NASA-CR-190636

An EVACS Simulation with Nested Transactions

David Auty
SofTech, Inc.

Collin Atkinson
UHCL

Charlie Randall
GHG Corporation

P.36

Release 01
June 30, 1992

(NASA-CR-190636) AN EVACS SIMULATION WITH NESTED TRANSACTIONS (Research Inst. for Computing and Information Systems) 36 p

N93-11717

Unclas

G3/61 0115104

Cooperative Agreement NCC 9-16
Research Activity No. SE.16
UHCL Subcontract 102

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

An EVACS Simulation with Nested Transactions

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Mr. David Auty of SofTech, Inc., Dr. Collin Atkinson of UHCL and Mr. Charlie Randall of GHG Corporation. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge III, Deputy Chief of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of UHCL, RICIS, NASA or the United States Government

MISSION

An EVACS Simulation with Nested Transactions

Release 01
June 30, 1992

University of Houston-Clear Lake Subcontract No. 102
NASA/JSC Cooperative Agreement NCC9-16
RICIS Project No. SE.16

Prepared by:

David Auty
SofTech, Inc.
1300 Hercules, Suite #105
Houston, TX 77058

Collin Atkinson
University of Houston-Clear Lake
2700 Bay Area Blvd
Box 444
Houston, TX 77058

Charlie Randall
GHG Corp.
1300 Hercules
Suite #111
Houston, TX 77058

An EVACS Simulation with Nested Transactions
Table of Contents

ii INTENTIONALLY BLANK PRECEDING PAGE BLANK NOT FILMED

An EVACS Simulation with Nested Transactions

1. Introduction

This report documents the recent effort of the MISSION Kernel Team on an EVACS simulation with nested transactions. The team has implemented the EVACS simulation [Atkinson92] along with a design for nested transactions. The EVACS simulation is a project wide aid to exploring Mission and Safety Critical (MASC) applications and their support software. For this effort it served as a trial scenario for demonstrating nested transactions and exercising the transaction support design.

The EVACS simulation is a simulation of some aspects of the Extra-Vehicular Activity Control System (EVACS), in particular, just the selection of communication frequencies. Its current definition is quite narrow, serving only as a starting point for prototyping purposes. (EVACS itself may be supplanted in a larger scenario of a lunar outpost with astronauts and a lunar rover).

Initially the simulation of frequency selection was written without consideration of nested transactions. This scenario was then modified to embed its processing in nested transactions. To simplify the prototyping effort, only two aspects of the general design for transaction support have been implemented: the basic architecture and state recovery.

The simulation has been implemented in the programming language Smalltalk. It consists of three components:

- Simulation support code which provides the framework for initiating, interacting and tracing the system.
- The EVACS application code itself, including its calls upon nested transaction support.
- Transaction support code which implements the logic necessary for nested transactions.

Each of these components deserves further description, but for now only the transaction support will be discussed.

2. A Transaction Taxonomy and Overview

An understanding of nested transactions comes from a progressive set of definitions. It begins with a relatively simple notion of actions and objects and adds complexity in several incremental steps. These steps include adding robustness to actions to form transactions, adding distribution to transactions and adding hierarchical nesting to transactions.

An *action* is a hierarchical composition of primitives (reads & writes) affecting several "objects" and which preserves system consistency. In its simplest form, an action is simply a read or write primitive affecting one object. More generally it consists of many reads and writes, affects many objects, and may be hierarchically composed of sub-actions. A primitive action inherently preserves consistency since it only affects one object. More complex combinations of primitives must preserve an overall consistency of system state in order to be properly considered as "actions".

An *object*, in this case, is a part of, or partition of, the total system state. In our primary reference on transactions [Moss85], an object is defined as a data item, but this concept of an object generalizes quite well to that of current object oriented definitions. A system is conveniently considered to consist of a collection of cooperating objects, each with potentially active and passive processing associated with them. An action can be equivalently defined as a unit of processing (a method or procedure) which interacts with many objects and which preserves a measure of consistency through its execution. Defining the measures of consistency, the steps which preserve consistency as well as the combinations of steps which may violate consistency temporarily, is an essential part of reliable system design.

A *transaction* is an action which exhibits *failure atomicity* and *serializability*. These two constraints provide the basis for constructing reliable systems out of multiple interacting actions. Failure atomicity refers to the property of either completing successfully or having no effect at all. This implies in the case of failure the restoration of objects which may have been altered during the transaction prior to the detection of failure. In practice, this can be achieved in many ways. [Moss85] describes two approaches as recovery from saved state and recovery via undo's, and presents details for the first of these which we will adopt. Maintaining recovery states is related to the technique of checkpointing known correct values as a computation proceeds. Recovery states are maintained in secondary storage which, depending on the degree of reliability required, may be itself duplicated or otherwise designed to maintain integrity (elsewhere referred to as stable storage or permanent storage).

Serializability refers to the nature of multiple actions which may interact through concurrent execution. If they are serializable, then one can establish after their completion a state which is equivalent to that which would be arrived at through some serial execution of the transactions. Stated another way, actions are serializable if they incorporate some mechanism of coordination which prevents their mutual corruption. Again this can be achieved in several ways. [Moss85] defines two approaches as access locking and timestamping with subsequent resolution. The approach we have taken uses access locking to ensure that a proper ordering of execution is achieved.

Object locking for transaction serialization is an extension to the common rules of locking for concurrency control. First note that we have chosen simple object reads and writes as primitives, thus the locking rules are for read/write access control. A read request is granted if no write request has been granted. A write request is granted if no other request, read or write, has been granted. Proper serializability requires further that no granting of access is released until all access is released when the action completes.

In summary, our approach to transactions requires the use of secondary storage to implement failure atomicity (recovery from failures as if the transaction never executed) and specialized object locking to ensure serializability (concurrent actions do not interfere).

A *distributed transaction* is a transaction which effects multiple objects at multiple sites. It adds to the paradigm of transaction processing the ability to recover from multiple and independently fallible processors and failed communications. Note that distribution of objects participating in a transactions does not require a change or extension to our general definition of transactions, i.e., distributed transactions obey the same rules for failure atomicity and serializability. Only the processing required to implement such transactions is modified. The modification consists of the addition of a *two-phase commit* protocol to ensure that all objects involved in the transaction are updated or reverted consistently.

The two-phase commit protocol requires that each participant object involved in the transaction first prepare to commit and respond that it is in fact prepared. Following successful processing of the preparation phase the transaction coordinator can logically toggle its own records to indicate commitment and broadcast this to all participants in the commitment phase. In this way, prior to commitment any participant not able to commit forces an abort. Following preparation all participants are able to fall forward or backwards. It is the singular action of the coordinator which transitions the transaction to commitment. Participants must then wait for the coordinator to signal which action they should take. In this way, assuming all node and communications failures are recoverable, no unrecoverable inconsistency of commitment or failure of the transaction can occur.

The Alpha kernel [Northcutt87] introduced, and we will assume for Mission as well, that all objects are truly independent; objects and messages can fail even though no physical distribution or node failure is involved. Thus, for the purposes of transaction processing, each object essentially becomes its own "virtual node". As a consequence of this perspective, any transaction requires the logic of distributed transactions (i.e., two-phase commits). Each object must handle its own participation in the transaction (i.e., handle enter_transaction, prepare_to_commit, complete_commit and abandon_transaction messages).

The final complexity which we add to this discussion is that of *nested transactions*. Nested transactions add the same feature of hierarchical composition as was defined for actions, allowing nested actions to be defined as nested transactions. The advantage of nested transactions is the partitioning of work being done which may require retries or alternative processing in the face of failure. If all processing which must commit or fail together must be executed as a single transaction, then failure requires reprocessing of the entire transaction. If instead the processing is broken into several sub-transactions, then failure of one sub-transaction can be handled independently of the other sub-transactions before signalling failure of the entire transaction. We still have the property that if the top-level transaction fails then all participants are restored as if no processing occurred, and we have the same property for the sub-transactions which allows for consistency of recovery within the transaction as well.

The introduction of nested transactions alters the general handling of transactions in two ways. First, the object locking rules must be modified to ensure proper coordination throughout the transaction and within the transaction. Secondly, recovery of nested transactions requires essentially a stack of recovery values being kept.

The change to the object locking rules relates to the handling of subtransaction completion. In normal transaction processing all object access required by the transaction is held until the transaction completes, and is then released. In the case of a subtransaction, the access restriction must be held until the entire top-level transaction completes. This is handled by having the subtransaction pass the object lock to its parent transaction for it to hold until completion. The parent may then pass the lock to its parent, if present, and so on until the top-level transaction is reached.

The second change to the object locking rules relates to the granting of access. Again, normal transaction access rules address "peer" level transactions attempting to access the same object. A special case exists if a subtransaction attempts to access an object which has already been accessed within a superior (e.g. parent) transaction. This can occur in two ways. It may be (a) that the object is required directly by a superior transaction and by the subtransaction, or it may be (b) that the object was required for a previous subtransaction. Case (a) is a difficult situation since it is not clear whether the superior transaction has completed its access in a consistent way at the time of the subtransaction's request for access. Unfortunately it is difficult to distinguish at runtime case (a) from case (b). Thus it is left either as a constraint on the programmer, as a constraint of the language, or to other pre-runtime controls not to implement case (a).

Case (b) is actually quite normal and acceptable. It requires, however, that the locking rules be defined to accommodate it. Note that at the end of the first subtransaction the object lock was passed up to the parent transaction. Thus when, during the second subtransaction, access to the object is requested the lock is owned by its parent. In this case, should be granted based on the possession of the lock by the parent. Generalized, the locking rules can be extended to the following:

- allow read access if all transactions holding a write lock are superiors of the subtransaction making the request, and
- allow write access if all transactions holding a lock in any mode are superiors of the subtransaction making the request.

The last note on nested transactions addresses the multiple levels of recovery required. For single level transactions a single recovery state is necessary for restoration. In the case of nested transactions, an object may be involved in several levels of nested transactions (e.g., case (b) just described). In fact, the rules of transaction participation and object locking prevent an object from participating in multiple transactions except when nested. Because an object may need to recover from a subtransaction failure prior to recovery from the parent transaction failure, a recovery state is necessary for the nested levels an object participates in as well as for the outer-most transaction level. A basic stack of recovery states meets this requirement.

3. A Design for Transaction Support

Our transaction design is based on two class definitions; objects of interest are either *transaction managers* or *transaction participants*. The application itself is defined as objects which inherit from the transaction participant class. This implies every application class is a subclass of the transaction participant class. The full processing of distributed nested transactions is incorporated into the definitions of these two object classes. This functionality includes two-phase commit, uniform recovery, concurrency control (lock management), lost-participant and manager recovery and schedulability / deadlock resolution. However, for the current prototype only the general architecture and recovery processing were implemented.

Note that the design was conceived with the idea in mind to eventually merge transaction semantics into the programming language itself. As a consequence and in consideration of existing languages (e.g., Smalltalk, Ada, Dragoon), it assumes a reasonable transformation of a "naive" application to one which incorporates transaction processing.

Transaction managers are defined to coordinate transaction participants and any subtransactions which are defined. Other than keeping a record of these participants and subtransactions, transaction managers are principally responsible for implementing the coordinator logic of two-phase commits as was described earlier. Transaction participants are defined to participate in transactions and, in particular, potentially nested transactions. Transaction participants are responsible for saving their current state, maintaining a stack of recovery states (in stable storage which can survive system crashes) and for properly responding to the various method calls associated with transactions: enter, prepare_commit, complete_commit and abandon_transaction.

The treatment of nested transactions deserves some special comment here. Our implementation of the transaction manager accommodates the situation of being nested within another transaction, but in general defines the processing to be identical for a sub-transaction as for a top-level transaction. This is possible partly because the treatment of state saving and recovery is handled by the participants. The singular addition required of a nested transaction manager is the passing of the participants list to the parent transaction manager.

Our design focuses more processing on the transaction participant. In particular, it is left to the participant to implement its own methods for saving and restoring its state. The transaction manager coordinates processing by issuing prepare_to_commit, complete_commitment or abandon_transaction commands, but does not receive or transmit participant states. Each participant thus keeps its own recovery stack.

A particularly significant aspect of the design is the dynamic nature of object participation. Objects participate in transactions when they are called upon, without any predefined list of participants being given to the transaction manager in advance. The process of entering into a transaction occurs as a part of calling an object. Prior to initiating the particular method of the call, the general transaction entry code is executed. Once entered, the object is a participant until the end of the transaction. The corresponding processing for leaving a transaction occurs at transaction commitment or abort.

Entering a transaction generally requires the saving of the current state of the object as a new entry on the recovery stack and notifying the transaction manager of the new participant. This is only done, however, if the object has not already participated in this transaction. The recovery state must always be the state of the object before any involvement in the transaction. To insure the recovery state is saved only once, a record is kept of the current transaction by each object. Thus as a part of transaction entry a comparison is made between the calling transaction and the current transaction. Only if they are different (the calling transaction is a subtransaction of the current transaction) is the state saved.

As was noted already, an object leaves a transaction at the time of transaction commitment or abort. Leaving a transaction implies poping the stack of recovery states. If the transaction commits the recovery value is tossed away. If the transaction aborts, the object assumes the recovery state as its current state, abandoning its previous current state.

There is a special case of leaving a nested transaction. If the transaction being left is nested (has a parent), then the object must be entered into the parent transaction. Again an entry check is made if the object had previously participated in the parent transaction. If this is the case then no further action should be taken. The object already has a recovery value from its earlier participation in the parent transaction on the stack which was made current when the subtransaction's recovery stack was popped.

If the object had not previously participated in the parent transaction (the subtransaction was first to call upon the object) then an entry into the parent transaction must take place. Note, however, that the recovery state to be pushed on the stack is the state of the object before its involvement in the subtransaction. This is the recovery value normally popped upon leaving the transaction. In fact, the recovery state needn't be popped at all (only to be pushed again), the recovery value can simply be left in place.

This processing ensures that all participants are kept in synchrony with the nesting of transactions which they are involved in. The recovery stack is not necessarily as deep as the nesting of transactions because the participants may not be entered into parent transactions until after a subtransaction commits or aborts. The process of being entered into the parent transaction as a part of leaving a nested transaction ensures that the proper set of recovery values is being maintained for each participant.

Appendix A - Bibliography

[Atkinson92]	Atkinson, Colin "Extending the EVAC System", A Working Paper for the Software Engineering Research Center on The MISSION Project, MISSION GEN/WP/101/01, Feb. 10, 1992.
[Moss85]	Moss, Elliot B. "Nested Transactions, An Approach to Reliable Distributed Computing", The MIT Press, 1985.
[Northcutt87]	Northcutt, J. D. "Mechanisms for Reliable Distributed Real-Time Operating Systems, The Alpha Kernel", Academic Press, 1987.

```
"
*****************************************************************************
Application : EVACS Simulation  -- modified to include transactions

A simulation of some aspects of the Extra-Vehicular Activity Control System
(EVACS).  In particular, this simulation looks only at the interaction between a
central controller and a set of MMUs, and more specifically at the selection
of communication frequencies.  The simulation has been extended to implement
frequency changes as a set of nested transactions. Changes must uniformly affect
both base station antennas and the Manned-Manuvering-Units (MMUs).  Different
scenarios of transaction success and failure can be run by having different
subtransactions of the scenario succeed or fail.

The simulation allows user control by the choice of frequency.  Each digit of the
three digit frequency controls one of the elements in the simulation and the
subtransactions it participates in.  In general values less than 5 succeed while
values 5 or greater fail.
   Digit 1 affects the central controller.
   Digit 2 affects the MMU.
   Digit 3 affects the antenna manager.
Also, digit 3 controls the antenna manager's antenna array.  These antennas
succeed if digit 3 is 4 or 5, but fail otherwise. For example:
  114Hz is complete success,
  914Hz is failure only of the central controller (root transaction)

Classes : EvacsRoot
            TransactionManager  PermanentStore
            TransactionParticipant
               Evacs
                 SimWindow  TextDisplayer  TextDisplayPane
                 CentralController  MMU  AntennaMgr  Antenna
          EvacsStack
Example : (Evacs new) start.

Classes are grouped into three categories:
  Transaction support,
  Simulation support, and
  Evacs application definition.
Classes definitions are presented in this order, then the class and instance
method definitions in the order:
  Simulation support,  Evacs application definition, Transaction support
which more closely presents the methods top-down in order of exection
*****************************************************************************
"!
```

```
" Transaction Support Classes " !

Object subclass: #EvacsRoot
  instanceVariableNames: ''
  classVariableNames: ''  poolDictionaries: ''
"   an empty class, no protocol or representation
    collects subclasses into one parent
" !

EvacsRoot subclass: #TransactionManager
  instanceVariableNames: 'id   participants   status
                          transactionHierarchy   subTs '
  classVariableNames: ''  poolDictionaries: ''
"   serves to coordinate transaction 2-phase commit and abort
  Class Methods
    runAsNewTransaction:id:parent:receiver:
  Instance Methods
    initWithID:,  setParents:,  processingComplete,   abort,
    registerParticipant:,  inheritParticipants:,  registerSubTransaction:,
    transactionHierarchy,  status
" !

EvacsRoot subclass: #TransactionParticipant
  instanceVariableNames: 'currentTM  permanentStore  status '
  classVariableNames: ''  poolDictionaries: ''
"   provides protocol and representation for objects which participate in
    transaction
  Class Methods
    new
  Instance Methods
    init,  currentState,  setStateTo:,  addState:,  restoreState:, prepared
    enter:, prepareCommitment,  completeCommitment,  abandonTransaction
" !

EvacsRoot subclass: #PermanentStore
  instanceVariableNames: ' currentState recoveryStack '
  classVariableNames: ''  poolDictionaries: ''
"   provides facility for saving an object's state & recovery states
  Class Methods
    new
  Instance Methods
    init,  push:,  update:,  pop,  readCurrent.  readTop
" !
" Simulation Support Classes " !

OrderedCollection subclass: #EvacsStack
  instanceVariableNames: ''   classVariableNames: ''   poolDictionaries: ''
"   subset of and renaming of orderedCollection methods, no new representation
  Class Methods (none)
  Instance Methods
    push:,  pop,  pushAll:,  readTop
" !

TransactionParticipant subclass: #Evacs
  instanceVariableNames: 'simWindow controller '
  classVariableNames: ''  poolDictionaries: ''
"   Collects subclasses into parent.  Defines shared representation
      (all subclasses get a reference to simWindow and controller).
    Defines method to initiate a simulation (start)
    All subclasses are potential transaction participants
" !

TextEditor subclass: #TextDisplayer
  instanceVariableNames: '' classVariableNames: '' poolDictionaries: ''
"   modified TextPane dispatcher, method modify always returns false
     (closing will not ask to have changes saved), no other changes
" !

TextPane subclass: #TextDisplayPane
  instanceVariableNames: '' classVariableNames: '' poolDictionaries: ''
"   modified TextPane, defaultDispatcherClass returns TextDisplayer
    no other changes
" !

Evacs subclass: #SimWindow
  instanceVariableNames: 'controllerFreq  antennaFreq  mmuFreq  inputPane
                          msgStream       displayPane '
  classVariableNames: ''  poolDictionaries: ''
"   Provides the display and interaction model for the simulation.
    Creates the window, panes (Input, Msg and Display) and menus objects
  Class Methods (none)
  Instance Methods
    externally called methods
      openWith,  antennaFreq:,  controllerFreq:,
                 anMMUFreq:,    textOut:
    internally called methods (called by window panes created by openWith)
      inputMenu, nullMsg,  defaultInput, displaySim:
      promptForFreq,  takeNewFreq,  callController:
" !
```

Evacs Simulation in SmallTalk
(class definitions)

```
" EVACS application classes " !

Evacs subclass: #CentralController
  instanceVariableNames: 'mmuArray antennaMgr frequency  mmusCount'
  classVariableNames: ''  poolDictionaries: ''
"   models the central controller (at base station) for the Evacs application
  Class Methods (none)
  Instance Methods
    setMaxMMUs:andSimWindow:,  currentState,  setStateTo:,  prepared,
    registerMMU:, changeFreq:
" !

Evacs subclass: #MMU
  instanceVariableNames: 'frequency number '
  classVariableNames: ''  poolDictionaries: ''
"   models behavior of an independent Manned Manuvering Unit
  Class Methods (none)
  Instance Methods
    setController:andSimWindow:,  currentState,  setStateTo:,  prepared,
    changeFrequencyTo:
" !

Evacs subclass: #AntennaMgr
  instanceVariableNames: 'antennaArray frequencyArray'
  classVariableNames: ''  poolDictionaries: ''
"   coordinates a collection of three antennas at the base station
  Class Methods
    newWith:
  Instance Methods
    setSimWindow:andMaxMMUs:,  antennaArray
    currentState,  setStateTo:,  prepared,  changeAntennasTo:
" !

Evacs subclass: #Antenna
  instanceVariableNames: 'frequencyArray '
  classVariableNames: ''  poolDictionaries: '' !
"   models behavior of an independent antenna at the base station
  Class Methods (none)
  Instance Methods
    setMaxMMUs:andSimWindow:andController:,
    currentState,  setStateTo:,  prepared,  changeFrequencyOfMMU:
" !
```

```
!Evacs class methods ! !
!Evacs methods !
"   Collects subclasses into parent.  Defines shared representation
      (all subclasses get a reference to simWindow and controller).
    Defines method to initiate a simulation (start)
    All subclasses are potential transaction participants
"
  start
    | maxMMUs |
    maxMMUs := 3.
    simWindow  := ( SimWindow new ).
    controller := ( CentralController new )
      setMaxMMUs: maxMMUs
      andSimWindow: simWindow.
    ( MMU new )
       setController: controller
       andSimWindow:  simWindow.
    ( MMU new )
       setController: controller
       andSimWindow:  simWindow.
    ( MMU new )
       setController: controller
       andSimWindow:  simWindow.

    ( simWindow openWith: controller ).
! !
! SimWindow class methods ! !
! SimWindow methods  !
"  Provides the display and interaction model for the simulation.
   Creates the window, panes (input, msg and display) and menus objects
  : contollerFreq  antennaFreq  mmuFreq  inputPane  msgStream  displayPane
  ***** externally called methods *****
  users of SimWindows can openWith, then update frequencies and write out msgs
"
  openWith: acontroller
    | topPane msgPane |
    controller := acontroller.
    controllerFreq := '0Hz'.
    antennaFreq    := '0Hz'.
    mmuFreq        := '0Hz'.
    topPane := (TopPane new) label: 'Kernel Simulation' .
    ( topPane addSubpane:
        ( inputPane     := (TextDisplayPane new)
            model: self;  name: #defaultInput;  menu:#inputMenu;
            framingRatio: (0 @ 0 extent: (2/3) @ (1/4)) )).
    ( topPane addSubpane:
        ( displayPane := (NoScrollGraphPane new)
             model: self;  name: #displaySim:;
             framingRatio: (0 @ (1/4) extent: (2/3) @ (3/4)) )).
    ( topPane addSubpane:
        ( msgPane      := (TextDisplayPane new)
             model: self;  name: #nullMsg;
             framingRatio: ((2/3) @ 0 extent: (1/3) @ 1) )).
    msgStream := ( msgPane dispatcher ).
    ( (topPane dispatcher) open; scheduleWindow  ).
!
  antennaFreq: newFreq
    antennaFreq := newFreq.
    ( displayPane update ).
!
  controllerFreq: newFreq
    controllerFreq := newFreq.
    ( displayPane update ).
!
  anMMUFreq: newFreq
    mmuFreq := newFreq.
    ( displayPane update ).
!
  textOut: aString
    ( msgStream nextPutAll: aString; cr ).
!
"continued"
```

```
" SimWindows methods continued "

" ***** internally called methods *****
  called by window panes (created by openWith:) at various times

  inputMenu establishes a menu providing command initiation for the user.
  User interaction in the system consists solely of selection of text in
  the inputPane and/or menu selection.  The menu provides two commands,
  implemented here by promptForFreq and takeNewFreq.  promptForFreq puts
  up a dialog box for user input. takeNewFreq takes whatever is currently
  selected in the inputPane as the user input.  promptForFreq and takeNewFreq
  are the only methods which call out from the window to the model, both
  calling controller.changeFreq
"
  nullMsg
      ^''.
!
  defaultInput
      ^( '999Hz 991Hz 919Hz 914Hz 199Hz 194Hz 119Hz 114Hz
          111Hz 115Hz 911Hz 915Hz' )
!
  inputMenu
    ^( (Menu labels: ( 'Prompt for Freq.\Selected New Freq'
                        breakLinesAtBackSlashes )
              selectors: #(promptForFreq takeNewFreq) )
         title: 'Operations' )
!
  promptForFreq
    | inputString |
    inputString := ( Prompter prompt:  'Please type desired frequecy'
                                 default: '114Hz' ).
    ( self textOut: ('initiating change to ', inputString) ).
    ( self callController: inputString ).
!
  takeNewFreq
    | inputString |
    inputString := ( inputPane selectedString ).
    ( self textOut: ('change requested, to ', inputString) ).
    ( self callController: inputString ).
!
"continued"
```

```
" SimWindows methods continued "

  callController: msg
    "with transaction code..."
    | success tm |
    tm := TransactionManager
            runAsNewTransaction:
              [ controller changeFreq: msg ]
              id:      'promptForFreq=>controller.changeFreq'
              parent:   currentTM
              receiver: controller.
    success := ( (tm status) = #completed ).
    ( self textOut: ('transaction success: ', (success printString)) ).
!
  displaySim: aRect
    | aPen afont aForm |
    aForm  := ( Form width: 600 height: 400 ).
    afont  := ( Font applicationFont ).
    aPen   := ( Pen new: aForm ).

    aPen place: 65 @ 25;
         centerText: 'CCU'          font: afont.
    aPen place: 50 @ 50; down; black;
         polygon: 35 sides: 4.
    aPen place: 66 @ 41;
         centerText: controllerFreq font: afont.

    aPen place: 115 @ 75;
         centerText: 'ANT'          font: afont.
    aPen place: 100 @ 100; down; black;
         polygon: 35 sides: 4.
    aPen place: 116 @ 91;
         centerText: antennaFreq    font: afont.

    aPen place: 165 @ 125;
         centerText: 'MMU'          font: afont.
    aPen place: 150 @ 150; down; black;
         polygon: 35 sides: 4.
    aPen place: 166 @ 141;
         centerText: mmuFreq        font: afont.

    ^aForm.
! !
```

```
!TextDisplayer class methods ! !
!TextDisplayer methods !
"   modified TextPane dispatcher, method modify always returns false
    (closing will not ask to have changes saved), no other changes
"
modified   "user modification not significant"
  ^ false
! !


!TextDisplayPane class methods ! !
!TextDisplayPane methods !
"   modified TextPane, defaultDispatcherClass returns TextDisplayer
    no other changes
"
defaultDispatcherClass
  ^ TextDisplayer
! !
! EvacsStack class methods ! !
! EvacsStack methods !
"   subset of and renaming of orderedCollection methods, no new representation
"
  push:  newObject
   ( super addFirst: newObject ).
!
  pop
   ^ ( super removeFirst )
!
  pushAll:  aCollection
   ( super addAllFirst: aCollection )
!
  readTop
    ^ ( contents at: startPosition ).
! !
```

```
!CentralController class methods  ! !
!CentralController methods !
"  models the central controller (base station) for the Evacs application
 subclass to: Evacs, subclass to: TransactionParticipant
  for Transaction Participant
  : currentTM  permanentStore  status
  for Evacs
  : simWindow   controller
  for CentralController
  : mmuArray  antennaMgr  frequency  MMUsCount
"
  setMaxMMUs:  maxMMUs
      andSimWindow:  aSimWindow
    mmuArray      := ( Array new: maxMMUs ).
    antennaMgr    := ( AntennaMgr new ) setSimWindow: aSimWindow
                                        andMaxMMUs:   maxMMUs.
    mmusCount     := 0.
    simWindow     := aSimWindow.
    controller    := self.
    ( antennaMgr antennaArray )
      do: [ :anAntenna |
        (anAntenna setMaxMMUs: maxMMUs
                   andSimWindow: simWindow
                   andController: controller).
     ].
!
  currentState
    ^ ( super addState:
          ( (Dictionary new) at:  #controllerSlot
                             put: frequency;
               yourself )).
!
  setStateTo: state
    ( super restoreState: state ).
    frequency    := ( state at: #controllerSlot ).
    ( simWindow controllerFreq: frequency ).
!
  prepared
    ^ ( (frequency at: 1) < $5 ). "1st digit of frequency < 5"
!
  registerMMU: mmu
    mmusCount := mmusCount + 1.
    ( mmuArray at: mmusCount put: mmu ).
    ^ mmusCount
!
"continued"
```

```
"CentralController methods continued"

  changeFreq: newFreq
    " Implements the essential function of EVACS sim, that of changing the
      frequencies of the MMUs and antennas in a coordinated fashion.  Changes
      are implemented as transactions to ensure integrity.  In the EVACS sim
      this method is also called as a top-level transaction thus all
      transactions here and subsequently created are sub-transactions.
    "
    | anMMU mmuNum tm success |
    frequency := newFreq.
    mmuNum := 1.
    anMMU  := ( mmuArray at: mmuNum ).

    tm := TransactionManager
            runAsNewTransaction:
              [ anMMU changeFrequencyTo: newFreq ]
              id:       'controller=>anMMU.changeFreq'
              parent:   currentTM
              receiver: anMMU.
    success := ( (tm status) = #completed ).
    ( simWindow textOut: ('transaction success: ', (success printString)) ).

    "to differentiate the MMUs from the antenna manager if the mmu fails, it
     is renentered into the parent transaction with a dummy value of 000Hz
    "
    (success)
      ifFalse: [
        ( anMMU enter: currentTM ).
        ( anMMU changeFrequencyTo: '000Hz' )].

    tm := TransactionManager
            runAsNewTransaction:
              [ antennaMgr changeAntennasTo: newFreq
                           for:              mmuNum ]
              id:       'controller=>antennaMgr.changeFreq'
              parent:   currentTM
              receiver: antennaMgr.
    success := ( (tm status) = #completed ).
    ( simWindow textOut: ('transaction success: ', (success printString)) ).

    ( simWindow controllerFreq: newFreq ).
! !
! MMU class methods ! !
! MMU methods !
"  models behavior of an independent Manned Manuvering Unit
 subclass of Evacs, subclass of TransactionParticipant
  for TransactionParticipant
  : currentTM  permanentStore  status
  for Evacs
  : simWindow   controller
  for MMU
  : number   frequency
"
  setController: theController
      andSimWindow:  theSimWindow
    controller := theController.
    number     := ( controller registerMMU: self ).
    simWindow  := theSimWindow.
!
  currentState
    ^ ( super addState:
          ( (Dictionary new) at:  #mmuSlot
                             put: frequency;
                yourself )).
!
  setStateTo: state
    ( super restoreState: state ).
    frequency   := ( state at: #mmuSlot ).
    ( simWindow anMMUFreq: frequency ).
!
  prepared
    ^ ( (frequency at: 2) < $5 ).
!
  changeFrequencyTo: newfrequency
      frequency := newfrequency.
      ( simWindow anMMUFreq: frequency ).
      ( simWindow textOut:
         ('This is an MMU and Im changing frequency to', frequency) ).
! !
! AntennaMgr class methods !
  newWith: aSimWindow
    ^ (super new) initWith: aSimWindow
! !
! AntennaMgr methods !
"   coordinates a collection of three antennas at the base station
 subclass of Evacs, subclass of TransactionParticipant
  for TransactionParticipant
  : currentTM  permanentStore  status
  for Evacs
  : simWindow   controller
  for antennaMgr
  : antennaArray   frequencyArray
"
  setSimWindow: aSimWindow
      andMaxMMUs: maxMMUs
    simWindow      := aSimWindow.
    antennaArray   := ( Array with: (Antenna new)
                              with: (Antenna new)
                              with: (Antenna new) ).
    frequencyArray := ( Array new: maxMMUs ).
!
  antennaArray
    ^ antennaArray
!
  currentState
    ^ ( super addState:
          ( (Dictionary new) at:  #antennaMgrSlot
                             put: (frequencyArray shallowCopy);
                yourself )).
!
  setStateTo: state
    ( super restoreState: state ).
    frequencyArray := ( state at: #antennaMgrSlot ).
    ( simWindow antennaFreq: (frequencyArray at: 1) ).
!
  prepared
    ^ ( (((frequencyArray at: 1) at: 3) = $4)
        | (((frequencyArray at:1) at: 3) = $5) ).
!
"continued"
```

```
"AntennaMgr methods continued"

  changeAntennasTo: newFreq
      for: anMMU
    | tm anAntenna |
    ( frequencyArray at: anMMU put: newFreq ).
    ( 1 to: 3 )
      do: [ :num |
        anAntenna := ( antennaArray at: num ).
        tm := TransactionManager
                runAsNewTransaction:
                  [( anAntenna  changeFrequencyOfMMU: 1 to: newFreq )]
                  id:      'antennaMgr=>anAntenna.changeFreq'
                  parent:   currentTM
                  receiver: anAntenna.
      ].
    ( simWindow antennaFreq: newFreq ).
! !
! Antenna class methods ! !
! Antenna methods !
"   models behavior of an independent antenna at the base station
 subclass of Evacs, subclass of TransactionParticipant
  for TransactionParticipant
  : currentTM  permanentStore  status
  for Evacs
  : simWindow   controller
  for Antenna
  : frequencyArray
"
  setMaxMMUs: maxMMUs
      andSimWindow:  aWindow
      andController: aController
    frequencyArray := ( Array new: maxMMUs ).
    simWindow      := aWindow.
    controller     := aController.
!
  currentState
    ^ ( super addState:
          ( (Dictionary new) at:  #antennaSlot
                             put: (frequencyArray shallowCopy);
                 yourself )).
!
  setStateTo: state
    ( super restoreState: state ).
    frequencyArray := ( state at: #antennaSlot  ).
    ( simWindow textOut:
        ('This is anAntenna, change be done to ', (frequencyArray at: 1)) ).
!
  prepared
    ^ ( ((frequencyArray at: 1) at: 3) < $5 ).
!
  changeFrequencyOfMMU: number to: frequency
      ( frequencyArray at: number put: frequency ).
      ( simWindow textOut:
          ('This is anAntenna, change be done to ', frequency) ).
! !
! TransactionManager class methods  !
  runAsNewTransaction: block
           id:         userId
           parent:     parentTransaction
           receiver:   participantObject
  " Creates transaction, executes block within it and invokes completion
    processing.  Receiver must be object receiving message in block "
    | newTM |
    newTM := (super new)
              initWithID: userId;
              setParents: parentTransaction.
    ( participantObject enter: newTM ).
    ( block value ).  "execute the transaction's code"
    ( newTM processingComplete ).
    ^ newTM
! !

! TransactionManager methods !
"   serves to coordinate transaction 2-phase commit and abort
  : id   participants   status   transactionHierarchy   subTs
"
  initWithID: userId
    " sets user id (string) and initializes collection variables and status "
    id           := userId.
    subTs        := ( Bag new ).
    participants := ( Set new ).
    status       := #created.
!
  setParents: parentTransaction
    " sets transaction hierarchy, including all parents and itself "
    transactionHierarchy := (EvacsStack new).
    ( parentTransaction notNil ) ifTrue: [
      ( transactionHierarchy pushAll: (parentTransaction transactionHierarchy) ).
      ( parentTransaction registerSubTransaction: self )
      ].
    ( transactionHierarchy push: self ).
!
" continued... "
" transactionManager methods cont. "

  processingComplete
    " initiate two phase commit:  send prepare message to all participants
      if participants all prepared, commit and send complete messages "
    | success parentTM |
    status  := #preparing.
    success := true.  "for now"
    participants  do: [ :participant |
      success := success & ( participant prepareCommitment ) ].
    (success)
      ifTrue: [
        status   := #committed.  " the binary arbiter of commitment "
        ( transactionHierarchy pop ).
        parentTM := ( transactionHierarchy readTop ).
        participants  do: [ :participant |
          ( participant completeCommitment: parentTM ) ].
        ( parentTM notNil ) ifTrue: [
          ( parentTM inheritParticipants: participants ) ].
        status := #completed.
      ]
      ifFalse: [( self abort )].
!
  abort
    " send abandonTransaction message to all participants "
    status := #aborted.
    participants do: [ :participant |
      ( participant abandonTransaction ) ].
!


  registerParticipant:  participantObject
    ( participants add: participantObject ).
!
  inheritParticipants: subTparticipants
    ( participants addAll: subTparticipants ).
!
  registerSubTransaction: transactionManager
    ( subTs add: transactionManager ).
!
  transactionHierarchy
    ^ transactionHierarchy
!
  status
    ^ status
! !
! TransactionParticipant class methods  !
  new
    ^(super new) init.
! !

! TransactionParticipant methods  !
"   provides protocol and representation for objects which participate in
    transaction
  : currentTM  permanentStore  status
"
  init
    permanentStore := ( PermanentStore new ).
    status         := #free.
!
  currentState        "returns state as collection of state information"
    "( self implementedBySubclass )"
    ^ self addState: (Dictionary new)  "subclass state, class state"
!
  setStateTo: state   "sets state to contents of stateCollection"
    "( self implementedBySubclass )."
    self restoreState: state.
!
  addState: state     "adds this class' instance variables to state object"
    ( state at:  #participantSlot
            put: (Array with: status
                        with: currentTM) ).
    ^ state
!
  restoreState: state "restores this class' instance variables"
    | thisClassState |
    thisClassState := (state at: #participantSlot).
    currentTM   := thisClassState at: 2.
    status      := thisClassState at: 1.
!
  prepared            "returns true if object is ready to commit"
    "( self implementedBySubclass )."
    ^ true  "by default"
!
"Continued"
```

```
"TransactionParticipant methods continued"
  enter: enteredTM
    | aCurrentState |
    "enter into transaction, if not current transaction save state"
    ( currentTM ~= enteredTM ) ifTrue: [
        ( permanentStore push: (self currentState) ). "recovery value"
        status     := #inTransaction.
        currentTM  := enteredTM.
        ( enteredTM registerParticipant: self ).
      ].
    ( permanentStore update: (self currentState) ).  "current value"
!
  prepareCommitment
    "check status, if ok prepare for commitment"
    ^ ( (self prepared) ifTrue: [
            ( permanentStore update: (self currentState) ).
            status := #prepared.
          ];
          yourself )
!
  completeCommitment: parentTM
    ( self restoreState: (permanentStore readTop) ).
       "class variables only, not newly calculated subclass variables"
    ( parentTM notNil )
      ifTrue: [ "enter parent transaction"
      " the following is equivalent to leaving the current transaction (pop)
        and entering the parent transaction (push if not current transaction)
        but inverted as an optimization (pop if parent is current transaction)"
        ( currentTM = parentTM )
          ifTrue:  [ ( permanentStore pop ). ]
          ifFalse: [ currentTM := parentTM.
                     status    := #inTransaction ]
      ]
      ifFalse: [ "leave current transaction"
        ( permanentStore pop )
      ].
!
  abandonTransaction
    | recoveryState |
    recoveryState := ( permanentStore pop ).
    ( self setStateTo: recoveryState ).
    ( permanentStore update: recoveryState ).
! !
! PermanentStore class methods  !
  new        ^(super new) init.
! !

! PermanentStore methods  !
"   provides facility for saving an object's state & recovery states
  : currentState   recoveryStack
"
  init
    recoveryStack := (EvacsStack new).
!
  push: recoveryState
    recoveryStack push: recoveryState.
!
  update: newCurrentState
    currentState := newCurrentState.
!
  pop        ^ recoveryStack pop
!
  readCurrent ^ currentState
!
  readTop    ^ (recoveryStack readTop)
! !


!CentralController methods !
  changeFreq: newFreq
    | anMMU mmuNum tm success anAntenna |
    frequency := newFreq.
    mmuNum := 1.
    anMMU  := ( mmuArray at: mmuNum ).

    " regardless of antenna manager's success or failure,
      antenna 1 will be entered into transaction directly  "
    anAntenna := ((antennaMgr antennaArray) at: 1).
    tm := TransactionManager
            runAsNewTransaction:
              [ anAntenna changeFrequencyofMMU: 1 to: newFreq ]
            id:      'controller=>anAntenna.changeFreq'
            parent:   currentTM
            receiver: anAntenna.

    tm := TransactionManager
            runAsNewTransaction:
              [ anMMU changeFrequencyto: newFreq ]
            id:      'controller=>anMMU.changeFreq'
            parent:   currentTM
            receiver: anMMU.
    success := ( (tm status) = #completed ).
    ( simWindow textOut: ('transaction success: ', (success printString)) ).

    "to differentiate the MMUs from the antenna manager if the mmu fails, it
     is renentered into the parent transaction with a dummy value of 000Hz "
    (success)
      ifFalse: [
        ( anMMU enter: currentTM ).
        ( anMMU changeFrequencyto: '000Hz' )].

    tm := TransactionManager
            runAsNewTransaction:
              [ antennaMgr changeAntennasTo: newFreq
                           for:              mmuNum ]
            id:      'controller=>anMMU.changeFreq'
            parent:   currentTM
            receiver: antennaMgr.
    success := ( (tm status) = #completed ).
    ( simWindow textOut: ('transaction success: ', (success printString)) ).

    ( simWindow controllerFreqChangedto: newFreq ).
! !
```

```
"construct application"
( (Smalltalk at: #Application ifAbsent: [])
    isKindOf: Class ) ifTrue: [
        ((Smalltalk at: #Application) for:'.Evacs Sim')
            addClass: EvacsRoot;
            addClass: TransactionManager;
            addClass: TransactionParticipant;
            addClass: PermanentStore;
            addClass: EvacsStack;
            addClass: Evacs;
            addClass: SimWindow;
            addClass: TextDisplayer;
            addClass: TextDisplayPane;
            addClass: CentralController;
            addClass: MMU;
            addClass: AntennaMgr;
            addClass: Antenna;
            comments: nil;
            initCode: nil;
            finalizeCode: nil;
            startUpCode: nil
        ]
!
```