Mission and Safety Critical (MASC)

An EVACS Simulation with Nested Transactions

N93-20314

(NASA-CR-192295) MISSION AND SAFETY CRITICAL (MASC): AN EVACS SIMULATION WITH NESTED TRANSACTIONS Interim Report (Research Inst. for Computing and Information Systems) 47 p

Unclas

G3/54 0148095

Project Report

GRANT
IN-54-CR
148095
P. 47

David Auty
SofTech, Inc.
Colin Atkinson
UHCL
Charlie Randall
GHG Corporation

Release 02: 9//92

Cooperative Agreement NCC 9-16
Research Activity No. SE.16

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

ricis

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

INTERIM REPORT: First complete draft

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

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.

MISSION

An EVACS Simulation with Nested Transactions Project Report

Release 02
September, 1992

University of Houston, Clear Lake
RICIS Project No. SE.16

Name:	An EVACS Simulation with Nested Transactions
ID:	MISSION / Kernel Team / Project Deliverable (PD) / #103 / ver. 02
Date:	September, 1992
Status:	First complete draft
Derivation:	ver. 01 + revisions, missing sections, code revisions
Author(s):	David Auty (SofTech), Colin Atkinson (UHCL), Charlie Randall (GHG)
Remarks:	

An EVACS Simulation with Nested Transactions
Table of Contents

An EVACS Simulation with Nested Transactions

1. Introduction

This report documents the EVACS Simulation with Nested Transactions, a recent effort of the MISSION Kernel Team. The EVACS simulation is a simulation of some aspects of the Extra-Vehicular Activity Control System, in particular, just the selection of communication frequencies. The simulation is a tool to explore mission and safety critical (MASC) applications. For the purpose of this effort, its current definition is quite narrow serving only as a starting point for prototyping purposes. (Note that EVACS itself has been supplanted in a larger scenario of a lunar outpost with astronauts and a lunar rover). The frequency selection scenario has been modified to embed its processing in nested transactions. Again as a first step, only two aspects of transaction support have been implemented in this prototype: architecture and state recovery. Issues of concurrency and distribution are yet to be addressed.

The simulation has been implemented in 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.

2. A Transaction Taxonomy and Overview

Our understanding of 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.

Actions

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

An *object*, in this case, is a part of, or partition of, the total system state. In Moss's work this is primarily 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 is 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.

Transactions

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. Moss 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 interaction between multiple transactions which may begin execution concurrently. 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, i.e., there is no significant interleaving of their execution which would corrupt system coherency. 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. Moss 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.

Access 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 definiton of transaction adds to the definition of actions 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, (that concurrent actions do not interfere).

Distirbuted Transaction

A *distributed transaction* is a transaction which effects multiple objects at multiple sites. It adds to the paradigm of transaction processing the ability to handle independently fallable 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 failures are recovered, no inconsistency of commitment or failure of the transaction can occur.

The Alpha kernel 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 procesing, 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).

Nested Transactions

The final complexity which we add to this discussion is that of *nested transactions*. Nested transactions add the same feature of heiarchical 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 first 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 passing the object lock to the 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 accomodate 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 it should be granted based on the possession of the lock by the parent. Generalized, the locking rules are modified to:

- 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 if necessary. 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 of the possibility of being involved in transactions at multiple levels, a recovery state is necessary for the outer-most transaction level an object participates in as well as for the nested levels. An object may need to recover from a subtransaction failure prior to recovery from the parent transaction failure. 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.

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, 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 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. Only the general architecture and recovery processing are discussed here.

Transaction Managers

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.

Transaction Participants

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 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.

Participant Entry Into a Transaction

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.

Leaving a Transaction

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 previous 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 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.

4. The Evacs Application

Emphasis has been placed on the construction of a simulation which illustrates the value of distributed nested transactions in a scenario of interest to NASA. The system which was eventually adopted is a simplified version of the type of system which will be required to monitor extra vehicular activities by astronauts from the future space station. We have chosen to call this the "Extra Vehicular Activity Control System" or EVACS for short. For the purpose of demonstrating DNTs we focus on a particular subsystem of the EVACS which is responsible for maintaining communication between astronauts and the space station. This is achieved through a highly simplified model, which is not intended to provide a realistic simulation of space station communications.

Communication between the astronauts and the space station is assumed to be via radio. Since astronauts may move freely around the space station, it is necessary to have several antennas, all transmitting the same signal, distributed around the exterior of the station. This ensures that no regions around the station will be cut from the radio signals. Each antenna is assumed to be controlled by an independent antenna software module.

Astronauts maneuver around the space station with the aided of a special "back-pack" known as a Manned Maneuvering Unit (MMU). This is assumed to contain a microprocessor running a software module which, in addition to monitoring the environment within the space suit and responding to the astronaut's commands, is responsible for maintaining appropriate communication channels with the space station. The final software object in the system is the central control unit, which is located at the central computing site on the space station, and is responsible for providing an interface to human monitors and controllers of the system.

For the purposes of communication, each active MMU is allocated a unique frequency. When a message needs to be transmitted from the station to an MMU, all antennas will simultaneously broadcast the message at the appropriate frequency. It is crucial therefore, that all the antennas, as well as each MMU, know and agree upon the frequency allocation. Should a mismatch arise for any reason, communication will be permanently lost, with no prospect of recovery.

Distributed Nested Transactions become of value when we consider the problem of changing the frequency allocated to a currently active MMU. There may be several reasons why such a change may be necessary. Perhaps a solar radiation burst is expected at that frequency, or interference is experienced from a piece of equipment. Assuming such a change is required, the main challenge is to perform the change in such a way that the system is not left in an inconsistent state as a result of some failure. Without DNTs there would be a real danger of this, since the job of updating the frequency involves updates to several independent objects. To guard against this possibility therefore, the change frequency operation is programmed as a distributed nested transaction.

The top-level transaction assumed to be an operation of the central control unit, which is responsible for determining when a change is required and what the new frequency should be. Usually the human operator will enter this information. The Central Control Unit is then responsible for informing each of the antennas and the MMU concerned and passing the appropriate information.

5. Simulation Support

The Smalltalk simulation draws upon several class definitions which provide simulation support. The category of simulation support includes:

- simulation startup,
- fault simulation (failure of objects involved in transactions)
- input events & parameterization (frequency selection)
- an active display of simulation objects
- an activity log.

These will be discussed in order.

The simulation startup (class Evacs) creates and initializes the top-level simulation objects. These objects are the simulation window (SimWindow), the central controller and a fixed number of MMUs (3). In general, initialization consits of linking each of these objects to each other (bi-directionally), except that the SimWindow does not requrire reference to the MMUs. Links are established from:

- the simWindow to the central controller,
- the MMUs to the simWindow and central controller, and
- the central controller to the simWindow.

The MMUs then register themselves with the central controller completing the link from central controller to MMUs. Once these objects are created and linked by initialization code the simulation is ready for execution. It will remain idle, though, until the user initiates an input event.

Fault simulation is an essential element of the simulation in order to exercise and demonstrate the details of transaction processing. Fault simulation is handled through specific evaluation of the frequency selected by the user. As part of transaction processing, each object participating in a transaction evaluates a function that determines the success or failure of that participant's involvement in the transaction. For our simulation, this function looks at the frequency selected and evalutates success or failure based on the frequency value. In particular, each participant class looks at a specific digit of the frequency for its evaluation. In general, a digit less than five generates success while digits five and greater generate failure. Thus frequencies like 111 and 222 are processed without failure, while 911, 191 and 119 each generate a failure in different classes in the system.

Processing within the simulation is driven by user initiated input events. Currently the simulation provides only the one event: change frequency for MMU #1. This event is in fact initiated by the user's clicking on one of two buttons. The processing for these buttons differ only in how the event is parameterized, i.e., how the user designates what the new frequency is to be. The change frequency event is also parameterized in terms of the simulation mode of operation as follows:

- proper: the simulation will treat sub-transaction failure as would be expected of the scenario being simulated; any sub-transaction failure will lead to the failure of the parent transaction,
- improper: the simultation will ignore sub-transaction failure in determining the success or failure of the parent transaction. While not realistic in terms of what would be expected for this scenario, improper behavior demonstrates more fully the nature of system-level nested transaction support.
- tough_case: this mode of behavior is an exageration of improper behavior in order to demonstrate a particulalry difficult aspect of nested transaction support. (See description following).

The window panes (a rectangular region of the window) required to support this one event and its parameterization include a button-pane with two push-buttons (non-persistent selection) for event initiation, a button-pane with three radio buttons (only one selected at a time) for mode selection and a text-pane for display of typical frequencies which the user can select from. The two event initiation buttons allow either frequency selection in the text-pane prior to button-pressing or frequency input to a promptor-window (dialog box in Macintosh lingo) following button-pressing.

The last two simulation support elements are provided in two output window panes: the simulation display (active display) and activity log. The simulation display is a graphic display containing rectangles on the scren for each of the principle simulation elements (central controller, antennas, antenna manager, and MMUs). Each rectangle has the object's name and its current frequency. As the change frequency event is processed, the user can see the objects' frequencies change.

The activity log is a simple text-pane with messages added throughout simulation processing. The messages include frequency change initiation, specific object frequency changes, transaction initation and transaction success/failure indications.

6. The Implementation's Use of Smalltalk

The simulation was implemented in Smalltalk for two reasons: it is an object oriented language and it provides significant rapid-prototyping support. For this simulation extensive use was made of pre-defined classes, including:

- object aggreagtions or collections: Dictionaries, Sets and Stacks,
- graphic display: Pen and GraphPane,
- the TextPane and TextEditor,
- the ButtonPane for initiating and parameterizing the user's change frequency request, and
- the Prompter class for putting up dialog boxes for the user.

In particular, however, it is likely that the Smalltalk windowing classes need additional discussion to clarify their processing.

6.1 An Introduction to Smalltalk Windowing Mechanisms

In general, the inteface between the window pane classes and the application is defined by three standard behaviors: changed-update, update-perform:name and user_action-perform:changeSelector described below. Note, however, that each subPane class elaborates and extends upon these standard protocols.

The Model-Pane-Dispatcher Framework
All predefined windowing classes in Smalltalk-V fit into the model-pane-dispatcher framework of inter-related objects. (Smalltalk-V is referenced here in place of the more general Smalltalk becaused of its unique handling of windowing). This corresponds directly to the original model-view-controller framework of Smalltalk-80, but the names are changed to reflect the significant modifications made in adapting the framework to the various Smalltalk-V platforms.

Each region of a window is handled by a Pane object. User interactions with this region (keystorkes, mouse clicks & drags, menu selections, etc.) are handled by a corresponding Dispatcher object. Both these objects are created by and interact with the model object. The richness of predefined windowing classes allows users to write simple model classes for their application and leave the compllexities of windowing to the predefined classes. The model object interacts primarily with the pane object and this will be the focus of the remaining discussion. Pane objects have their own interaction with the dispatcher objects but these details are not necessary to an understanding of how to use them.

The Sponsor-Dependents Relationship
The relationship established between model and pane objects is a classic Smalltalk relationship, that of sponsor and dependents. The sponsor-dependent relationship is built into the Object class itself and is thus available throughout Smalltalk applications. It has specific relevance here. Any object may establish itself as a sponsor object which can adopt any number of dependents (a one to many relationship). The Object instance method addDependent: takes the dependent object as parameter and adds it to the dependents collection for the receiving object. When told which object is their sponsor (i.e., their model), pane objects add themselves as dependents to that sponsor.

The Changed-Update Behavior

The Object class implements a number of methods which make specific reference to the dependents collection. Chief amongst these is the changed method. When an object is called with the *changed* message, the object itself may take no internal action, but each dependent in the dependents collection is called with the *update* message. Thus without having to explicitly keep track of dependents, a sponsor object can signal its dependents that processing is due.

The changed-update behavior is a convenient mechanism for coordination between sponsors and dependents and it forms a base for interactions between models and panes. Whether or not the changed method is used, pane objects must implement the update method for the purpose of responding to the changes in the model. In fact, in the Evacs simulation, the update method of one pane is called directly by the model in recognition of the fact that only a specific pane update is appropriate to the application. The changed mechanism is not used but the changed-update paradigm is none-the-less depended upon.

The Perform:(Selector) Mechanism

The changed-update behavior is the basic mechanism for providing output through windows. Whenever the output is to change, the *changed* method or specifc *update* methods must be called for the changes to seen. In the Evacs application, as object frequencies change the model object is notified (SimDisplay) and the corresponding display pane is called with the udpdate message.

Another Smalltalk mechanism is needed, however, to complete the output processing and to provide for user-interaction as input. The problem for output is passing to the pane object the data which is being displayed by that pane which presumably has changed. If it were always the case that output changes were initiated by the model object itself, then passing the changed data along with the update message would appear to be the most obvious solution to this problem. In practice, however, the model may want to udpate all dependent panes through the changed-update mechanism, and it may not be the case that the model object itself initiated the output-changed request. It turns out to be much more convenient for the pane to request the necessary information from the model whenever needed.
The perform:(selector) mechanism was adopted for the implementation of the pre-defined window classes whenever the pane needs to call out to the model. While the sponsor-dependents framework was a natural one for establishing a general protocol for the model's references to the pane, the perform:(selector) mechansim provides a more flexible approach for the pane's references to the model. Again this mechanism is built into the Object class. The Object class implements the *perform* method, in which the name of the method to be called is passed as parameter. A method name is called a selector in Smalltalk in reference to the selection of one of a number of methods when calling an object.

In the case of the model-pane relationship, the pane is given reference to its model-as-sponsor object and one or more symbol values which are method selectors for the model. The pane object can then invoke the model at any time with the perform message and an appropriate selector. There are two conventions for using this mechanism: the update-perform:name behavior for output and the user_action-perform:changeSelector behavior for input.

The Update-Perform:Name Behavior

To complete the description of output processing, we have a potentially three-tiered behavior in which the model is called with the *changed* message, the pane is called with the *update* message, and the pane uses the perform:name mechanism to call the model for the data which is to be displayed. Within the pane object the symbol for this last call back to the model is kept in the named instance variable *name.*, thus the title update-perform:name. (I would like to hereby note my preference for the more mnemonic instance variable name displayDataSelector, but I wasn't asked). *Name:* is also the name of the method which receives this selector for the pane. Note, of course, that the data required depends entirely on the nature of the pane class definition. Thus this general protocol has "strings attached" in the sense that by convention the model always passes in a *name* selector, but its implementation must return a pane-specific data item for the pane to continue properly. Two different cases of its use are discussed in the next section.

The User_Action-Perform:ChangeSelector Behavior

The perform:(selector) mechansim is equally essential for user-interaction as input to the model. While there are many interactions possible and complex processing is required on the part of dispatchers, it all boils down to some user-interaction, filtered through the dispatcher and window-pane, to be signalled back to the model as a change. (Note that the normal changed-update behavior is from the model to the pane). The general mechanism for this is the perform:changeSelector. Again, the instance variable name used within the pane is changeSelector which must have been set to a method name of the model during initialization. This general behavior also has "strings attached" in the sense that whle all panes use the same convention, they likely will require a method which accepts a pane-specific parameter or two and can process them accordingly. Further, different pane classes may choose to supplement this mechanism with more direct calls to the model if it does not supply a changeSelector method.

The textDispatcher as Stream Coincidence

While in general it is not necessary to understand the details of the dispatcher and its interaction with the corresponding pane, in the case of the textDispatcher it is conincidentally more convenient for the model to call the dispatcher for some output rather than the textPane. This is the consequence of the general nature of the update-perform:name mechanism and the interface provided for the textDispatcher's own processing. In general the update-perform:name mechanism expects the model to reply with the new data to be displayed. For the textPane this would be a text string. Indeed if the intention is to provide the entire string to be displayed then this mechanism should be used. Often, however, the intention is to add text to a scrolling window of text without replacing what is there, as is the case of the message log in the Evacs simulation. In this case the update-perform:name mechanism cannot be used.

The textDispatcher class definition is actually named textEditor due to its involved user interface. The textEditor allows users to select, cut and paste in the text pane, all without the model having to define the underlying implementation. Perhaps surprisingly, the model is not even informed of this interaction. It requires a supplemental user interaction to signal the model that processing is expected, at which point the model can query for selected or changed text. If nothing else, Smalltalk provides the standard *save* menu item wihch is interpreted by the textEditor and the associated textPane as the occasion for invoking the change selector of the model.

Thus it is the textEditor which incorporates methods to add text to the current display string. These methods were chosen to match a standard protocol in Smalltalk, the protocol found in the Stream object class, a sub-class of Collection. A stream is a collection of items (characters in this case) which may be bounded or unbounded, but which have a current position attribute which allows get and put operations without reference to location (the current position is used). Additional methods may be used to change the current location, such as top, bottom, and locate. To add a string to the display text of a particular textPane, the expression (textEditor nextPutAll: string) is used, where textEditor is the textPane's associated Dispatcher.

Putting it all together

The only required window parameter which we have not discussed is the specification of the particular region of the window which a pane is to handle. This is commonly done by a defining a fractional rectangle which frames a region within a 1X1 window. This is used to scale the window's frame to that of the pane. The given rectangle is called the framingRatio. While other options exist, these are left as advanced topics for discussion elsewhere.

For an output pane, the necessary parameters then are the framingRatio, the model object (typically self for the creating object) and the "name" selector which when called will provide the pane's output data. For an input pane, the necessary parameters are the framingRatio, the model, and the changeSelector. A pane which serve for both input and output, such as textPanes, may require both the name and changeSelector. Each Pane subclass expects these parameters before being opened. Note, however, that each sub-Pane class defiition adds its own requirements and its own interpretation of the general protocol.

6.2 The Evacs Simulation's Use of Window Classes

As was noted earlier, the Evacs Simulation uses three different window classes (panes): the graphPane, textPane and buttonPane.

The Smalltalk-V graphPane

The Smalltalk-V graphPane is used for output display in the Evacs Simulation. The graphPane itself is quite simple as an output Pane. Its implementation of the update method passes to the "named" method of the model the actual size of the pane's window-area and expects back a filled in display form. In the Evacs Simulation, this is provided by the displaySim method (name => #displaySim) and the displayNode objects which know how to draw themselves given a "pen" associated with the display form. The management of multiple displayNode objects is handled by the SimDisplay object which serves as the sponsor/model for the graphPane.

The Smalltalk-V textPane

The Smalltalk-V textPane is used both for message output and frequency input. In both cases a model, framingRatio and name selector are provided as paramenters. In neither case is a change selector provided since the save menu item is not used for controlling the simulation.

For the message pane, the SimWindow object itself serves as sponsor/model. A trivial method which returns the null string serves as the name selector. It is necessary as it is called as a part of the open processing of a text pane, although, in this case no initial text is to be displayed. Following this initial call it would only be called as part of the save menu item processing, except that this has been disabled. Immediately following creation, the SimWindow object calls for the message pane's associated textEditor which is then used as a stream for subsequent message out operations.

For the input pane, a separate SimInput object serves as sponsor/model. The static text, which the user can select within, is supplied by the method given as the name selector (defaultInput). A pane specific method defined for textPanes, selectedString, is called upon to retrieve the selected text. This is done when the ChangeFreq button is pushed as described below.

The Smalltalk-V buttonPane

The Smalltalk-V buttonPane is used in two places in the input area of the SimWindow. The SimInput object mentioned above serves as sponsor/model. For button panes, no name selector is needed since they serve only as input panes. In practice, the change selector is not needed for button panes and it also is not provided here. Instead, the implicit method name buttonPressed is implemented in SimInput. A buttonPane without a changeSelector attempts to call this method name directly whenever a button is pressed. The same buttonPressed method serves both button panes. An additional required parameter for button panes is the list of button names. These button names are used when a button is pressed as a parameter to the buttonPressed method to indicate which button was pressed. The button pane implementation is smart enough to take the size of the button names list as the number of buttons to create within the pane's area, to size each appropriately and to display each with its name centered inside.

Two optional parameters to button panes are used alternately on the two button pane instances in SimInput. For the event initiation buttons, a parameter is passed to indicate "push-button" behavior is desired. When this is specified, pushing a button is a non-persistent event, i.e., the button is not "selected". Selection on pressing is the standard behavior for button panes. This is the desired behavior for the mode selection butons, so this optional parameter is omitted. Note, however, that when multiple buttons are specified for a single button pane, selection on pressing means deselection of the previously selected button. This is termed "radio-button" behavior. Selective buttons which do not interact with each other require separate button panes. The additional parameter required for the mode selection button pane is the indication of which button is to be initially selected.

```
"
*****************************************************************************
Application : EVACS Simulation  -- includes transaction support

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 manned manuvering uints (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 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.
   Digit 4 controls the antenna array.
For example:
  1111Hz is complete success,
  9111Hz is failure only of the central controller (root transaction)

Classes : EvacsRoot
            Checkpointable  TransactionManager
               TransactionParticipant
                  Evacs
                    SimWindow   SimNodes    SimDisplay  SimInput
                    CentralController  MMU  AntennaMgr  Antenna
          EvacsStack   TextDisplayer  TextDisplayPane
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 (compilation order), then the class and instance
method definitions in the approximate order:
  Simulation support,  Evacs application definition, Transaction support
which more closely presents the methods top-down in order of exection
*****************************************************************************
"!
```

EVACS Simulation in Smalltalk
(Class Definitions)

```
" 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: #Checkpointable
  instanceVariableNames: ' currentState recoveryStack visibleState'
  classVariableNames: ''  poolDictionaries: ''
"   provides facility for saving an object's state & recovery states
  Class Methods
    new
  Instance Methods
    init,  pushCheckpoint,    restoreCheckpoint,  discardCheckpoint,
           saveCurrent,       restoreCurrent
" !

Checkpointable subclass: #TransactionParticipant
  instanceVariableNames: 'currentTMs  status '
  classVariableNames: ''  poolDictionaries: ''
"   provides protocol and representation for objects which participate in
    transaction
  Class Methods
    new
  Instance Methods
    init,   addState:,  restoreState:, prepared
    enter:, prepareCommitment,  completeCommitment,  abandonTransaction
" !
" 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
" !
" Simulation Support Classes " !
Evacs subclass: #SimWindow
  instanceVariableNames: 'topPane    simDisplay  simInput
                          msgStream  crPrinted '
  classVariableNames: ''  poolDictionaries: ''
"  Provides the display and interaction model for the simulation.
   Creates the window; calls upon clients for display and interaction.
   Handles message pane itself
  Class Methods
     new
  Instance Methods
     init,          openWith:,
     addMMU:,       simMode,
     nullMsg,       logText:,         logNoCr:,
     antennaFreq:,  controllerFreq:,  antennaMgrFreq:,  anMMUFreq:
" !

Evacs subclass: #SimDisplay
  instanceVariableNames: 'displayPane  displayForm  mmuCnt  simNodes '
  classVariableNames: ''  poolDictionaries: ''
"  Handles display pane of simWindow, showing key simulation objects and their
   status.  Uses SimNodes to handle multiple display objects
  Class Methods (none)
  Instance Methods
   initWith:,  addMMU,  openWith:,  update:with:,  displaySim:
" !

Evacs subclass: #SimNode
  instanceVariableNames: 'name  freq  locateBlock '
  classVariableNames: ''  poolDictionaries: ''
"  captures information about and draws nodes for SimWindow's displayPane
  Class Methods (none)
  Instance Methods
   name:,  freq:,  locate:,  drawUsing:font:
" !

Evacs subclass: #SimInput
  instanceVariableNames: 'inputPane  modePane  doitPane  simMode '
  classVariableNames: ''  poolDictionaries: ''
"  Handles input text and button panes of window, providing user control
   over simulation
  Class Methods (none)
  Instance Methods
   openWIth:andController:andSimWindow,  defaultInput,     callController:,
   buttonPressed:,  ChangeFreq,  Prompt,  Proper,  Improper,  ToughCase,  mode
" !
```

EVACS Simulation in Smalltalk
(Class Definitions)

```
" EVACS application classes " !

Evacs subclass: #CentralController
  instanceVariableNames: 'mmuArray antennaMgr frequency  mmusCount success'
  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 success'
  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  antNum '
  classVariableNames: ''  poolDictionaries: '' !
"  models behavior of an independent antenna at the base station
  Class Methods (none)
  Instance Methods
    setMaxMMUs:andSimWindow:,
    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 !
 new
   ^ ( super new ) init
! !
! SimWindow methods  !
"  Provides the display and interaction model for the simulation.
   Creates the window; calls upon clients for subpane display and interaction
   (SimDisplay and SimInput).  Handles message pane itself.
 subclass to: Evacs, EvacsRoot
 for Evacs      : simWIndow  controller
 for SimWindow  : topPane  simDisplay  simInput  msgStream  crPrinted
"
 init
   simDisplay := ( SimDisplay new ) initWith: self.
   simInput   := ( SimInput new )   init.
   crPrinted  := false.  " crPrinted is used by msgPane methods "
!

" continued "
```

```
"SimWindow methods continued"

  openWith: acontroller
  " creates topPane and 5 subPanes.  Display area is divided generally into
    3 areas: input, display (output) and message log.  The input area is
    further divided into a textDisplay pane and two button panes.  The msgPane
    takes up the entire right half of the display area.  The displayPane takes
    up the lower three quarters of the left half.  The textDisplay for input
    appears in the upper one eighth with the buttons directly under that.  The
    mode buttons take up three eights of the button space with the doit pane
    filling the 1/8 X 1/8 space directly to the left of the mode bottons.
    The doit pane is so named because these buttons initiate simulation
    processing.
  "
    | msgPane  displayPane  inputPane  modePane  doitPane
      msgFrame displayFrame inputFrame modeFrame doitFrame |
    controller   := acontroller.
    msgFrame     := ( (1/2) @ 0      extent: (1/2) @ 1     ).
    displayFrame := ( 0 @ (1/4)      extent: (1/2) @ (3/4) ).
    inputFrame   := ( 0 @ 0          extent: (1/2) @ (1/8) ).
    modeFrame    := ( 0 @ (1/8)      extent: (3/8) @ (1/8) ).
    doitFrame    := ( (3/8) @ (1/8) extent: (1/8) @ (1/8) ).

    topPane      := (TopPane new) label: 'Evacs Simulation' .
    ( topPane addSubpane:
       ( msgPane    := (TextDisplayPane new)    framingRatio: msgFrame     )).
    ( topPane addSubpane:
       ( displayPane:= (NoScrollGraphPane new)  framingRatio: displayFrame )).
    ( topPane addSubpane:
       ( inputPane  := (TextDisplayPane new)    framingRatio: inputFrame   )).
    ( topPane addSubpane:
       ( modePane   := (ButtonPane new)         framingRatio: modeFrame    )).
    ( topPane addSubpane:
       ( doitPane   := (VerticalButtonPane new) framingRatio: doitFrame    )).

    ( msgPane model: self; name: #nullMsg ).
        msgStream  := ( msgPane dispatcher ).
    ( simDisplay
       openWith: displayPane ).
    ( simInput
       openWith: (Array  with: inputPane  with: modePane  with: doitPane)
       andController: controller  andSimWindow: self  ).

   ( (topPane dispatcher) open; scheduleWindow  ).
 !
 " continued "
```

```
"SimWindow methods continued
  SimWindow users, after creating the simWindow, must tell it about each MMU
  created.  SimInput provides for simulation mode selection which can be queried
  via the simMode method.  Methods to handle msgPane provide an initial null
  message and methods to add text.  Changes to simulation frequencies are
  announced via calls to the four <>Freq: methods.
  : topPane  simDisplay  simInput  msgStream  crPrinted
"
  addMMU
    ( simDisplay addMMU ).
!
  simMode
    ^ ( simInput mode )
!
  nullMsg
      ^''.
!
  logText: aString
    crPrinted ifFalse: [ msgStream cr ].
    ( msgStream nextPutAll: aString; cr ).
    crPrinted := true.
!
  logNoCr: aString
    ( msgStream nextPutAll: (aString, '  ') ).
    crPrinted := false.
!
  controllerFreq: aFreq
    ( simDisplay update: 'centralController'  with: aFreq ).
!
  antennaMgrFreq: aFreq
    ( simDisplay update: 'antennaMgr'  with: aFreq ).
!
  antennaFreq: aFreq
          for: antNum
    | antName |
    antName := ( 'antenna', (antNum printPaddedTo: 1) ).
    ( simDisplay update: antName   with: (aFreq, ' 0Hz 0Hz') ).
!
  anMMUFreq: aFreq
    ( simDisplay update: 'MMU1'  with: aFreq ).
! !
! SimDisplay class methods ! !
! SimDisplay methods !
"  Handles display pane of simWindow, showing key simulation objects and their
   status.  Uses SimNodes to handle multiple display objects
  subclass to: Evacs, EvacsRoot
  for Evacs
  : simWindow
  for SimDisplay
  : displayPane  displayForm  mmuCnt  simNodes
"
  initWith: aSimWindow
  " Creates nodes for central controller, antenna manager, and 3 antennas.
    Divides display area (displayForm) into a 3X4 grid (horizontal X vertical).
    Position code blocks return positions in the center of one of these cells.
  "
    | pos22 pos32 pos41 pos42 pos43 |
    simWindow := aSimWindow.
    mmuCnt    := 0.
    pos22 := [( (displayForm width // 2)      @ (displayForm height // 8 * 3) )].
    pos32 := [( (displayForm width // 2)      @ (displayForm height // 8 * 5) )].
    pos41 := [( (displayForm width // 6 )     @ (displayForm height // 8 * 7) )].
    pos42 := [( (displayForm width // 6 * 3)  @ (displayForm height // 8 * 7) )].
    pos43 := [( (displayForm width // 6 * 5 ) @ (displayForm height // 8 * 7) )].
    simNodes  := Dictionary new.
    simNodes at: 'centralController'  put:
      ( (SimNode new)  name: 'Central Controller';
                       freq: 'OHz';          locate: pos22 ).
    simNodes at: 'antennaMgr'  put:
      ( (SimNode new)  name: 'Antenna Manager';
                       freq: 'OHz';          locate: pos32 ).
    simNodes at: 'antenna1'  put:
      ( (SimNode new)  name: 'Ant1';
                       freq: 'OHz OHz OHz';  locate: pos41 ).
    simNodes at: 'antenna2'  put:
      ( (SimNode new)  name: 'Ant2';
                       freq: 'OHz OHz OHz';  locate: pos42 ).
    simNodes at: 'antenna3'  put:
      ( (SimNode new)  name: 'Ant3';
                       freq: 'OHz OHz OHz';  locate: pos43 ).
!
"continued"
```

```
" SimDisplay methods continued
  : simWindow  displayPane  displayForm  mmuCnt  simNodes
"
  addMMU
  " creates SimNode for an new MMU.  MMUs are displayed in the first row of the
    displayForm's 3X4 grid.  mmuPos selects the horizontal position based on the
    mmu's number, and always selects the first row.
  "
    | nameStr mmuNum mmuPos |
    mmuNum  := (mmuCnt := mmuCnt + 1).
    nameStr := ( 'MMU', (mmuNum printPaddedTo: 1) ).
    mmuPos  := [ ( (displayForm width // 6 * (mmuNum * 2 - 1) )
                 @ (displayForm height // 8 ) ) ].
    simNodes at: nameStr  put:
      ( (SimNode new)  name: nameStr;
                       freq: '0Hz';    locate: mmuPos ).
!
  openWith: aDisplayPane
    displayPane := aDisplayPane  model: self;
                                 name:  #displaySim:.
!
  update: nodeName    with: aFreq
    ( (simNodes at: nodeName) freq: aFreq ).
    ( displayPane update ).
    ( simWindow logNoCr: (nodeName, ': ', (aFreq copyFrom: 1 to: 6)) ).
!
  displaySim: framingRect
     | aPen aFont |
     displayForm  := ( Form width:  (framingRect width)
                            height: (framingRect height) ).
     aFont        := ( Font applicationFont ).
     aPen         := ( Pen new: displayForm ).
     simNodes do: [ :simNode |
       (simNode drawUsing: aPen
                   font: aFont ).].
     ^displayForm.
! !
! SimInput class methods ! !
! SimInput methods !
"  Handles input text and button panes of window, providing user control
   over simulation
 subclass to: Evacs, EvacsRoot
 for Evacs
 : simWindow controller
 for SimInput
 : inputPane modePane doitPane simMode
"
 openWith: panesArray   andController: aController   andSimWindow: aSimWindow
   inputPane  := (panesArray at: 1)
     model: self;   name: #defaultInput.
   modePane   := (panesArray at: 2)
     model: self;   buttons: #( Proper Improper ToughCase );  push: 1.
   doitPane   := (panesArray at: 3)
     model: self;   buttons: #( ChangeFreq  Prompt );  pulse: true.
   simWindow  := aSimWindow.
   controller := aController.
   simMode    := #proper.
!
 defaultInput
    ^( ('1111Hz 1119Hz  1191Hz 1199Hz    1911Hz 1919Hz  1991Hz 1999Hz\',
        '9111Hz 9119Hz  9191Hz 9199Hz    9911Hz 9919Hz  9991Hz 9999Hz')
        breakLinesAtBackSlashes )
!
 callController: msg
   "with transaction code..."
   | success tm |
    ( simWindow logText: '' ).
    ( simWindow logText: ('initiating change to ', msg) ).
   tm := TransactionManager  runAsNewTransaction:
          [ controller changeFreq: msg ]
          id:      'simWindow->controller.changeFreq'
          parent: nil   receiver: controller  simWindow: simWindow.
!
 buttonPressed: aSymbol
    ( self perform: aSymbol ).
!
" continued ..."
" SimInput methods continued
  : simWindow controller inputPane modePane doitPane simMode
"
  ChangeFreq
      | inputString |
      inputString := ( inputPane selectedString ).
       ((inputString size) = 6) ifTrue: [
           ( self callController: inputString ).
         ]
         ifFalse: [
           ( simWindow logText:
               ('You must select a frequency first, try again') ).
         ]
!
  Prompt
     | inputString |
    inputString := ( Prompter prompt: 'Please type desired frequency'
                               default: '114Hz' ).
    ( self callController: inputString ).
!
  Proper
    simMode := #proper.
!
  Improper
    simMode := #improper.
!
  ToughCase
    simMode := #toughCase.
!
  mode
    ^ simMode
! !
! SimNode class methods ! !
! SimNode methods !
"  captures information about and draws nodes for SimWindow's displayPane
 : name  freq  locateBlock
"


 name: aString
   name := aString.
!
 freq: aString
   freq := aString.
!
 locate: aBlockReturnPoint
   locateBlock := aBlockReturnPoint.
!
drawUsing: aPen  font: aFont
    | centerPoint  height width origin vOffset|
    " draws a rectangle at centerPoint (calculated by locateBlock)
       rectangle is two-line height and just wide enough for embedded text
       SimNode name and frequceny are centered text items
     vOffset is vertical offset from upper bound to text baseline
    "
    centerPoint := ( locateBlock value ).
    width       := ( (name size) max: (freq size) ) * (aFont width) + 15.
    height      := ( (aFont height) * 2 + 15 ).
    origin      := ( (centerPoint x) - ( width // 2 ) )
                     @ ( (centerPoint y) - ( height // 2 ) ).
    vOffset     := ( (aFont height) // 2 + 5 ).

    ( aPen drawRectangle: (origin extent: (width @ height)) ).
    ( aPen place: ( (centerPoint x)  @ ( (origin y) + vOffset ))).
    ( aPen centerText: name
             font: aFont ).
    ( aPen place: ( (centerPoint x)  @ ( (centerPoint y) + vOffset ))).
    ( aPen centerText: freq
             font: aFont ).
! !
```

EVACS Simulation in Smalltalk
(EVACS application code)

```
!CentralController class methods  ! !
!CentralController methods !
"  models the central controller (base station) for the Evacs application
 subclass to: Evacs, subclass to: TransactionParticipant
  for Transaction Participant
  : currentTMs  permanentStore  status
  for Evacs
  : simWindow   controller
  for CentralController
  : mmuArray  antennaMgr  frequency  MMUsCount  success
"
  setMaxMMUs:  maxMMUs
     andSimWindow:  aSimWindow
    | antCnt |
    mmuArray     := ( Array new: maxMMUs ).
    antennaMgr   := ( AntennaMgr new ) setSimWindow: aSimWindow
                                     andMaxMMUs:    maxMMUs.
    mmusCount    := 0.       simWindow    := aSimWindow.
    controller   := self.    frequency    := '0Hz'.
!
  addState
    ( visibleState at:  #controllerSlot
                 put: frequency ).
    ( super addState ).
!
  restoreState
    frequency   := ( visibleState at: #controllerSlot ).
    ( super restoreState ).
    ( simWindow controllerFreq: frequency ).
!
  prepared
    | localResult |
    localResult := ( (frequency at: 1) < $5 ). "1st digit of frequency < 5"
    ( (simWindow simMode) = #proper )
      ifTrue:  [ ^ success & localResult ]
      ifFalse: [ ^ localResult ].
!
  registerMMU: mmu
    mmusCount := mmusCount + 1.
    ( mmuArray at: mmusCount put: mmu ).
    ( simWindow addMMU ).
    ^ 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 anAntenna |
    frequency := newFreq.
    mmuNum := 1.
    anMMU  := ( mmuArray at: mmuNum ).
    success:= true.
    ( (simWindow simMode) = #toughCase )
      ifTrue: [
        anAntenna := ((antennaMgr antennaArray) at: 1).
        tm       := TransactionManager runAsNewTransaction:
           [ anAntenna changeFrequencyOfMMU: 1 to: newFreq ]
           id:     'controller=>anAntenna.changeFreq'
           parent: (currentTMs readTop) receiver: anAntenna simWindow: simWindow.
        success := success & ( (tm status) = #completed ).
      ].

    tm       := TransactionManager runAsNewTransaction:
       [ anMMU changeFrequencyTo: newFreq ]
       id:     'controller=>anMMU.changeFreq'
       parent: (currentTMs readTop)   receiver: anMMU  simWindow: simWindow.
    success := success & ( (tm status) = #completed ).

    "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: (currentTMs readTop) ).
        ( anMMU changeFrequencyTo: '0000Hz' )].

    tm       := TransactionManager runAsNewTransaction:
       [ antennaMgr changeAntennasTo: newFreq  for: mmuNum ]
       id:  'controller=>antennaMgr.changeFreq'
       parent: (currentTMs readTop)  receiver: antennaMgr  simWindow: simWindow.
    success := success & ( (tm status) = #completed ).

    ( simWindow controllerFreq: newFreq ).
 ! !
! MMU class methods ! !
! MMU methods !
"   models behavior of an independent Manned Manuvering Unit
 subclass of Evacs, subclass of TransactionParticipant
  for TransactionParticipant
  : currentTMs  permanentStore  status
  for Evacs
  : simWindow   controller
  for MMU
  : number   frequency
"
  setController: theController
      andSimWindow:  theSimWindow
    controller := theController.
    number     := ( controller registerMMU: self ).
    simWindow  := theSimWindow.
    frequency  := '0Hz'.
!
  addState
    ( visibleState at:  #mmuSlot
                  put: frequency ).
    ( super addState ).
!
  restoreState
    frequency   := ( visibleState at: #mmuSlot ).
    ( super restoreState ).
    ( simWindow anMMUFreq: frequency ).
!
  prepared
    ^ ( (frequency at: 2) < $5 ).
!
  changeFrequencyTo: newfrequency
      frequency := newfrequency.
      ( simWindow anMMUFreq: frequency ).
! !
! AntennaMgr class methods ! !
! AntennaMgr methods !
"   coordinates a collection of three antennas at the base station
 subclass of Evacs, subclass of TransactionParticipant
  for TransactionParticipant
  : currentTMs  permanentStore  status
  for Evacs
  : simWindow   controller
  for antennaMgr
  : antennaArray   frequencyArray   success
"
   setSimWindow: aSimWindow
       andMaxMMUs: maxMMUs
     | anAntenna |
     simWindow      := aSimWindow.
     antennaArray   := ( Array with: (Antenna new)
                               with: (Antenna new)
                               with: (Antenna new) ).
     frequencyArray := ( Array new: maxMMUs ) atAllPut: '0Hz'.
     ( 1 to: 3 ) do: [ :index |
       anAntenna := ( antennaArray at: index ).
        ( anAntenna setMaxMMUs: maxMMUs
                    antNum:      index
                    andSimWindow: simWindow).
      ].
!
  addState
    ( visibleState at:  #antennaMgrSlot
                 put: (frequencyArray shallowCopy) ).
    ( super addState ).
!
  restoreState
    frequencyArray := ( visibleState at: #antennaMgrSlot ).
    ( super restoreState ).
    ( simWindow antennaMgrFreq: (frequencyArray at: 1) ).
!
  prepared
    | localResult |
    localResult := ( ((frequencyArray at: 1) at: 3) < $5 ).
    ( (simWindow simMode) = #proper )
      ifTrue:  [ ^ success & localResult ]
      ifFalse: [ ^ localResult ].
!
"continued"
```

```
"AntennaMgr methods continued"

  antennaArray
    ^ antennaArray
!
  changeAntennasTo: newFreq
      for: anMMU
    | tm anAntenna |
    success := true.
    ( 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',
                      (num printPaddedTo: 1), '.changeFreq')
                parent: (currentTMs readTop)    receiver: anAntenna
                simWindow: simWindow.
        success := success & ( (tm status) = #completed ).
      ].
    ( simWindow antennaMgrFreq: newFreq ).
! !
! Antenna class methods ! !
! Antenna methods !
"  models behavior of an independent antenna at the base station
 subclass of Evacs, subclass of TransactionParticipant
  for TransactionParticipant
  : currentTMs  permanentStore  status
  for Evacs
  : simWindow   controller
  for Antenna
  : frequencyArray
"
  setMaxMMUs:     maxMMUs
      antNum:      anInt
      andSimWindow: aWindow
    antNum := anInt.
    frequencyArray := ( Array new: maxMMUs ) atAllPut: '0Hz'.
    simWindow      := aWindow.
!
  addState
    ( visibleState at:  #antennaSlot
                put: (frequencyArray shallowCopy) ).
    ( super addState ).
!
  restoreState
    frequencyArray := ( visibleState at: #antennaSlot ).
    ( super restoreState ).
    ( simWindow antennaFreq: (frequencyArray at: 1)
                       for: antNum ).
!
  prepared
    ^ ( ((frequencyArray at: 1) at: 4) < $5 ).! !
!
  changeFrequencyOfMMU: number to: frequency
      ( frequencyArray at: number put: frequency ).
      ( simWindow antennaFreq: frequency
                        for:   antNum  ).
! !
```

```
!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 ).
! !
! Checkpointable class methods  !
 new       ^(super new) init.
! !

! Checkpointable methods  !
"  provides facility for saving an object's state & recovery states
 : currentState   recoveryStack  visibleState
"
 init
   recoveryStack := (EvacsStack new).
!
 pushCheckpoint
   visibleState := ( Dictionary new ).
   ( self addState ).
   ( recoveryStack push: visibleState ).
!
 restoreCheckpoint
   visibleState := ( recoveryStack pop ).
   ( self restoreState ).
!
 discardCheckpoint
   ( recoveryStack pop ).
!
 saveCurrent
   visibleState := ( Dictionary new ).
   ( self addState ).
   currentState := visibleState.
!
 restoreCurrent
   visibleState := currentState.
   ( self restoreState ).
! !
! TransactionManager class methods  !
  runAsNewTransaction: block
       id:         userId             parent:    parentTransaction
       receiver:   participantObject  simWindow: aSimWindow
  " Creates transaction, executes block within it and invokes completion
    processing.  Receiver must be object receiving message in block "
    | newTM |
    ( aSimWindow logText: ( 'Beginning Transaction: ', userId )).
    newTM := (super new)
               initWithID: userId;
               setParents: parentTransaction.
    ( participantObject enter: newTM ).
    ( block value ).  "execute the transaction's code"
    ( newTM processingComplete ).
    ( aSimWindow logText: ( userId, ':  ', (newTM status) )).
    ^ 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) ).
      ( transactionHierarchy push: parentTransaction ).
      ( parentTransaction registerSubTransaction: self )
      ].
!
  transactionHierarchy
    ^ transactionHierarchy
!
  status
    ^ status
!
" 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 "
       parentTM := ( transactionHierarchy readTop ).
       participants  do: [ :participant |
         ( participant completeCommitment: parentTM ) ].
       ( parentTM notNil ) ifTrue: [
         ( parentTM inheritParticipants: participants ) ].
       status := #completed.
     ]
     ifFalse: [
       status := #failed.
       participants do: [ :participant | ( participant abandonTransaction ) ].
     ].
!
 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 ).
! !
! TransactionParticipant class methods  ! !
! TransactionParticipant methods  !
"  provides protocol and representation for objects which participate in
   transaction
 : currentTMs  status
"
 init
   currentTMs := (EvacsStack new).
   status     := #free.
   ( super init ).
!
 addState
   ( visibleState at:  #participantSlot
                 put: (Array with: status
                            with: currentTMs) ).
!
 restoreState
   currentTMs := ( (visibleState at: #participantSlot) at: 2 ).
   status     := ( (visibleState at: #participantSlot) at: 1 ).
!
 prepared            "returns true if object is ready to commit"
   "( self implementedBySubclass )."
   ^ true  "by default"
!
"Continued"
```

```
"TransactionParticipant methods continued"
  enter: enteredTM
    "enter into transaction, if not current transaction save state"
    ( (currentTMs readTop) ~= enteredTM ) ifTrue: [
        ( self pushCheckpoint ). "recovery value"
        ( currentTMs push: enteredTM ).
        ( enteredTM registerParticipant: self ).
        status  := #inTransaction.
      ].
    ( self saveCurrent ).
!
  prepareCommitment
    "check status, if ok prepare for commitment"
    ^ ( (self prepared) ifTrue: [
            ( self saveCurrent ).
            status := #prepared.
          ];
          yourself )
!
  completeCommitment: parentTM
    ( currentTMs pop ).
    ( parentTM isNil )
      ifTrue:  [ "leave top transaction"
        ( self discardCheckpoint ).
        status := #free.
      ]
      ifFalse: [ "enter parent transaction"
        ( (currentTMs readTop) = parentTM ) ifTrue: [
            ( self discardCheckpoint ).
          ]
        ifFalse: [
            ( currentTMs push: parentTM ).
          ].
      ].
    ( self saveCurrent ).
!
  abandonTransaction
    ( self restoreCheckpoint ).
    ( self saveCurrent ).
! !
```

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