COMPUTING LABORATORY

Using objects and actions to provide fault tolerance in distributed, real-time applications.

S.K. Shrivastava and A. Waterworth

TECHNICAL REPORT SERIES
No 349 October, 1991
Using objects and actions to provide fault tolerance in distributed, real-time applications

S.K. Shrivastava and A. Waterworth

Abstract

In this paper, we develop an object-oriented model for structuring distributed real-time applications. Atomic actions (atomic transactions) and exception handling techniques are used to introduce fault tolerance. Additional techniques are then developed to permit application and device specific commit and abort processing. Objects can be replicated to increase their availability. We examine the reasons why some of the existing real-time object models are not suitable for active replication and why the model proposed here represents an improvement. Realistic examples are used to illustrate the practical utility of our approach.
SHRIVASTAVA, Santosh Kumar

Using objects and actions to provide fault tolerance in distributed, real-time applications.
[By] S.K. Shrivastava and A. Waterworth

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1991.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 349)

Abstract

In this paper, we develop an object-oriented model for structuring distributed real time applications. Atomic actions (atomic transactions) and exception handling techniques are used to introduce fault tolerance. Additional techniques are then developed to permit application and device specific commit and abort processing. Objects can be replicated to increase their availability. We examine the reasons why some of the existing real time object models are not suitable for active replication and why the model proposed here represents an improvement. Realistic examples are used to illustrate the practical utility of our approach.

About the author

Professor Shrivastava joined the Computing Laboratory in August 1975, where he is a Professor.
Adrian Waterworth gained his B.Sc. in Computing Science at the University of Newcastle upon Tyne in 1988, after which he became a Ph.D. student. He has recently became a Research Associate in the BAe Dependable Computing Systems Centre at Newcastle where he has worked since 1st October 1991.

Suggested keywords

ATOMIC ACTIONS
DISTRIBUTED SYSTEMS
EXCEPTION HANDLING
FAULT TOLERANCE
OBJECT-ORIENTED SYSTEMS
REAL-TIME SYSTEMS
REPLICATION

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404  001.6425
U.D.C.  519.687  681.322.06
Using Objects and Actions to provide Fault Tolerance in Distributed, Real-Time Applications.

S.K. Shrivastava and A. Waterworth.
Computing Laboratory, University of Newcastle upon Tyne, NE1 7RU, UK.

Abstract.

In this paper, we develop an object-oriented model for structuring distributed real-time applications. Atomic actions (atomic transactions) and exception handling techniques are used to introduce fault tolerance. Additional techniques are then developed to permit application and device specific commit and abort processing. Objects can be replicated to increase their availability. We examine the reasons why some of the existing real-time object models are not suitable for active replication and why the model proposed here represents an improvement. Realistic examples are used to illustrate the practical utility of our approach.


1. Introduction.

Real-time systems pose challenging problems in system structuring: services must be provided under stringent timing constraints and the continuous availability of those services is often required, despite underlying failures [1]. Not surprisingly, the problem of designing fault tolerant real-time systems has attracted much attention. In this paper, we develop an object-based model for structuring distributed real-time systems. We show that the atomic action (atomic transaction) based approach to the provision of fault tolerance can be extended to real-time systems. The practical utility of the approach promoted in this paper is then demonstrated by applying it to some realistic problems in real-time control.

2. Real-Time Objects and Atomic Actions.

The concept of software objects interacting by means of atomic actions forms the basis for a computational model which has been widely advocated for the construction of reliable distributed applications [2,3]. An object is an instance of some class and it consists of a number of instance variables, which define its internal state, and a set of operations or methods, which define its externally visible behaviour. Since the state of an object can only be accessed by invoking one of its methods and all inter-object communication takes place through such method invocations, the object-based structuring technique possesses a number of advantages. For example, transparent support for distributed processing can be provided by making use of a suitable remote procedure call (RPC) mechanism. More importantly, fault tolerance can be provided by using atomic actions to control all invocations. An atomic action possesses the properties of:

- **Failure Atomicity.** Actions either complete successfully (commit) having their intended effect or fail entirely (abort) having no effect.

- **Serializability.** The concurrent execution of two or more atomic actions is equivalent to some serial order of execution. (Non-interference of concurrent actions.)

and

- **Permanence of Effect.** The effects of committed actions are reflected in the
system state and not lost as a result of subsequent failures.

The use of atomic actions thus ensures that objects only undergo consistent state changes, even in the presence of failures.

Since the object and action model is considered suitable for structuring distributed applications concerned with the long term storage and manipulation of structured data (e.g. office information systems), it is natural to enquire whether the same structuring technique can be applied in the arena of real-time systems, where application software interacts with external devices (e.g. sensors and actuators). Our investigations, reported here, indicate that this is indeed the case, provided that certain enhancements are made. The nature of these enhancements is discussed in more detail below.

First of all, specific notations and supporting mechanisms are required for specifying and enforcing timing constraints (e.g. deadlines). Several researchers have investigated this area [4,5,6,7,8] and a number of suitable mechanisms are already well-known. Secondly, since objects in real-time systems need to interact with external devices, commit and abort processing for atomic actions may have to be application or device specific. For example, aborting an action which opened a valve may well involve closing that valve. Finally, a number of real-time applications are reactive in nature and are hard to structure in terms of clients and servers (the traditional approach to structuring distributed applications). In such reactive systems, an object often has to respond to external events while also carrying out its own local processing. Not surprisingly, this has led to the development of the concept of the active object – an object with one or more internal threads (or processes) which permit concurrent processing. For example, the ARTS real-time object model [7] permits multiple threads within an object. These threads rely upon the use of locks and critical regions to synchronize their activities when accessing shared data inside the object. A broadly similar approach has also been advocated in the MARUTI system [8].

Unfortunately, although such active objects are potentially useful in real-time applications software, they also introduce a problem. In many cases, objects may be replicated on distinct processing nodes within a distributed system to improve their availability. Furthermore, for real-time applications, the preferred replication strategy is often active replication, in which all functioning replicas of an object receive and process all requests. Active replication requires that all non-faulty replicas behave identically. This makes it necessary to identify all sources of non-determinism in replicas and ensure that non-deterministic decisions are resolved identically. The state machine model of active replication [9] gives one way of structuring computations such that this can be achieved. Taking objects to be state machines, the basic requirement is that the computations which take place within the method invocations of an object should be deterministic. This ensures that, if all the replicas of an object have identical states, then the execution of the same method at all the correctly functioning replicas will produce identical state transformations at these replicas, giving rise to identical results. Given this determinacy property, it is then only necessary to ensure that:

- if a correctly functioning replica executes a particular method, then all other correctly functioning replicas will also execute that method (agreement property)

and

- all correctly functioning replicas execute methods in an identical order (order property).

Since, in general, invocation messages may get lost and arrive in arbitrary order at replicas, the agreement and order properties are necessary for ensuring that (potentially) non-deterministic selection of method invocation messages is resolved identically at all the non-faulty replicas. These properties can be enforced in several ways. For example, in one architecture supported by the Delta-4 system [10], an atomic broadcast protocol is used for method invocations, simultaneously satisfying both agreement and order requirements, while an alternative Delta-4 architecture uses an explicit agreement protocol for method invocations and a separate “leader-follower” protocol for enforcing order [11].

Non-determinism caused by internal concurrency within an object (e.g. as a result of multiple threads) cannot be handled directly by the above protocols. Intuitively, it should be clear that, if an object contains several concurrent, interacting threads and that object is replicated on distinct nodes, then, unless the pattern of interactions between threads is made identical at all non-faulty replicas, the states of these replicas could diverge, defeating the purpose of replication. There is, however, no simple means of enforcing identical thread interactions, short of forcing identical, low-level scheduling decisions for all replicas. While this is a feasible approach, which has actually been demonstrated within the Delta-4 project, we feel that it relies too heavily upon being
able to modify low-level thread scheduling. We therefore propose a model which is not explicitly based on threads and which only needs the underlying support of the agreement and order protocols.

We require methods to be structured as atomic actions, so that their executions can be free from interference; further, atomic actions provide a neat method for dealing with exceptional conditions since an action can either commit, performing normal services, or abort, returning an exceptional response. We use background processes within active objects to accept specified method invocations and respond to device specific and other events such as deadline expiry and timeouts. The systematic use of such method selection techniques can then ensure that active replication of objects is possible.


A typical object declaration is shown in Figure 1. As well as containing declarations for the object's state variables and code for the object's methods, it can also include code for a number of object specific commit and abort operations. These operations can then be called from within any of the method's methods as shown.

```
OBJECT Example IS
STATE
  // State variables.
  COMMIT
    COMMIT_1 : // Code for device specific
                // commit.
  ABORT
    ABORT_1 :  // Code for device specific
                // abort.

METHOD P(...) SIGNALS e1, e2
  action
    ... if <Some Condition> then
    begin
      Abort ;       // Default abort.
      signal e1 ;
    end ;
    Some_Object.Q(...) [ Fail :
      begin
        Abort(ABORT_1) ;
        signal e2 ;
      end ] ;
    other_Object.R(...) ;
  ... end action : Commit(COMMIT_1)
                  [ Abort : signal Fail ] :
  ... // Other methods.
END.
```

Fig. 1. Object Declaration.

We assume that the default commit and abort operations perform state-based commit and abort processing (as in "traditional" atomic actions) and any action for which commit or abort processing has not been specified is assumed to use the default operations. Also, a few simplifying restrictions are imposed. We permit concurrent actions within an object, provided that they do not modify shared data; further, we assume that all atomic actions are top-level. Thus far, most of the real-time applications which we have studied seem to consist of sequences of top-level (mutually independent) actions, however it remains to be seen whether or not this is a general feature of the real-time applications domain.

When an error is detected, it is handled using the well-known termination model of exception handling [12, 13]. Every method of an object is expected to provide either a normal service (expected and desired) or, failing that, an abnormal service (expected, but undesired). The former constitutes the normal return of the called method whilst the latter defines the exceptional return. As stated before, method bodies are executed as top-level atomic actions, so a normal return from some method, P, would mean that the execution of P committed, while an exceptional return would mean that the execution of P aborted. In the above example, the method P is capable of signalling exceptional returns e1 and e2 to its caller. An exception is signalled after aborting the current activation of the method and the caller of the method can declare a specific handler for coping with signalled exceptions (see the handler for the Fail exception within the body of P). We assume that every procedure is capable of signalling a default exception Fail and that every procedure body has associated with it a default exception handler which aborts the action and signals the fail exception. This handler is explicitly shown in the example. The default handler is executed whenever an exception is encountered for which no specific handler is available. This would happen, for example, if the operation R were to return an exception.

Timing constraints upon operations are specified and enforced using a "temporal scope" notation (see Fig. 2) which is capable of raising a number of standard timing exceptions such as Deadline (indicating a missed deadline), Start_Time (indicating a missed start time) and Period (indicating the expiry of the current period for periodic tasks). Handlers can then be programmed for these exceptions in the usual way.
every Period_value do
begin
    Operation_A;
end [ Period : Exception_Handler ];

at Begin_Time do
    Operation_B;
finishby End_Time
    [ Start_Time : Handler_1;
      Deadline : Handler_2 ];

Fig. 2. Notations for timing constraints.

This particular method of expressing real-time constraints depends upon all objects within the system having a uniform notion of the passage of time, so we assume that some kind of global time base, possibly implemented with synchronized clocks, is available.

The object defined in Figure 1 is passive, as it has no process associated with it. Such an object acts as a simple server, executing method calls as they arrive from clients. If the object is replicated, then the underlying agreement and order protocols are expected to ensure that all replicas will execute identical methods in identical order. If the object is shared amongst several clients, then the methods of the object will also need to be protected by some form of concurrency control mechanism to ensure correct, serializable behaviour between method invocations from different sources. For the purposes of this paper and to save space, we assume that such concurrency control is being employed; we will be expanding upon this important aspect of the real-time programming problem in later works.

Frequently, an object needs to carry out internal processing while continuing to accept method invocations. To support this, an object may also include one or more processes. These may either perform simple background tasks or they may be more sophisticated processes which exercise control over the selection of method executions as shown below:

PROCESS
    select
        accept Method_i(...) \rightarrow ...;
        accept Method_j(...) \rightarrow ...;
        delay(Timeout_value) \rightarrow ...;
    end select;

Fig. 3. Select and Accept Constructs.

This notation is based upon the Ada select and accept primitives. The process waits for any of the specified method invocation messages to arrive, with an optional timeout. If several invocations are received, then any one of them is chosen for execution, after which the specified further processing is performed (which may involve further selects) and then the select statement is exited. In a separate paper, we have described how this type of select statement can be implemented in state machine based replicated systems [14]. Essentially, this involves forcing identical selection of method invocation messages at all non-faulty replicas. Expiry of the timer is also treated as an incoming method invocation message, so the entire select statement will be executed in an identical manner at all non-faulty replicas. The same basic technique can be used for the implementation of temporal scope commands since a timing constraint expressed as, for example:

do S finishby t
    [ Deadline : begin
        Abort;
        signal Fail
    end ];

permits an equivalent implementation of the form:

par
    S; send(null, self)
||
    select
        accept null \leftrightarrow skip;
        delay(t) \leftrightarrow begin
            Abort;
            signal Fail
        end;
    end select;
end par;

in which two parallel activities are created, one executing the statement S and sending a null message to itself and the other executing a timed select statement which will ensure that the statement S is executed before the expiry of time t or that the current action is aborted. It is important to realize that, for replicated real-time objects, timeout and deadline events must be agreed and ordered at all correctly functioning replicas in exactly the same way as method invocations. The need for some kind of mechanism like the one shown here is therefore unavoidable.

In the following section, we present some realistic examples to illustrate the practical utility of our overall approach.

4. Examples

In this section, we give three example applications, each illustrating some of the ideas developed earlier. For example, the call control application relies heavily upon the use of the select and accept
mechanisms to provide event handling. Asynchronous invocations of actions are also used to allow objects to initiate remote activities while continuing to carry out local processing. (Asynchronous invocations are specified by the caret character: e.g. `Call_Control.Answer_Call`.) On the other hand, the assembly line application makes use of programmed abort operations and only uses `select` and `accept` to enforce ordering constraints upon invocations. Similarly, the last example demonstrates the use of programmed aborts, but it also makes more extensive use of exception handling, both for timing constraints and method invocations.

Throughout all of these examples, internal calls to hardware operations are represented as ordinary invocations, but with their names in upper case. Hence, any operation which is listed in upper case (but which is neither a reserved word in this notation, nor the value of an enumerated type such as `CALL_IN_PROGRESS`) can be assumed to be a call to some low-level (usually hardware) routine. Also, all three examples happen to require only default (state-based) commit operations.

4.1. Example 1: Dedicated (Hotline) Communication System.

This example shows a simple call control application. A call controller object supports a dedicated bi-directional "hot-line" service between two telephones. The actual sequence of events when a call is made is as follows:

1. Receiver lifted at caller's telephone, initiating call.
2. Callee's telephone rings for a pre-determined length of time.
3. If the handset is lifted at the callee's telephone while it is still ringing, the two telephones are connected by the call controller until the call completes. Otherwise, the controller will stop the callee's telephone ringing when a pre-determined "ring-time" expires and disconnect both sides of the call.
4. If a call is successfully connected, it continues until such time as one (or both) of the parties hangs up. If only one of the participants clears the call in this way, a "Call-Disconnected" signal is sent to the other telephone.
5. If, for some reason, a caller hangs up just as a callee responds, the call is cleared by the controller and the "Call-Disconnected" signal is sent to the callee's telephone.
6. Finally, when a call is cleared by the controller (for example because it was not answered quickly enough), the "Call-Disconnected" signal is sent to both telephones to ensure that both parties replace their receivers.

In the following implementation, we have broken down the application software into two distinct object types - a pair of telephone objects (one for each telephone) and a single call controller object. The telephone objects offer an interface, consisting of the operations (methods) `Ring` and `Stop_Ring`, to the call controller. These operations allow the call controller to initiate or terminate ringing at a telephone. The latter operation also returns status information to the call controller, allowing it to deduce the current state of a telephone object. The other methods offered by a telephone object are `Receiver_Lifted` and `Receiver_Dropped`, both of which are assumed to be invoked internally in response to appropriate hardware signals and which allow the telephone object to start or answer calls and clear calls respectively. (The actual business of someone speaking into the telephone is not part of the control software and it takes place in the time between the receiver being lifted and subsequently replaced - i.e. between the Receiver_Lifted and Receiver_Dropped invocations.)

The call controller is an active object which offers methods for starting, answering and clearing calls and which takes care of the management of the actual connection between the caller and callee telephones. The process within the call controller object waits for an incoming `Start_Call` invocation (using the `accept` statement) and then, having received one, waits for the call to be answered, abandoned or to timeout using non-deterministic `select`. If the call is answered, the call controller waits for one of the parties to clear (terminate) the call and then forces the other party to disconnect, after which the controller begins a new cycle waiting for the next call to begin. Similar processing (waiting for a call to be cleared) is carried out by the controller if a call times out or is abandoned by the caller before the callee answers.

Note that the definition of the call controller shown here contains some features which relate to a more general call control application (that is, one supporting multiple calls between a number of telephones). In particular, the use of a database or directory to map telephone numbers to telephone object identifiers would only be required in the more general case.

Interface definition for telephone:

```plaintext
INTERFACE Telephone IS
  Ring:
  Stop_Ring returns Call_Status;
END.
```
Object definition for telephone:

```object
OBJECT Telephone IS
    STATE
        TelephoneNo MyNo :
        Call_Status Current_State :
    METHOD Receiver_Lifted
        action
            if Current_State = CALL_COWING_IN
                begin
                    STOP_RING :
                    Call_Control.Answer_Call :
                end
            else
                Call_Control.Start_Call(MyNo) :
                Current_State := CALL_IN_PROGRESS :
            end action :
    METHOD Receiver_Dropped
        action
            Call_Control.Clear_Call(MyNo) :
            Current_State := NO_CALL :
        end action :
    METHOD Ring
        action
            START_RING :
            Current_State := CALL_COWING_IN :
        end action :
    METHOD Stop_Ring returns Call_Status
        action
            if Current_State = CALL_COWING_IN
                begin
                    STOP_RING :
                    Current_State := NO_CALL :
                end :
            return Current_State :
        end action :
END.
```

Interface definition for call controller:

```interface
INTERFACE Call_Controller IS
    Start_Call( IN TelephoneNo ) :
    Answer_Call :
    Clear_Call( IN TelephoneNo ) :
END.
```

Object definition for call controller:

```object
OBJECT Call_Controller IS
    STATE
        DirectoryPhones :
        // Maps telephone numbers to
        // telephone object ID's.
        TelephoneCaller, Callee :
        // Hold caller and callee ID's
        // while call is in progress.
    METHOD Start_Call( IN TelephoneNo CallerNo )
        action
            CalleeNo := <The other telephone> :
            Caller := Phones[CallerNo] :
            Callee := Phones[CalleeNo] :
            CALL := CALL_IN_PROGRESS :
            CALL := CALL_COWING_IN :
            CALL := CALL_IN_PROGRESS :
        end action :
    METHOD Answer_Call
        action
            CONNECT(Caller, Callee) :
        end action :
    METHOD Clear_Call( IN TelephoneNo T )
        action
            DISCONNECT(Phones[T]) :
            if Phones[T] = Caller then
                Caller := NO_CONNECTION :
                else
                    Callee := NO_CONNECTION :
            end :
        end action :
    PROCEDURE Disconnection( IN Telephone T )
        begin
            CONNECT(T, DISCONNECTED_SIGNAL) :
            select
                accept Clear_Call →
                delay(D_Time) → DISCONNECT(T) :
                else
            end select :
        end :
END.
```

4.2. Example 2: Assembly Line Quality Controller.

In this example (taken from [15]), the system under consideration is an assembly line in a plant.
where containers of chemicals are processed. Occasionally, a container may be defective, in which case it must be carefully removed and discarded, preferably without stopping the line. This task is carried out by two robot arms which also serve the line in other capacities. Defective containers are detected by a Quality Control system, which then coordinates its activity with the two arms in order to lift the container from the line. In order to allow a faulty container to reach the arms, the lifting operation cannot begin until 5 seconds after detection. It must then be completed within 10 seconds of detection to make way for the next container to arrive. Hence, before a faulty container can be lifted, each arm must know that the operating conditions will allow it to lift the container within the specified deadline. If the container cannot be grasped correctly or cannot be lifted, then the assembly line must be safely stopped, the container removed manually and the line reset. Similarly, if the deadline expires while the arms are still in the process of lifting their load, the assembly line must be stopped and the arms cleared by an operator in order to prevent spillage or other hazards.

We have divided the system into separate Quality Monitor and Arm Controller objects. The former is an active object which continuously monitors the containers on the line. When a faulty container is detected, a local Remove Container operation is invoked, which calls parallel Prepare_Lift operations at the two arm controllers. If both of these preparatory operations succeed, then two parallel Perform_Lift operations are invoked, otherwise the line is reset. The expiry of the deadline for removing the container during either Prepare_Lift or Perform_Lift causes both arms to lock themselves in their current position and the quality monitor to shut down the assembly line. Since the arms also serve the line in other capacities as part of their normal function, select and accept are used within the arm controller objects to ensure that invocations other than Perform_Lift cannot be accepted after Prepare_Lift has been carried out.

The assembly line quality controller does not have an interface since it does not export any of its operations (all activity is internal). The object definition, however, is as follows:

**OBJECT Quality_Monitor IS**

**STATE**

Arm_Controller Arms[2] ;

**ABORT**

HALT : // Stop assembly line and alert

**METHOD Remove_Container( IN Time Start_Time,**

**IN Time My_DL )**

**METHOD**

integer i ; action

at Start_Time do

par for i := 1 to 2

Arms[i].Prepare_Lift(My_DL)

[ Fail : Abort(HALT) ] ;

end par ;

par for i := 1 to 2

Arms[i].Perform_Lift(My_DL) ;

end par

finishby My_DL [ Fail, Deadline :

Abort(HALT) ] ;

end action ;

**PROCESS**

begin

every Period do

if <Current container defective> then

Remove_Container(NOW+5, NOW+10) ;

end ;

END.

**INTERFACE Arm_Controller IS**

Emergency_Release :

Prepare_Lift( IN Time ) ;

Perform_Lift( IN Time ) ;

END.

**OBJECT Arm_Controller IS**

**STATE**

Position Current ;

**ABORT**

LOCK : // Lock arm in position.

**METHOD Prepare_Lift( IN Time My_DL ) action**

do

ARM_MOVE(<Pick up position>) : finishby My_DL[ Fail, Deadline :

begin

Abort(LOCK) ;

signal Fail

end ] ;

end action ;

**METHOD Perform_Lift( IN Time My_DL ) action**

do

ARM_GRAB ;

// Work out motion plan to raise

// arm to required position.

ARM_MOVE(<Motion plan>) ;

finishby My_DL [ Fail, Deadline :

begin

Abort(LOCK) ;

signal Fail

end ] ;

end action ;

**PROCESS**

begin

cycle
select
    accept Prepare_Lift →
      accept Perform_Lift :
        // Other robot arm services.
    end select ;
end cycle ;
END.

4.3. Example 3 : Controller for a Walking Robot.

This example is taken from [6] and deals with the A.S.V. (Autonomous Suspension Vehicle), a six-legged walking robot which moves by taking a sequence of steps determined by some overall motion plan. Each leg is controlled by a separate actuator and a robot step consists of moving legs forward individually and then pushing all of the legs backwards simultaneously. To maintain stability, certain constraints are placed upon the motion of the legs. First of all, legs 1, 2, 5 & 6 (the ones at the corners) can only be moved when all of the other legs are safely on the ground. However, legs 3 & 4 (the middle ones) can both be moved at the same time, so long as legs 1, 2, 5 & 6 are safely in place. If stability is threatened due to the violation of these constraints, recovery actions must be taken such as forcibly placing a failed leg on the ground. Similarly, if stability is threatened by an unbalanced load then all legs may have to be placed on the ground. There is also a requirement that each leg does not remain off the ground for longer than a specified amount of time, τ, in order to prevent excess strain on the robot body.

This application can be structured using two types of object: a master A.S.V. controller object and six identical leg controller objects. The master controller offers an interface which includes the operation Move. This operation takes a destination position and a deadline as its parameters and attempts to move the robot to the destination within the given time. The movement is performed as a series of Step invocations, which are local to the main controller and which each invoke six parallel Move operations, followed by six parallel Push operations on the leg controller objects. The leg movement operations obtain permission to move by calling a Move_Permission object (not shown here) which grants permission provided that the stability constraints for leg movement are not violated. Synchronization for the leg push operations is achieved by giving them a common start time (since all of the legs must push at the same time).

A number of programmed abort operations are used in both the master controller and the leg controller objects. These allow objects to perform operation based recovery such as a leg controller placing a failed leg on the ground or the A.S.V. controller re-attempting a failed Step operation.

Note that the master controller is an active object which performs background processing to monitor the overall stability of the A.S.V. In the event of the robot becoming unstable, this background process forces the A.S.V. to shut down and alerts the operator.

Interface definition for main A.S.V. controller:

INTERFACE A.S.V IS
    Move( IN Position, IN Time )
    SIGNALS Danger, Out_of_Time ;
END.

Object definition for main A.S.V. controller:

OBJECT A.S.V IS
    STATE
        Position Current ;
        Leg_Controller Legs[6] ;
    ABORT
        PANIC : begin
            HALT_ASV ;
            // Set off alarm to
            // alert operator.
        end ;
        STEP_FAILED :
            // Application specific
            // recovery operations.
METHOD Step( IN Position dest, IN Time My Deadline)
    SIGNALS Danger
action
    Position LegPos[6] ;
    integer i :
    Time Push_Time ;
    do
        par for i := 1 to 6
            LegPos[i] := // New position for leg i.
            Legs[i].Move(LegPos[i])
            [ Fail : // Record leg i move
              // failure. ] ;
        end par ;
    If <Any move failed> then
        Abort(STEP_FAILED) ;
    Push_Time := NOW + 8 ;
    par for i := 1 to 6
        Legs[i].Push(Push_Time)
            [ Fail : // Record leg i push
              // failure.]
            Push_Failure :
                signal Danger ] ;
    end par ;
    If <Any push failed> then
        Abort(STEP_FAILED) ;
finishby My Deadline
    [ Deadline: Abort(STEP_FAILED) ] ;
end action ;
METHOD Move( IN Position Dest, IN Time Master_DL )

SIGNS Danger, Out_of_Time

action
  Time i_deadline ; // Intermediate step deadline.
  Position i_pos ; // Intermediate step positions.
  boolean Step_OK ;

do while Current != Dest do
  begin
    i_deadline := // Function of Master_DL
    and current time
    i_pos := // Function of current and
              // destination positions.
    Step_OK := TRUE ;
  do Step(i_pos, i_deadline)
    [ Fail : Step_OK := FALSE ;
    Danger :
      If <ASV unstable> then
      begin
        Abort(PANIC)
        signal Danger ;
      end ;
    else
      Step_OK := FALSE
    ] ;
  finishby i_deadline
  [ Deadline : Step_OK := FALSE ; ] ;
  If Step_OK then
  Current := i_pos
  else
    // Calculate position.
  finishby Master_DL
end action ;

MOVE : begin
  If <leg clear of ground>
  then
    MOVE_LEG(LOWER) ;
    Current := // Final position
    // of leg
    Move_Permission.Release(My_Type) ;
  end ;

METHOD Move( IN Position Dest )

action
  If Move_Permission.Request(My_Type) != GRANTED
  then Abort ;
    // Now, move leg from current position to
    // destination position. Must perform move
    // within T time units to minimize stresses
    // on robot body.
  do
    MOVE_LEG(Dest) ;
  finishby (NOW+T) [ Fail, Deadline :
    begin
      Abort(MOVE) ;
      signal Fail ;
    end ] ;
  Move_Permission.Release(My_Type) ;
end action ;

METHOD Push( IN Time Start_Time )

SIGNS Push_Failure

action
  // At specified time, push leg backwards.
  at Start_Time do
  PUSH_LEG [ Fail :
    If <this is first failure>
    then
      begin
        Abort(PUSH) ;
        signal Fail ;
      end ;
    else signal Push_Failure ] ;
end action ;

END.

5. Discussion and Concluding Remarks.

The previous examples illustrate that real-time applications have to deal with a variety of exceptional cases. We have therefore based our ideas on a framework which permits well-known exception handling techniques to be applied to programs structured as atomic actions controlling operations on objects. Other researchers have also investigated the use of atomic actions in real-time systems [6,15]. While the approach suggested here is closer to that proposed in [6], we have tried to adopt a more systematic and flexible set of notations and we have been able to solve a number of real-time application problems. The timed atomic action technique developed in [15] gives an alternative approach to structuring a sub-class of client-server applications (such as the assembly line system), however our attempts to express the telephone and A.S.V.
applications using timed commitment were not successful. We believe that this was mainly due to the enforced coordinator-cohort structure of timed atomic commitment, coupled with the lack of provision for event handling. Furthermore, an attempt to program the telephone and assembly line applications using the CHAOS\textsuperscript{art} approach [6] was also unsuccessful. While this could, in part, be due to our being unfamiliar with the full range of mechanisms offered by the CHAOS\textsuperscript{art} system, we do feel that there is a need for a more expressive and flexible approach to timing constraints than that currently supported by CHAOS\textsuperscript{art}. It was this that led us to investigate the mechanisms reported here.

Throughout the development of our model, we have been careful to bear in mind the constraints imposed by active replication. We have shown how objects constructed according to the techniques proposed here can, in principle, be replicated and, in this respect, we have presented convincing arguments in favour of employing structured constructs which control the activities within an object.

We are carrying out a more detailed investigation of the approach presented here by studying a variety of other real-time examples. At the same time, a more detailed investigation of the performance implications of our approach and the use of atomic action structures for building fault tolerant applications (e.g. using coloured actions [16]) is being made, taking into account real-time constraints.

Acknowledgements.

This work has been supported in part by grants from the UK Science and Engineering Research Council, the ESPRIT Delta-4 project (No. 2252) and British Aerospace. Dr. H. Green and colleagues from GEC-Plessey Telecoms were kind enough to supply the telephone controller example. Thanks are also due to D. T. Seaton at Ferranti and our colleagues in the Computing Laboratory at the University of Newcastle upon Tyne for helpful discussions and comments.

References.


