

# *Atomic Actions, Concurrent Processes and Reliability*



## Goal

To understand how concurrent processes can reliably cooperate in the presence of errors

# Topics



- Atomic actions
- Backward error recovery
- Forward error recovery
- Asynchronous notifications
- POSIX signals
- Asynchronous Event Handling in RTJ
- Asynchronous Transfer of Control (ATC) in Ada
- ATC in Java

# *Atomic Actions — Motivation*

- Concurrent processes enable parallelism in the real world to be reflected in programs
- The interaction between 2 processes has been expressed in terms of a single communication; this is not always the case
- E.g., withdrawal from a bank account may involve a ledger process and a payment process in a sequence of communications to authenticate the drawer, check the balance and pay the money
- It may be necessary for more than two processes to interact in this way to perform the required action
- The processes involved must see a consistent system state
- With concurrent processes, it is all too easy for groups of processes to interfere with one other
- What is required is for each group of processes to execute their joint activity as an indivisible or atomic action

# *Atomic Actions — Definition*

An action is atomic if the processes performing it:

- are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the action
- do not communicate with other processes while the action is being performed
- can detect no state change except those performed by themselves and if they do not reveal their state changes until the action is complete
- can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent

# *Nested Actions*



- 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 processes 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 process.
- The outer level action would then no longer be indivisible

# *Atomic Transactions*



- In 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 fail
- By failure, it is meant that an error has occurred from which the transaction cannot recover — normally a processor failure
- With an atomic transaction, all components used are returned to their original state (that is the state they were before the transaction commenced)
- Atomic transactions are sometimes called recoverable actions or atomic actions

# *Properties of Atomic Transactions*

- **Failure** atomicity — the transaction must complete successfully or have no effect
- **Synchronization** atomicity (or **isolation**) — partial execution cannot be observed by any concurrently executing transaction
- Not entirely suitable for programming fault-tolerant systems 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
- Atomic transactions provide a form of backward error recovery but 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 system database

# *Requirements for Atomic Actions*

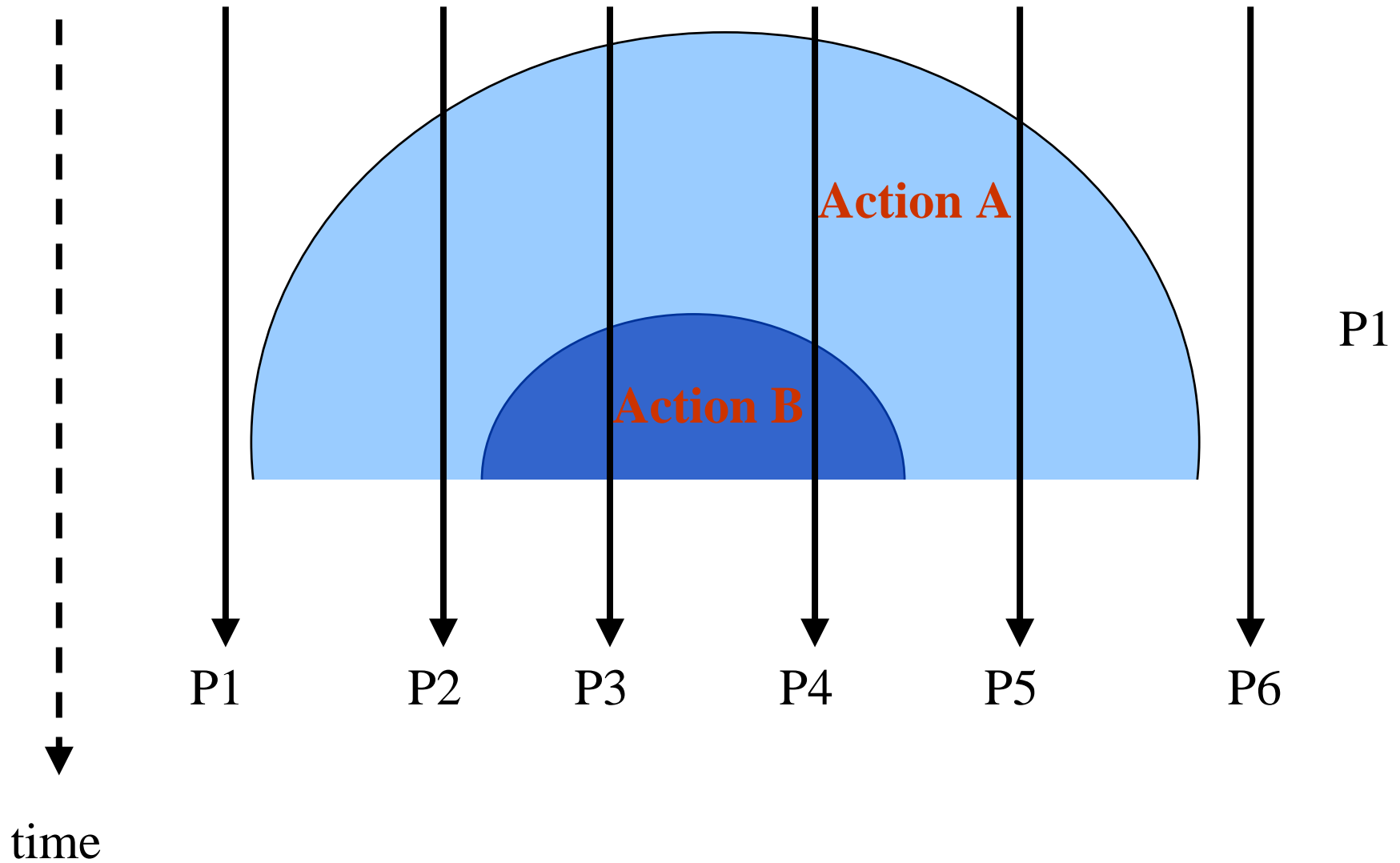
- Well-defined boundaries
  - A start, end and a side boundary
  - The start and end boundaries are the locations in each process where the action is deemed to start and end
  - The side boundary separates those processes involved in the action from those in the rest of the system
- Indivisibility
  - No exchange of information between processes active inside and those outside (resource managers excluded).
  - The value of any shared data after the actions is determined by the strict sequencing of the actions in some order
  - There is no synchronization at the start. Processes can enter at different times
  - Processes are not allowed to leave the atomic action until all are willing and able to leave



# *Requirements for Atomic Actions*

- Nesting
  - Atomic actions may be nested as long as they do not overlap with other atomic actions
  - 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
  - Sequential execution could impair the performance of the overall system and 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 serialising their executions
- They must allow recovery procedures to be programmed

# *Nested Atomic Actions*



# Language Structure

```
action A with (P2, P3, . . .) do
```

```
. . .
```

```
-- can only communication with p2, P3 etc
```

```
-- and use local variables
```

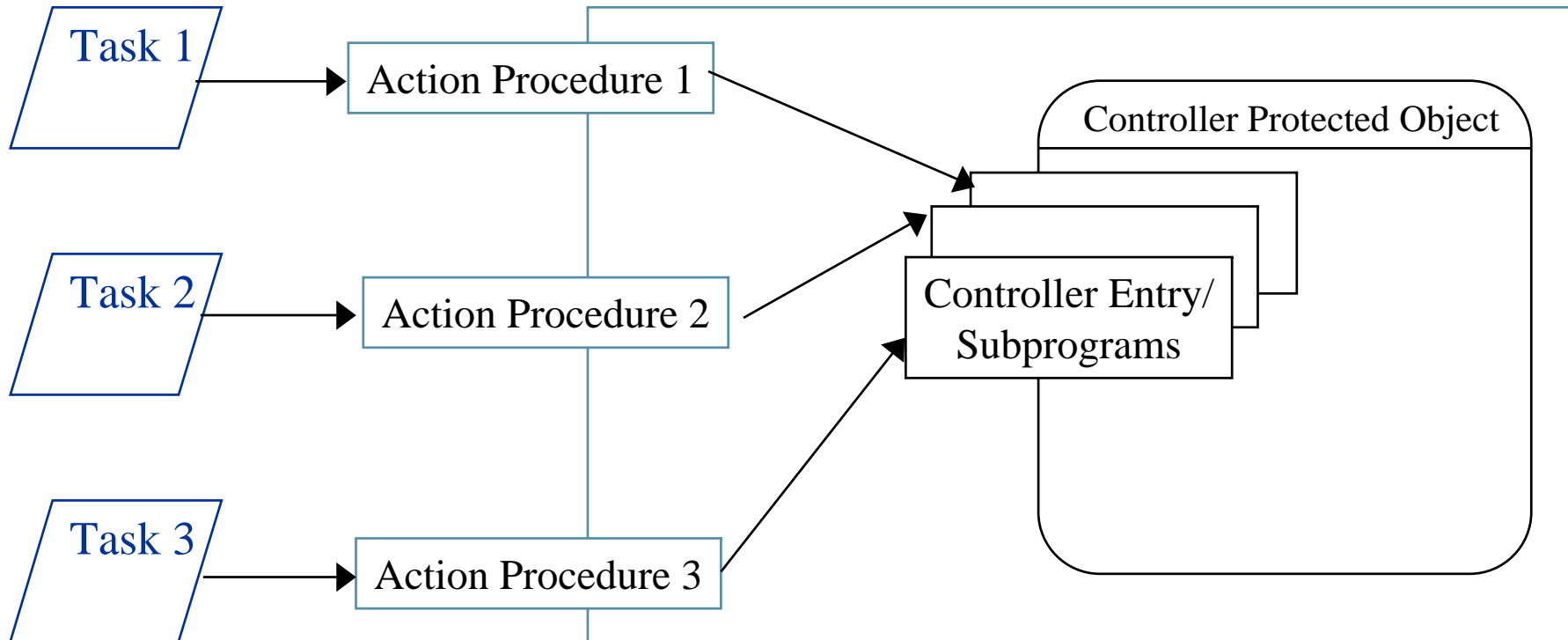
```
end A;
```

- No mainstream language or OS supports atomic action

# *Atomic Actions in Ada*

- The extended rendezvous in Ada enables a common form of atomic action where a task communicates with another task to request some computation; the called task undertakes this execution and then replies via the out parameters of the rendezvous
- The atomic action takes the form of an accept statement; it possesses synchronization atomicity as long as;
  - it does not update any variable that another task can access, and
  - it does not rendezvous with any other task
- An atomic action in Ada for three tasks could be programmed with a nested rendezvous, however, this would not allow any parallelism within the action
- An alternative model is to create an action controller and to program the required synchronization

# Ada and Atomic Actions



- Each atomic action is implemented by a package
- Roles are identified, each role is represented by a procedure in the package specification
- A task must associate itself with a role
- Each role can only have one active task

# Ada Structure

Action Controller

Role 1

Entry Protocol

action component

Exit Protocol

leave

Role 2

Entry Protocol

action component

Exit Protocol

leave

Role 3

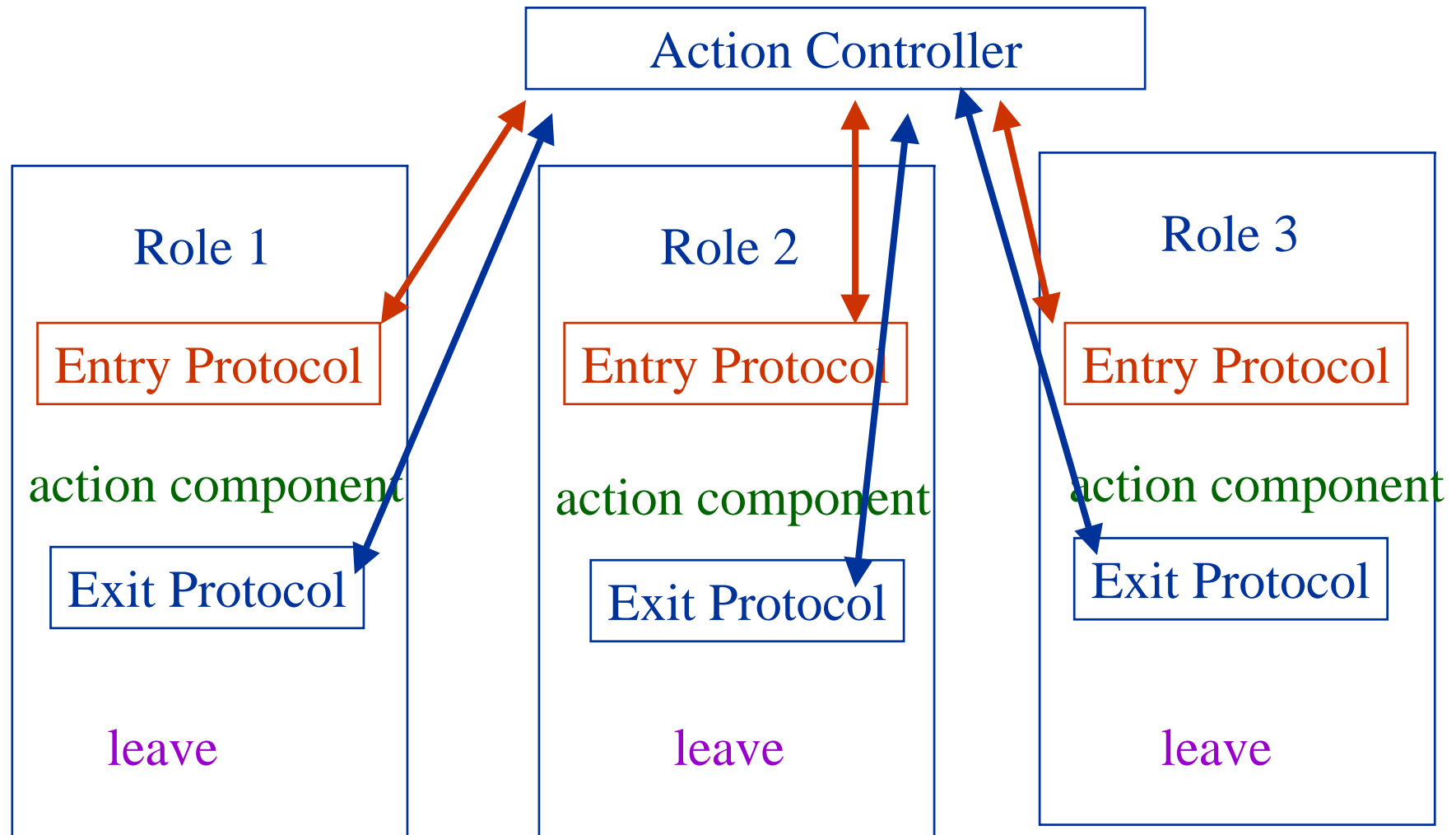
Entry Protocol

action component

Exit Protocol

leave

# Structure



# Ada Code

```
package Action_X is  
    procedure Code_For_First_Task(--params); -- Role1  
    procedure Code_For_Second_Task(--params); -- Role2  
    procedure Code_For_Third_Task(--params); -- Role3  
end Action_X;
```

```
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;  
  -- similarly for second, 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;
```

# Ada Code

```
procedure Code_For_First_Task(--params) is  
begin  
    Action_Controller.First;  
    -- acquire resources; the action itself,  
    -- communication 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;
```

- No recovery yet
- Only part encapsulation — can not stop communication with other tasks (unless insist on no with clauses??)
- Action controller could use semaphores, monitors etc

# *Atomic Actions in Java*

- 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

# Structure

Action Controller

Role 1

Entry Protocol

action component

Exit Protocol

leave

Role 2

Entry Protocol

action component

Exit Protocol

leave

Role 3

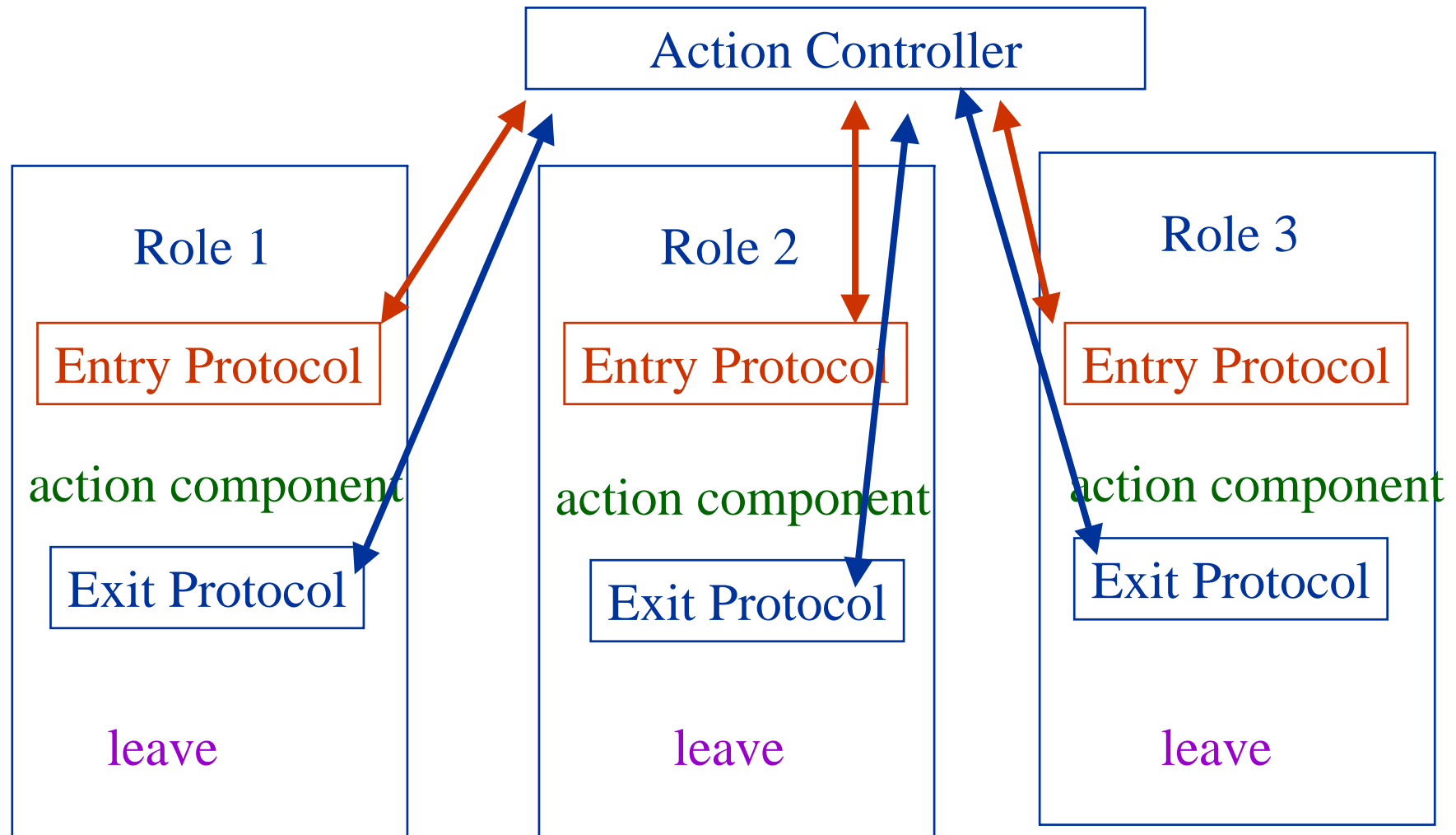
Entry Protocol

action component

Exit Protocol

leave

# Structure



```
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 processes 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 }
    }
};
```

Entry protocol

Exit protocol

```
public void role2()  
{  
    // similar to role1  
}  
  
public void role3()  
{  
    // similar to role1  
}  
}
```

# *A Four-Way Atomic Action*

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

public class NewAtomicActionControl
    extends AtomicActionControl
    implements FourWayAtomicAction
{
    public NewAtomicActionControl()
    {
        C = 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();  
    }  
}  
}
```

Have overridden the finish methods

All calls now dispatch to this method,  
consequently it must call the parent  
method

```
public void role4()
{
    boolean done = false;
    while(!done) {
        try {
            // As C is of type Controller, it must first
            // be converted to a RevisedController in order
            // to call the fourth method
            ((RevisedController)C).fourth();
            done = true;
        } catch (InterruptedException e) { // ignore }
    }
    // .... perform action
    done = false;
    while(!done) {
        try {
            Control.finished();
            done = true;
        } catch (InterruptedException e) { // ignore }
    }
}
}
```

# *Backward Error Recovery — Conversations*

- Consider 3 processes, each process names participates in the action via an action statement
- Within the statement, there is a recovery block: eg P1:

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

- On entry, the state of a process is saved; the set of entry points forms the recovery line

# *Conversations*



- Whilst inside, a process is only allowed to communicate with other processes active in the conversation and general resource managers
- In order to leave, all processes active in must have passed their acceptance test
- If passed, the conversation is finished and all recovery points are discarded
- If any process fails the test, all processes have their state restored and they execute their alternative modules
- Conversations can be nested, but only strict nesting is allowed
- If all alternatives fail, recovery must be performed at a higher level



# *Conversations*

- In the original definition of conversations, all processes taking part must have entered before any of the other processes can leave
- Here, if a process does not enter, as long as the other processes active in the conversation do not wish to communicate with it then the conversation can complete
- If a process does attempt communication, it can either block and wait for the process to arrive or it can continue
- This allows conversations to be specified where participation is not compulsory
- It allows processes with deadlines to leave the conversation, continue and if necessary take some alternative action

# *Criticisms of Conversations*

- Conversations can be criticised; when a conversation fails, all the processes are restored and all enter their alternatives
- This forces the same processes to communicate again to achieve the desired effect
- This may be not what is required; in practice when one process fails to achieve its goal in a primary module, it may wish to communicate with a completely new group of processes in its secondary module
- Also, the acceptance test for this secondary module may be quite different
- There is no way to express these requirements using conversations
- Dialogs and Colluquys ———SEE BOOK

# Atomic Actions and Forward Error Recovery

- If an exception occurs in one process, it is raised asynchronously in all processes active in the action

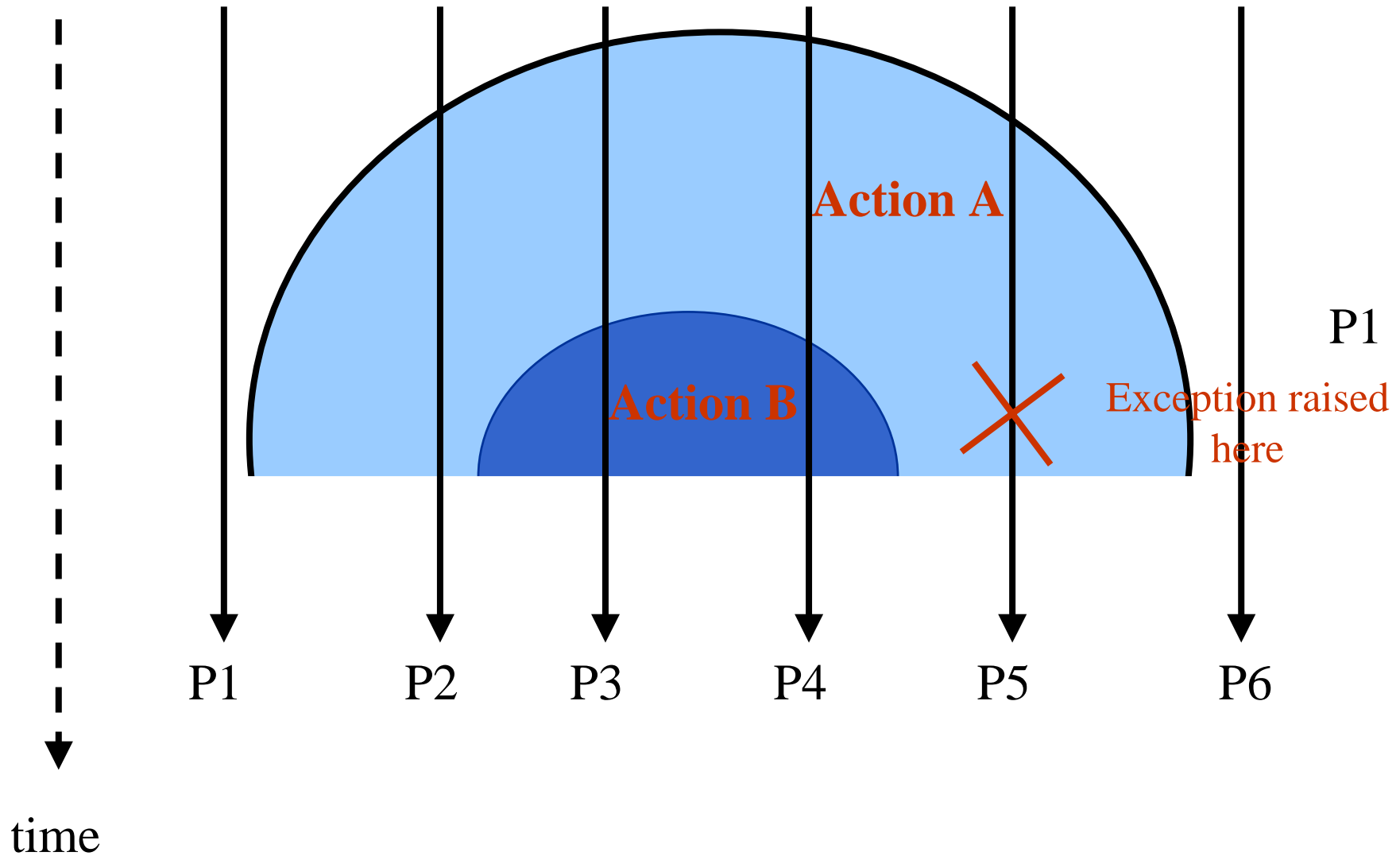
```
action A with (P2, P3) do
  -- the action
exception
  when exception_a =>
    -- sequence of statements
  when exception_b =>
    -- sequence of statements
  when others =>
    raise atomic_action_failure;
end A;
```

- Both termination and resumption models are possible
- If there is no handler in any one processes 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 processes

# *Resolution of Concurrently Raised Exceptions*

- Many process may raise different exceptions at the same time; this is likely if the error can not be uniquely identified by the error detection facility in action components
- If two exceptions are simultaneously raised, there may be two separate handlers in each process; the two exceptions in conjunction constitute a third which is the exception which indicates that both the other two exceptions have occurred.
- To resolve concurrently raised exceptions, exception trees can be used; here the handler is that at the root of the smallest subtree that contains all the exceptions
- It is not clear how to combined any parameters associated with this exception
- Each atomic action component can declare its own unique exception tree

# Exceptions and Nested Atomic Actions



# *Exceptions and Nested Atomic Actions*



- One process active in an action may raise an exception when other processes in the same action are involved in a nested action
- All processes involved must participate in the recovery action; unfortunately, the internal action is indivisible!

# *Exceptions and Nested Atomic Actions*

Two possible solutions to this problem

- 1 Hold back the raising of the exception until the internal action has finished
  - The exception may be associated with the missing of a deadline
  - The error condition detected may indicate that the internal action may never terminate because some deadlock condition has arisen
- 2 Allow internal actions to have a predefined abortion exception; this indicates that an exception has been raised in a surrounding action and that the pre-conditions under which it was invoked are no longer valid
  - If raised, the internal action should abort itself. Once the action has been aborted, the containing action can handle the original exception. If cannot abort itself, it must signal failure exception.
  - If no abortion exception is defined, the surrounding action must wait for the internal action to complete

# *Asynchronous Notifications*

- None of the major RT languages/OSs support atomic actions
- They do support asynchronous notifications: a mechanism whereby one process can gain the attention of another without the latter waiting
- This can be used as a basis for error recovery between concurrent systems
- As with exception handling: resumption and termination models:
- The resumption model behaves like a software interrupt
- With the termination model, each process specifies a domain of execution during which it is prepared to receive an asynchronous event; after an event has been handled, control is returned to the interrupted process at a location different to that where the event was delivered



# *The User Need for Asynchronous Notification*

- **Fundamental requirement:** to enable a process to respond **quickly** to a condition detected by another process
- Error recovery — to support atomic actions
- Mode changes — where changes between modes are expected but cannot be planned.
  - a fault may lead to an aircraft abandoning its take-off and entering into an emergency mode of operation;
  - an accident in a manufacturing process may require an immediate mode change to ensure an orderly shutdown of the plant.
  - The processes 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

# *The User Need for Asynchronous Notification*

- 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
  - 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 process 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

# *Polling*



Polling for the notification is too slow. It can be argued that the process could be aborted and recreated quickly enough, however, this is probably more error prone than providing direct support

# *Asynchronous Event Handling*

- RTJ asynchronous events (ASE) are similar to POSIX signals (there is a class which allows POSIX signals to be mapped onto RTJ events)
- There are three main classes associated ASEs:
  - `AsynEvent`
  - `AsyncEventHandler`
  - `BoundAsyncEventHandler`
- Each `AsynEvent` can have one or more handlers
- When the event occurs all the handlers associated with the event are **scheduled** for execution
- The firing of an event can also be associated with the occurrence of an implementation-dependent external action by using the `bindTo` method

# *Asynchronous Events*

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

    public synchronized void addHandler(AsyncEventHandler handler);
    public synchronized void removeHandler(
        AsyncEventHandler handler);
    public void setHandler(AsyncEventHandler handler);
    public void bindTo(java.lang.String happening);
    // bind to external event

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

    public void fire();
    // Execute the run() methods of the set of handlers
    ...
}
```

# *Asynchronous Event Handlers*

```
public abstract class AsyncEventHandler implements Schedulable
{
    public AsyncEventHandler(SchedulingParameters scheduling,
        ReleaseParameters release, MemoryParameters memory,
        MemoryArea area, ProcessingGroupParameters group);

    public void addToFeasibility();
    public void removeFromFeasibility();

    protected int getAndClearPendingFireCount();

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

    public final void run();

    ...
}
```

# *Bound Asynchronous Event Handlers*

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

# *Timers (see later)*

```
public abstract class Timer extends AsyncEvent
{
    protected Timer(HighResolutionTimer t, Clock c,
                    AsyncEventHandler handler);
    public ReleaseParameters createReleaseParameters();

    public AbsoluteTime getFireTime();
    public void reschedule(HighResolutionTimer time);
    public Clock getClock();

    public void disable();
    public void enable();

    public void start(); // start the timer ticking
}
```



# *POSIX Signals*

- Used for a class of **environment-detected synchronous errors** (such as divide by zero, illegal pointer)
- There are a number of pre-defined signals each of which is allocated an integer value. e.g. SIGALARM, SIGILL
- Also an implementation-defined number of signals which are available for application use: SIGRTMIN .. SIGRTMAX
- Each signal has a default handler, which usually terminates the receiving process
- A process can block, handle or ignore a signal
- A signal which is not blocked and not ignored is delivered as soon as it is generated; a signal which is blocked is pending delivery

# *C Interface to POSIX Signals*

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

```
typedef struct {  
    int si_signo; /* signal number */  
    int si_code; /* cause of signal */  
    union sigval si_value; /* integer or pointer */  
} siginfo_t;
```

Mainly  
used for  
message  
queues,  
timers and  
real-time  
signals

```
typedef ... sigset_t; /* implementation dependent */
```

```
struct sigaction { /* information about the handler */
    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 handler */
```

Diagram illustrating the parameters of the `sigaction` function:

- signal**: Points to the `int sig` parameter.
- handler**: Points to the `const struct sigaction *reaction` parameter.
- old handler**: Points to the `struct sigaction *old_reaction` parameter.

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

```
int sigsuspend(const sigset_t *sigmask);
```

```
/* wait for a non-blocked signal and the handler to complete */
```

```
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

```
/* as sigsuspend, but handler not called */
```

```
/* information returned returned */
```

```
int sigtimedwait(const sigset_t *set, siginfo_t *info,  
                 const struct timespec *timeout);
```

```
/* as sigwaitinfo with timeout */
```

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

/* allow a signal set to be created and manipulated */
int sigemptyset(sigset_t *s); /* initialise a set to empty */
int sigfillset(sigset_t *s); /* initialise 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 member */
```

```
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 */
```

# *Mode Change Example*

```
#include <sig.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_int;
}
```

```
int main()
{
    sigset_t mask, omask;
    struct sigaction s, os;
    int local_mode;

    SIGEMPTYSET(&mask); /* create a signal mask */
    SIGADDSET(&mask, MODE_CHANGE);

    s.sa_flags = SA_SIGINFO; /* use 3 argument handler */
    s.sa_mask = mask; /* additional signals blocked
                       during handler */
    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:
            ...
    }
    SLEEP(. . .);
}
...
}

```

signals  
masked  
here

Signal occurring here  
are acted upon  
immediately;  
however, the  
application only  
responds on each  
iteration

Note, sleep wakes up if signal called

# *POSIX Threads and Atomic Actions*

- Two approaches to implementing an atomic action-like structure between threads:
  - 1 Use a **signals**, **setjmp** and **longjmp** to program the required coordination. Unfortunately, longjmp and all the thread system calls are **async-signal unsafe**. This means that any communication and synchronization between threads using mutexes and condition variables must be encapsulated between calls for blocking and unblocking signals. The resulting algorithm can become complex.
  - 2 Use thread creation and cancelling to program the required recovery. As threads are cheap, this approach does not have the same performance penalty as more heavy-weight process structure.
- The need for these approaches comes from the use of the resumption model; a more straightforward structure is obtainable if a termination model is supported

# *Asynchronous Notification in Ada*



- The abort statements
- Asynchronous Transfer of Control (the asynchronous select statement)

# *The Abort Statement*



- Intended for use in response to those error conditions where recovery by the errant task is not deemed possible
- Any task may abort any other named task
- Tasks which are aborted become abnormal and are prevented from interacting with other tasks
- Any non-completed tasks that depend on the aborted tasks also become abnormal
- When a task becomes abnormal, every construct it is executing is aborted immediately unless it is included in an abort-deferred operation

# *Abort Completion*

- If a construct is blocked outside an abort-deferred operation (other than at an entry call), it is immediately completed
- Other constructs must complete no later than
  - the end of activation of a task
  - when it activates 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 the sequence of statements in an exception handler

**A task which does not reach any of these points need not be terminated!**

**Real-Time Annex requires any delay to be documented**

# *Abort-deferred Operations*

- a protected action
- waiting for an entry call to complete (after having initiated the attempt to cancel it as part of the abort)
- waiting for termination of dependent tasks
- executing an `Initialize`, `Finalize`, or assignment operation of a controlled object
- certain actions within these operations result in bounded error:
  - the execution of an asynchronous select statement
  - the creation of tasks whose master is within the operation

# *Use of the Abort Statement*

- "An abort statement should be used only in situations requiring unconditional termination." ARM 9.8
- "The existence of this statement causes intolerable overheads in the implementation of every other feature of tasking. Its 'successful' use depends on a valid process aborting a wild one before the wild one aborts a valid process — or does any other serious damage. The probability of this is negligible. If processes can go wild, we are much safer without aborts." C.A.R. Hoare (On Ada 83)

Even so, the abort is considered to be a valid real-time requirement, and Ada makes every effort to ensure that the facility can be used as safely as possible, given its inherently dangerous nature.

# *The Asynchronous Select Statement*

```
asynchronous_select ::=  
    select  
        triggering_alternative  
    then abort  
        abortable_part  
    end select;
```

```
triggering_alternative ::=  
        triggering_statement  
        [sequence_of_statement]
```

```
triggering_statement ::= entry_call_statement |  
                        delay_statement
```

```
abortable_part ::= sequence_of_statements
```

**must not contain an accept statement**



# *Semantics I*

- First the triggering statement is executed
- If the entry call is queued (or the delay time has not passed), the abortable part begins its execution
- If the abortable part completes before the completion of the entry call (or before the delay time expires), the entry call (or delay) is cancelled
- When cancellation of the entry call or the delay completes, the select statement is finished
- Cancellation of the delay happens immediately, cancellation of the entry call may have to wait if the rendezvous or protected action is in progress (until it has finished)

# *Semantics II*

- If the triggering event completes, the abortable part is aborted (if not already completed) and any finalisation code is executed
- When these activities have finished, the optional sequence of statements following the triggering event is executed
- Note: If the triggering entry call is executed, then even if the abortable part completes, the optional sequence of statements following the triggering event is executed
- If the triggering event occurs before the abortable can start, the abortable part is not executed

# Example: Rendezvous Available Immediately

```
task Server is  
    entry ATC_Event;  
end Server;  
task body Server is  
begin  
    ...  
    accept ATC_Event do  
        Seq2;  
    end ATC_Event;  
    ...  
end Server;
```

```
task To_Interrupt;  
task body To_Interrupt is  
begin  
    ...  
    select  
        Server.ATC_Event;  
        Seq3;  
    then abort  
        Seq1;  
    end select  
    Seq4;  
end To_Interrupt;
```

# *No Rendezvous before Seq1 Finishes*

```
task Server is  
    entry ATC_Event;  
end Server;  
task body Server is  
begin  
    ...  
    accept ATC_Event do  
        Seq2;  
    end ATC_Event;  
    ...  
end Server;
```

```
task To_Interrupt;  
task body To_Interrupt is  
begin  
    ...  
    select  
        ... aborted ...  
        Seq3;  
    then abort  
        Seq1;  
    end select  
    Seq4;  
end To_Interrupt;
```

# *Rendezvous Finishes before Seq1*

```
task Server is  
    entry ATC_Event;  
end Server;  
task body Server is  
begin  
    ...  
    accept ATC_Event do  
        Seq2;  
    end ATC_Event;  
    ...  
end Server;
```

```
task To_Interrupt;  
task body To_Interrupt is  
begin  
    ...  
    select  
        Server.ATC_Event;  
        Seq3;  
    then abort  
        aborted  
    end select  
    Seq4;  
end To_Interrupt;
```

# *Rendezvous Finishes after Seq1*

```
task Server is  
    entry ATC_Event;  
end Server;  
task body Server is  
begin  
    ...  
    accept ATC_Event do  
        Seq2;  
    end ATC_Event;  
    ...  
end Server;
```

```
task To_Interrupt;  
task body To_Interrupt is  
begin  
    ...  
    select  
        Server.ATC_Event;  
        Seq3;  
    then abort  
        Seq1;  
    end select  
    Seq4;  
end To_Interrupt;
```

# *Sequence of Events*

**if** the rendezvous is available immediately **then**

Server.ATC\_Event is issued

Seq2 is executed

Seq3 is executed

Seq4 is executed

**else if** no rendezvous starts before Seq1 finishes **then**

Server.ATC\_Event is issued

Seq1 is executed

Server.ATC\_Event is cancelled

Seq4 is executed

# *Sequence of Events Continued*

else if the rendezvous finishes before Seq1 finishes then

Server.ATC\_Event is issued

partial execution of Seq1 occurs *concurrently* with Seq2

Seq1 is aborted and finalised

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

execution of Seq2 completes

Seq3

Seq4

end




# *Exceptions and ATC*



- With the asynchronous select statement, two activities are potentially executing concurrently
- Both can raise exceptions
- The one from the abortable part is lost, if the abortable part is aborted

# Example of ATC — Error Recovery

```
type Error_ID is (Err1, Err2, Err3);
package Error_Notification is new Broadcasts(Error_ID);
Error_Occurred : Error_Notification.Broadcast;
task type Interested_Party;
task Error_Monitor;
task body Error_Monitor is
begin
    ...
    -- when error detected
    Error_Occurred.Send(Error);
    ...
end Error_Monitor
```



a protected type

# *Error Recovery II*

```
task body Interested_Party is
  Reason : Error_ID;
begin
  loop
    select
      Error_Occurred.Receive(Reason) ;
    case Reason is
      when Err1 => -- appropriate recovery action
      when Err2 => -- appropriate recovery action
      when Err3 => -- appropriate recovery action
    end case;
    then abort
      loop -- normal operation end loop;
    end select;
  end loop;
end Interested_Party;
```

# *Deadline Overrun Detection*

```
with Ada.Real_Time; use Ada.Real_Time;

task Critical;

task body Critical is
    Deadline : Real_Time.Time := ...;
begin
    ...
    select
        delay until Deadline;
        -- recovery action
    then abort
        -- enter time critical section of code
    end select;
    ...
end Critical;
```

# Mode Changes

```
with Persistent_Signals; use Persistent_Signals;
with Calendar; use Calendar;
...
type Mode is (Non_Critical, Critical);
Change_Mode : Persistent_Signal;
task Sensor_Monitoring;
task body Sensor_Monitor is
    Current_Mode : Mode := Non_Critical;
    Next_Time : Time := Clock;
    Critical_Period : constant Duration := 1.0;
    Non_Critical_Period : constant Duration := 10.0;
    Current_Period : Duration := Non_Critical_Period;
begin
```

# Mode Change II

```
loop
  select Change_Mode.Wait;
    if Current_Mode = Critical then
      Current_Mode := Non_Critical;
      Current_Period := Non_Critical_Period;
    else Current_Mode := Critical;
      Current_Period := Critical_Period; end if;
    Next_Time := Clock; -- say
  then abort
    loop
      -- read sensor etc.
      delay until Next_Time;
      Next_Time := Next_Time + Current_Period;
    end loop;
  end select;
end loop;
end Sensor_Monitor;
```

# *Understanding ATC*



- Interaction with the Delay Statement
- Interaction with Timed Entry Calls
- Interaction with Multiple Entry Calls
- Nested ATC
- Interaction with Exceptions

# *Interaction with the Delay Statement*

```
task body A is
  T : Time;
  D : Duration;
begin
  ...
  select
    delay until T;
  then abort
    delay D;
  end select;
end A;
```

```
task body B is
  T : Time;
  D : Duration;
begin
  ...
  select
    delay D;
  then abort
    delay until T;
  end select;
end B;
```

**Are these equivalent?**



# *Interaction with Timed Entry Calls*

**task body A is**

T : Time;

**begin**

**select**

**delay until** T;

S1;

**then abort**

Server.Entry1;

S2;

**end select;**

**end A;**

**task body B is**

T : Time;

**begin**

**select**

Server.Entry1;

S1;

**then abort**

**delay until** T;

S2

**end select;**

**end B;**

**task body C is**

T : Time;

**begin**

**select**

Server.Entry1;

S1;

**or**

**delay until** T;

S2

**end select;**

**end C;**

Very similar structures, all slightly different behaviours

# *Rendezvous Starts and Finishes Before Timeout*

```
task body A is
  T : Time;
begin
  select
    delay until T;
  S1;
  then abort
    Server.Entry1;
  abort;
end select;
end A;
```

```
task body B is
  T : Time;
begin
  select
    Server.Entry1;
  S1;
  then abort
    abort;
  S2;
end select;
end B;
```

```
task body C is
  T : Time;
begin
  select
    Server.Entry1;
  S1;
  or
    until T;
  delay until T;
  S2;
end select;
end C;
```

## *Rendezvous Starts Before Timeout but Finishes After Timeout*

```
task body A is
  T : Time;
begin
  select
    delay until T;
  S1;
then abort
  Server.Entry1;
  S2;
end select;
end A;

task body B is
  T : Time;
begin
  select
    Server.Entry1;
  S1;
then abort
  delay until T;
  abort;
end select;
end B;

task body C is
  T : Time;
begin
  select
    Server.Entry1;
  S1;
or
  delay until T;
  S2;
end select;
end C;
```

# *Timeout Occurs Before the Rendezvous Starts*

**task body A is**

T : Time;

**begin**

**select**

**delay until T;**

S1;

**then abort**

Server.Entry1;

S2;

**end select;**

**end A;**

**task body B is**

T : Time;

**begin**

**select**

Server.Entry1;

S1;

**then abort**

**delay until T;**

abort

**end select;**

**end B;**

**task body C is**

T : Time;

**begin**

**select**

Server.Entry1;

S1;

**or**

**delay until T;**

S2;

**end select;**

**end C;**

# Timed Entry Calls

- Rendezvous with `Server` starts and finishes before timeout
  - `A` executes the rendezvous and then attempts `S2`, if `S2` does not complete before the timeout it is abandoned and `S1` is executed
  - `B` executes the rendezvous and then `S1`
  - `C` executes the rendezvous and `S1`
- The rendezvous starts before the timeout but finishes after the timeout
  - `A` executes the rendezvous and `S1`
  - `B` executes the rendezvous, `S1` and part of `S2`
  - `C` executes the rendezvous and `S1`

# *Timed Entry Calls II*

- The timeout occurs before the rendezvous started
  - A executes S1
  - B executes part or all of S2 and possibly the rendezvous and S1
  - C executes S2

# *A Timed Entry Call?*

```
task body C is
  T : Time;
begin
  Occurred := False;
  select
    delay until T;
  then abort
    Server1.Entry1(Occurred);
    -- Occurred set to True in Server1
  end select;
  if Occurred then
    S1;
  else
    S2;
  end if;
end C;
```

# *Interaction with Multiple Entry Calls*

```
task body A is
```

```
  T : Time;
```

```
begin
```

```
  ...
```

```
  select
```

```
    TaskC.Entry1;
```

```
  then abort
```

```
    TaskD.Entry1;
```

```
  end select;
```

```
end A;
```

```
task body B is
```

```
  T : Time;
```

```
begin
```

```
  ...
```

```
  select
```

```
    TaskD.Entry1;
```

```
  then abort
```

```
    TaskC.Entry1;
```

```
  end select;
```

```
end B;
```



# Consider:

1. **TaskC.Entry1** becomes available first:
  - **TaskA** will rendezvous with **TaskC** and possibly **TaskD** (if the rendezvous becomes available before the rendezvous with **TaskC** completes)
  - **TaskB** will rendezvous with **TaskC** and possibly **TaskD**
2. **TaskD.Entry1** becomes ready first:
  - Similar to above
3. **TaskC.Entry1** and **TaskD.Entry1** are both ready:
  - **TaskA** will rendezvous with **TaskC** only
  - **TaskB** will rendezvous with **TaskD** only

# *Nested Asynchronous Select Statements*

```
task body A is
begin
  ...
  select
    B.Entry1;
  then abort
    select
      C.Entry1;
    then abort
      Seq;
    end select;
  end select;
  ...
end A;
```

Here task A will wait for an entry call to become complete from tasks B or C. If none arrive before Seq has finished its execution, C will be cancelled and then B will potentially also be cancelled.

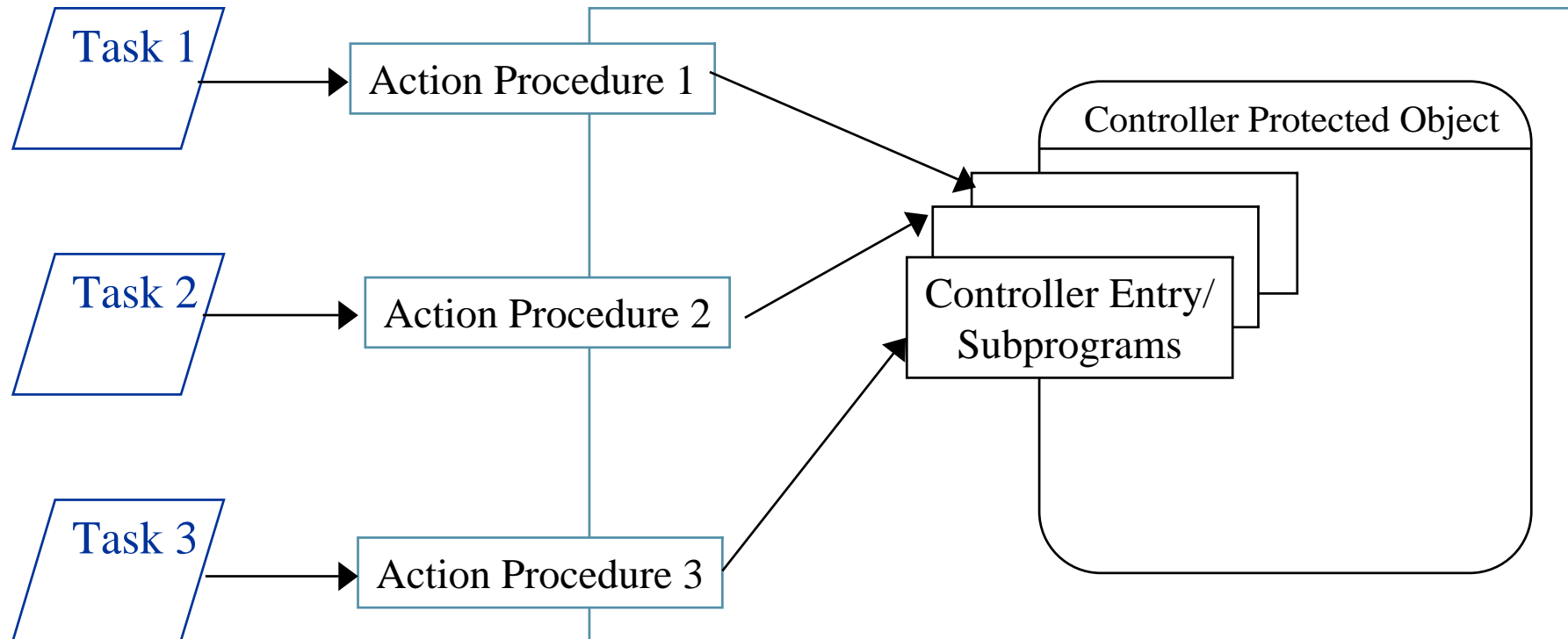
# *Interaction with Exceptions*

```
task body Server is
begin
    ...
    accept ATC_Event do
        Seq2;
    end ATC_Event;
    ...
end Server;

select
    Server.ATC_Event;
    Seq3;
then abort
    Seq1
end select
...
```

- If Seq1 raises an exception and the triggering event does not occur, the exception is propagated from the select statement
- If the triggering event occurs, then any exception raised by S1 is lost (to avoid the possibility of parallel exceptions being raised from the select statement)

# Ada and Atomic Actions



- Each atomic action is implemented by a package
- Roles are identified, each role is represented by a procedure in the package specification
- A task must associate itself with a role
- Each role can only have one active task

# *Backward Error Recovery: Conversations*

```
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;
```

- Each procedure will contain a recovery block
- We will use ATC to inform each task if one of the other tasks in the conversation has failed

```
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;
```

# *The Protected Controller*

- The `Wait_Abort` entry is the asynchronous event on which the tasks will wait whilst performing their part of the action
- Each task calls `Done` if it has finished without error; only when all three tasks have called `Done` will they be allowed to leave
- If a task recognises an error condition (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
- 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 and undertake recovery
- Once the event has been handled, all tasks must wait on `Cleanup` so that they all can leave the conversation together

```
-- local PO 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

  procedure T1_Primary is
  begin
    select
      Controller.Wait_Abort;
      Controller.Cleanup; -- wait for all to finish
      raise Primary_Failure;
    then abort
      begin
        -- code to implement atomic action,
        if Acceptance_Test = Failed then
          Controller.Signal_Abort;
        else
          Controller.Done;
        end if;
      exception
        when others =>
          Controller.Signal_Abort;
      end;
    end select;
  end T1 Primary;

```

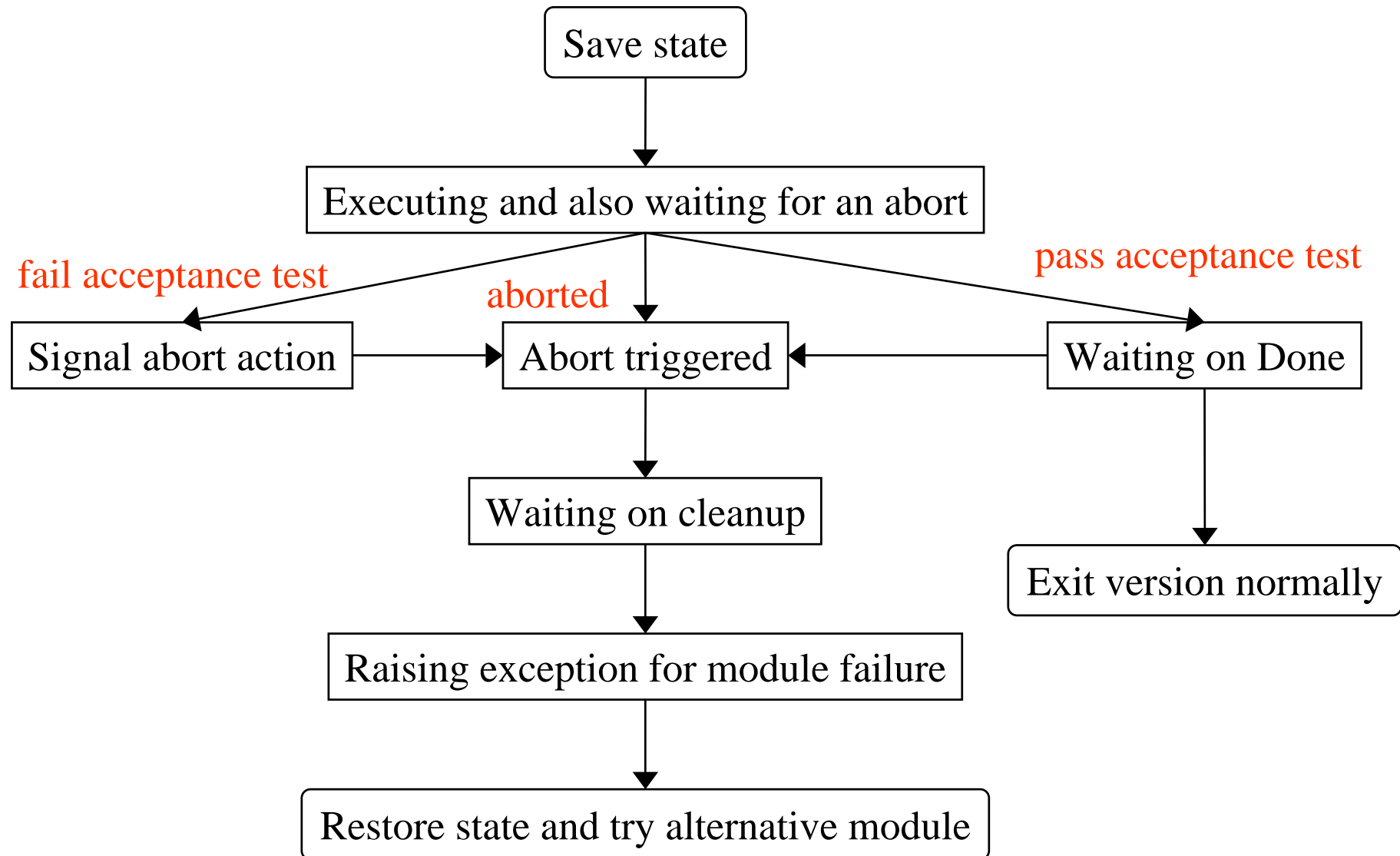
similarly for T1\_Secondary  
and T1\_Tertiary

```
begin
  My_Recovery_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_Recovery_Cache.Restore(...);
      when Secondary_Failure => Recovery_Cache.Restore(...);
      when Tertiary_Failure => Recovery_Cache.Restore(...);
      raise Atomic_Action_Failure;
      when others => Recovery_Cache.Restore(...);
      raise Atomic_Action_Failure;
    end;
  end loop;
end T1_Part;

end Conversation;
```

similarly for T2  
and T3

# State Transition Diagram



# *Forward Error Recovery*

```
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;
```

```
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);
    procedure Signal_Abort(E: Exception_Id);
    entry Wait_Cleanup(Result : out Vote_t);
  private
    Killed : Boolean := False;
    Releasing_Cleanup : Boolean := False;
    Releasing_Done : Boolean := False;
    Reason : Exception_Id := Null_Id;
    Final_Result : Vote_t := Commit;
    Informed : Integer := 0;
end Controller;
```

```
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;
```



```

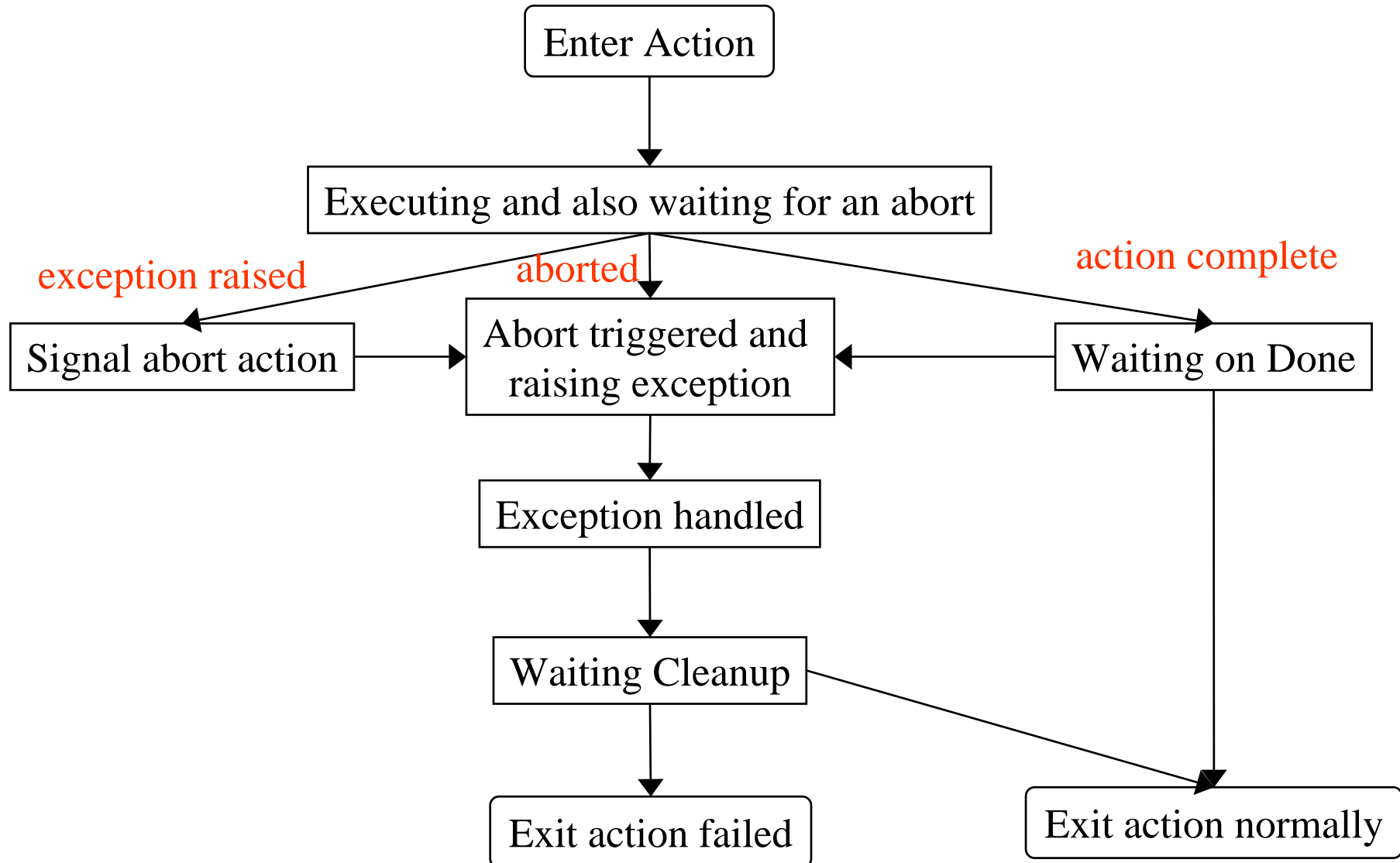
procedure T1(Params: Param) is
  X : Exception_Id; Decision : Vote_T;
begin
  select
    Controller.Wait_Abort(X); Raise_Exception(X);
  then abort
    begin
      -- code to implement atomic action
      Controller.Done;
    exception when E: others =>
      Controller.Signal_Abort(Exception_Identity(E));
    end;
  end select;

exception
  when E: others => 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_Part;

  -- similarly for T2 and T3
end Action;

```

# *F.E.C.: State Transition Diagram*



# *Asynchronous Transfer of Control in Java*

- Early versions of Java allowed one thread to asynchronously effect another thread through

```
public final void suspend() throws SecurityException;
```

```
public final void resume() throws SecurityException;
```

```
public final void stop() throws SecurityException;
```

```
public final void stop(Throwable except)  
                    throws SecurityException;
```

- The `stop` method, causes the thread to stop its current activity and throw a `ThreadDeath` exception
- All of the above methods are now obsolete and therefore should not be used

# ATC in Java

- Standard Java now only supports (in the Thread class):

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

```
public void destroy();
```

- When a thread interrupts another thread:
  - If the interrupted thread is blocked in `wait`, `sleep` or `join`, 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
- The `destroy` method is similar to the Ada abort facility and destroys the thread without any cleanup

# *ATC in RTJ*

- Similarities between Ada and RTJ models
  - it is necessary to indicate which regions of code can receive the ATC request
  - ATC are deferred during task/thread interaction and finalization
- Differences
  - The RTJ model is integrated with the Java exception handling facility whereas the Ada model is integrated into the select statement and entry-handling mechanisms
  - The RTJ model requires each method to indicate that it is prepared to allow the ATC to occur; ATC are deferred until the thread is executing within such a method.
  - 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

# *ATC in RTJ*

- Allows one thread to interrupt another thread
- Integrated into the Java exception handling and interrupt facility
- RTJ requires each method to indicate if it is prepared to allow an ATC
- Use of ATC requires
  - declaring an `AsynchronouslyInterruptedException` (AIE)
  - identifying methods which can be interrupted using a throw clause
  - signaling an `AsynchronouslyInterruptedException` to a thread (t)
- Calling `interrupt()` throws a **generic AIE**
- ATC are deferred during synchronized methods and finally clauses

# Example of ATC

```
import nonInterruptibleServices.*;

public class InterruptibleService
{ // AIE is short for AsynchronouslyInterruptedException
  public AIE stopNow = AIE.getGeneric();

  public boolean Service() throws AIE
  {
    //code interspersed with calls to nonInterruptibleServices
  }
}

public InterruptibleService IS = new InterruptibleService();

// code of thread, t
if(IS.Service()) { ... }
else { ... };

// now another thread interrupts t:
t.interrupt;
```

# *Semantics: when AIE is signalled*

- If  $t$  is executing within an ATC-deferred section, the AIE is marked as pending
- If  $t$  is executing in a method which has no AIE declared in its throws list, the AIE is marked as pending
- A pending AIE is thrown as soon as  $t$  returns to (or enters) a method with an AIE declared in its throws list
- If  $t$  is executing within a try block within a method which has declared an AIE in its throws list, the try block is terminated and control is transferred to catch clauses; if no appropriate catch clause is found, the AIE is propagated to the calling method
- If an appropriate handler is executed, processing of the AIE is completed (unless the AIE is propagated from within the handler)
- If  $t$  is executing outside a try block within a method which has declared an AIE in its throws list, the method is terminated and the AIE is thrown immediately in the calling method



# *Semantics: when AIE is signalled*

- If `t` is blocked inside a `wait`, `sleep` or `join` method called from within a method which has an AIE declared in its `throws` list, `t` is rescheduled and the AIE is thrown
- If `t` is blocked inside a `wait`, `sleep` or `join` method called from within a method which has **no** AIE declared in its `throws` list, `t` is rescheduled and the AIE is marked as pending

# *Catching an AIE*

- Once an ATC has been thrown 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.
- If it is not, the exception should be propagated to the calling method.
- The `happened` method defined in the class `AsynchronouslyInterruptedException` is used for this purpose

# Example Continued

```
import NonInterruptibleServices.*;
public class InterruptibleService
{
    public AIE stopNow = AIE.getGeneric();

    public boolean Service() throws AIE
    {
        try {
            //code interdispersed with calls to NonInterruptibleServices
        }
        catch AIE AI {
            if(stopNow.happened(true)) { //handle the ATC }
            // no else clause, the true parameter indicates that
            // if the current exception is not stopNow,
            // it is to be immediately propagated
            // to the calling method
        }
    }
}
```

# *Alternative Handler*

```
catch AIE AI {  
    if(stopNow.happened(false)) {  
        //handle the ATC  
    } else {  
        //cleanup  
        AI.propagate();  
    }  
}
```

# AIE

```
public class AsynchronouslyInterruptedException extends
    java.lang.InterruptedException
{
    ...
    public synchronized void disable();
    public boolean doInterruptible (Interruptible logic);

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

    public boolean happened (boolean propagate);

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

    public void propagate();
}
```

# *Interruptible*

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

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

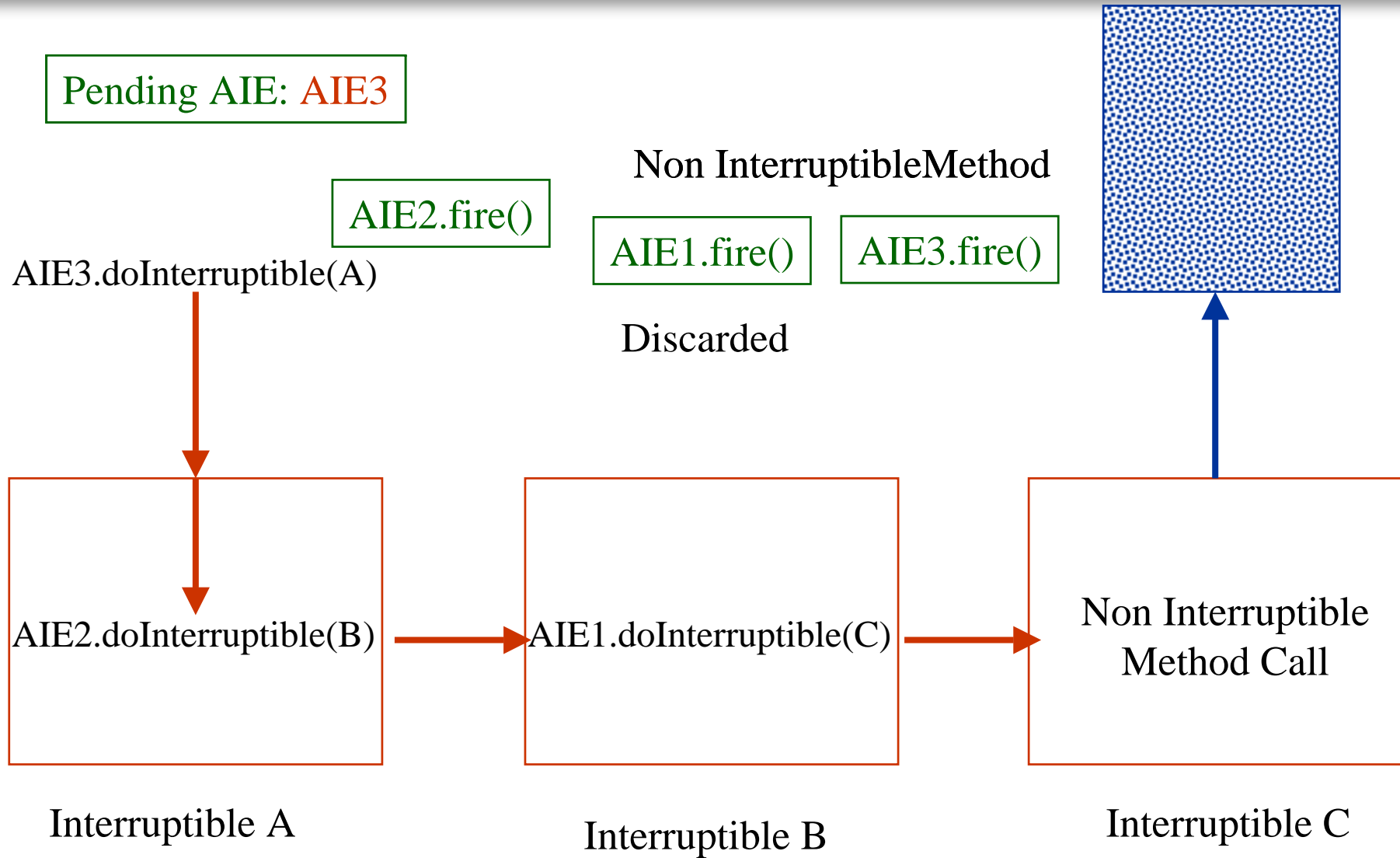
- An object which wishes to provide an interruptible method does so by implementing the Interruptible interface.
- The `run` method is the method that is interruptible; the `interruptedAction` method is called by the system if the `run` method is interrupted

# *Interruptible*

- Once this interface is implemented, the implementation can be passed as a parameter to the `doInterruptible` method in the AIE class
- The method can then be interrupted by calling the `fire` method in the AIE class
- Further control over the AIE is given by
  - `disable`
  - `enable`
  - `isEnabled`
- A disabled AIE is deferred until it is enabled

Only one task can be executing a `doInterruptible` at once

# Nested ATCs





# *Timeouts (see later)*

- With Real-Time Java, there is a subclass of `AsynchronouslyInterruptedException` called `Timed`
- Both absolute and Relative times can be used

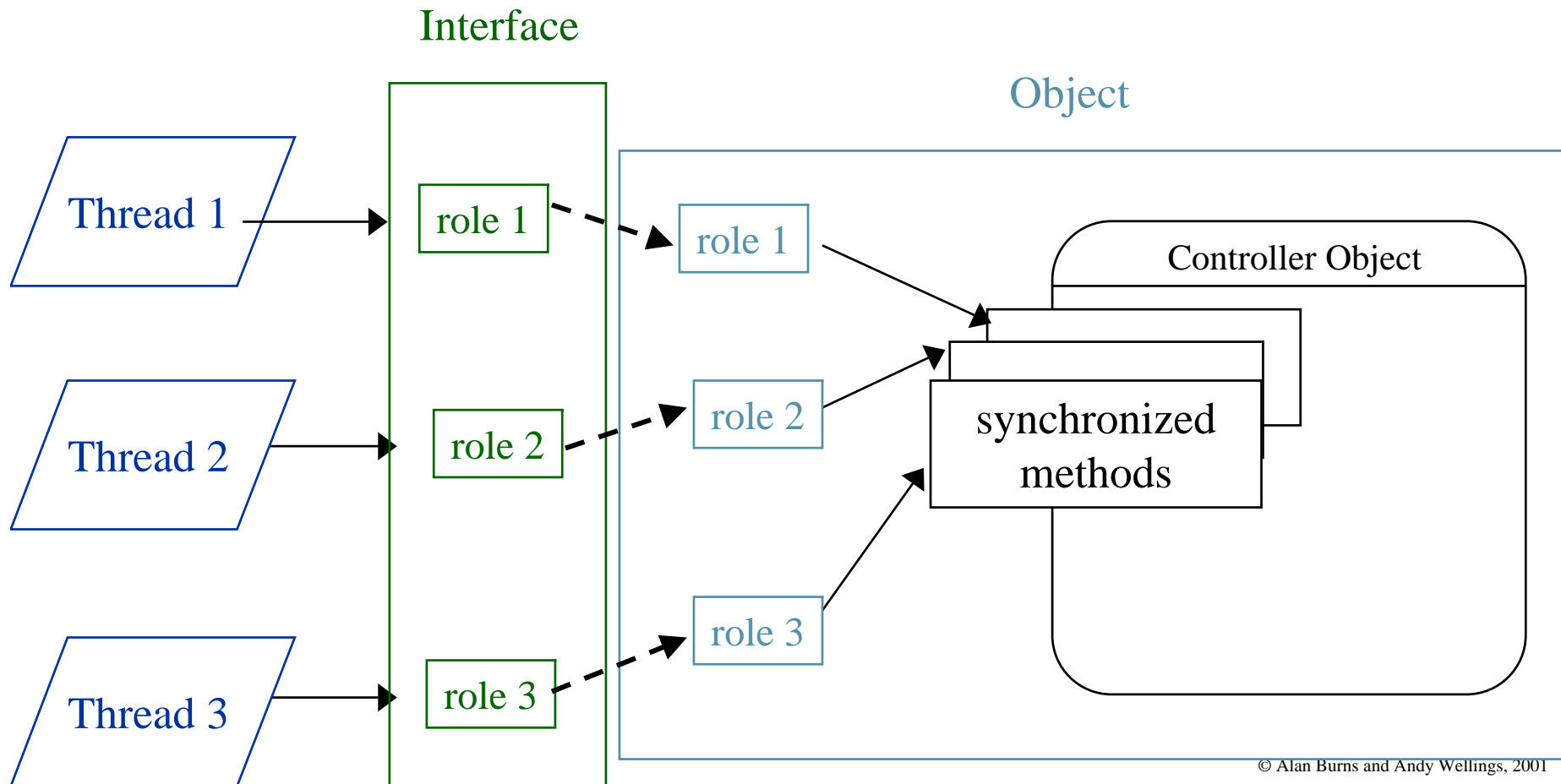
```
public class Timed extends AsynchronouslyInterruptedException
    implements java.io.Serializable
{
    public Timed(HighResolutionTime time) throws
        IllegalArgumentException;

    public boolean doInterruptible (Interruptible logic);

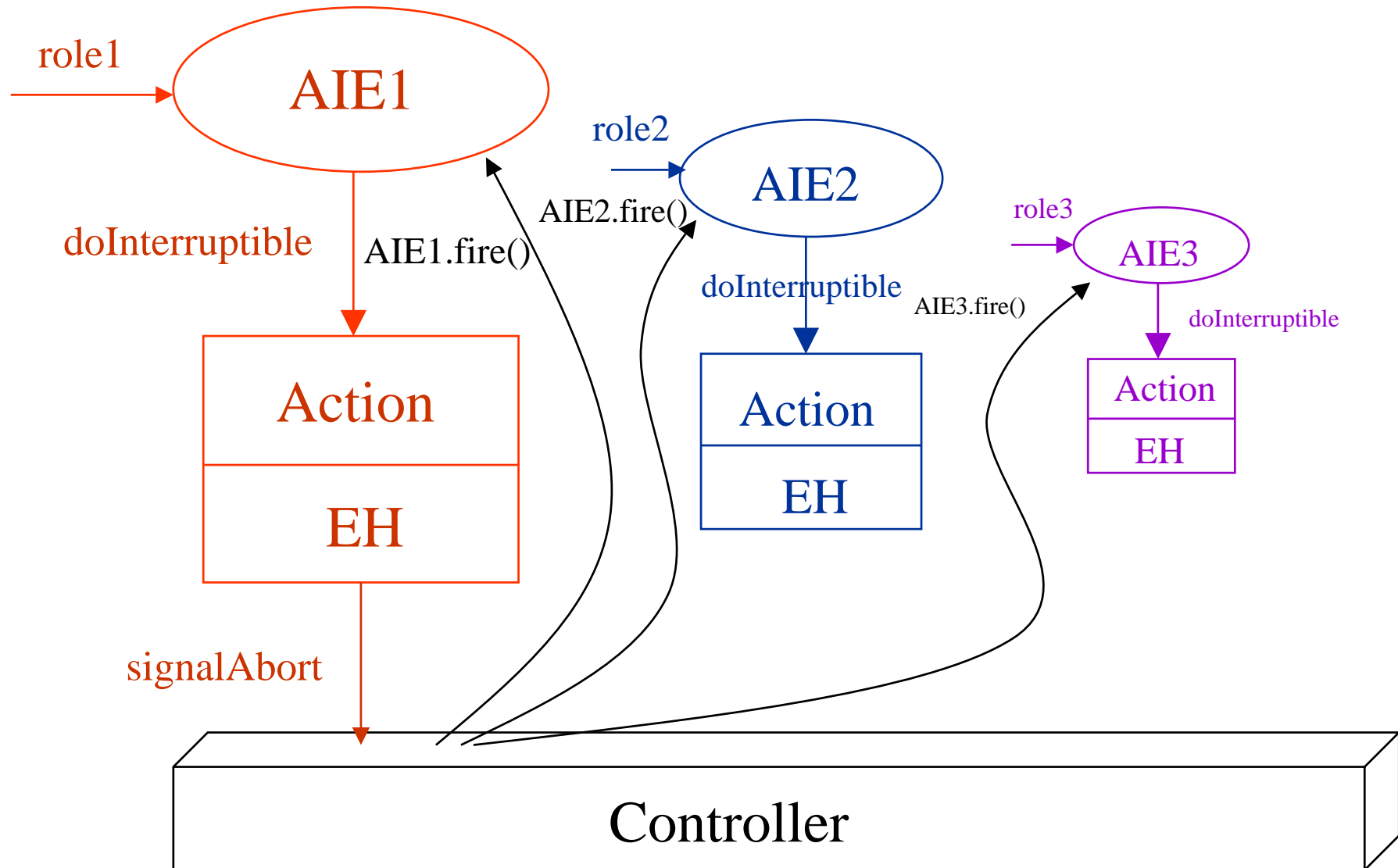
    public void resetTime(HighResolutionTime time);
}
```

# *RTJ and Atomic Actions*

- Consider forward error recovery, use same approach as Ada



# Use of doInterruptible



# *RTJ and Atomic Actions*

```
import javax.realtime.AIE;
```

```
public class AtomicActionException extends AIE
```

```
{
```

```
    public static Exception cause;
```

```
    public static boolean wasInterrupted;
```

```
}
```

} shared between  
objects of the  
class

```
public class AtomicActionFailure extends Exception
```

```
{};
```

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

```
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() { // constructor
        // 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();
    }
    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,  
                        AIE  
{  
    boolean Ok;  
    // no AIE until inside the atomic action  
    boolean done = false;  
    while(!done) {  
        try {  
            Control.first();  
            done = true;  
        } catch (InterruptedException e) {  
            // ignore  
        }  
    }  
}
```



Entry protocol

```

Ok = aael.doInterruptible
  (new Interruptible() {
    public void run(AIE e) throws AIE {
      try {
        // perform action
        // if necessary call e.disable() and e.enable()
        // defer AIE
        Control.done();
      }
      catch(Exception x) {
        if(x.getClass().getName() == "AIE")
          ((AIE) x).propagate();
        else
          Control.signalAbort(x);
      }
    }
  }
  public void interruptAction(AIE 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, AIE  
{// similar to role1 };
```

```
public void role3() throws AtomicActionFailure, AIE  
{// similar to role1 };
```

```
}
```

# Summary



- When processes interact, it is necessary to constrain their IPC so that recovery procedures can be programmed
- Atomic actions are 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 processes 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

# Summary

- A conversation is an atomic action with backward error recovery facilities
- On entry, the state of the process is saved; whilst inside, a process is only allowed to communicate with other processes active in the conversation and general resource managers
- In order to leave, all processes active in the conversation must have passed their acceptance test
- If any process fails its acceptance test, all processes have their state restored 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

# Summary

- Forward error recovery via exception handlers can also be added to atomic actions
- If an exception is raised by one process then all active in the action must handle it
- Two issues with this approach are the resolution of concurrently raised exceptions and exceptions in internal actions
- An asynchronous notification mechanism can be used to help program recovery
- POSIX provides signal and a thread cancelling mechanism
- A signal can be handled, blocked or ignored; unfortunately, it is not easy to program recoverable actions using a resumption model of asynchronous events



# Summary



- Ada and Real-Time provide an asynchronous transfer of control mechanism based on the termination model
- Ada's is built on top of the select statement
- RTJ is integrated into the exception and thread interrupt mechanisms
- This termination approach, in combination with exceptions, allows for an elegant implementation of a recoverable action