are useful for those applications which involve the manipulation of databases, they are not suitable for programming fault-tolerant systems *per se*. This is because they imply that some form of recovery mechanism will be supplied by the system. Such a mechanism would be fixed, with the programmer having no control over

its operation. Although atomic transactions provide a form of backward error recovery, they do not allow recovery procedures to be performed. Notwithstanding these points, atomic transactions do have a role in protecting the integrity of a real-time database system.

7.1.3 Requirements for atomic actions

If a real-time programming language is to be capable of supporting atomic actions, it must be possible to express the requirements necessary for their implementation. These requirements are independent of the notion of a task and the form of intertask communication provided by a language. They are the following.

- **Well-defined boundaries** – each atomic action should have a start, an end and a side boundary. The start boundary is the location in each task involved in the atomic action where the action is deemed to start. The end boundary is the location in each task involved in the atomic action where the action is deemed to end. The side boundary separates those tasks involved in the atomic action from those in the rest of the system.
- **Indivisibility (isolation)** – an atomic action must not allow the exchange of any information between the tasks active inside the action and those outside (resource managers excluded). If two atomic actions do share data then the value of that data after the atomic actions is determined by the strict sequencing of the two actions in some order.
  There is no implied synchronization at the start of an atomic action. Tasks can enter at different times. However, there is an implied synchronization at the end of an atomic action; tasks are not allowed to leave the atomic action until all tasks are willing and able to leave.
- **Nesting** – atomic actions may be nested as long as they do not overlap with other atomic actions. Consequently, in general, only strict nesting is allowed (two structures are strictly nested if one is completely contained within the other).
- **Concurrency** – it should be possible to execute different atomic actions concurrently. One way to enforce indivisibility is to run atomic actions sequentially. However, this could seriously impair the performance of the overall system and therefore should be avoided. Nevertheless, the overall effect of running a collection of atomic actions concurrently must be the same as that which would be obtained from serializing their executions.
- As it is the intention that atomic actions should form the basis of damage confinement, they must allow recovery procedures to be programmed.

Figure 7.1 diagrammatically represents the boundaries of a nested atomic action in a system of six tasks. Action B involves only tasks P_3 and P_4, whereas action A also includes P_2 and P_5. The other tasks (P_1 and P_6) are outside the boundaries of both atomic actions.

It, perhaps, should be noted at this point that some definitions of atomic actions require that all tasks be synchronized on *both* entry and exit of the action.

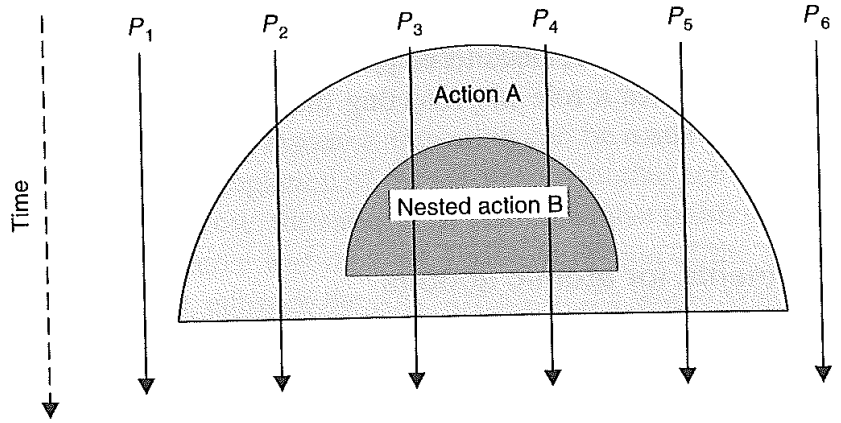

Figure 7.1 Nested atomic actions.

7.2 Atomic actions in C/Real-Time POSIX, Ada and Real-Time Java

Atomic actions provide structuring support for the software of large embedded systems. To get the full benefit of this aid requires the support of the real-time language. Unfortunately, such support is not directly provided by any of the major languages. This section considers the suitability of C/Real-Time POSIX, Ada and Real-Time Java for programming atomic actions. Following this, a possible language framework is given, and then this framework is extended to provide forward and backward error recovery.

The problem of resource allocation is postponed until Chapter 8. For now, it is assumed that resources have two modes of use: sharable and non-sharable, with some resources being amenable to both sharable and non-sharable modes. Furthermore, it is assumed that all actions are two-phased, and that the resource manager will ensure that appropriate usage is made of the resources. Also tasks within an action synchronize their own access to the resource to avoid any interference.

Atomic actions could be encapsulated within the monitor-like construct that is found in each of the languages. However, this would not allow any parallelism within an action. Hence, an alternative approach is adopted. In a similar fashion to the solution to the reader/writers problem presented in Section 5.9.1, the entry and exit protocols are implemented with a monitor-like construct and then the application code is surrounded by calls to the routines. Figure 7.2 illustrates an atomic action that requires three tasks (T_1, T_2 and T_3). Each task has its own role to play in the action. These roles and represented by the action procedures: for example, T_1 calls Action Procedure 1, etc. These action procedures then liaise with an action controller to provide the required synchronization. Figure 7.3 shows the structure of the roles.

7.2.1 Atomic actions and C/Real-Time POSIX mutexes

Using C and the Real-Time POSIX API, support for atomic actions following the structure given above can be provided as follows. Here only a two thread action is illustrated, but the approach is easily extended to more threads.

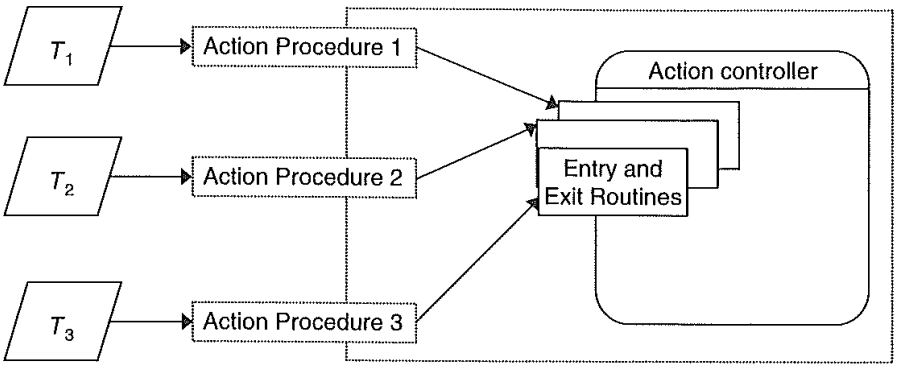

Figure 7.2 The structure of an action controller.

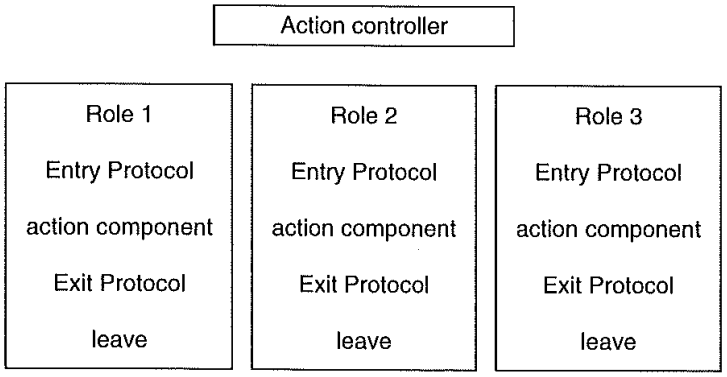

Figure 7.3 Using the action controller.

First, the mutex, condition variables and shared variables needed by the action controller are encapsulated in a `typdef`. Several boolean variables are used to represent the state of the action, and a mutex and four condition variables (and their associated attribute objects) provide the necessary synchronization.

```
#include <pthread.h>

typedef unsigned int boolean;
#define false 0
#define true (!false)

typedef struct {
  boolean first_process_active;
  boolean second_process_active;
  boolean first_process_finished;
  boolean second_process_finished;

  pthread_mutexattr_t mutex_att;
  pthread_mutex_t mutex;

  pthread_condattr_t cond_att;
  pthread_cond_t no_first_process, no_second_process;
  pthread_cond_t atomic_action_ends1, atomic_action_ends2;
} monitor;
```

The entry and exit protocols in support of the first role can now be given:

```
void entry_1(monitor *M) {
  PTHREAD_MUTEX_LOCK(&M->mutex);
    while(M->first_process_active)
      PTHREAD_COND_WAIT(&M->no_first_process, &M->mutex);
    M->first_process_active = true;
}

void exit_1(monitor *M) {
  while(!M->second_process_finished)
    PTHREAD_COND_WAIT(&M->atomic_action_ends2, &M->mutex);
  PTHREAD_COND_SIGNAL(&M->atomic_action_ends1);
  M->first_process_active = false;
  PTHREAD_COND_SIGNAL(&M->no_first_process);
  PTHREAD_MUTEX_UNLOCK(&M->mutex);
}
```

To enter the action, the action controller first obtains a mutual exclusion lock so it knows that its action will not be interfered with. It then checks that there is no other thread executing this role. If there is, the controller forces the calling thread to wait until the role is available. If the role is available, the thread is allowed to continue and a note is made that the role is now active.

When the role is finished, the exit protocol is called. Here, the controller checks to see if the other thread has finished its role. If it hasn't the first thread is made to wait. If it has, the action controller signals that the first thread has finished its role, then returns the data structure to a position from which a new action can start. The second role is similarly structured.

On exit from its role, the thread signals that it has finished and then waits for the other thread to finish. Of course, the data structure must be initialized via some appropriate routine:

```
void init_action(monitor *M) {
 // initialize all mutex and conditions with the appropriate
 // attributes; initialize booleans to false
}
```

The structure of the application code takes the following form:

```
void code_for_first_process(monitor *M) {
  entry_1(M);
    // get resource in non-sharable mode
    // update resource

    // signal second process that it is ok
    // for it to access resource

    // any final processing
  exit_1(M);

}
```

```
void code_for_second_process (monitor *M) {
  entry_2(M);
    // initial processing

    // wait for first process to signal
    // that it is ok to access resource

    // access resource

    // release resource
  exit_2(M);
}
```

Although the above approach has been able to implement the synchronization aspects of an atomic action, there is no way to ensure that the application threads do not communicate with any other threads.

7.2.2 Atomic actions in Ada

The Ada approach to implementing an atomic action is illustrated below for a three task system. It is similar to the C/Real-Time POSIX approach but uses guards instead of condition variables. Here, the action is encapsulated within an Ada package:

```
package Action_X is
  procedure Code_For_First_Task(--params);
  procedure Code_For_Second_Task(--params);
  procedure Code_For_Third_Task(--params);
end Action_X;
```

The body of the package contains the action controller, which is implemented as a protected type.

```
package body Action_X is
  protected Action_Controller is
    entry First;
    entry Second;
    entry Third;
    entry Finished;
  private
    First_Here : Boolean := False;
    Second_Here : Boolean := False;
    Third_Here : Boolean := False;
    Release : Boolean := False;
  end Action_Controller;

  protected body Action_Controller is
    entry First when not First_Here is
    begin
      First_Here := True;
    end First;
```

```
  entry Second when not Second_Here is
  begin
    Second_Here := True;
  end Second;

  entry Third when not Third_Here is
  begin
    Third_Here := True;
  end Third;

  entry Finished when Release or Finished'Count = 3 is
  begin
    if Finished'Count = 0 then
      Release := False;
      First_Here := False;
      Second_Here := False;
      Third_Here := False;
    else
      Release := True;
    end if;
  end Finished;
end Action_Controller;
```

In the above code, the action is synchronized by the `Action_Controller` protected object. This ensures that only three tasks can be active in the action at any one time and that they are synchronized on exit. The boolean `Release` is used to program the required release conditions on `Finished`. The first two calls on `Finished` will be blocked as both parts of the barrier expression are false. When the third call comes, the `Count` attribute will become three; the barrier becomes open and one task will execute the entry body. The `Release` variable ensures that the other two tasks are both released. The last task to exit must ensure that the barrier is closed again.

The application code also has a similar structure to the C/Real-Time POSIX approach.

```
procedure Code_For_First_Task(--params) is
begin
  Action_Controller.First;
  -- acquire resources
  -- the action itself, communicates with tasks executing
  -- inside the action via resources
  Action_Controller.Finished;
  -- release resources
end Code_For_First_Task;

-- similar for second and third task
begin
  -- any initialization of local resources
end Action_X;
```

More details on how to program atomic actions in Ada can be found in Wellings and Burns (1997).

7.2.3 Atomic actions in Java

The previous sections have illustrated the basic structure for programming atomic actions. The Java approach could follow a similar structure. However, this section takes the opportunity to expand on this approach so that the Java support can easily be extended using inheritance.

First, an interface can be defined for a three-way atomic action:

```
public interface ThreeWayAtomicAction {
  public void role1();
  public void role2();
  public void role3();
}
```

Using this interface, it is possible to provide several action controllers that implement a variety of models. Applications can then choose the appropriate controller without having to change their code.

The following action controller implements the same semantics as that given previously for C/Real-Time POSIX and Ada. A synchronized `Controller` class implements the required entry and exit synchronization protocols. The `finished` method is, however, a little more complex than its Ada counterpart. This is due to the semantics of `wait` and `notifyAll`. In particular, it is necessary to count the tasks as they leave the action (using `toExit`) in order to know when to reset the internal data structures for the next action. In Ada this was achieved via the `Count` attribute.

```
public class AtomicActionControl
       implements ThreeWayAtomicAction {
  protected Controller Control;
  public AtomicActionControl()  { // constructor
    Control = new Controller();
  }

  class Controller {
    protected boolean firstHere, secondHere, thirdHere;
    protected int allDone;
    protected int toExit;
    protected int numberOfParticipants;

    Controller() {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      numberOfParticipants = 3;
      toExit = numberOfParticipants;
    }

    synchronized void first() throws InterruptedException {
      while(firstHere) wait();
      firstHere = true;
    }
```

```
  synchronized void second() throws InterruptedException {
    while(secondHere) wait();
    secondHere = true;
  }

  synchronized void third() throws InterruptedException {
    while(thirdHere) wait();
    thirdHere = true;
  }

  synchronized void finished() throws InterruptedException {
    allDone++;
    if(allDone == numberOfParticipants) {
      notifyAll();
    } else while(allDone != numberOfParticipants) {
        wait();
    };
    toExit--;
    if(toExit == 0) {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      toExit = numberOfParticipants;
      notifyAll(); // release all waiting for the next action
    }
  }
}

public void role1() {
  boolean done = false;
  while(!done) {
    try {
      Control.first();
      done = true;
    } catch (InterruptedException e) {
      // ignore
    }
  }

  // .... perform action

  done = false;
  while(!done) {
    try {
      Control.finished();
      done = true;
    } catch (InterruptedException e) {
      // ignore
    }
  }
}

public void role2() {
  // similar to role1
}
```

```
  public void role3() {
    // similar to role1
  }
}
```

Given the above framework, it is now possible to extend this to produce, say, a four-way action:

```
public interface FourWayAtomicAction
       extends ThreeWayAtomicAction {
  public void role4();
}
```

and then

```
public class NewAtomicActionControl extends AtomicActionControl
             implements FourWayAtomicAction {
  public NewAtomicActionControl() {
    Control = new RevisedController();
  }

  class RevisedController extends Controller {
    protected boolean fourthHere;

    RevisedController() {
      super();
      fourthHere = false;
      numberOfParticipants = 4;
      toExit = numberOfParticipants;
    }

    synchronized void fourth() throws InterruptedException {
      while(fourthHere) wait();
      fourthHere = true;
    }

    synchronized void finished() throws InterruptedException {
      super.finished();
      if(allDone == 0) {
        fourthHere = false;
        notifyAll();
      }
    }
  }
  public void role4() {
    boolean done = false;
    while(!done) {
      try {
        // As Control is of type Controller, it must first
        // be converted to a RevisedController in order
        // to call the fourth method
        ((RevisedController)Control).fourth();
        done = true;
      } catch (InterruptedException e) {
```

```
        // ignore
      }
    }

    // .... perform action

    done = false;
    while(!done) {
      try {
        Control.finished();
        done = true;
      } catch (InterruptedException e) {
        // ignore
      }
    }
  }
}
```

Note that it has been necessary to override the `finished` method. Care must be taken here because of Java's automatic run-time dispatching of method calls. All calls to `finished` in the original code will dispatch to the overridden method.

7.3 Recoverable atomic actions

Although the various models described above have enabled a simple atomic action to be expressed, they all rely on programmer discipline to ensure that no interactions with external tasks occur (apart from with resource allocators). Moreover, they assume that no task within an atomic action is aborted; if the real-time language supports an abort facility then a task could be asynchronously removed from the action leaving the action in an inconsistent state.

In general, none of the mainstream languages or operating systems directly supports backward or forward error recovery facilities in the context of atomic actions. (However, C/Real-Time POSIX, Ada and Java do provide asynchronous notification mechanisms which can be used to help program recovery – see Sections 7.5, 7.6 and 7.7.) Language mechanisms have been proposed in research-oriented systems. In order to discuss these mechanisms, a simple language framework for atomic actions is introduced. The proposed recovery mechanisms are then discussed in the context of this framework.

To simplify the framework, only static tasks will be considered. Also it will be assumed that all the tasks taking part in an atomic action are known at compile-time. Each task involved in an action declares an action statement which specifies: the action name, the other tasks taking part in the action, and the code to be executed by the declaring task on entry to the action. For example a task P_1 which wishes to enter into an atomic action A with tasks P_2 and P_3 would declare the following action:

```
action A with (P2, P3) do
    -- acquire resources
    -- communicate with P2 and P3
    -- release resources
end A;
```

It is assumed that resource allocators are known and that communication inside the action is restricted to the three P tasks (together with external calls to the resource allocators). These restrictions are checked at compile-time. All other tasks declare similar actions, and nested actions are allowed as long as strict nesting is observed. Note that if the tasks are not known at compile-time, then any communication with a task will be allowed only if both tasks are active in the same atomic action.

The imposed synchronization on the action is as follows. Tasks entering the action are not blocked. A task is blocked inside the action only if it has to wait for a resource to be allocated, or if it attempts to communicate with another task inside the action and that task is either active in the action, but not in a position to accept the communication, or is not as yet active in the action.

Tasks may leave the action only when all tasks active in the action wish to leave. This was not the case in the examples given earlier. There it was assumed that all tasks must enter the action before any could leave. Here it is possible for a subset of the named tasks to enter the action and subsequently leave (without recourse to any interactions with the missing tasks). This facility is deemed to be essential in a real-time system where deadlines are important. It solves the **deserter** problem where all tasks are held in an action because one task has not arrived. This issue will be considered along with error recovery in the next two subsections.

7.3.1 Atomic actions and backward error recovery

Atomic actions are important because they constrain the flow of information around the system to well-defined boundaries and therefore can provide the basis for both damage confinement and error recovery. In this section, backward error recovery between concurrent tasks is described.

In Chapter 2, it was shown that when backward error recovery is applied to groups of communicating tasks, it is possible for all the tasks to be rolled back to the start of their execution. This was the so-called *domino effect*. The problem occurred because there was no consistent set of recovery points or a recovery line. An atomic action provides that recovery line automatically. If an error occurs inside an atomic action then the tasks involved can be rolled back to the start of the action and alternative algorithms executed; the atomic action ensures that tasks have not passed any erroneous values through communication with tasks outside the action. When atomic actions are used in this way they are called **conversations** (Randell, 1975).

With conversations each action statement contains a recovery block. For example:

```
action A with (P2, P3) do
   ensure <acceptance test>
   by
      -- primary module
   else by
      -- alternative module
   else by
      -- alternative module
   else error
end A;
```

Other tasks involved in the conversation declare their part in the action similarly. The basic semantics of a conversation can be summarized as follows.

- On entry to the conversation, the state of a task is saved. The set of entry points forms the recovery line.
- While inside the conversation, a task is allowed only to communicate with other tasks active in the conversation and general resource managers. As conversations are built from atomic actions, this property is inherited.
- In order to leave the conversation, all tasks active in the conversation must have passed their acceptance test. If this is the case, then the conversation is finished and all recovery points are discarded.
- *If any task fails its acceptance test, all tasks have their state restored to that saved at the start of the conversation and they execute their alternative modules.* It is, therefore, assumed that any error recovery to be performed inside a conversation *must* be performed by *all* tasks taking part in the conversation.
- Conversations can be nested, but only strict nesting is allowed.
- If all alternatives in the conversation fail then recovery must be performed at a higher level.

It should be noted that in conversations, as defined by Randell (1975), all tasks taking part in the conversation must have entered the conversation before any of the other tasks can leave. This differs from the semantics described here. If a task does not enter into a conversation, either because of tardiness or because it has failed, then as long as the other tasks active in the conversation do not wish to communicate with it, the conversation can complete successfully. If a task does attempt to communicate with a missing task then either it can block and wait for the task to arrive or it can continue. Adopting this approach has two benefits (Gregory and Knight, 1985).

- It allows conversations to be specified where participation is not compulsory.
- It allows tasks with deadlines to leave the conversation, continue and if necessary take some alternative action.

Although conversations allow groups of tasks to coordinate their recovery, they have been criticized. One important point is that when a conversation fails, all the tasks are restored and all enter their alternative modules. This forces the same tasks to communicate again to achieve the desired effect; a task cannot break out of the conversation. This may be not what is required. Gregory and Knight (1985) point out that in practice when one task fails to achieve its goal in a primary module through communication with one group of tasks, it may wish to communicate with a completely new group of tasks in its secondary module. Furthermore, the acceptance test for this secondary module may be quite different. There is no way to express these requirements using conversations.

7.3.2 Atomic actions and forward error recovery

It was pointed out in Chapter 2 that, although backward error recovery enables recovery from unanticipated errors, it is difficult to undo any operation that may have been

performed in the environment in which the embedded system operates. Consequently forward error recovery and exception handling must be considered. In this section, exception handling between the concurrent tasks involved in an atomic action is discussed.

With backward error recovery, when an error occurs all tasks involved in the atomic action participate in recovery. The same is true with exception handling and forward error recovery. If an exception occurs in one of the tasks active in an atomic action then that exception is raised in *all* tasks active in the action. The exception is said to be **asynchronous** as it originates from another task. The following is a possible Ada-like syntax for an atomic action supporting exception handling:

```
action A with (P2, P3) do
   -- the action
exception
   when exception_a =>
        -- sequence of statements
   when exception_b =>
        -- sequence of statements
   when others =>
      raise atomic_action_failure;
end A;
```

With the termination model of exception handling, if all tasks active in the action have a handler and all handle the exception without raising any further exception, then the atomic action completes normally. If a resumption model is used, when the exception has been handled, the tasks active in the atomic action resume their execution at the point where the exception was raised.

With either model, if there is no exception handler *in any one of the tasks active in the action* or one of the handlers fails then *the atomic action fails* with a standard exception `atomic_action_failure`. This exception is raised in all the involved tasks.

There are two issues which must be considered when exception handling is added to atomic actions: resolution of concurrently raised exceptions and exceptions in nested actions (Campbell and Randell, 1986). These are now briefly reviewed.

Resolution of concurrently raised exceptions

It is possible for more than one task active in an atomic action to raise different exceptions at the same time. As Campbell and Randell (1986) point out, this event is likely if the errors resulting from some fault cannot be uniquely identified by the error-detection facility provided by each component of the atomic action. If two exceptions are simultaneously raised in an atomic action, there may be two separate exception handlers in each task. It may be difficult to decide which one should be chosen. Furthermore, the two exceptions in conjunction constitute a third exception which is the exception that indicates that both the other two exceptional conditions have occurred.

In order to resolve concurrently raised exceptions, Campbell and Randell propose the use of an **exception tree**. If several exceptions are raised concurrently then the exception used to identify the handler is that at the root of the smallest subtree that contains all the exceptions (although it is not clear how to combine any parameters

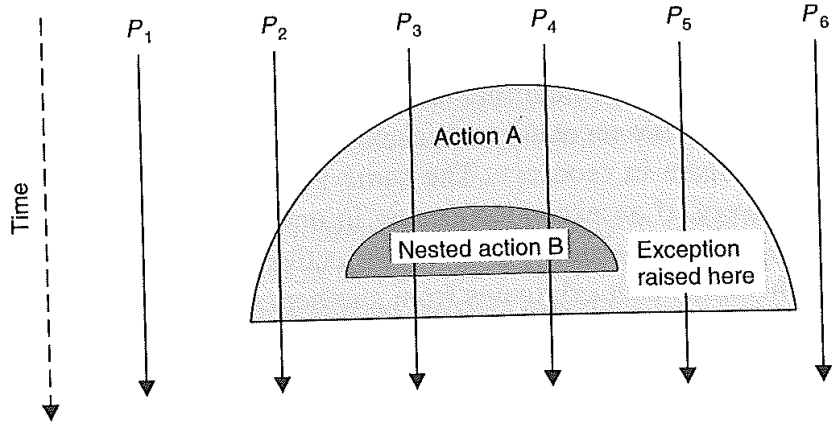

Figure 7.4 An exception in a nested atomic action.

associated with this exception). Each atomic action component can declare its own exception tree; the different tasks involved in an atomic action may well have different exception trees.

Exceptions and internal atomic actions

Where atomic actions are nested, it is possible for one task active in an action to raise an exception when other tasks in the same action are involved in a nested action. Figure 7.4 illustrates the problem.

When the exception is raised, all tasks involved must participate in the recovery action. Unfortunately, the internal action, by definition, is indivisible. To raise the exception in that action would potentially compromise that indivisibility. Furthermore, the internal action may have no knowledge of the possible exceptions that can be raised.

Campbell and Randell (1986) have discussed two possible solutions to this problem. The first solution is to hold back the raising of the exception until the internal action has finished. This they reject because:

- In a real-time system, the exception being raised may be associated with the missing of a deadline. To hold back the recovery procedure may seriously place in jeopardy the action's timely response.
- The error condition detected may indicate that the internal action may never terminate because some deadlock condition has arisen.

For these reasons, Campbell and Randell allow internal actions to have a predefined abortion exception. This exception is raised to indicate to the action that an exception has been raised in a surrounding action and that the preconditions under which the action was invoked are no longer valid. If such an exception is raised, the internal action should invoke fault-tolerant measures to abort itself. Once the action has been aborted, the containing action can handle the original exception.

If the internal action cannot abort itself then it must signal an atomic action failure exception. This then may be combined with the outstanding exception so as to affect the choice of recovery performed by the surrounding action. If no abortion exception is

defined, the surrounding action must wait for the internal action to complete. Alternatively, a default handler could be provided which would raise the atomic action failure exception.

7.4 Asynchronous notification

Although forward and backward error recovery have been discussed separately, in reality they may need to be combined in many real-time systems. Backward error recovery is needed to recover from unanticipated errors, and forward error recovery is needed to undo or ameliorate any interaction with the environment. Indeed, forward error handling can be used to implement a backward error recovery scheme – see Section 7.6.3.

As discussed in Section 7.2, none of the major real-time languages supports atomic actions, and it is necessary to use more primitive language facilities to achieve the same effect. The same is true for recoverable actions. One of the main requirements for a recoverable action is to be able to gain the attention of a task involved in an action and notify it that an error has occurred in another task. Most languages and operating systems support some form of asynchronous notification mechanism. As with exceptions, there are two basic models: resumption and termination.

The resumption model of asynchronous notification handling (often called **event handling**) behaves like a software interrupt. A task indicates which events it is willing to handle; when the event is signalled, the task is interrupted (unless it has temporally inhibited the event from being delivered) and an event handler is executed. The handler responds to the asynchronous event and then the task continues with its execution from the point at which it was interrupted. This, of course, sounds very similar to the resumption model of exception handling given in Section 3.2.4. The main difference is that the event is usually *not* signalled by the affected task (or because of an operation the affected task is performing), but is signalled *asynchronously*. However, many operating systems do not provide a special exception-handling facility for synchronous exception handling, but use the asynchronous events mechanisms instead. The C/Real-Time POSIX signal facility is an example of an asynchronous event model with resumption.

Note that, with the resumption model, the flow of control of a task is only temporarily changed; after the event has been handled the task is resumed. In a multithreaded process, it is possible to associate a distinct thread with the event and to schedule the thread when the event is signalled. Real-Time Java provides support for this model.

With the termination model of asynchronous notification, each task specifies a domain of execution during which it is prepared to receive an asynchronous notification that will cause the domain to be terminated. This form is often called **asynchronous transfer of control** or ATC. If an ATC request occurs outside this domain, it may be ignored or queued. After the ATC has been handled, control is returned to the interrupted task at a location different from that where the ATC was delivered. This, of course, is very similar to the termination model of exception handling. The Ada and Real-Time Java languages support asynchronous transfer of control mechanisms.

An extreme form of asynchronous notification with termination semantics is to abort the task and allow another task to perform some recovery. All operating systems and most concurrent programming languages provide such a facility. However, aborting

a process can be expensive and is often an extreme response to many error conditions. Aborting a thread is less expensive but still potentially dangerous as it can leave resources in an undefined state. Consequently, some form of safe asynchronous notification mechanism is also required.

The inclusion of an asynchronous notification mechanism into a language (or operating system) is controversial, as it complicates the language's semantics and increases the complexity of the run-time support system. This section thus first considers the application requirements which justify the inclusion of such a facility. The C/Real-Time POSIX, Ada and Real-Time Java models of asynchronous notification are then discussed.

7.4.1 The user need for asynchronous notification

The fundamental requirement for an asynchronous notification facility is to enable a task to respond *quickly* to a condition which has been detected by another task. The emphasis here is on a quick response; clearly a task can always respond to an event by simply polling or waiting for that event. The notification of the event could be mapped onto the task's communication and synchronization mechanism. The handling task, when it is ready to receive the event, simply issues the appropriate request.

Unfortunately, there are occasions when polling for events or waiting for the event to occur is inadequate. These include the following.

- **Error recovery** – this chapter has already emphasized that when groups of tasks undertake atomic actions, an error detected in one task requires all other tasks to participate in the recovery. For example, a hardware fault may mean that the task will never finish its planned execution because the preconditions under which it started no longer hold; the task may never reach its polling point. Also, a timing fault might have occurred, which means that the task will no longer meet the deadline for the delivery of its service. In both these situations, the task must be informed that an error has been detected and that it must undertake some error recovery as quickly as possible.
- **Mode changes** – a real-time system often has several modes of operation. For example, a fly-by-wire civil aircraft may have a take-off mode, a cruising mode and a landing mode. On many occasions, changes between modes can be carefully managed and will occur at well-defined points in the system's execution, as in a normal flight plan for a civil aircraft. Unfortunately, in some application areas, mode changes are expected but cannot be planned. For example, a fault may lead to an aircraft abandoning its take-off and entering an emergency mode of operation; or an accident in a manufacturing task may require an immediate mode change to ensure an orderly shutdown of the plant. In these situations, tasks must be quickly and safely informed that the mode in which they are operating has changed, and that they now need to undertake a different set of actions.
- **Scheduling using partial/imprecise computations** – there are many algorithms where the accuracy of the results depends on how much time can be allocated to their calculation. For example, numerical computations, statistical estimations and heuristic searches may all produce an initial estimation of the required result, and then refine that result to a greater accuracy. At run-time, a certain amount of time

can be allocated to an algorithm, and then, when that time has been used, the task must be interrupted to stop further refinement of the result.

- **User interrupts** – in a general interactive computing environment, users often wish to stop the current processing because they have detected an error condition and wish to start again.

7.5 Asynchronous notification in C/Real-Time POSIX

The C/Real-Time POSIX model supports two forms of asynchronous notification: a resumption facility based on signals, and a thread cancellation mechanism.

7.5.1 C/Real-Time POSIX signals

C/Real-Time POSIX signals support the resumption model. They are also used for a class of environment-detected synchronous errors (such as divide by zero, illegal pointer and so on).

There are a number of predefined signals, each of which is allocated an integer value. There are also an implementation-defined number of signals which are available for application use. Each signal has a default handler, which usually terminates the receiving process. Example signals are: `SIGABRT` for abnormal termination, `SIGALARM` for alarm clock expiry, `SIGILL` for illegal instruction exception, `SIGRTMIN` for the identifier of the first real-time application-definable exception, and `SIGRTMAX` for the identifier of the last real-time application-definable exception. Only those signals whose numbers lie between `SIGRTMIN` and `SIGRTMAX` are considered to be real-time by POSIX. A real-time signal is one which can have extra information passed to the handler by the process which generated it; in addition, they are queued.

The program can specify that it wants to receive signals as a result of certain events occurring; for example, when a message is received on a message queue (see the `mq_notify` function in Program 6.1) or a timer expires (see the `timer_create` function in Program 10.7). An application can configure its requirements for generating and receiving signals via the use of several data structures. These are shown in Program 7.1.

The `sigevent` structure is used to pass information to the signal-handling subsystem concerning the generation of a signal. For example, `timer_create` functions take an object of this type to indicate what should happen when the timer expires. The `sigev_signo` indicates the identity of the signal to be generated, and `sigev_value` indicates what data should be passed to the handler. The `sigev_notify` indicates how the program should be notified when the signal is generated. There are three possibilities.

- `SIGEV_NONE` – no notification occurs when the signal is generated.
- `SIGEV_SIGNAL` – a signal is queued with the application-defined data.
- `SIGEV_THREAD` – a new pthread should be created with a start routine (see `pthread_create` function in Program 4.5) defined by `sigev_notify_function` and thread attributes defined by `sigev_notify_attributes`.

Program 7.1 The C/Real-Time POSIX interface to data structures supporting signal generation.

```
/* used with message queue notification, timers etc */

struct sigevent {
  int sigev_notify;
    /* SIGEV_SIGNAL, */
    /* SIGEV_THREAD or SIGEV_NONE */

  int sigev_signo;  /* signal to be generated */

  union sigval sigev_value; /* value to be queued */

  void (*)sigev_notify_function(union sigval s);
  /* function to be treated as thread */
  pthread_attr_t *sigev_notify_attributes;
  /* thread attributes */
};

union sigval {
  int sival_int;
  void *sival_ptr;
};
```

If a signal is to be delivered to the process itself rather than via creating a thread, then further types and functions are available in C/Real-Time POSIX to specify the program's response. These are shown in Program 7.2. Most of the predefined C/Real-Time POSIX signals are delivered in this manner.

There are three ways in which a process can deal with a signal.

- It can **block** the signal and either handle it later or accept it.
- It can **handle** the signal by setting a function to be called whenever it occurs.
- It can **ignore** the signal altogether (in which case the signal is simply lost).

A signal that is not blocked and not ignored is **delivered** as soon as it is **generated**. A signal that is blocked is **pending** delivery, or may be **accepted** by calling one of the `sigwait()` functions.

Blocking a signal

C/Real-Time POSIX maintains the set of signals that have been currently masked by a process. The function `sigprocmask` is used to manipulate this set. The `how` parameter is set to: `SIG_BLOCK` to add signals to the set, `SIG_UNBLOCK` to subtract signals from the set, or `SIG_SETMASK` to replace the set.[1] The other two parameters contain pointers

[1] `SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK` are compile-time constants.

Program 7.2 An abridged C/Real-Time POSIX interface to signals.

```
typedef ... sigset_t;

/* the following manipulates the signal mask */
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
  /* how = SIG_BLOCK -> the set is added to the current set */
  /* how = SIG_UNBLOCK -> the set is subtracted from the */
  /*       current set */
  /* how = SIG_SETMASK -> the given set becomes the mask */

/* the following routines allow a signal */
/* set to be created and manipulated */
int sigemptyset(sigset_t *s); /* initialize a set to empty */
int sigfillset(sigset_t *s); /* initialize a set to full */
int sigaddset(sigset_t *s, int signum); /* add a signal */
int sigdelset(sigset_t *s, int signum); /* remove a signal */
int sigismember(const sigset_t *s, int signum);
   /* returns 1 if a member */

/* the following support signal handling */

typedef struct { /* signal parameters */
  int si_signo;
  int si_code;
  union sigval si_value;
} siginfo_t;

struct sigaction {
  void (*sa_handler) (int signum); /* non real-time handler */
  void (*sa_sigaction) (int signum, siginfo_t *data,
                        void *extra); /*real-time handler */
  sigset_t sa_mask; /* signals to mask during handler */
  int sa_flags; /*indicates if signal is to be queued */
};

int sigaction(int sig, const struct sigaction *reaction,
              struct sigaction *old_reaction);
/* sets up a signal handler, reaction, for sig */

/* the following functions allow a  */
/* process to wait for a signal */

int sigsuspend(const sigset_t *sigmask);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);

/* the following functions allow a  */
/* signal to be sent */
int kill (pid_t pid, int sig);
   /* send the signal sig to the process pid */
int sigqueue(pid_t pid, int sig, const union sigval value);
   /* send signal and data */

/* All the above functions return -1 when errors have occurred. */
/* A shared variable errno contains the reason for the error */
```

to the set of signals to be added/subtracted/replaced (`set`) and the returned value of the old set (`oset`). Various functions (`sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset` and `sigismember`) allow a set of signals to be manipulated.

When a signal is blocked, it remains pending until it is unblocked or accepted. When it is unblocked, it is then delivered. Note, some signals *cannot* be blocked.

Handling a signal

A signal handler can be set up using the function `sigaction`. The `sig` parameter indicates which signal is to be handled, `reaction` is a pointer to a structure containing information about the handler, and `old_reaction` points to information about the previous handler. Essentially, the information about a handler contains a pointer to the handler function (`sa_handler` if the signal is a non-real-time signal, or `sa_sigaction` if the signal is a real-time one), the set of signals to be masked during the handler's execution (`sa_mask`), and whether the signal is to be queued (indicated by setting `sa_flags` to the symbolic constant `SA_SIGINFO` – only signals whose value lies between `SIGRTMIN` and `SIGRTMAX` can be queued). The `sa_handler` member indicates the action associated with the signal and can be:

- `SIG_DFL` – default action (usually to terminate the process);
- `SIG_IGN` – ignore the signal;
- pointer to a function – to be called when the signal is delivered.

For non-real-time signals, only an integer parameter can be passed to the handler when the signal is generated. The value of this parameter normally indicates the signal itself (the same handler can be used for more than one signal). However, for the real-time signals, more data can be passed via a pointer to the `siginfo_t` structure. This structure contains the signal number (again), a code which indicates the cause of the signal (for example, a timer signal) and an integer or pointer value.

If more than one *real-time* signal is queued, the one with the lowest value is delivered first (that is, `SIGRTMIN` is delivered before `SIGRTMIN + 1` and so on).

A process can also wait for a signal to arrive using the functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait`. The function `sigsuspend` replaces the mask with that given by the parameter to the call and suspends the process until:

(1) a non-blocked signal is delivered, and

(2) the associated handler is executed.

If the handler terminates the process, the `sigsuspend` function never returns; otherwise it returns with the signal mask reset to the state that existed *prior* to the call of `sigsuspend`.

The `sigwaitinfo` function also suspends the calling process until the signal arrives. However, this time the signal must be blocked, and thus the handler is not called. Instead, the function returns the selected signal number and stores the information about the delivered signal in the `info` argument. The function `sigtimedwait` has the same

semantics as `sigwaitinfo`, but allows a timeout to be specified for the suspension. If no signals are delivered by the timeout, `sigwaitinfo` returns with –1 and `errno` set to `EAGAIN`.

Care must clearly be taken when using signals for condition synchronization. There is a potential race condition between checking to see whether a signal has already arrived and issuing a request which causes suspension. The appropriate protocol is to block the signal first, then test to see if it has occurred and, if not, suspend and unblock the signal using one of the above functions.

Ignoring a signal

A signal can be ignored by simply setting the value of `sa_handler` to `SIG_IGN` in a call to the function `sigaction`.

Generating a signal

There are two ways in which a process can generate a signal to be sent to another process. The first is via the `kill` function and the second is via the `sigqueue` function. The latter can only send real-time signals.

Note, however, that a process can also request that a signal be sent to itself: when a timer expires (for example, `SIGALRM` – see Section 13.2.3), when asynchronous I/O completes, by the arrival of a message on an empty message queue (see Section 6.7), or by using the C raise statement.

A simple example of C/Real-Time POSIX signals and processes

As an illustration of C/Real-Time POSIX signals, consider the program fragment below. A process performs some computation periodically. The actual computation to be performed depends on a system-wide mode of operation. A mode change is propagated to all processes via an application-defined real-time signal `MODE_CHANGE`. The signal handler `change_mode` simply changes a global variable `mode`. The processes access `mode` at the beginning of each iteration. To ensure that the mode does not change while it is accessing it, the `MODE_CHANGE` signal is blocked.

```
#include <signal.h>

#define MODE_A 1
#define MODE_B 2
#define MODE_CHANGE SIGRTMIN + 1

int mode = MODE_A;

void change_mode(int signum, siginfo_t *data, void *extra) {
  /* signal handler */
  mode = data -> si_value.sival_int;
}
```

```
int main() {

  sigset_t mask, omask;
  struct sigaction s, os;
  int local_mode;

  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, MODE_CHANGE);

  s.sa_flags = SA_SIGINFO;
  s.sa_mask = mask;
  s.sa_sigaction = &change_mode;
  s.sa_handler = &change_mode;

  SIGACTION(MODE_CHANGE, &s, &os); /* assign handler */

  while(1) {

    SIGPROCMASK(SIG_BLOCK, &mask, &omask);
      local_mode = mode;
    SIGPROCMASK(SIG_UNBLOCK, &mask, &omask);

    /* periodic operation using mode*/
    switch(local_mode) {
      case MODE_A:
        ...
        break;
      case MODE_B:
        ...
        break;
      default:
        ...
    }
  }
  ...
}
```

Signals and threads

The original C/POSIX signal model came from Unix and was extended to make it more appropriate for real-time when the Real-Time Extensions to POSIX were specified. With the POSIX Thread Extensions, the model has become more complex and is a compromise between a per process model and a per thread model. The following points should be noted.

- Signals which are generated as a result of a synchronous error condition, such as memory violation, are delivered only to the thread that caused the signal.
- Other signals may be sent to the process as a whole; however, each one is only delivered to a single thread in the process.
- The `sigaction` function sets the handler for *all* threads in the process.

- The functions `kill` and `sigqueue` still apply to processes. A new function `pthread_kill`

  ```
  int pthread_kill(pthread_t thread, int sig);
  ```

  allows a process to send a signal to an individual thread.
- If more than one thread is eligible to have a signal delivered to it, it is not defined which thread is chosen.
- If the action specified by a handler for the signal is for termination, *the whole process is terminated*, not just the thread.
- Signals can be blocked on a per thread basis using a function `pthread_sigmask` which has the same set of parameters as `sigprocmask`. The use of the function `sigprocmask` is not specified for a multithreaded process.
- The functions `sigsuspend`, `sigwaitinfo` or `sigtimedwait` operate on the calling thread not the calling process.
- A new function `sigwait`

  ```
  int sigwait(const sigset_t *set, int *sig);
  ```

  allows a thread to wait for one of several blocked signals to occur. It behaves the same as `sigwaitinfo()` except that the information associated with the signal is not returned. The signals are specified in the location referenced `set`. The function returns zero when a successful wait has been performed and the location referenced by `sig` contains the received signal.

  If one of the signals is already pending when the function is called, the function returns immediately. If more than one is pending, it is not defined which one is chosen unless only real-time signals are pending. In this case, the one with the lowest value is chosen.
- If a signal action is set by a thread to 'ignore', it is unspecified whether the signal is discarded immediately it is generated or remains pending.

Although C/Real-Time POSIX allows a thread or a process to handle an asynchronous event, care must be taken because some of the POSIX system calls are termed **async-signal unsafe** and **async-cancel unsafe**. It is undefined what happens if a signal interrupts an async-unsafe function that was called from a signal-catching function. For example, it is not safe to use the function `pthread_cond_signal` in a signal handler because of the race condition it introduces with the function `pthread_cond_wait`.

7.5.2 Asynchronous transfer of control and thread cancellation

C/Real-Time POSIX supports the C `setjmp` and `longjmp` mechanisms. As discussed in Section 3.3.3 these can be used to implement a termination model of exception handling. There are also versions of these that can be used in conjunction with signals (`sigsetjmp` and `siglongjmp`). These as well as saving the thread's state, save and restore the signal mask respectively. Hence it is conceivable that an ATC could be

Program 7.3 The C/Real-Time POSIX interface to thread cancellation.

```
#define PTHREAD_CANCEL_ASYNCHRONOUS ...
#define PTHREAD_CANCEL_ENABLE ...
#define PTHREAD_CANCEL_DEFERRED ...
#define PTHREAD_CANCEL_DISABLE ...
#define PTHREAD_CANCELED ...

int   pthread_cancel(pthread_t thread);
void  pthread_cleanup_push(void (* routine)(void *), void * arg);
void  pthread_cleanup_pop(int execute);


int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

implemented using a combination of these mechanisms. Unfortunately, these non-local jump functions and all the thread system calls are async-signal unsafe. C/Real-Time POSIX does not define what happens when an async-signal unsafe function is interrupted by a signal handler and the handler calls an async-signal unsafe function.

Threads can also be asynchronously terminated using the C/Real-Time POSIX thread cancellation facilities. However, a thread can control if and when it is cancelled via its cancellation state and cancellation type. The state can be **enabled** or `disabled`. If disabled, any attempt to cancel the thread is held pending. If enabled, cancellation will occur at point determined by the cancellation type. If the type is **asynchronous**, the cancellation may occur at any time. If `deferred`, the cancellation will occur at the next cancellation point that the thread reaches in its execution. The defaults for a newly created thread are enabled and deferred.

The full set of cancellation points is defined in the C/Real-Time POSIX standard. However, it is worth noting that most of the C/Real-Time POSIX pthread functions are *not* cancellation points.

In support of safe cancellation, C/Real-Time POSIX also allows a thread to define one or more routines that will be executed when cancellation is acted upon. These routines are stored on a stack, and the application can push and pop them during its execution. On cancellation (or when `pthread_exit` is called), they are popped off (in LIFO order) and executed.

Program 7.3 gives the full C/Real-Time POSIX API that supports cancellation and cleanup routines. Note that a thread cancels another thread by calling the function `pthread_cancel`, and a thread can introduce a cancellation point into its exection by calling `pthread_testcancel`.

7.5.3 C/Real-Time POSIX and atomic actions

Given the close interaction between activities in an atomic action, it is more appropriate to consider the action to take place between C/Real-Time POSIX threads rather than

C/Real-Time POSIX processes. Atomic actions are easiest to implement if a termination model of asynchronous transfer of control is used.

Given the problems of using a combination of `setjmp/longjmp` and signals, it is best to use thread creation and cancellation to program the required recovery. As C/Real-Time POSIX threads are designed to be efficient, this approach does not have the same performance penalty that would be associated with a more heavyweight process structure.

7.6 Asynchronous notification in Ada

The asynchronous notification facilities in Ada allow an application to respond to:

- events being signalled asynchronous from the external environment – this is in support of interrupt handling and will be considered in detail in Section 14.3;
- events being triggered by the passage of time – the handling for these events is executed at the priority of the clock device and they are considered in detail in Section 10.4.1;
- asynchronous transfer of control (ATC) requests on a task – supporting a termination model;
- task abortion.

There is no generalized mechanisms for a resumption model of asynchronous notification, hence this section will focus on ATC and task abortion.

7.6.1 Asynchronous transfer of control

Ada provides a structured form of asynchronous notification handling called **asynchronous transfer of control** (ATC). To emphasize that ATC is a form of communication and synchronization, the mechanism is built on top of the inter-task communication facility.

The Ada `select` statement was introduced in Chapter 6. It has the following forms:

- a selective accept (to support the server side of the rendezvous) – this was discussed in Section 6.5;
- a timed and a conditional entry call (to either a task or a protected entry) – this is discussed in Section 9.4.2;
- an asynchronous select – discussed here.

The asynchronous select statement provides an asynchronous notification mechanism with termination semantics.

The execution of the asynchronous select begins with the issuing of the triggering entry call or the issuing of the triggering delay. If the triggering statement is an entry call, the parameters are evaluated as normal and the call issued. If the call is queued, then a sequence of statements in an abortable part is executed.

If the triggering statement completes before the execution of the abortable part completes, the abortable part is aborted. When these activities have finished, the optional sequence of statements following the triggering statement is executed.

If the abortable part completes before the completion of the entry call, an attempt is made to cancel the entry call and, if successful, the execution of the asynchronous select statement is finished. The following illustrates the syntax:

```
select
  Trigger.Event;
  -- optional sequence of statements to be
  -- executed after the event has been received
then abort
  -- abortable sequence of statements
end select;
```

Note that the triggering statement can be a delay statement and, therefore, a timeout can be associated with the abortable part (see Section 9.4.3).

If the cancellation of the triggering event fails because the protected action or rendezvous has started, then the asynchronous select statement waits for the triggering statement to complete before executing the optional sequence of statements following the triggering statement.

Clearly, it is possible for the triggering event to occur even before the abortable part has started its execution. In this case the abortable part is not executed and therefore not aborted.

Consider the following example:

```
task Server is
    entry Atc_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
  ...
  accept Atc_Event do
    Seq2;
  end Atc_Event;
  ...
end Server;

task body To_Be_Interrupted is
begin
  ...
  select  -- ATC statement
    Server.Atc_Event;
    Seq3;
  then abort
    Seq1;
  end select;
  Seq4;
  ...
end To_Be_Interrupted;
```

When the above ATC statement is executed, the statements which are executed will depend on the order of events that occur:

```
if the rendezvous is available immediately then
   Server.Atc_Event is issued
   Seq2 is executed
   Seq3 is executed
   Seq4 is executed (Seq1 is never started)
elsif no rendezvous starts before  Seq1 finishes then
   Server.Atc_Event is issued
   Seq1 is executed
   Server.Atc_Event is cancelled
   Seq4 is executed
elsif the rendezvous finishes before Seq1 finishes then
   Server.Atc_Event is issued
   partial execution of Seq1 occurs concurrently with Seq2
   Seq1 is aborted and finalized
   Seq3 is executed
   Seq4 is executed
else (the rendezvous finishes after Seq1 finishes)
   Server.Atc_Event is issued
   Seq1 is executed concurrently with partial execution of Seq2
   Server.Atc_Event cancellation is attempted but is unsuccessful
   execution of Seq2 completes
   Seq3 is executed
   Seq4 is executed
end if
```

Note that there is a race condition between `Seq1` finishing and the rendezvous finishing. The situation could occur where `Seq1` does finish but is nevertheless aborted.

Ada allows some operations to be **abort deferred**. If `Seq1` contains an abort-deferred operation, then its cancellation will not occur until the operation is completed. An example of such an operation is a call on a protected object.

The above discussion has concentrated on the concurrent behaviour of `Seq1` and the triggering rendezvous. Indeed, on a multiprocessor implementation it could be the case that `Seq1` and `Seq2` are executing in parallel. However, on a single-processor system, the triggering event will only ever occur if the action that causes it has a higher priority than `Seq1`. The normal behaviour will thus be the preemption of `Seq1` by `Seq2`. When `Seq2` (the triggering rendezvous) completes, `Seq1` will be aborted before it can execute again. And hence the ATC is 'immediate' (unless an abort-deferred operation is in progress).

Exceptions and ATC

With the asynchronous select statement, potentially two activities occur concurrently: the abortable part may execute concurrently with the triggering action (when the action is an entry call). In either one of these activities, exceptions may be raised and unhandled. Therefore, at first sight it may appear that potentially two exceptions can be propagated simultaneously from the select statement. However, this is not the case: one of the exceptions is deemed to be lost (that raised in the abortable part when it is aborted), and hence only one exception is propagated.

7.6.2 Task abortion

Tasks in Ada can be aborted using an abort statement; any task may abort any other named task by executing this statement

Once aborted tasks are said to become *abnormal*, and are prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately. However, some implementations may not be able to facilitate immediate shut down, and hence all Ada requires is that the task terminate before it next interacts with other tasks. Note that the Real-Time Systems Annex does require 'immediate' to be just that on a single processor system.

After a task has been marked as abnormal, execution of its body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort-deferred operation*. The execution of an abort-deferred operation is allowed to complete before it is aborted. The same rules for aborting a task body also apply to aborting a sequence of statements in the asynchronous select statement discussed earlier.

If a construct, which has been aborted, is blocked outside an abort-deferred operation (other than at an entry call), the construct becomes abnormal and is immediately completed. Other constructs must complete no later than the next *abort completion point* (if any) that occurs outside an abort-deferred operation.

An abort completion point occurs at:

- the end of activation of a task;
- the point where the execution initiates the activation of another task;
- the start or end of an entry call, accept statement, delay statement or abort statement;
- the start of the execution of a select statement, or of the sequence of statements of an exception handler.

The following operations are defined to be abort-deferred:

- a protected action;
- waiting for an entry call to complete;
- waiting for termination of dependent tasks;
- the execution of an 'initialize' procedure, a 'finalize' procedure, or an assignment operation of an object with a controlled part.

In Ada, the equivalent of C/Real-Time POSIX's cleanup handlers can be programmed using `Controlled` types. Objects of these types must have their finalization routines executed when the task is aborted.

7.6.3 Ada and atomic actions

It was shown in Section 3.4 that backward error recovery in a sequential system could be implemented by exception handling. In this section, the Ada ATC facility and exception handling is used to implement backward and forward error recovery. It is assumed that the underlying Ada implementation and run-time are fault free, and therefore the strong typing provided by Ada will ensure that the Ada program itself remains viable.

Backward error recovery

The following package is a generic version of the one was given in Section 3.4 for saving and restoring a task's state.

```
generic
 type Data is private;
package Recovery_Cache is
  procedure Save(D : in Data);
  procedure Restore(D : out Data);
end Recovery_Cache;
```

Consider three Ada tasks which wish to enter into a recoverable atomic action. Each will call their appropriate procedure in the package given below:

```
package Conversation is

  procedure T1(Params : Param); -- called by task 1
  procedure T2(Params : Param); -- called by task 2
  procedure T3(Params : Param); -- called by task 3

  Atomic_Action_Failure : exception;

end Conversation;
```

The body of the package encapsulates the action and ensures that only communication between the three tasks is allowed during the conversation.[2] The `Controller` protected object is responsible for propagating any error condition noticed in one task to all tasks, saving and restoring any persistent data in the recovery cache, and ensuring that all tasks leave the action at the same time. It contains three protected entries and a protected procedure.

- The `Wait_Abort` entry represents the asynchronous event on which the tasks will wait while performing their part of the action.
- Each task calls `Done` if it has finished its component of the action without error. Only when all three tasks have called `Done` will they be allowed to leave.
- Similarly, each task calls `Cleanup` if it has had to perform any recovery.
- If any task recognizes an error condition (either because of a raised exception or the failure of the acceptance test), it will call `Signal_Abort`. This will set the flag `Killed` to `True`, indicating that the tasks must be recovered.

Note that, as backward error recovery will be performed, the tasks are not concerned with the actual cause of the error. When `Killed` becomes `True`, all tasks in the action receive the asynchronous event. Once the event has been handled, all tasks must wait on the `Cleanup` entry so that they can all terminate the conversation module together.

[2] In practice, this might be difficult to ensure because of Ada's scope rules. One way of increasing the security would be to require that the `Conversation` package is at the library level and its body only references pure (state-free) packages. The solution presented here assumes that the tasks are well behaved. It also assumes, for simplicity, that the correct tasks call `T1`, `T2` and `T3` at the correct times.

```
with Recovery_Cache;
package body Conversation is

  Primary_Failure, Secondary_Failure,
         Tertiary_Failure: exception;
  type Module is (Primary, Secondary, Tertiary);

  protected Controller is
    entry Wait_Abort;
    entry Done;
    entry Cleanup;
    procedure Signal_Abort;
  private
    Killed : Boolean := False;
    Releasing_Done : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Informed : Integer := 0;
  end Controller;

  -- any local protected objects for communication between actions

  protected body Controller is
    entry Wait_Abort when Killed is
    begin
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;

    procedure Signal_Abort is
    begin
      Killed := True;
    end Signal_Abort;

    entry Done when Done'Count = 3 or Releasing_Done is
    begin
      if Done'Count > 0 then
        Releasing_Done := True;
      else
        Releasing_Done := False;
      end if;
    end Done;

    entry Cleanup when Cleanup'Count = 3 or Releasing_Cleanup is
    begin
      if Cleanup'Count > 0 then
        Releasing_Cleanup := True;
      else
        Releasing_Cleanup := False;
      end if;
    end Cleanup;
  end Controller;
```

```
  procedure T1(Params : Param) is separate;
  procedure T2(Params : Param) is separate;
  procedure T3(Params : Param) is separate;

end Conversation;
```

The code for each task is contained within a single procedure: e.g. `T1`. Within such a procedure, three attempts are made to perform the action. If all attempts fail, the exception `Atomic_Action_Failure` is raised. Each attempt is surrounded by a call that saves the state and restores the state (if the attempt fails). Each attempt is encapsulated in a separate local procedure (`T1_Primary`, etc.), which contains a single 'select and then abort' statement to perform the required protocol with the controller. The recovery cache is used by each task to save its local data.

```
separate(Conversation)
procedure T1(Params : Param) is
  procedure T1_Primary is
  begin
    select
      Controller.Wait_Abort; -- triggering event
      Controller.Cleanup; -- wait for all to finish
      raise Primary_Failure;
    then abort
      begin
        -- code to implement atomic action,
        -- the acceptance test might raise an exception
        if Accept_Test = Failed then
          Controller.Signal_Abort;
        else
          Controller.Done; -- signal completion
        end if;
      exception
        when others =>
          Controller.Signal_Abort;
      end;
    end select;
  end T1_Primary;

  procedure T1_Secondary is ... ;
  procedure T1_Tertiary is ... ;

  package My_Cache is new Recovery_Cache(..); -- for local data

begin
  My_Cache.Save(..);
  for Try in Module loop
    begin
      case Try is
        when Primary => T1_Primary; return;
        when Secondary => T1_Secondary; return;
        when Tertiary => T1_Tertiary;
      end case;
```

```
    exception
      when Primary_Failure =>
        My_Cache.Restore(..);
      when Secondary_Failure =>
        My_Cache.Restore(..);
      when Tertiary_Failure =>
        My_Cache.Restore(..);
        raise Atomic_Action_Failure;
      when others =>
        My_Cache.Restore(..);
        raise Atomic_Action_Failure;
    end;
  end loop;
end T1;

-- similarly for T2 and T3
```

Figure 7.5 illustrates a simple state transition diagram for a participating task in a conversation.

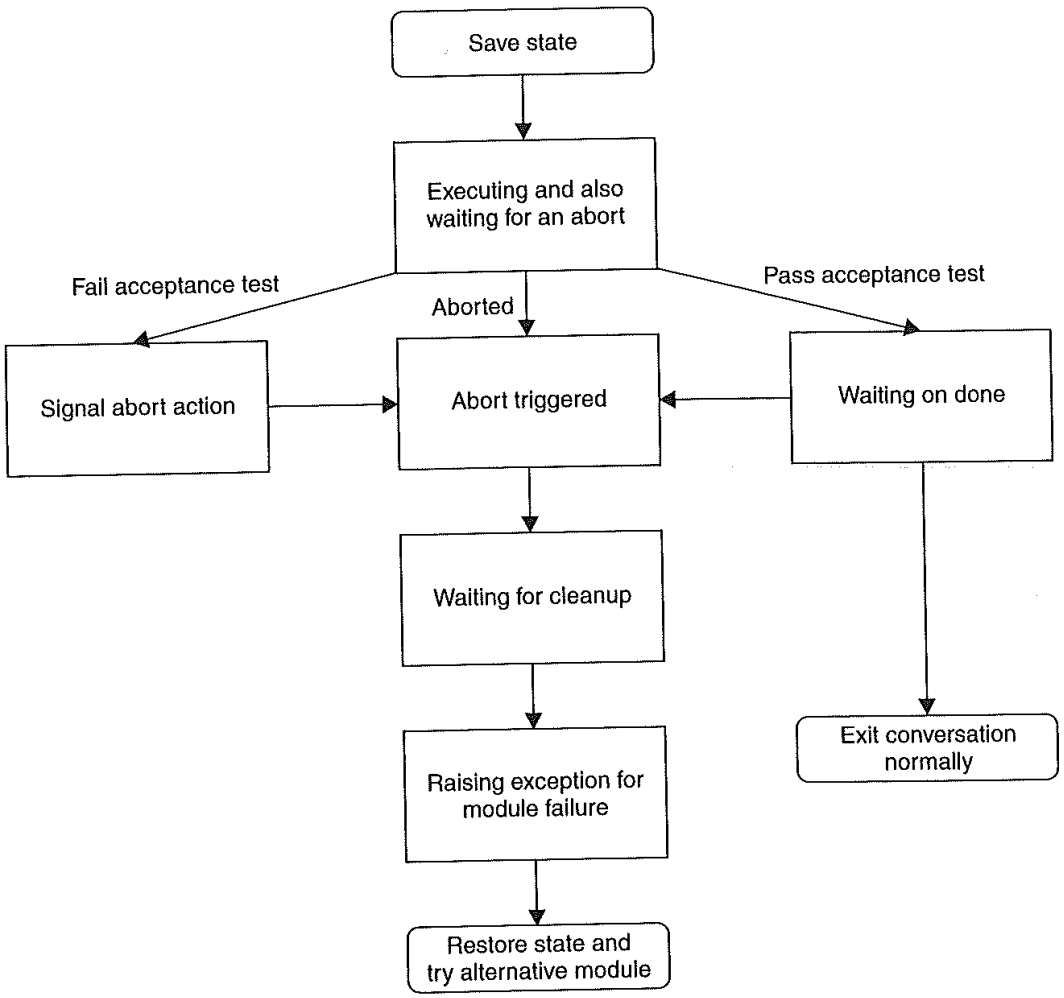

Figure 7.5 Simple state transition diagram for a conversation.

Forward error recovery

Ada's ATC facility can be used with exceptions to implement atomic actions with forward error recovery between concurrently executing tasks. Consider again the following package for implementing an atomic action between three tasks:

```
package Action is
  procedure T1(Params : Param); -- called by task 1
  procedure T2(Params : Param); -- called by task 2
  procedure T3(Params : Param); -- called by task 3

  Atomic_Action_Failure : exception;
end Action;
```

As with backward error recovery, the body of the package encapsulates the action and ensures that only communications between the three tasks are allowed. The `Controller` protected object is responsible for propagating any exception raised in one task to all tasks, and for ensuring that all tasks leave the action at the same time.

```
with Ada.Exceptions;
use Ada.Exceptions;
package body Action is
  type Vote_T is (Commit, Aborted);
  protected Controller is
    entry Wait_Abort(E: out Exception_Id);
    entry Done;
    procedure Cleanup (Vote: Vote_T);
    entry Wait_Cleanup(Result : out Vote_T);
    procedure Signal_Abort(E: Exception_Id);
  private
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id;
    Final_Result : Vote_T := Commit;
    Informed : Integer := 0;
  end Controller;

  -- any local protected objects for communication between actions

  protected body Controller is
    entry Wait_Abort(E: out Exception_Id) when Killed is
    begin
      E := Reason;
      Informed := Informed + 1;
      if Informed = 3 then
        Killed := False;
        Informed := 0;
      end if;
    end Wait_Abort;

    entry Done when Done'Count = 3 or Releasing_Done is
    begin
```

```
    if Done'Count > 0 then
      Releasing_Done := True;
    else
      Releasing_Done := False;
    end if;
  end Done;

  procedure Cleanup (Vote: Vote_T) is
  begin
    if Vote = Aborted then
      Final_Result := Aborted;
    end if;
  end Cleanup;

  procedure Signal_Abort(E: Exception_Id) is
  begin
    Killed := True;
    Reason := E;
  end Signal_Abort;

  entry Wait_Cleanup (Result: out Vote_T)
        when Wait_Cleanup'Count = 3 or Releasing_Cleanup is
  begin
    Result := Final_Result;
    if Wait_Cleanup'Count > 0 then
      Releasing_Cleanup := True;
    else
      Releasing_Cleanup := False;
      Final_Result := Commit;
    end if;
  end Wait_Cleanup;
end Controller;

procedure T1(Params: Param) is
  X : Exception_Id;
  Decision : Vote_T;
begin
  select
    Controller.Wait_Abort(X); -- triggering event
    Raise_Exception(X); -- raise common exception
  then abort
    begin
      -- code to implement atomic action
      Controller.Done; --signal completion
    exception
      when E: others =>
        Controller.Signal_Abort(Exception_Identity(E));
    end;
  end select;
exception
    -- if any exception is raised during the action
    -- all tasks must participate in the recovery
    when E: others =>
      -- Exception_Identity(E) has been raised in all tasks
```

```
        -- handle exception
        if Handled_Ok then
          Controller.Cleanup(Commit);
        else
          Controller.Cleanup(Aborted);
        end if;
        Controller.Wait_Cleanup(Decision);
        if Decision = Aborted then
          raise Atomic_Action_Failure;
        end if;
    end T1;

    procedure T2(Params : Param) is ...;

    procedure T3(Params : Param) is ...;
end Action;
```

Each component of the action (`T1`, `T2` and `T3`) has identical structure. The component executes a select statement with an abortable part. The triggering event is signalled by the `Controller` protected object if any component indicates that an exception has been raised and not handled locally in one of the components. The abortable part contains the actual code of the component. If this code executes without incident, the `Controller` is informed that this component is ready to commit the action. If any exceptions are raised during the abortable part, the `Controller` is informed and the identity of the exception passed. Note that, unlike backward error recovery (given earlier), here the cause of the error must be communicated.

If the `Controller` has received notification of an unhandled exception, it releases all tasks waiting on the `Wait_Abort` triggering event (any task late in arriving will receive the event immediately it tries to enter into its select statement). The tasks have their abortable parts aborted (if started), and the exception is raised in each task by the statement after the entry call to the controller. If the exception is successfully handled by the component, the task indicates that it is prepared to commit the action. If not, then it indicates that the action must be aborted. If any task indicates that the action is to be aborted, then all tasks will raise the exception `Atomic_Action_Failure`. Figure 7.6 shows the approach using a simple state transition diagram.

The above example illustrates that it is possible to program atomic actions with forward error recovery in Ada. However, there are two points to note about this example.

- Only the first exception to be passed to the `Controller` will be raised in all tasks. It is not possible to get concurrent raising of exceptions, as any exception raised in an abortable part when it is aborted is lost.
- The approach does not deal with the deserter problem. If one of the participants in the action does not arrive, the others are left waiting at the end of the action. To cope with this situation, it is necessary for each task to log its arrival with the action controller (see Exercise 7.7).

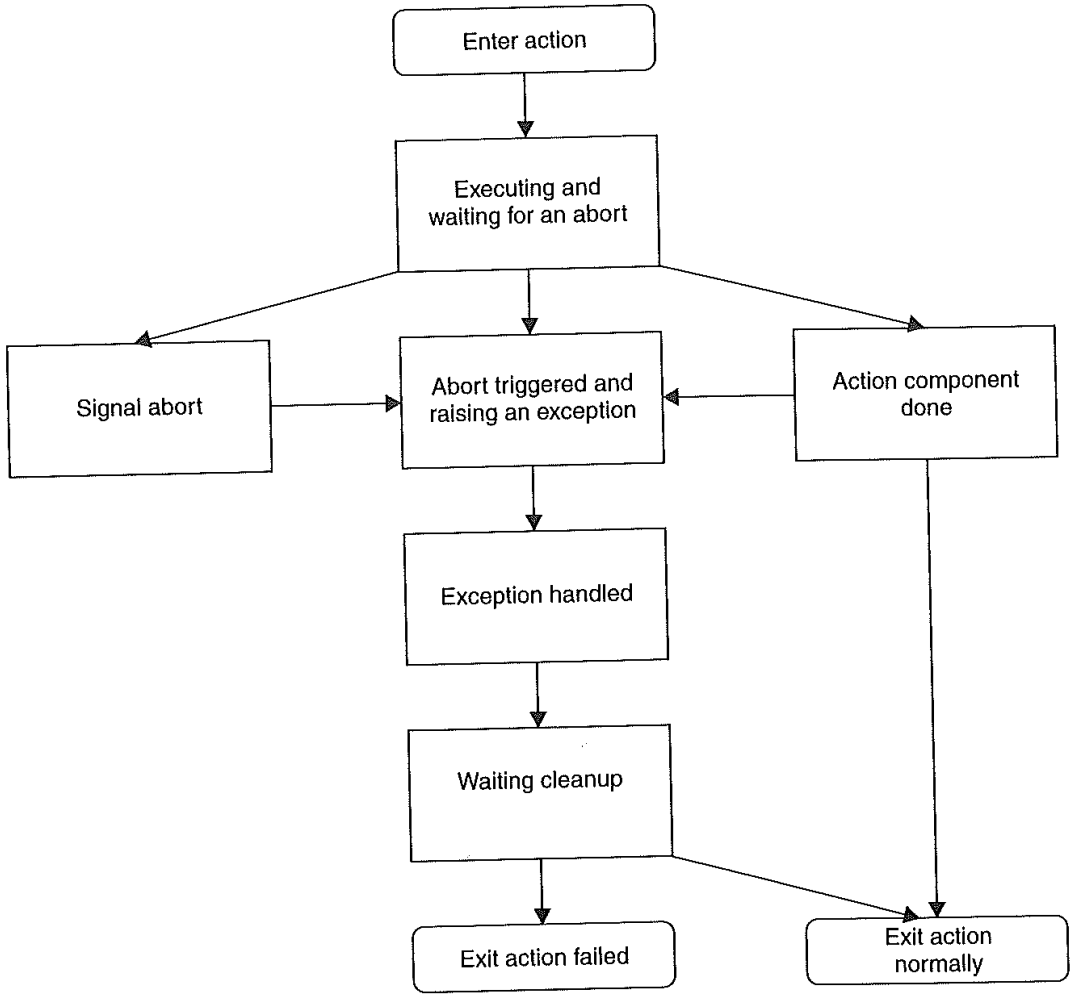

Figure 7.6 Simple state transition diagram illustrating forward error recovery.

7.7 Asynchronous notification in Real-Time Java

Real-Time Java supports both the resumption and termination models of asynchronous notification, but provides no direct support for asynchronous thread termination. The resumption model is provided by an asynchronous event-handling facility, where the handlers are scheduled entities rather than executed in the context of an interrupted thread. The termination model is provided within the context of an asynchronous exception-handling model for real-time threads. The latter can be used to program safe asynchronous termination. Note that facilities discussed apply only to Java systems that support the Real-Time Specification for Java; they cannot be used in a normal Java environment.

7.7.1 Asynchronous event handling

The equivalent of a C/Real-Time POSIX signal in Real-Time Java is an asynchronous event. Indeed, there is even a class `POSIXSignalHandler` which allows POSIX signals to be mapped onto Real-Time Java events for the occasion when Real-Time Java is being implemented on top of a POSIX-compliant operating system (see Section 14.4.2).

Program 7.4 shows the three main classes associated with asynchronous events in Real-Time Java. Each `AsyncEvent` can have one or more `AsyncEventHandlers`.

Program 7.4 The `AsyncEvent`, `AsyncEventHandler` and `BoundAsyncEventHandler` classes.

```
public class AsyncEvent {
  public AsyncEvent();

  public void addHandler(AsyncEventHandler handler);
  public void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);
  // Associate a new handler with this event,
  // removing all existing handlers.

  public void bindTo(java.lang.String happening);
  // bind to external event
  public void unbindTo(java.lang.String happening);

  public ReleaseParameters createReleaseParameters();
  // creates a ReleaseParameters object representing the
  // characteristics of this event

  public void fire();
  // Schedule for execution the set of handlers for this event.
  public boolean handledBy(AsyncEventHandler handler);
  // Returns true if this event is handled by this handler.
}

public abstract class AsyncEventHandler implements Schedulable {
  public AsyncEventHandler();
  // parameters are inherited from the current thread

  public AsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group);
  ... // other constructors available

  // methods which implement the Schedulable interface,
  // see Chapter 12

  protected int getAndClearPendingFireCount();
  // Atomically set to zero the number of pending executions
  // of this handler and returns the value from before
  // it was cleared.
  protected int getAndDecrementPendingFireCount();
  protected int getAndIncrementPendingFireCount();

  public void handleAsyncEvent();
  // Override this method to define the action to be
  // taken by this handler

  public final void run();
}

public abstract class BoundAsyncEventHandler extends AsyncEventHandler {
  public BoundAsyncEventHandler();
  // other constructors
}
```

When the event occurs (indicated by a call to the `fire` method), all the handlers associated with the event are scheduled for execution according to their `SchedulingParameters` – see Section 12.7. Note that the firing of an event can also be associated with the occurrence of an implementation-dependent external action by using the `bindTo` method.

Each handler is scheduled once for each outstanding event firing. Note, however, that the handler can modify the number of outstanding events by the methods in the `AsyncEventHandler` class.

Although an event handler is a schedulable entity, the goal is that it will not suffer the same overhead as an application thread. Consequently, it cannot be assumed that there is a separate implementation thread for each handler, as more than one handler may be associated with a particular implementation thread. If a dedicated thread is required, the `BoundAsyncEventHandler` should be used.

7.7.2 Asynchronous transfer of control in Real-Time Java

Early versions of Java allowed one thread to asynchronously affect another thread; however, this support has now been deprecated. Standard Java now only supports the following:

```
public void interrupt() throws SecurityException;
public boolean isInterrupted();
```

One thread can signal an interrupt to another thread by calling the `interrupt` method. The result of this depends on the current status of the interrupted thread.

- If the interrupted thread is blocked in the `wait`, `sleep` or `join` methods, it is made runnable and the `InterruptedException` is thrown.
- If the interrupted thread is executing, a flag is set indicating that an interrupt is outstanding. *There is no immediate effect on the interrupted thread.* Instead, the called thread must periodically test to see if it has been 'interrupted' using the `isInterrupted` method.

By itself, this does not meet the user needs outlined in Section 7.4.1.

Real-Time Java provides an alternative approach for interrupting a thread, based on asynchronous transfer of control (ATC). The Real-Time Java ATC model is similar to the Ada one in that it is necessary to indicate which regions of code can receive the ATC request. However, the Real-Time Java model is different in two important respects.

(1) The Real-Time Java model is integrated with the Java exception-handling facility, whereas the Ada model is integrated into the select statement and entry-handling mechanisms.

(2) The Real-Time Java model requires each method to indicate that it is prepared to allow the ATC to occur. ATC is deferred until the thread is executing within such a method. In contrast, Ada's default is to allow the ATC if a subprogram has been called from within the select-then-abort statement; a deferred response must be explicitly handled.

Both languages defer the ATC during interaction with other threads/tasks (for example synchronized statements/methods in Java and protected actions and rendezvous in Ada) or within constructors and finalization (finally) clauses.

The Real-Time Java ATC model brings together the Java exception-handling model and an extension of thread interruption. Essentially the model is that when a real-time thread (or more generally a schedulable object – see Section 12.7) is interrupted, an asynchronous exception (`AsynchronouslyInterruptedException`) is delivered to the real-time thread rather than the thread having to poll for the interruption as would be the case with conventional Java. `AsynchronouslyInterruptedException` is a checked exception.

The notion of an asynchronous exception is not new and has been explored in previous languages. The main problem with them is how to program safely in their presence. As discussed in Chapter 3, most exception-handling mechanisms have exception propagation within a termination model. Consider a thread that has called method A, which has called method B, which has called method C. When an exception is raised within method C, if there is no local handler, the call to method C is terminated and a handler is sought in method B (the exception propagates up the call chain). If no handler is found in B, the exception is propagated to A. When a handler is found, it is executed, and the program continues to execute in the context in which the handler was found. There is no return to the context where the original exception was thrown. This model makes it difficult to write code that is tolerant of an asynchronous exception being thrown at it. Every method would need a handler for the root class of all asynchronous exceptions.

The Real-Time Java solution to this problem is to require that all methods (including constructors) that are prepared to allow the delivery of an asynchronous exception, place the exception in their `throws` lists; Real-Time Java calls such methods **AI-methods** (Asynchronously Interruptible). If a method does not do this, then the asynchronous exception is not delivered but held pending until the real-time thread is in a method that has the asynchronous exception in its throw clause. Hence, code that has been written without being concerned with ATC can execute safely even in an environment where ATCs are being used. Furthermore, to ensure that ATCs can be handled safely, Real-Time Java requires that:

(1) ATCs are deferred during the execution of synchronized methods or statements and static initializers. This is to ensure that any shared data is left in a consistent state; Real-Time Java calls these sections of code and the methods that are not AI methods collectively **ATC-deferred** sections.

(2) An ATC can only be handled from within code that is an ATC-deferred section; this is to avoid the handler for one ATC being interrupted by another ATC being delivered.

The full model is best explained in two stages. The first is the low-level support and overall approach; the second is the use of the high-level support to provide a structured means for handling ATC. Use of the basic ATC facilities requires three activities:

(1) declaring an `AsynchronouslyInterruptedException` (AIE);

(2) identifying methods which can be interrupted;

(3) signalling an `AsynchronouslyInterruptedException` to a thread.

Program 7.5 The Real-Time Java `AsynchronouslyInterruptedException` class.

```
public class AsynchronouslyInterruptedException extends
             java.lang.InterruptedException
{
  public  AsynchronouslyInterruptedException();


  public boolean clear();
  public boolean disable();
  // only valid within a doInterruptible,
  // returns true if successful
  public boolean doInterruptible (Interruptible logic);
  // Only one specific Interruptible can be running per thread
  // at any one time.
  // Returns True, if the Interruptible is executed, false if one
  // is already in progress for this thread.

  public boolean enable();
  public boolean fire();

  public static AsynchronouslyInterruptedException getGeneric();
  // returns the AsynchronouslyInterruptedException which
  // is generated when RealtimeThread.interrupt() is invoked

  public boolean isEnabled();
}
```

Program 7.5 shows the specification of the `AsynchronouslyInterrupted-Exception` class. The methods will be explained in due course, but for now all that is required is to know that for each thread there is an associated generic AIE.

An AIE may be placed in the `throws` list associated with a method. For example, consider the following class which provides an interruptible service using a package which declares non-interruptible services (that is, ones which do not have `throws` lists containing `AsynchronouslyInterruptedExceptions`).

```
import nonInterruptibleServices.*;

public class InterruptibleService {
  public AsynchronouslyInterruptedException stopNow =
         AsynchronouslyInterruptedException.getGeneric();

  public boolean service()
        throws AsynchronouslyInterruptedException {
    //code interspersed with calls to NonInterruptibleServices
  }
}
```

Now assume that a real-time thread `rtThread` has called an instance of this class to provide the `service`:

```
public InterruptibleService IS = new InterruptibleService();

// code of thread, rtThread
if(IS.service()) { ... } else { ... };
```

and that another real-time thread interrupts `rtThread`:

```
rtThread.interrupt();
```

The consequences of this call depend on the current state of `rtThread` when the call is made.

- If `rtThread` is executing within an ATC-deferred section – that is, executing within a synchronized method (or block), a static initializer or within a method that has no `AsynchronouslyInterruptedException` declared in its `throws` list (such as those in the package `nonInterruptibleServices`) – the AIE is marked as pending. The exception is delivered as soon as `rtThread` leaves the ATC-deferred region and is executing in a method with an `AsynchronouslyInterruptedException` declared in its `throws` list (such as the `service` method).
- If `rtThread` is executing within an AI-method (and it is not within a synchronized block), then the method's execution is interrupted, and control is transferred (propagated) up the call chain until it finds a try block in an ATC-deferred region that has a catch clause naming `AsynchronouslyInterruptedException` (or a parent class). Any synchronized methods or statements that are terminated by this propagation have their monitor locks released (and their `finally` clauses executed). The handler is then executed.
- If `rtThread` is blocked inside a `sleep`, `join`, `MemoryArea.join` (or `joinAndEnter`) or `waitForNextPeriodInterruptible` method called from within an AI-method, `rtThread` is rescheduled and the `AsynchronouslyInterruptedException` is delivered.
- If `rtThread` is blocked inside a `wait`, `sleep`, `join`, `MemoryArea.join` (or `joinAndEnter`) or `waitForNextPeriodInterruptible` method called from within an ATC-deferred region, `rtThread` is rescheduled and the `AsynchronouslyInterruptedException` is thrown as a synchronous exception (it is a subclass of the `InterruptedException`) and it is also marked as pending. Even if the synchronous exception is handled, the asynchronous exception is redelivered as soon as `rtThread` enters an AI-method.

Once an ATC has been delivered and control is passed to an appropriate exception handler, it is necessary to ascertain whether the caught ATC is the one expected by the interrupted thread. If it is, the exception can be handled. The `clear` method defined in the class `AsynchronouslyInterruptedException` is used for this purpose. The `clear` method simply resets the pending state if the AIE is current and returns

`true`, indicating that the current AIE is now no longer pending. If the flag is not reset, the AIE will then be redelivered when control next enters an AI-method.

```
import NonInterruptibleServices.*;
public class InterruptibleService {
  public AsynchronouslyInterruptedException stopNow =
          new AsynchronouslyInterruptedException();

  public boolean Service()
       throws AsynchronouslyInterruptedException {
    try {
      // code interdispersed with calls to
      // NonInterruptibleServices
    }
    catch AsynchronouslyInterruptedException AIE {
      if(stopNow.clear()) {
        // handle the ATC
       } else {
        // Cleanup and leave the current AIE still pending.
       }
     }
   }
}
```

Here, when the AIE is thrown, control is passed to the catch clause at the end of the try block. A handler is found for `AsynchronouslyInterruptedExceptions`. In order to determine whether the current `AsynchronouslyInterruptedException` is `stopNow`, a call is made to the `clear` method. This returns `true` if `stopNow` is the current exception. If it is not the current exception then, the real-time thread does some cleanup routines before exiting the handler. In the latter case, the AIE is still pending.

The `Interruptible` interface

The above discussion illustrates the basic mechanisms provided by Real-Time Java for handling ATCs. To facilitate their structured use, the language also provides an interface called `Interruptible` – see Program 7.6.

An object which wishes to provide an interruptible method does so by implementing the `Interruptible` interface. The `run` method is the method that is interruptible; the `interruptAction` method is called by the system if the `run` method is interrupted.

Program 7.6 The Real-Time Java `Interruptible` interface.

```
public interface Interruptible {
  public void interruptAction (
          AsynchronouslyInterruptedException exception);

  public void run (AsynchronouslyInterruptedException exception)
       throws AsynchronouslyInterruptedException;
}
```

Once this interface is implemented, the implementation can be passed as a parameter to the `doInterruptible` method in the `AsynchronouslyInterruptedException` class. The method can then be interrupted by calling the `fire` method in the `AsynchronouslyInterruptedException` class. Further control over `AsynchronouslyInterruptedException` is given by the `disable`, `enable` and `isEnabled` methods. A disabled `AsynchronouslyInterruptedException` is deferred until it is enabled. An example of this latter use will be given in Section 7.7.3.

Note that only one `doInterruptible` method for a particular `AsynchronouslyInterruptedException` can be active at one time. If a call is outstanding, the method returns immediately with a `false` value.

Multiple `AsynchronouslyInterruptedExceptions`

Given that `AsynchronouslyInterruptedException` can be deferred, it is possible for multiple ATCs to be deferred. This can happen when the `run` method of one class (which implements the `Interruptible` interface) calls a `doInterruptible` on an AIE. The associated `run` method may also call another `doInterruptible`. Hence it is possible for a thread to be executing nested `doInterruptibles`. Consider the following example:

```
import javax.realtime.*;

public class NestedATC {
  AsynchronouslyInterruptedException AIE1 = new
        AsynchronouslyInterruptedException();
  AsynchronouslyInterruptedException AIE2 = new
        AsynchronouslyInterruptedException();
  AsynchronouslyInterruptedException AIE3 = new
        AsynchronouslyInterruptedException();

 public void method1() {
   // ATC-deferred region
 }

 public void method2() throws AsynchronouslyInterruptedException {
    AIE1.doInterruptible
       (new Interruptible()
         {
            public void run(AsynchronouslyInterruptedException e)
                 throws AsynchronouslyInterruptedException
            {
              method1();
            }
            public void interruptAction(
                 AsynchronouslyInterruptedException e)
            {
               if(AIE1.clear()) {
                 // recovery here
               } else {
                // cleanup
              }
```

```
            }
          }
        );
    }

    public void method3() throws AsynchronouslyInterruptedException {
       AIE2.doInterruptible
          (new Interruptible()
            {
              public void run(AsynchronouslyInterruptedException e)
                    throws AsynchronouslyInterruptedException {
                method2();
              }
              public void interruptAction(
                    AsynchronouslyInterruptedException e){
                 if(AIE2.clear()) {
                   // recovery here
                 } else {
                  // cleanup
                }
              }
            }
          );
    }

    public void method4() throws AsynchronouslyInterruptedException {
       AIE3.doInterruptible
          (new Interruptible()
            {
              public void run(AsynchronouslyInterruptedException e)
                    throws AsynchronouslyInterruptedException {
                method3();
              }
              public void interruptAction(
                    AsynchronouslyInterruptedException e) {
                 if(AIE3.clear()) {
                   // recovery here
                 } else {
                  // cleanup
                }
              }
            }
          );
      }

    }
```

Now suppose that a thread `t` has created an instance of `NestedATC` and has called `method4`, which has called `method3`, which has called `method2`, which has called `method1` which is an ATC-deferred region. Assume that the thread is interrupted by a call to `AIE2.fire()`. This is held pending. If `AIE3` now comes in then `AIE2` is discarded as `AIE3` is at a higher level in the nesting. If `AIE1` comes in then `AIE1` is discarded (because it is at a lower level). Once `method1` has returned, the currently pending AIE is thrown.

7.7.3 Real-Time Java and atomic actions

This section illustrates how Real-Time Java's ATC facilities can be used to implement atomic actions with forward error recovery.

First, an `AtomicActionException` is defined along with an `AtomicActionFailure` exception.

```
import javax.realtime.AsynchronouslyInterruptedException;

public class AtomicActionException extends
          AsynchronouslyInterruptedException {
  public static Exception cause;
  public static boolean wasInterrupted;
}

public class AtomicActionFailure extends Exception;
```

Using a `ThreeWayRecoverableAtomicAction` similar to that defined earlier:

```
public interface ThreeWayRecoverableAtomicAction {
  public void role1() throws AtomicActionFailure;
  public void role2() throws AtomicActionFailure;
  public void role3() throws AtomicActionFailure;
}
```

a `RecoverableAction` class can be implemented with a similar structure to that given for Ada.

```
import javax.realtime.*;

public class RecoverableAction
          implements ThreeWayRecoverableAtomicAction {
  protected RecoverableController Control;
  private final boolean abort = false;
  private final boolean commit = true;

  private AtomicActionException aae1, aae2, aae3;

  public RecoverableAction()  // constructor {
    Control = new RecoverableController();
      // for recovery
      aae1 = new AtomicActionException();
      aae2 = new AtomicActionException();
      aae3 = new AtomicActionException();
  }

  class RecoverableController {
    protected boolean firstHere, secondHere, thirdHere;
    protected int allDone;
    protected int toExit, needed;
    protected int numberOfParticipants;
    private boolean committed = commit;
```

```
RecoverableController() {
  // for synchronization
  firstHere = false;
  secondHere = false;
  thirdHere = false;
  allDone = 0;
  numberOfParticipants = 3;
  toExit = numberOfParticipants;
  needed = numberOfParticipants;
}

synchronized void first() throws InterruptedException {
  while(firstHere) wait();
  firstHere = true;
}

synchronized void second() throws InterruptedException {
  while(secondHere) wait();
  secondHere = true;
}

synchronized void third() throws InterruptedException {
  while(thirdHere) wait();
  thirdHere = true;
}

synchronized void signalAbort(Exception e) {
 allDone = 0;
 AtomicActionException.cause = e;
 AtomicActionException.wasInterrupted = true;
 // raise an AsynchronouslyInterruptedException
 // in all participants
 aae1.fire();
 aae2.fire();
 aae3.fire();
}

private void reset() {
  firstHere = false;
  secondHere = false;
  thirdHere = false;
  allDone = 0;
  toExit = numberOfParticipants;
  needed = numberOfParticipants;
  notifyAll();
}

synchronized void done() throws InterruptedException {
  allDone++;
  if(allDone == needed) {
    notifyAll();
  } else while(allDone != needed) {
      wait();
      if(AtomicActionException.wasInterrupted) {
        allDone--;
        return;
      }
```

```
      }
      toExit--;
       if(toExit == 0) {
         reset();
       }
    }

    synchronized void cleanup(boolean abort) {
      if(abort) { committed = false; };
    }

    synchronized boolean waitCleanup()
             throws InterruptedException {
      allDone++;
      if(allDone == needed) {
        notifyAll();
      } else while (allDone != needed) {
          wait();
      }
      toExit--;
      if(toExit == 0) {
        reset();
      }
      return committed;
    }
}

public void role1() throws AtomicActionFailure,
                    AsynchronouslyInterruptedException {
    boolean Ok;
    // entry protocol
    // no AIE until inside the atomic action
    boolean done = false;
    while(!done) {
      try {
        Control.first();
        done = true;
      } catch(InterruptedException e) {
        // ignore
      }
    }

    // the following defines an interruptible
    // section of code, and a routine to be called
    // if the code is interrupted
    Ok = aae1.doInterruptible
        (new Interruptible()
          {
            public void run(AsynchronouslyInterruptedException e)
                  throws AsynchronouslyInterruptedException {
              try {
               // perform action
               // if necessary call e.disable() and e.enable() to
               // defer AIE
               Control.done();
              }
              catch(Exception x) {
```

```
                Control.signalAbort(x);
              }
            }

            public void interruptAction(
                  AsynchronouslyInterruptedException e) {
              // no action required
            }
          }
        );
    if(!Ok) throw new AtomicActionFailure();

    if(aae1.wasInterrupted) {
      try {
        // try to recover
        Control.cleanup(commit);
        if(Control.waitCleanup() != commit) {
           throw new AtomicActionFailure();
        };
      }
      catch(Exception x) {
        throw new AtomicActionFailure();
      }
    };
  }

  public void role2() throws AtomicActionFailure,
                AsynchronouslyInterruptedException
  {/* similar to role1 */};

  public void role3() throws AtomicActionFailure,
                AsynchronouslyInterruptedException
  {/* similar to role1 */};

}
```

Note that the absence of direct language support for asynchronous transfer of control makes the Java solution appear more complicated than its Ada counterpart. This is mainly due to the requirement to construct `Runnable` objects to pass to the `doInterruptible` method.

Summary

Reliable execution of tasks is essential if real-time embedded systems are to be used in critical applications. When tasks interact, it is necessary to constrain their intertask communication so that recovery procedures can be programmed, if required. Atomic actions have been discussed in this chapter as a mechanism by which programs, consisting of many tasks, can be structured to facilitate damage confinement and error recovery.

Actions are atomic if they can be considered, so far as other tasks are concerned, to be indivisible and instantaneous, such that the effects on the

system are as if they are interleaved as opposed to concurrent. An atomic action has well-defined boundaries and can be nested. Resources used in an atomic action are allocated during an initial *growing phase*, and released either as part of a subsequent *shrinking phase* or at the end of the action (if the action is to be recoverable).

The syntax of an atomic action can be expressed by an action statement. The following statement executed within task P_1 indicates that P_1 wishes to enter into an atomic action with P_2 and P_3:

```
action A with (P2,P3) do
    -- sequence of statements
end A;
```

P_2 and P_3 must execute similar statements.

A *conversation* is an atomic action with backward error recovery facilities (in the form of recovery blocks).

```
action A with (P2,P3) do
   ensure <acceptance test>
   by
      -- primary module
   else by
      -- alternative module
   else error
end A;
```

On entry to the conversation, the state of the task is saved. While inside the conversation, a task is only allowed to communicate with other tasks active in the conversation and general resource managers. In order to leave the conversation, all tasks active in the conversation must have passed their acceptance test. If any task fails its acceptance test, all tasks have their state restored to that saved at the start of the conversation and they execute their alternative modules. Conversations can be nested and if all alternatives in an inner conversation fail then recovery must be performed at an outer level.

Forward error recovery via exception handlers can also be added to atomic actions. If an exception is raised by one task then all tasks active in the action must handle the exception.

```
action A with (P2,P3) do
   -- the action
exception
   when exception_a =>
       -- sequence of statements
   when others =>
      raise atomic_action_failure;
end A;
```

Two issues that must be addressed when using this approach are the resolution of concurrently raised exceptions and exceptions in internal actions.

Few mainstream languages or operating systems directly support the notion of an atomic action or a recoverable atomic action. However, most communication and synchronization primitives allow the isolation property of an action to be programmed. To implement a recoverable action requires an asynchronous notification mechanism. This can either have resumption semantics (in which case it is called an asynchronous event-handling mechanism) or it can have termination semantics (in which case it is called asynchronous transfer of control). C/Real-Time POSIX supports asynchronous events using signals and a thread-cancelling mechanism. A signal can be handled, blocked or ignored. Real-Time Java also supports asynchronous events.

Both Ada and Real-Time Java provide the termination model of asynchronous transfer of control. The Ada mechanism is built on top of the select statement. Real-Time Java's ATC, in contrast, is integrated into its exception and thread interrupt mechanisms. These termination approaches, in combination with exceptions, allow for an elegant implementation of a recoverable action.

Further reading

Anderson, T. and Lee, P. A. (1990) *Fault Tolerance Principles and Practice*, 2nd ed. London: Prentice Hall.

Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-Time Programming in Ada*. Cambridge: Cambridge University Press.

Lea, D. (2000) *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. Harlow: Prentice Hall.

Lynch, N. A. (ed.) (1993) *Atomic Transactions*, Morgan Kaufmann Series in Data Management Systems. San Mateo, California: Morgan Kaufmann.

Northcutt, J. D. (1987) *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel*. Orlando: Academic Press.

Shrivastava, S. K., Mancini, L. and Randell, B. (1987) *On the Duality of Fault Tolerant Structures*, Lecture Notes in Computer Science, Volume 309, pp. 19–37. Springer-Verlag.

Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

7.1 Distinguish between an atomic *action* and an atomic *transaction*. What is the relationship between an atomic transaction and a conversation?

7.2 Rewrite the `Action_X` Ada package given in Section 7.6.3 so that it becomes a general-purpose package for controlling a three-task conversation. (Hint: use generics.)

7.3 Can your solution to Exercise 7.2 be extended to cope with an arbitrary number of tasks participating in the atomic action?

7.4 What would be the implications of extending Ada to enable one task to raise an exception in another?

7.5 Compare and contrast asynchronous notification and exception handling.

7.6 In Section 7.6.3, backward and forward error recovery is shown between three Ada tasks. Show how they can be combined into a single solution that gives both forward and backward error recovery between the same three tasks.

7.7 Update the solution given in Section 7.6.1 to deal with the deserter problem.

7.8 Consider the following four fragments of code:

```
-- Fragment 1
select
    T.Call; -- an entry call to task T
    Flag := A;
or
    delay 10.0;
    Flag := B;
    -- code taking 2 seconds to execute
end select;

-- Fragment 2
select
    T.Call; -- an entry call to task T
    Flag := A;
else
    delay 10.0;
    Flag := B;
    -- code taking 2 seconds to execute
end select;

-- Fragment 3
select
    T.Call; -- an entry call to task T
    Flag := A;
then abort
    delay 10.0;
    Flag := B;
    -- code taking 2 seconds to execute
end select;

-- Fragment 4
select
    delay 10.0;
    Flag := A;
then abort
    T.Call; -- an entry call to task T
    Flag := B;
    -- code taking 2 seconds to execute
end select;
```

A rendezvous with `T.Call` takes 5 seconds to execute. What is the value of the `Flag` variable after the execution of each of the four fragments in each of the following cases? You may assume a `Flag` assignment statement takes zero execution time.

(1) `T.Call` is available when the select is executed.
(2) `T.Call` is NOT available when the select is executed and does not become available in the next 14 seconds.
(3) `T.Call` is NOT available when the select is executed, but does become available after 2 seconds.
(4) `T.Call` is NOT available when the select is executed but does become available after 8 seconds.

7.9 Consider the following package specification which provides a procedure to search part of a large character array for a *unique* fixed-length string. The procedure returns the position of the start of the string if it is found.

```
package Search_Support is
  type Array_Bounds is range 1 .. 1_000_000_000;
  type Large_Array is array(Array_Bounds) of Character;
  type Pointer is access Large_Array;

  type Search_String is new String(1..10);

  procedure Search(Pt: Pointer;
                   Lower, Upper: Array_Bounds;
                   Looking_For : Search_String;
                   Found : out Boolean;
                   At_Location : out Array_Bounds);
end Search_Support;
```

Three tasks wish to perform a concurrent search of the array for the same string; they are derived from a common task type.

```
task type Searcher(Search_Array: Pointer;
                   Lower, Upper: Array_Bounds) is
  entry Find(Looking_For : Search_String);
  entry Get_Result(At_Location : out Array_Bounds);
end Searcher;
```

The string to be found is passed via an initial rendezvous with the tasks.
Sketch the body of the task type (and any other objects you might need) so that when one task finds the string, all other tasks are *immediately* informed of the string's location so that further fruitless search is avoided. Assume that the `SearchString` will be found by one of the three tasks. Furthermore, all tasks must be prepared to pass back the result via the `GetResult` entry.

7.10 Consider the following Ada code fragment:

```
Error_1, Error_2 : exception;
task Watch;
task Signaller;

protected Atc is
  entry Go;
  procedure Signal;
private
```

```
  Flag : Boolean := False;
end Atc;

protected body Atc is
  entry Go when Flag is
  begin
    raise Error_1;
  end Go;

  procedure Signal is
  begin
    Flag := True;
  end Signal;
end Atc;

task body Watch is
begin
  ...
  select
    Atc.Go;
  then abort
    -- code taking 100 millisecond
    raise Error_2;
  end select;
  ...
exception
  when Error_1 =>
    Put_Line("Error_1 Caught");
  when Error_2 =>
    Put_Line("Error_2 Caught");
  when others =>
    Put_Line("Other Errors Caught");
end Watch;


task body Signaller is
begin
  ...
  Atc.Signal;
  ...
end Signaller;
```

Describe carefully the possible executions of this program fragment assuming that context switches between tasks can happen at any time.

7.11 A particular POSIX-based application consists of several periodic processes and has two modes of operation: MODE A and MODE B. The application has one process which only operates in MODE A. Sketch the design of this process assuming that when the system wishes to undertake a mode change, it sends a signal to all processes indicating the current mode of operation. Assume also the existence of a routine called `WAITNEXTPERIOD`, which will suspend the process until its next period of execution is due. Note that a mode change should only affect the process at the beginning of each period.

7.12 Illustrate how Ada's OOP model can be used to produce extensible atomic actions.

7.13 Compare and contrast the Ada and Java models for asynchronous transfer of control.

7.14 To what extent can standard Java be used to implement atomic actions?

7.15 Why have the Java routines `resume()`, `stop()` and `suspend()` been made obsolete?

7.16 Redo Exercise 7.11 for Real-Time Java.

7.17 Show how Ada's termination model of exception handling can be implemented in response to the receipt of a POSIX signal.