COSC460
Computer Science Honours Project
University of Canterbury

A SMALLTALK QUEUEING NETWORK SIMULATOR

Supervisor : Dr. Wolfgang Kreutzer

By Warwick Irwin

ABSTRACT

This report gives complete documentation for a working implementation of a discrete, event driven Smalltalk-80 modelling context described in Goldberg and Robson "Smalltalk-80 : The Language and its Implementation". It also documents DEMOS-derived enhancements to this simulation system and methods for improved presentation of results.

The system is running under Apple's level0 image on a Macintosh plus computer. It uses only standard Smalltalk-80 and will be portable to any other standard Smalltalk system.

An extremely brief introduction to Smalltalk, source code and example programs are given as appendices.

TABLE OF CONTENTS

1.0 INTRODUCTION
1.1 Background 3
1.2 Objectives of this Project 5
1.3 Structure of this Report 5

2.0 THE GIVEN SYSTEM
2.1 Overview 7
2.2 Distributions 8
2.3 Entities 13
2.4 Simulation Control 16
2.5 Resources 21
2.6 Data Collection 27
2.7 Summary of Errors 32

3.0 ENHANCEMENTS
3.1 Report Generation 39
3.2 Distributions 42
3.3 Data Collection 45
3.4 Graphics 48
3.5 Conditional Queueing 51

4.0 COMPARISON WITH DEMOS
4.1 Comparison of Features 58
4.2 A Worked Example 61

5.0 BIBLIOGRAPHY. 65

Appendix A SMALLTALK BASICS 66

1.0 INTRODUCTION

1.1 BACKGROUND

1.1.1 The Given System

Part 3 of Goldberg and Robson "Smalltalk-80 : The Language and it's Implementation" is devoted to describing a Smalltalk modelling context which may be used as a basis for writing simulations. The primary purpose of this section of Goldberg and Robson's book is to provide a practical example of Smalltalk programming. This example is, however, a complete simulation system and if extended to include some additional capabilities, could become a powerful tool for the development of simulation applications.

As this whole report revolves around Goldberg and Robson's book, for the sake of simplicity the book will be referred to 'the Blue Book'. The simulation system given in part 3 will be called 'the Blue Book System'.

Queueing Networks

The system supports simulations built around the concept of a queueing network. This is a very general model structure, in which entities (the components whose behavior is being modeled), move through a network of activities. Whenever an activity (eg. acquiring a resource) cannot be performed immediately (eg if the resource is occupied elsewhere), the entity must wait in a queue with any other similarly delayed entities until the activity is possible. Figure 1.1 gives a simple example in which two types of entity arrive, acquire some resources, then leave.

Event -driven

The simulator is event-driven. A list of scheduled events is maintained, simulated time is advanced to the time at which the next event is to occur, and that event is executed.

Discrete

An event occurs at a specific time, not over a period of time. This means that a simulated action with a particular duration will be represented by two events: The commencement and the termination of the action. Variables take on discrete values which endure until an event causes them to change. Processes involving variables changing continuously over a period of time cannot be directly represented by the system.

Process Orientation

Entities are simulated by Smalltalk objects. These objects exist as independent processes, running (conceptually) concurrently. This allows a very direct translation of a queueing network model into a program.

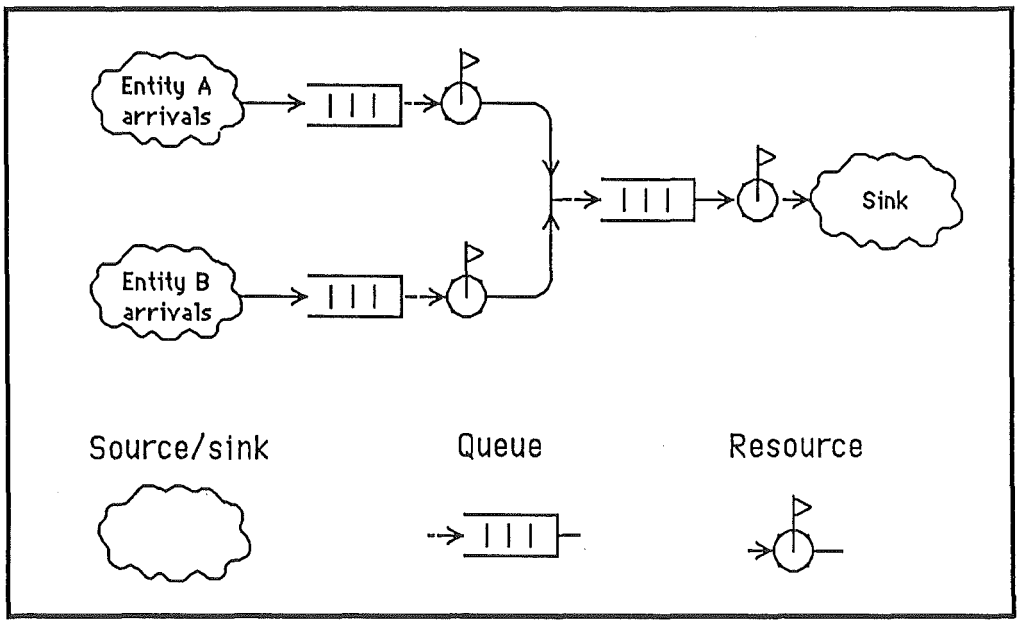


Figure 1.1

1.1.2 Demos

DEMOS is a Simula based modelling context developed by Graham Birtwistle. It is essentially very similar to the Blue Book System, because it is also a discrete, event-driven, process-oriented queueing network simulator. DEMOS has proven itself a useful sumulation tool.

In some areas, Goldberg and Robson's system is more versatile than Birtwistle's. But at the same time, DEMOS has some extremely useful features, without which the Blue Book System is severely disadvantaged.

1.2 OBJECTIVES OF THIS REPORT

Goal This project was to produce an extended working version of Goldberg and Robson's Smalltalk simulation framework on a Macintosh Plus. The extensions were to be oriented toward making the Blue Book System as capable as DEMOS, and to also take advantage of Smalltalk's graphics interface.

Approach The following steps have been taken toward achievement of this goal:

Typing in the original simulator from the Blue Book.
Testing and correcting each class as it was entered.
Running many example simulations and removing bugs.
Performing a feature by feature comparison with DEMOS to find those which should be adopted.
Design and implementation (where possible) of the additions.
Testing with further example simulations.

1.3 STRUCTURE OF THIS REPORT

Chapter 2 Chapter two covers implementation of the given system. Before attempting this section, readers unfamiliar with Smalltalk should read the very simple introduction in Appendix A. The purpose of this chapter is to give the reader an understanding of the system framework, so the chapter on additions to the system may be clear. It gives an overview of the system, then details specifics of distributions, entities, simulation control, resources and statistics gathering. The last and most important section explains and corrects all errors discovered.

Chapter 3 Chapter three details implementation of features adopted from DEMOS. The areas covered are: Distributions, statistics gathering, reporting, graphics and conditional queueing.

Chapter 4 A feature by feature comparison with DEMOS and an example program written in both languages.

Chapter 5 Bibliography.

Appendix A A far too brief introduction to the ideas behind Smalltalk, and some warnings about using the simulator in level0.

2.0 THE GIVEN SYSTEM

2.1 OVERVIEW

Programming With Classes In a conventional operating system, such as the environment in which DEMOS was designed, a simulation would be written as one program (possibly in more than one file), and the definitions of the controlling simulation system referred to in the source code. In Smalltalk, classes added by a programmer assume exactly the same status as the classes already making up the system. This means the simulator is already part of the programming environment. Writing a simulation is a matter of adding or modifying appropriate classes, which will make use of the simulator features provided.

Structure of Simulations Every simulation has one controlling object, and any number of participating objects or entities. The controlling object is an instance of a subclass of `Simulation`. The entities are instances of subclasses of `SimulationObject`.

Entities Class `SimulationObject` is a generalised entity. It provides methods for resource acquisition and time delays. Actual entity types are defined by creating subclasses of `SimulationObject`. These subclasses must define the tasks of the entity. Each new instance of these subclasses is a single entity.

Simulation Control Class `Simulation` is a generalised simulation controller. It maintains the queue of scheduled events and the simulated time. Entities are activated and deactivated by `Simulation` using semaphores. Methods for scheduling events are provided, and references to all resources and queues are stored.

Resources Instances of `ResourceProvider` are resource objects which store a queue of events waiting to acquire them. Instances of `ResourceCoordinator` represent the relationship between server and client entities. They have a queue of customers awaiting a server, or of servers awaiting a customer, depending on demand.

Distributions — Simulations depend heavily on the availability of 'random' numbers drawn from particular probability distributions. These numbers are used, for example, to determine the length of service of a customer, or the duration until another customer arrives.

2.2 DISTRIBUTIONS

2.2.1 Purpose — As stated above, values drawn randomly from distributions are often used in simulations to represent time intervals. Continuous probability distributions are provided to supply this need. It is also often necessary to decide whether an event occurs, or how many times it occurs. Such questions are answered by discrete distributions.

Below is a brief outline of the purpose of the discrete distributions available in the Blue Book System:

Bernoulli — The probability of success in a two-state trial.

Binomial — The number of successes in N trials.

Geometric — The number of trials required before a success occurs.

Poisson — The number of events occurring in a given time interval.

The continuous distributions also supported are:

Uniform — An interval of values of equal probability.

Exponential — A time before the next event occurs.

Gamma — The time before the N^{th} event occurs.

Normal — General case of other continuous distributions if sufficient independent cases are used. (Central limit theorem.)

2.2.2 Functional Description

All distributions provide the following functions:

ProbabilityDistribution instance protocol

next	Return a number drawn randomly from the distribution.
density: x	The probability density function at x.
distribution: aCollection	For discrete distributions, this is the probability a trial will yield one of the elements of aCollection. For continuous distributions, aCollection must be a sequence of contiguous values of the random variable. The value returned is an estimate of the probability of a trial being in the range covered by aCollection.

Example

If `aDist` is an instance of a `Poisson` distribution with a mean number of ten events happening per unit interval, a random number may be drawn from the distribution by `aDist next`. This might return, for example, `8`.

Sending `aDist density: 15` would return `0.034718`, the probability of exactly fifteen events occuring in one interval.

`Poisson` is a discrete distribution, so sending `aDist distribution: #(8 9 10)` would return `0.362818`, the probability of eight or nine or ten events occurring in one trial.

If `aDist` was an instance of `Normal`, a continuous distribution, `aDist distribution: #(8 9 10)` would estimate the probability of a random sample lying in the interval from eight to ten. The calculation is made by determining the density at each point in the collection and assuming the density is linear between them. Clearly a more accurate estimate may be gained by including more points in the interval.

Instance Creation

In addition, individual distributions add specific methods for instance creation:

SampleSpace class protocol

data: aCollection — Create a new instance and set the list of values from which the sample is to be drawn to aCollection.

Bernoulli class protocol

parameter: aNumber — Create a new instance and set p, the probability of success, to aNumber.

Binomial class protocol

events:n mean:m — Create a new instance and set it to n Bernoullis, with a total probability of success of m.

Geometric class protocol

mean: m — Create a new instance and set the mean number of trials before success to m.

Poisson class protocol

mean:p — Create a new instance and set the average number of events per unit interval to p.

Uniform class protocol

from: begin to: end — Create a new instance with lower bound set to begin and upper bound to end.

Exponential class protocol

mean: p — Create a new instance with μ equal to p in units of time per event.

parameter: p — Create a new instance with μ equal to p in events per unit time.

Gamma class protocol

events: k mean: p — Create a new instance as for k exponentials summing to a mean of p.

Normal class protocol

mean: a deviation: b — Create a new instance with mean a and standard deviation b.

Parameters

Most classes also include methods to return mean and variance of the random variable produced by the distribution. These are:

Bernoulli instance protocol	
mean	The mean probability of success, p.
variance	p x 1-p.
Geometric instance protocol	
mean	The mean no. of trials before a success.
variance	$1\text{-}p + p^2$, where p is the probability of a single trial being successful.
Poisson instance protocol	
mean	The mean number of events in a unit interval.
variance	Identical to the mean.
Uniform instance protocol	
mean	(end - begin) + 2.
variance	$(\text{end} - \text{begin})^2 + 12$.
Exponential instance protocol	
mean	The mean time per event.
variance	$1 + \mu^2$. Where μ is mean number of events per unit time
Gamma instance protocol	
mean	The mean time before the N^{th} event.
variance	N times the variance of a single exponential.
Normal instance protocol	
mean	The mean time per event.
variance	The square of the standard deviation supplied when the instance was created.

Example

If `aDist` was created with `aDist ← Uniform from:0.5 to:1.5`, `aDist mean` would return `1.0`.

2.2.3 Implementing the Distributions

Class `ProbabilityDistribution` is the superclass of all distributions. It has two immediate subclasses: `DiscreteProbability` and `ContinuousProbability`. The actual distributions are implemented as further subclasses of these two.

Random Sampling

`ProbabilityDistribution` maintains a class variable, `U`, which is an instance of `Random`. This random number on [0,1] is the starting point for producing random numbers from distributions in the method `next`. It is treated as a point on the y-axis of the cumulative distribution function, and the corresponding x-axis value determined. This is taken as the value of the random variable. This technique is known as the inverse distribution method. Because the inverse transformation is unique to each distribution, the `inverseDistribution` method must be implemented in the classes of the distributions themselves. (In some cases, `next` is over-ridden instead of implementing `inverseDistribution`.)

Probabilities

The sole purpose of the classes `DiscreteProbability` and `ContinuousProbability` is to provide different methods for the `distribution: aCollection` message.

In the discrete case, the value calculated is the sum of the probabilities of the elements in `aCollection`.

In the continuous case, the area under the density curve is calculated by performing a trapezoidal integration on the elements of `aCollection`. This method simply assumes the curve is linear between the points in the collection.

There is one exception to this calculation technique for continuous probabilities. Since the formula for the cumulative probability of exponential distributions is known, the `distribution: aCollection` method is over-ridden with `distribution: anInterval`. Instead of making a trapezoidal estimate, the lower cumulative probability is subtracted from the upper one to give the exact value required.

2.3 ENTITIES

2.3.1 Purpose

Class `SimulationObject` forms the framework for all entities in the simulation. It provides methods required by entities, and a structure expected by `Simulation`. Entities are encoded as subclasses of `SimulationObject`.

A subclass may include a method `tasks`, which contains the actions that will be performed by the entity, and a method `initialize`. When an entity is first activated by a `startUp`, these two methods will be invoked.

Both `tasks` and `initialize` are implemented in `SimulationObject`, but perform no actions. This allows a default entity, which will do nothing, to work in a simulation.

2.3.2 Functional Description

`SimulationObject` has a class variable `ActiveSimulation`, which refers to the current controlling `Simulation` instance. Nearly all `SimulationObject` methods work by sending messages to `ActiveSimulation`. It would have been possible to omit most of the `SimulationObject` methods and have entities send messages straight to the active simulation instead of to self. The approach chosen has the advantage, however, of making monitoring of events much easier, as will be seen in a later section.

User-supplied Methods

The user may supply some methods in a subclass of `SimulationObject` in order to tailor the general entities for a specific purpose. There are two methods which `Simulation` assumes a user will implement. A user may add any other methods desired to the `SimulationObject` subclass. (These will, of course, not be used automatically)

SimulationObject instance protocol

initialize	Instantiate instance variables.
tasks	Send messages to self (and other classes if desired) to perform the entity's tasks.

Existing Methods

The following `SimulationObject` methods are already available. With the exception of `startUp`, they are sent to self in the user's `tasks` method.

SimulationObject instance protocol

startUp	Send enter: self to the controlling simulation, execute tasks, then finishUp.
finishUp	Send exit: self to simulation.
holdFor: aTimeDelay	Do not continue until aTimeDelay has passed.
acquire: amount ofResource: resourceName	Retrieve the ResourceProvider named resourceName from the active simulation and get it to return a staticResource for amount when sufficient resource is available.
acquire: amount ofResource: ResourceName withPriority: priorityNumber	As above, with higher priorities served first.
produce: amount ofResource: ResourceName	Retrieve the ResourceProvider named resourceName from the active simulation and tell it to add amount to the quantity available.
release: aStaticResource	aStaticResource represents an amount acquired from a resource (as returned by acquire:ofResource). Return the amount to the resource.
inquireFor: amount ofResource resourceName	Ask the appropriate ResourceProvider whether amount is available.
resourceAvailable: resourceName	Answer whether the ResourceProvider has been created.
acquireResource: resourceName	Get the ResourceCoordinator called resourceName from the active simulation and use it to return a DelayedEvent containing a customer entity.

produceResource: resourceName	Get the ResourceCoordinator called resourceName and seek service as a customer.
resume: anEvent	anEvent is a DelayedEvent containing a customer (as returned by acquireResource:). Allow the customer to continue with its tasks.
numberOfProvidersOfResource: resourceName	Ask the named ResourceCoordinator how many customer entities await service.
numberOfRequestersOfResource: resourceName	Ask the named ResourceCoordinator how many server entities are waiting to serve.
stopSimulation	Terminate the simulation.

Example

In a simulation in which Royalist entities queue to shake the Queen's hand for an Exponentially distributed length of time, the definition of class Royalist could look like:

```
SimulationObject subclass Royalist
instance methods
tasks
        | theQueen |
        theQueen ← self acquire: 1 ofResource: 'Queen'.
        self holdFor: (Exponential mean: 10) next.
        self release: theQueen
```

2.3.3 Implementing Entities

SimulationObject does little real work, so implementation is very straightforward. Acquisition and production of static and coordinated resources is done by asking ActiveSimulation to provide an appropriate resource object, then passing on the message to that object. All other messages are passed on to ActiveSimulation.

2.4 SIMULATION CONTROL

Class Simulation

A single object is used to control a simulation. This object is an instance of a user-written subclass of `Simulation`. There may be only one such object active at any time, because only one simulation may be run at a time. The object's function is to administrate the simulation, maintain the event queue and simulated time, and provide access to all resources available to that simulation.

`Simulation` provides all necessary protocol for a simulation to run, with the exception of the simulation-specific definitions of how entities will enter the simulation, and what resources will be accessible to entities. Methods to perform these functions are added to the `Simulation` subclass created by the user.

The method `proceed` contains the main loop of a simulation. It removes the event at the head of the queue, advances simulated time to the time at which the event is to occur, and releases the event to run. Since `proceed` performs one event, a simulation is executed by sending the controlling object this message repeatedly.

Functional Description

A translation of the Blue Book definitions follows:

User-supplied Methods

Like `SimulationObject`, the user must supply some methods in a subclass of `Simulation`. There are five methods which `Simulation` assumes a user will implement. Other methods may also be added.

Simulation subclass instance protocol

defineArrivalSchedule This method includes messages to self to schedule the creation of new entities. It may use any of:

schedule:at:
scheduleArrivalOf:at:
scheduleArrivalOf:accordingTo:
scheduleArrivalOf:accordingTo:startingAt:

(See later in this section.)

defineResources

This method also sends messages to self. Their purpose is to allow creation of resources at the beginning of a simulation. Methods available are:

produce:of:
coordinate:

(See later in this section.)

initialize

This message is called immediately when a Simulation is created. Its purpose is to allow the user to instantiate any variables added in the subclass.
For example, initialize might contain:

myOwnVariable ← Histogram from: 0 to: 100 by: 10.

enter: anObject
and
exit: anObject

These messages are sent by an entity when it enters the simulation and when it exits. If a user wishes to take some special action at these times, these messages provide the opportunity. Their most frequent purpose is to update statistics. For example:

```
exit: anObject
        anObject duration < 10 ifTrue:
                [fastCount ← fastCount + 1].
```

Existing Methods

The remaining methods are already part of `Simulation`'s repertoire. They may be invoked by sending a message to self in the above methods, or by sending a message to `ActiveSimulation` from a `SimulationObject`.

Scheduling

Simulation subclass instance protocol

schedule: actionBlock
after: timeDelay
and
schedule: actionBlock
at: timeInteger

Both methods insert actionBlock into the event queue. When the given time arrives, the block will be executed. For example, the following statement might be included in defineArrivalSchedule :

```
self schedule: [self finishUp] at: 1000.
```

scheduleArrivalOf:
aSimulationObject
at: timeInteger

aSimulationObject , will be sent the message startUp at timeInteger.

	scheduleArrivalOf: **aSimulationObjectClass** **accordingTo:** **aProbabilityDistribution** and **scheduleArrivalOf:** **aSimulationObjectClass** **accordingTo:** **aProbabilityDistribution** **startingAt: timeInteger**	A new SimulationObject will be created and started at intervals drawn from aProbabilityDistribution. Note that the class of the SimulationObject is passed.
	delayFor: timeDelay and **delayUntil: timeInteger**	Do not return until simulated time is as specified.
Resource Creation	**produce: amount** **of: resourceName**	Create a ResourceProvider with amount available. This method may also be used to increase the amount of an already existing resource.
	coordinate: resourceName	Create a ResourceCoordinator, to synchronise activities of server-client entities.
Accessing Resources	**provideResourceFor:** **resourceName**	Return the ResourceProvider named resourceName
	includesResourceFor: **resourceName**	Returns whether or not the resource exists. The Blue Book incorrectly states: "If such a resource does not exist, then report an error." (Page 448) In fact the method more usefully returns false.
Running the Simulation	**startUp**	Send defineArrivalSchedule and defineResources messages. Inform classes SimulationObject and Resource that this is the active simulation.
	proceed	Wait until all entities are passive, advance simulated time, and execute the next event.

	finishUp	Empty the event queue.
Time	**time**	Return the simulated time. Allows SimulationObjects to ask: ActiveSimulation time.

Example For the example of `Royalists` queueing to meet the `Queen`, the controlling `Simulation` might look like:

```
Simulation subclass MeetTheQueen
instance methods
defineArrivalSchedule
        self scheduleArrivalOf: Royalist
                accordingTo: (Exponential mean: 2.0).
        self schedule: [self finishUp] at: 100.0

defineResources
        self produce: 1 of: 'Queen'
```

The simulation may be run by typing:

```
aGardenParty ← MeetTheQueen new startUp.
[aGardenParty proceed] whileTrue
```

Note that, although the simulation runs, no output will be produced. Obtaining results is covered in section 2.6.

Implementing Class Simulation The main concerns of `Simulation` are managing resources, and scheduling and running events.

Resource Handling Resource handlers `ResourceProvider` and `ResourceCoordinator` are classes which administrate queues for resources. Their implementation is discussed in section 2.5.

`Simulation` manages resources by keeping a set containing instances of `ResourceProvider` and `ResourceCoordinator`. Each instance in the set represents one resource available to the entities in the simulation. New instances are created and added to the set via the methods for `produce:of:` and `coordinate:`. These messages are usually sent from `defineResources`, but will work if sent to `ActiveSimulation` from an entity.

Scheduling

All scheduling methods eventually rely on `delayUntil:`.

`DelayUntil:` works by not returning until simulated time is the time asked for. The method which sent it will continue normally as soon a `delayUntil:` returns.

`DelayUntil:` creates an instance of `DelayedEvent`. This a package containing a private semaphore and a `condition`. The `condition` is set to be the time at which `delayUntil:` is to return, and the whole package is inserted into the event queue. The event queue is a `SortedCollection`, and `DelayedEvents` sort according to their `condition`, so the event with the earliest time will be at the head of the queue. The semaphore in the `DelayedEvent` is then told to wait, blocking execution of the current method. It cannot return until the semaphore is signalled.

Running

`DelayUntil:` puts methods to sleep, `proceed` awakens them. In `proceed`, the `DelayedEvent` at the head of the queue is removed and simulated time is set to the `condition` of the `DelayedEvent`. The semaphore in the `DelayedEvent` is signalled, allowing the method which executed the original `delayUntil:` to continue.

It is essential that all actions which should occur at a particular time are completed before simulated time is advanced. To ensure all entities are passive before `proceed` releases another event, the variable `processCount` is kept. Whenever a new process is created, the method `newProcessFor:` increments `processCount` via the `Simulation` method `startProcess`. When that process terminates, `stopProcess` decrements `processCount`. `DelayUntil:` must also send `stopProcess` before telling the `DelayedEvent` semaphore to wait and `startProcess` afterwards. `Proceed` will not execute another event until `processCount` is zero, as returned by `readyToContinue`. The statement `processor yield` is used to send the `proceed` to the back of the ready queue repeatedly until no other processes are active.

2.5 RESOURCES

There are two essentially different types of resource: static resources and coordinated resources.

Static resources are concerned with the availability of a simple commodity to entities. The commodity has no tasks to perform and need only be represented as an amount. Such resources are simulated by instances of `ResourceProvider`.

Coordinated resources are concerned with the availability of entities to other entities. These resources model server-customer relationships. A server acquires a customer via the coordinated resource, and a customer makes itself available to the coordinated resource for acquisition by a server. This mechanism allows synchronisation of entities to perform a common activity. These relationships are managed by instances of `ResourceCoordinator`.

Variations on the way static resources are manipulated by entities allow different static resource behavior.

A resource which begins the simulation with a certain quantity and is consumed during execution can be modelled by initially creating the resource via `produce:ofResource:` in `defineResources`. Entities then consume the resource by sending the message `acquire:ofResource`.

Resources which cannot be consumed should be treated as for the above case, but when an entity finishes with a resource, it should send `release:` to return the acquired commodity.

Resources which are produced by one type of resource and consumed by another are modelled by the producer sending `produce:ofResource` and the consumer sending `acquire:ofResource`.

Functional Description

`Resource` is an abstract superclass of `ResourceProvider` and `ResourceCoordinator`. It provides means of enqueueing requests, and protocol to allow `Simulation` to manage the resources. Protocol for `Resource` follows.

Class Resource

Resource class protocol

named: resourceName	This is a class method to create a new instance and set its name.

Resource instance protocol

name	Return the resource name.
addRequest: aDelayedEvent	This method should only be used by the subclasses. It adds aDelayedEvent to the queue and blocks the semaphore.

Resource-Provider

The ResourceProvider messages for acquisition and production are normally sent by SimulationObject methods, not directly by entities.

ResourceProvider instance protocol

acquire: amountNeeded **withPriority: priorityNumber**	Wait until sufficient resource is available, then return a StaticResource instance for amountNeeded. StaticResource is merely a means of giving an entity something to hold onto and release. Its protocol will be explained later in this section. Higher numbered priorities get precedence. This message is sent by SimulationObject methods acquire:ofResource: and acquire:ofResource:withPriority.
produce: amount	If the resource does not yet exist, create it. Add amount to the amount of resource available. This message is sent by SimulationObject's method produce:ofResource: .

It is frequently necessary for entities to determine how much of a resource is available, so the following method is provided.

ResourceProvider instance protocol

amountAvailable	Return the remaining quantity of resource.

Example

The previous example, MeetTheQueen demonstrates the handling of a resource, 'Queen' which is not consumable. Alternatively, if the Queen was to be modelled as having a handshake capacity of 30, the example would appear as:

```
Simulation subclass MeetTheQueen
instance methods
defineArrivalSchedule
        self scheduleArrivalOf: Royalist
                accordingTo: (Exponential mean: 2.0).

defineResources
        self produce: 30 of: 'Queen'
```

```
SimulationObject subclass Royalist
instance methods
tasks
        (self inquireFor: 1 ofResource: 'Queen')
                ifTrue: [self acquire: 1 of: 'Queen'.
                        self holdFor:
                                (Exponential mean: 10) next]
                ifFalse: [self stopSimulation]
```

The methods `produce:of:`, `acquire:of:` and `inquireFor:ofResource:` are defined in `SimulationObject`. The `SimulationObject` methods obtain the `Resource` and send `acquire:withPriority`, `produce:` and `inquireFor` respectively.

StaticResource

There are two `StaticResource` methods of importance to the user:

StaticResource instance protocol

consume: aNumber	Reduce the remaining amount of this StaticResource by aNumber.
release	Return the remaining amount to the resource which created it.

Resource-Coordinator

Like `ResourceProvider`, `ResourceCoordinator` is usually only manipulated indirectly by the user. Its methods are:

ResourceCoordinator instance protocol

acquire	This message results from a server sending self acquireResource: resourceName. It returns a delayedEvent whose condition is the acquired customer. When the customer has been served, the server should release it with self resume: aDelayedEvent.

producedBy: aCustomer	This message results from a customer sending self produceResource: resourceName. The method will not return until the server which acquired aCustomer releases it.

Example

If the Queen was to be modelled as an entity rather than as a resource, the example with unlimited handshaking capacity could be written as:

```
Simulation subclass MeetTheQueen
instance methods
defineArrivalSchedule
        self scheduleArrivalOf: Royalist
                accordingTo: (Exponential mean: 2.0).

defineResources
        self coordinate: 'Handshake'

SimulationObject subclass Royalist
instance methods
tasks
        self provideResourceFor: 'Handshake'

SimulationObject subclass Queen
instance methods
tasks
        | aPerson |
        aPerson ← self acquireResource: 'Handshake'.
        self holdFor: (Exponential mean: 10) next.
        self resume: aPerson
```

As with `ResourceProvider` in the previous example, the `ResourceCoordinator` methods are invoked here via `SimulationObject` methods.

Implementing Resources

Resources require maintenance of a queue of entities awaiting service. The class `Resource` defines `pending` for this purpose. As in `Simulation's` event queue, elements of the queue are `DelayedEvents`. In the event queue, the condition of a `DelayedEvent` is the time at which the event should occur. In `pending`, the condition of each element is a `StaticResource`, a customer, or uninitialized depending on the situation.

Resource-Provider

When an instance of `ResourceProvider` is sent an `acquire:` message, a `DelayedEvent` containing a `StaticResource` for the required amount is added to the queue. If the amount available is sufficient, the method will return the `StaticResource` immediately. If the request is too large to be granted at once, the `DelayedEvent` is `paused`. It will be tested again whenever any other entity accesses the resource, and when enough resource is free, the `DelayedEvent` will be `resumed`, and its contents returned by `acquire:`

The source code is slightly tricky to comprehend for the case when a request is satisfied immediately. The `DelayedEvent` is told to `resume` before being told to `wait`. The effect is that there is no wait, since semaphores 'remember' signals.

Priorities are implementedby having `pending` as a `SortedCollection`. The `DelayedEvents` in `pending` sort according to their `conditions`, which are `StaticResources`. Since `StaticResources` sort by their priority, the entire queue is ordered by priority. Within priorities, service is FIFO.

Resource-Coordinator

The queue for a coordinated resource contains either customers or servers, depending on which is in greater supply. A variable `whoIsWaiting` flags what the queue consists of.

If customers are waiting, the queue consists of `DelayedEvents` whose `conditions` are the customers themselves. New customers sending `producedBy: self` are simply added to the queue. When a server tries to acquire a customer, the head `DelayedEvent` is removed and returned to the server.

If servers are waiting, `pending` contains an empty `DelayedEvent` for each server. Additional servers sending `acquire:` add another `DelayedEvent` and wait. When a customer sends `producedBy: self`, the head `DelayedEvent` has its `condition` set and is `resumed`. The waiting server will wake up, and return the `condition` of its `DelayedEvent`. This `condition` is itself a `DelayedEvent`, which contains the `paused` customer. It is now the responsibility of the server to `resume` the customer's `DelayedEvent` when service is completed.

2.6 DATA COLLECTION

Data collection is required for two reasons: (1) To provide the user with statistical information about the performance of the simulation, and (2) To help the user verify the correct functioning of the model. For the first category, the Blue Book provides classes `SimulationObjectRecord` and `Histogram`. For the second category, class `EventMonitor` is available.

SimulationObjectRecord

A `SimulationObjectRecord` records the entry time and duration of an entity.

SimulationObjectRecord instance protocol

entrance: currentTime	Record the entrance time as currentTime.
exit: currentTime	Calculate the duration.
entrance	Return the entrance time.
exit	Record the entrance time plus duration.
duration	Return the duration.
printOn: aStream	Print entrance time and duration.

SimulationObjectRecords are not very useful without some organisation. The Blue Book defines class StatisticsWithSimulation for this purpose. If the user creates the simulation controlling class as a subclass of StatisticsWithSimulation instead of Simulation, then a SimulationObjectRecord will be stored for every entity in the simulation.

StatisticsWithSimulation is a subclass of Simulation, and adds an instance variable to store all SimulationObjectRecords. It creates and updates the records in the methods enter: and exit: which are automatically sent by entities. The user should be careful not to override these methods.

To print the statistics the following method exists:

StatisticsWithSimulation instance protocol

printStatisticsOn: aStream	Print data for all entities.

The following simulation gives an example of use:

```
StatisticsWithSimulation subclass CarRace
instance methods
defineArrivalSchedule
        self scheduleArrivalOf: (Car new) at: 0.0.
        self scheduleArrivalOf: (Truck new) at: 0.0.
        self scheduleArrivalOf: (Car new) at: 1.0.
        self scheduleArrivalOf: (Car new) at: 2.0.

SimulationObject subclass Car
instance methods
tasks
        self holdFor: (Uniform from: 10 to: 20) next

SimulationObject subclass Truck
instance methods
tasks
        self holdFor: 45.0
```

Object	Entrance Time	Duration
a Car	0.0	15.0577
a Truck	0.0	45.0
a Car	1.0	13.3023
a Car	2.0	9.1344

Histogram

The stated purpose of this class is to produce statistics on throughput of the simulation. In fact, Histogram is not specific about what its data applies to, and may be used for other purposes, such as graphing queue lengths.

Histogram class protocol

from: lowerNum to: upperNum by: step	Create a new instance. Data will be grouped into intervals of step.

Histogram instance protocol

store: aValue	Record aValue in the Histogram.
printStatisticsOn: aStream	Produce output on aStream.

Below is an example program using histograms:

```
Simulation subclass anotherRace
instance variable names  durations
instance methods
initialize
        durations ← Histogram from: 10 to: 20 by: 2

defineArrivalSchedule
        self scheduleArrivalOf: Car
                accordingTo: (Exponential mean: 2.0).
        self schedule: [self finishUp] at: 100.0
```

```
storeTime: aValue
        durations store: aValue

printStatisticsOn: aStream
        durations printStatisticsOn: aStream

SimulationObject subclass Car
instance methods
tasks
        | myDuration |
        myDuration ← (Uniform from: 10 to: 20) next.
        self holdFor: myDuration.
        ActiveSimulation storeTime: myDuration
```

Output will look similar to:

```
        Number of       Minimum         Maximum         Average
        Objects         Value           Value           Value
        46              10.1828         18.1100         14.3743

        Number of
Entry   ObjectsFrequency
10-12   10              0.2173913       | XXXXXXXXXX
12-14   8               0.173913        | XXXXXXXX
14-16   8               0.173913        | XXXXXXXX
16-18   12              0.2608695       | XXXXXXXXXXXX
18-20   8               0.173913        | XXXXXX
```

In level0, histograms are actually messier than this because the size of tabs is too small and cannot be adjusted. Some editing is required to get the neat result above. A full Smalltalk system would allow this problem to be solved.

EventMonitor This device is used to produce a trace of entities. The only protocol a user needs to be aware of is how to specify the destination of output.

EventMonitor class protocol

file: aFile	aFile will recieve all traces.

`EventMonitor` is a subclass of `SimulationObject`. It re-implements all the task language methods by printing the time and a description of the event before and after passing the message to the superclass. Any entity to be traced must be defined as a subclass of `EventMonitor`.

An example of output from a simulation with reader and writer entities being traced is:

0.0	Reader 1 enters
0.0	Reader 1 requests 1 of File
1.34	Reader 2 enters
2.11	Writer 3 enters
2.431	Writer 3 produces 2 of File
2.431	Writer 3 exits
2.431	Reader 1 obtained 1 of File
2.723	Reader 1 exits
4.150	Writer 4 enters
.	.
.	.

Note that entity numbering is consecutive, even though entities may be of different types. Each entity type may have its own sequence of numbers if the user adds to the entity definition a class variable which will act as a counter. The method below must also be added:

EventMonitor subclass instance protocol

setLabel	Increment the counter and set label to be a string containing the value of the counter. Label is defined in EventMonitor.

For example, a `Reader` might add a class variable, `ReaderNumber`, and the method:

```
SimulationObject subclass Reader
instance methods
setLabel
        ReaderNumber ← ReaderNumber + 1.
        label ← ReaderNumber printString
```

2.7 SUMMARY OF ERRORS

This section provides corrections for all errors discovered in the given system.

Distributions `ProbabilityDistribution` omits the method:

```
atEnd
        ↑false
```

Although this message is never sent directly by the simulation system, it is required by the method for `do:`, which is used to produce a stream of samples drawn from the distribution. The message `do:`is sent in `Simulation's` method for `schedule:startingAt:andThenEvery:`.

Binomial omits methods for mean and variance. The superclass Bernoulli will return incorrect values. The methods should be included as:

```
mean
        ↑super mean * N

variance
        ↑super variance * N
```

Gamma omits the method for next or inverseDistribution. The superclass Exponential responds with a value drawn from a single exponential distribution rather than the sum of N exponentials. A solution is:

```
next
        | t |
        t ← 0.
        N timesRepeat: [t ← t + super next].
        ↑t
```

There is another error in Gamma. The instance creation method is given as:

```
events: k mean: p
        k ← k truncated.
        k > 0
                ifTrue: [↑(self parameter: k/p) setEvents: k]
                ifFalse: [self error:
                        'The number of events must be
                                greater than 0']
```

Attempting to modify k produces an error. The method should be:

```
events: k mean: p
        | t |
        t ← k truncated.
        t > 0
                ifTrue: [↑(self parameter: t/p) setEvents: t]
                ifFalse: [self error:
                        'The number of events must be
                                greater than 0']
```

Simulation Control

Class `Simulation` contains a number of errors. In particular, the fundamental method `proceed` does not work correctly.

The `delayUntil:` method contains a typographical error. The given code is:

```
delayUntil: aTime
        | delayEvent |
        delayEvent ← DelayedEvent onCondition: timeInteger
```

A solution:

```
delayUntil: aTime
        | delayEvent |
        delayEvent ← DelayedEvent onCondition: atime
```

The method proceed is supplied as:

```
proceed
        | eventProcess |
        [self readyToContinue]
                whileFalse: [Processor yield].
        eventQueue isEmpty
                ifTrue: [↑self finishUp]
                ifFalse: [eventProcess ← eventQueue
                                                removeFirst.
                        currentTime ← eventProcess time.
                        eventProcess resume]
```

The line `currentTime ← eventProcess time` is incorrect. The `eventProcess` is a `DelayedEvent` whose `condition` is the time the event should occur. The line should be:

```
currentTime ← eventProcess condition
```

Some example programs given by Goldberg and Robson rely on the (very useful) construct: `[aSimulation proceed] whileTrue`, or variations of it. For this to work, `proceed` needs to return true when items remain in the event queue. It already returns false correctly when the queue is empty. The statement `↑true` should be appended to the `ifFalse:` clause.

A more complex error in `proceed` occurs because of the parallel process nature of entities.

The first action performed by an entity after it resumes execution is to signal its existence to the simulation by causing `processCount` to be incremented. After telling an entity to resume, `proceed` will terminate. Usually, `proceed` is invoked again immediately, and tests `processCount` to allow any still executing events to complete. A problem arises, however, when `proceed` reaches the point of testing `processCount` before the resumed entity has incremented it. In this situation, the

next event may be resumed prematurely, or if the queue is now empty, the simulation terminates.

The solution is to yield the processor after resuming the entity, to allow the entity to begin execution.

The final method for `proceed` is now:

```
proceed
        | eventProcess |
        [self readyToContinue]
                whileFalse: [Processor yield].
        eventQueue isEmpty
                ifTrue: [↑self finishUp]
                ifFalse: [eventProcess ← eventQueue
                                                removeFirst.
                        currentTime ← eventProcess condition.
                        eventProcess resume.
                        Processor yield.
                        ↑true]
```

Resources

The class Resource is an abstract superclass of `ResourceProvider` and `ResourceCoordinator`. The purpose of `Resource` is to combine common features of the two subclasses. Unfortunately, it attempts to combine too much. The queue of requests, pending, is defined as a `SortedCollection`. This works for `ResourceProvider`, allowing requests to be sorted by priority. For `ResourceCoordinator`, however, there is no priority mechanism and nothing to be gained by sorting requests. In fact, trying to sort the elements in the queue of a `ResourceCoordinator` causes an error. This is because the `delayedEvents` sort by their `conditions`, which are the customers requesting service. Since customer entities do not implement the message <=, an error arises.

It would be possible to let users add a <= message to customers, allowing an arbitrary customer priority system, and include a default method in `SimulationObject`. This is unlikely to be the intention of the designers, and produces a non-orthogonal system because no priority mechanism can be provided for servers. A more straightforward solution is to define `pending` to be an `OrderedCollection` within `ResourceCoordinator`.

The original `Resource` method for instantiating `pending` was:

```
setName: aString
        resourceName ← aString.
        pending ← SortedCollection new
```

This has been altered to:

```
setName: aString
        resourceName ← aString
```

The deleted statement has been copied into `ResourceProvider's` initialization method:

```
setName: aString with: amount
        super setName: aString.
        pending ← SortedCollection new.
        amountAvailable ← amount
```

`ResourceCoordinator` has also had new pending instantiation added.

```
setName: aString
        super setName: aString.
        pending ← OrderedCollection new.
        whoIsWaiting ← #none
```

Data Collection

The EventMonitor class sends the wrong message in one method. The original was:

```
produceResource: resourceName
        super produce: amount
                ofResource: resourceName
```

This should be:

```
produceResource: resourceName
        super produceResource: resourceName
```

3.0 ENHANCEMENTS

This section describes features added to the Blue Book System. Most of these ideas are inspired by DEMOS. In some cases, the algorithm used is translated directly from the Simula source code.

Design Philosophy

All borrowed features have been modified to integrate with the existing system, and provide interfaces like those of the supplied classes.

Not all DEMOS features have been transferred. Those that would require large changes to the original code and are not essential have been omitted. These omissions are discussed in the next chapter.

3.1 Reporting

The Blue Book System provides limited report writing capability. In contrast, DEMOS uses automatic reporting. DEMOS reporting may be switched on or off, but it is otherwise fairly inflexible. Rather than introduce the same limitations to the Smalltalk system, a user-defined approach to reports has been taken.

The `printStatisticsOn:` methods form the basis of report writing. Those which were supplied by the original system have been extended to produce a standard format, and all new data collectors written adhere to this design.

More importantly, a `printStatisticsOn:` method has been added to `ProbabilityDistribution` and `Resource`.

To obtain a report, the user is required to declare an instance variable in the `Simulation` for every object which is to report. At the end of the simulation, these variables should be told to print. For this purpose, the method `printStatisticsOn:` has also been added to `Simulation`. Like other user supplied methods, the default implementation performs no actions. The user may override this method in a subclass to print reports on whatever subjects required, in any order. Standard reports may also be interspersed with user text.

The new reporting format requires a name for each object told to report. Resources already have a name, but ProbabilityDistributions and data collectors do not. To remedy this, `ProbabilityDistribution` and `Histogram` have had a `name` instance variable added, and all instance creation methods prefixed with name:. For example, a `Uniform` may now be created with `name:from:to:` as well as the original `from:to:`. If the older method is used, `name` will be prompted for. Note that this means the new method must be used in some circumstances, for example if each new entity creates its own distribution, otherwise the user will be supplying an endless stream of names.

Example A version of the Queen-meeting simulation and its output follows:

```
Simulation subclass MeetTheQueen
instance variables    royalistArrivals handshakes
instance methods
initialize
        super initialize.
        royalistArrivals ← (Exponential named: 'Arrivals'
                                        mean: 2.0).
        handShakes ← Histogram named: 'Handshake Time'
                                        from: 0 to: 30 by: 5.

defineArrivalSchedule
        self scheduleArrivalOf: Royalist
                accordingTo: royalistArrivals.
        self schedule: [self finishUp] at: 100.0

defineResources
        self produce: 1 of: 'Queen'

recordShake: shakeTime
        handshakes store: shakeTime
```

```
printStatisticsOn: aStream
        royalistArrivals printStatisticsOn: aStream.
        (self provideResourceFor: 'Queen')
                        printStatisticsOn: aStream.
        handshakes printStatisticsOn: aStream

SimulationObject subclass Royalist
instance methods
tasks
        | theQueen shakeTime |
        theQueen ← self acquire: 1 ofResource: 'Queen'
        shakeTime ← (Exponential named: 'Shake'
                                                mean: 10) next.
        self holdFor: shakeTime.
        ActiveSimulation recordShake: shakeTime.
        self release: theQueen
```

After running the simulation, output might look like:

```
----------Arrivals----------
Exponential Parameter 0.5
Number of observations 60


-------------Queen------------
ResourceProvider
Number of departures 11
Mean queue length 22.0138
Mean wait 36.6897
```

```
---------Handshake Time-------
Histogram

Number of       Minimum         Maximum         Average
Objects         Value           Value           Value
10              1.90644         21.9652         8.72547

                Number of
Entry           ObjectsFrequency
0-5             4               0.4             |XXXX
5-10            3               0.3             |XXX
10-15           1               0.1             |X
15-20           1               0.1             |X
20-25           1               0.1             |X
25-30           0               0.0             |
```

3.2 Distributions

There are four DEMOS distributions not included in the Blue Book System. The first two given below are discrete and the second two continuous. Because the distribution framework already exists, adding new ones is simplified. For a discussion of this framework, see Chapter 2.

Functional Description

Details of the method behaviors are given below:

Constant

Constant class protocol

value: x	Create a new instance which will always return value.

Constant instance protocol

mean	Return value.
variance	Return 0.
density: x	If x is value return 1, otherwise return 0.

	inverseDistribution: x	Return value.
RandInt	Randint class protocol	
	from: min to: max	Create a new instance. All integers between min and max inclusive are equally likely.
	Randint instance protocol	
	lowerBound	Return minimum value.
	upperBound	Return maximum value.
	mean	Return the halfway point.
	variance	Return variance, calculated from deviations from mean.
	density: x	If x is an integer within the legal range return 1 / the number of integers in the interval, otherwise return 0.
	inverseDistribution: x	Return a random integer from the interval.
Erlang	Erlang class protocol	
	mean: p withk: kValue	Create a new instance which is equivalent to the sum of kValue Exponentials and has a total mean of p. Here p is in units of time per event.
	parameter: p withk: kValue	Create a new instance which is equivalent to the sum of kValue Exponentials and has a total rate of p. Here p is events per unit of time.
	Erlang instance protocol	
	mean	Return the mean number of events per unit interval.
	variance	$1 / (k \times mean^2)$
	density: x	Not implemented.
	inverseDistribution: x	Return the cumulative distribution inverse of x.

Empirical

Empirical class protocol

cumulativeProbabilities: anArray — Create a new instance with the cumulative distribition function defined by joining the points in anArray. Each array element is an instance of Point with x being the independent variable and y the cumulative probability. Both x and y values must form a strictly ascending sequence. The first y must be 0, and the last 1.

Empirical instance protocol

densityArray — Return the array of points.

inverseDistribution: x — Return the inverse of x by finding the adjacent probabilities and interpolating between them.

Example

These distributions are used in the same format as those already described. Output style is also the same, for example:

```
----------A Randint---------
RandInt from 1 to 100
Number of observations 34
```

Implementing the Distributions

A discussion of those implementation decisions which were non-trivial follows.

Erlang

This class could have been implemented as an `Exponential` subclass, and `inverseDistribution` calculated by summing `k` superclass samples. The same effect can be achieved, however, by first multiplying together k random seeds, then calculating the inverse using the same formula as used in `Exponential`. This means `Erlang` may be a direct subclass of `ContinuousProbability`.

Empirical

The `density:`, `mean` and `variance` messages are too difficult to implement for arbitrary functions.

The inverseDistribution: method steps along the array of points until a cumulative probability value is found which is greater than the supplied seed. The estimated sample value is found by interpolating between this point and the previous one.

3.3 Data Collection

The biggest downfall of the Blue Book System is the lack of statistical instrumentation. The classes presented here attempt to remedy this problem.

Count

This class simply sums its inputs. In fact it is not really necessary, since the user could easily write code to perform the necessary functions, but it has been included to allow count statistics to be treated in the same way as other statisitics.

Count class protocol

named: aString	Create a new instance and set its name.

Count instance protocol

update: aNumber	Add value to the sum.
total	Return the sum.
printStatisticsOn: aStream	Output result.

Count output looks like:

```
----------A Count----------
Count
Total 847
```

Tally

This is a tally in the DEMOS sense. (The Blue Book example uses 'tally' to mean a simple count [page 474]). An instance of `Tally` performs the functions of `Count`, and also calculates the mean, variance and extreme values.

Tally class protocol

named: aString	Create a new instance and set its name.

Tally instance protocol

update: value	Add value to the sum.

total	Return the sum.
mean	Return the mean.
variance	Return the variance.
minimum	Return the minimum.
maximum	Return the maximum.
printOn: aStream	Output results.

An example of output is:

```
------------A Tally-----------
Tally
Total 71
Number of observations 15
Minimum 1  Maximum 9
Average 4.73333
Variance 1747.85
```

Accumulate

The purpose of this class is to produce time-weighted statistics. This means average and variance are calculated from the duration for which a value existed rather than the number of discrete times that value occurred.

Accumulate class protocol

named: aString	Create a new instance and set its name.

Accumulate instance protocol

startWith: initialValue	Commence accumulating with first value of initialValue.
start	Commence accumulating with first value of 0.
update: value	Set accumulate to value now.
mean	Return mean up to the last update time.
variance	Return variance up to the last update time.

minimum and **maximum**	Return extreme values.
integral	Return the sum of all values times their durations.
includeLast	Include continuation of the last value up to the current time.
printStatisticsOn: aStream	Execute includeLast and output results.

Output looks like:

```
---------An Accumulate--------
Accumulate
Number of events 115
Minimum 7.85122  Maximum 21.221
Average 16.34982
Variance 1947.38812
```

Regression

Instances of `Regression` accept a series of (x,y) points and determine the best fit line through them by minimizing the sum of the squared deviations of the parts from the fitted line. The algorithm is straight from DEMOS. Relevant protocol is:

Regression class protocol

named: aString	Create a new instance and set its name.

Regression instance protocol

updateAt: xValue store: yValue	Record the point.
printOn: aStream	Fit the line and print a DEMOS-like report giving estimated intercept and slope.

Output has the form:

```
---------A Regression--------
Regression
Number of points 13
Mean x 7.69131  Mean y 14.0769
Residual standard deviation 3.24097
Regression coefficient 1.92443
Intercept -0.72636
SD Regression coefficient 0.240417
Correllation coefficient 0.93443
```

Implementing Data Collectors

Accumulate

A slight modification to `Simulation` was required to inform class Accumulate which simulation is active. This is necessary for `Accumulate` to access the simulated time.

```
activate
        SimulationObject activeSimulation: self.
        Resource activeSimulation: self.
        Accumulate activeSimulation: self
```

3.4 Graphics

Pictorial representation of `Histograms` and `Regressions` has been implemented. In fact, a general graph drawing facility has been developed, to allow users to represent any collection of points in a number of possible graph forms.

Axes Functionality

The heart of the graphing feature is class `Axes`. This is a subclass of `Form`, so any set of axes may be manipulated by a user with any standard `Form` protocol. `Axes` adds methods to `Form` so that labelled x and y axes with drawn scales may be formed.

By the very nature of graphs, a large amount of information is required from the user. To simplify cases where users are unconcerned about details, default values exist for most variables.

Axes class protocol

new	Create a new set of default axes.
title: heading xAxis: x yAxis: y	Create a new set of default axes, with a central heading, the x-axis labelled x and the y-axis labelled y.

Axes instance protocol

origin: origin xLength: x yLength:y	Set the origin of the axes to be at the coordinates of the point given by origin, the length of the x-axis to be x and the length of the y-axis to be y.
xLower: min upper: max step: increment	Set the lowest scale value on the x-axis to min, and the highest to max, with steps of increment.
yLower: min upper: max step: increment	Set the lowest scale value on the y-axis to min, and the highest to max, with steps of increment.
draw	Plot the axes and return self.

An example of a form returned by `Axes` is given in figure 3.1.

Axes Implementation

`Axes` relies on the standard system class `Commander` to mark regular intervals along both lines using the `lineUpFrom:to:` method. Unfortunately, level0 has had this class removed, and so it had to be re-implemented. Similarly, the Pen method `location` was needed and had to be restored. Source code is supplied in Appendix B.

BarGraph

This a subclass of `Axes`. It is actually a histogram, but that name was already in use. This class adds vertical bars to the axes.

BarGraph instance protocol

data: anArray	Set the data which will be graphed. There will be one bar for each element in anArray.
drawGraph	Draw the axes and bars and return self.

Graphing Histogram

An instance of `BarGraph` depicting the supplied data can be obtained by the following methods:

Histogram instance protocol

graphForm: title xAxis: x yAxis: y	Return a graph with specified title and axis labels.
graphForm	Return a graph with default title and axis labels.

An example of a form returned by `Histogram` is given in figure 3.2.

ScatterGraph

Another `Axes` subclass. It adds methods for plotting points and lines between points.

ScatterGraph instance protocol

data: pointCollection	Set the points to be plotted to aCollection.
plotPoints	Draw the axes and mark given points on it with x's and return self.
joinPoints	Draw the axes, add lines joining the given points and return self.

Examples of forms returned by `ScatterGraph` are given in figure 3.3 and 3.4.

Graphing Regression

An instance of `ScatterGraph` depicting the supplied data can be obtained by the following methods:

Regression instance protocol

graphForm: title xAxis: x yAxis: y	Return a graph with specified title and axis labels.
graphForm	Return a graph with default title and axis labels.

An example of a form returned by `Regression` is given in figure 3.5.

3.5 Conditional Queueing

Conditional Queues are waiting lines in which entities are delayed until a specific condition arises. They are almost essential in non-trivial simulations, but were not included in the original design.

Fortunately, these queues fit very neatly into the `Resource` structure already implemented. For the purposes of this simulator, conditional queues are defined to be resources.

When an entity is to wait, it informs the `ConditionQueue` it will join, and specifies the condition on which it is waiting. Whenever any entity performs some action which could release items awaiting a condition, the entity must tell the `ConditionQueue` to `test`.

ConditionQueue class protocol

named: aString	Create a new instance and set its name.

ConditionQueue instance protocol

waitUntil: ConditionBlock	Join the tail of the queue. Test and release items from the queue head if possible.
test	Test and release items from the head of the queue.

Unlike DEMOS, there is no way provided to test all entries in a queue. Instead, seperate queues should be provided for each group requiring FIFO service.

As with other `Resources`, users do not directly access `ConditionQueue`, but go indirectly through `SimulationObject` methods. The additional `SimulationObject` methods are:

SimulationObject instance protocol

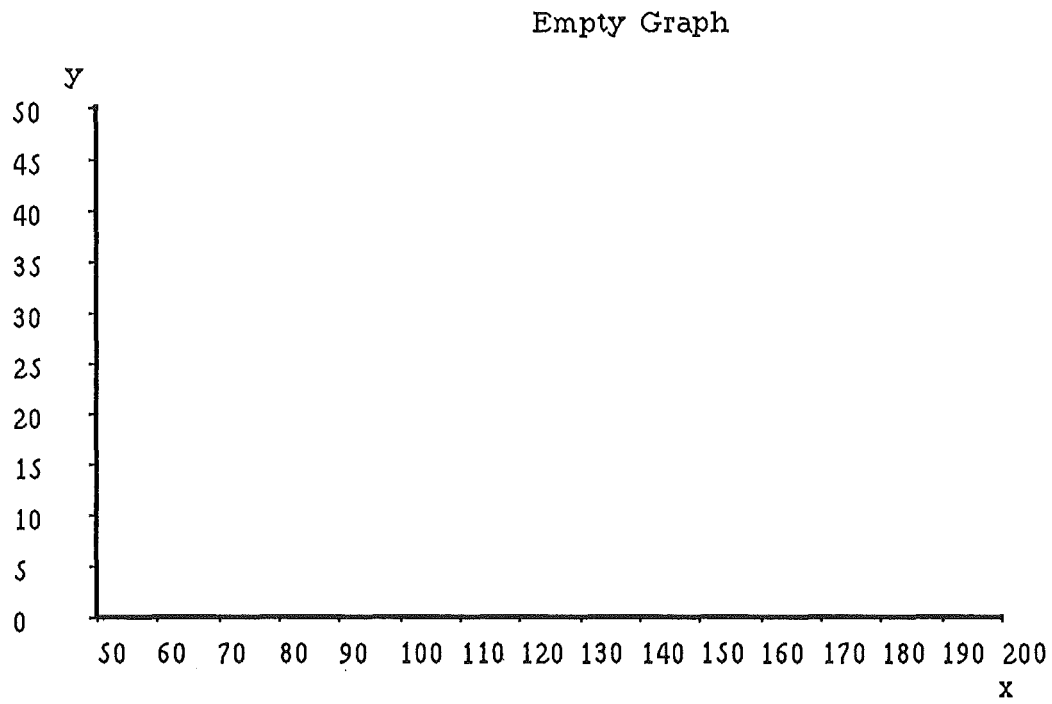
joinCondQueue: queueName until: conditionBlock	Send waitUntil: to the ConditionQueue called queueName.
testCondQueue: queueName	Send test to the ConditionQueue called queueName.

Because `SimulationObjects` rely on `Simulation` to supply a `ConditionQueue` of

the specified name, `Simulation` must have the following message added:

Simulation instance protocol

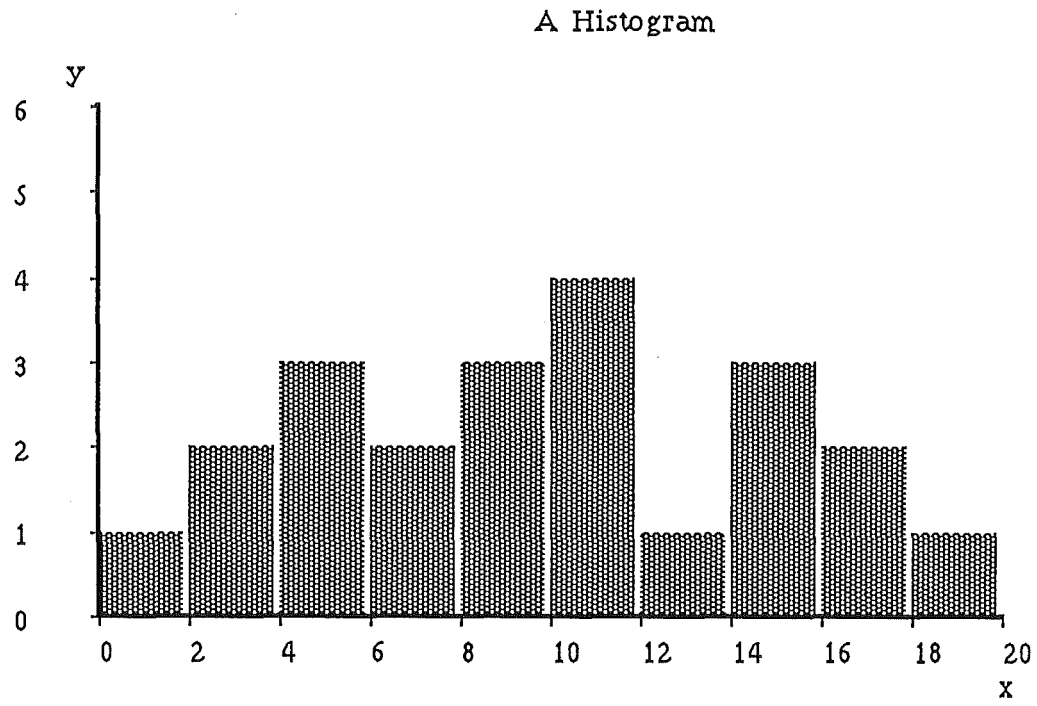
makeCondQueue: queueName	Create and store a ConditionQueue called queueName. This message should be sent by a user subclass in defineArrivalSchedule.



Created by:

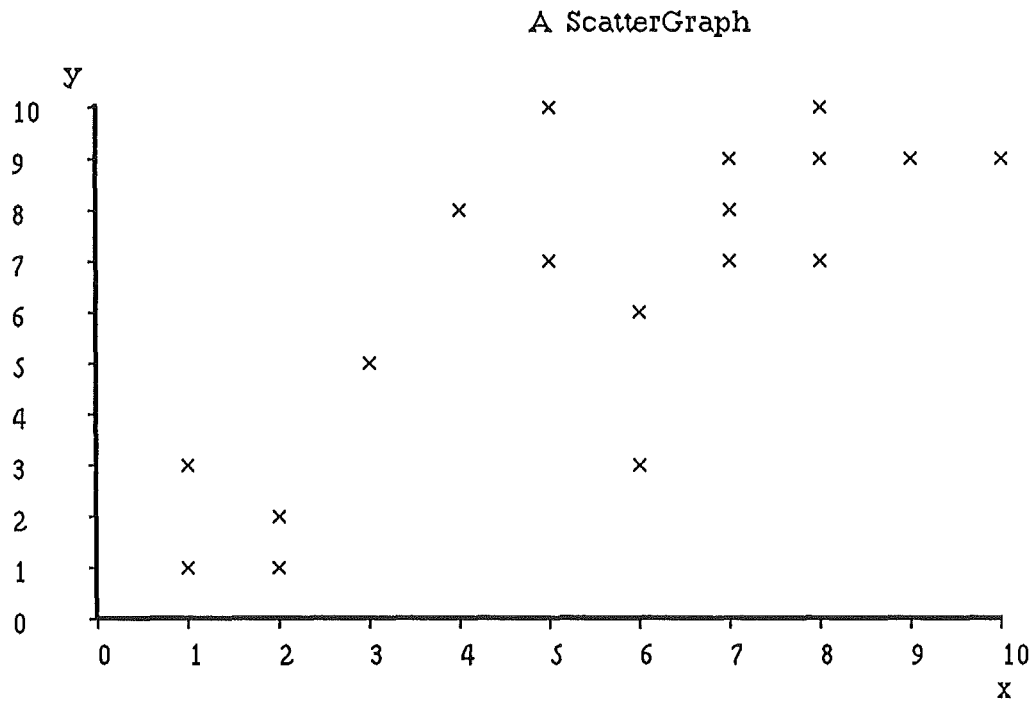
```
aGraph ← Axes title: 'Empty Graph' xAxis: 'x' yAxis: 'y'.
aGraph extent: 500@400.
aGraph origin: 120@340 xLength: 350 yLength: 200.
aGraph xLower: 50 upper: 200 step: 10.
aGraph yLower: 0 upper: 50 step: 5.
aGraph draw display.
```

Figure 3.1



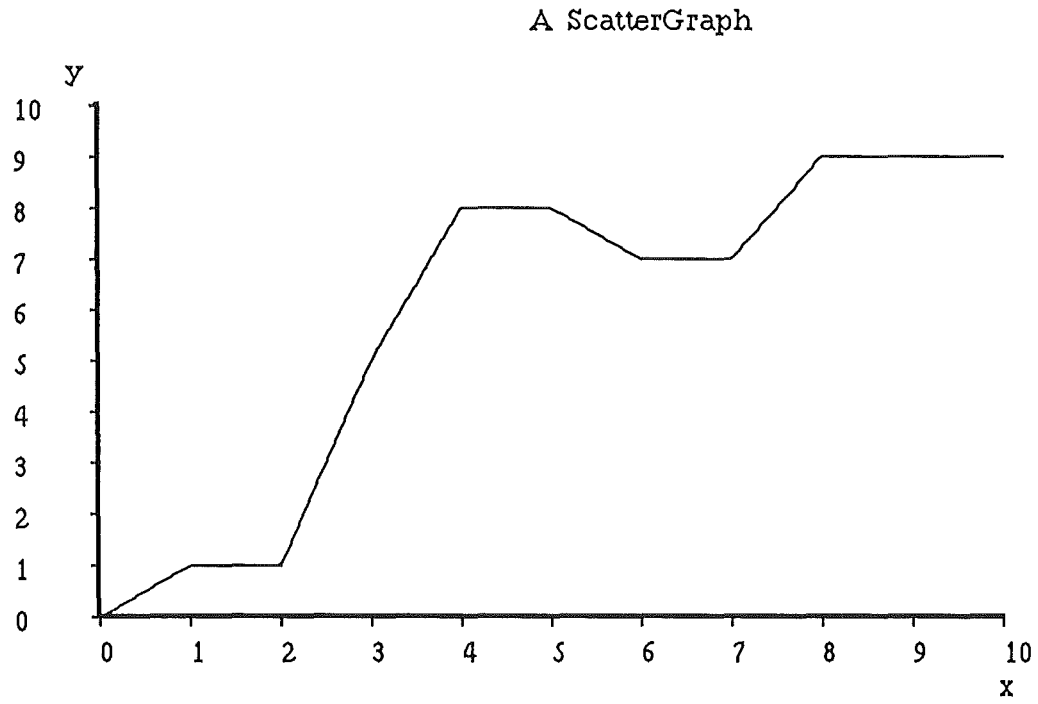

```
aGraph ← Histogram from: 0 to: 20 by: 2.
aGraph store: 10.  aGraph store: 14. ....
(aGraph graphForm: 'A Histogram' xAxis: 'x' yAxis: 'y') display
```

Figure 3.2



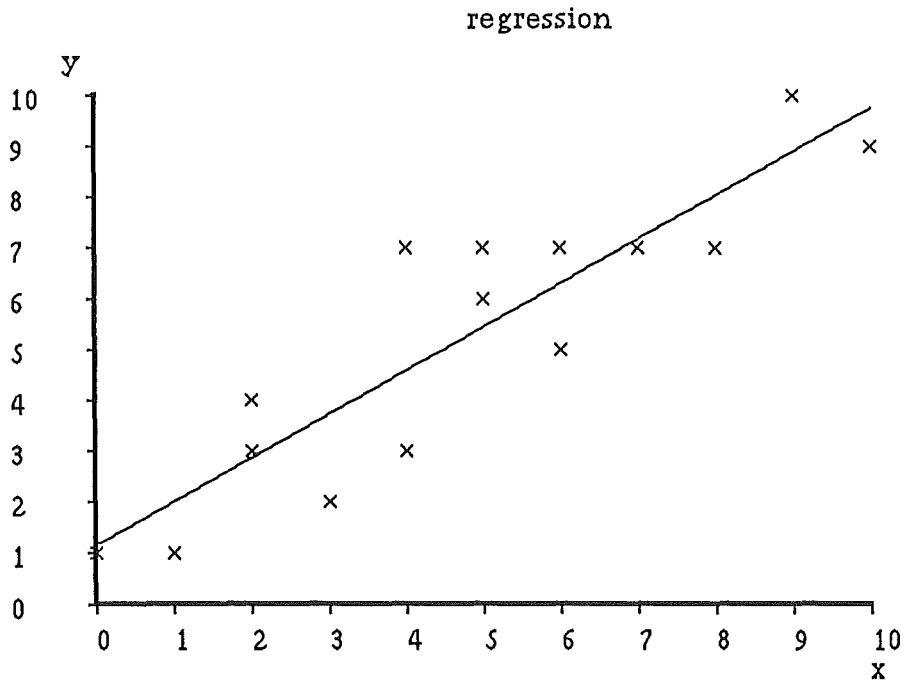

```
aGraph ← ScatterGraph title: 'A ScatterGraph' xAxis: 'x' yAxis: 'y'.
aGraph extent: 500@400.
aGraph origin: 120@300 xLength: 350 yLength: 200.
aGraph xLower: 0 upper: 10 step: 1.
aGraph yLower: 0 upper: 10 step: 1.
points ← OrderedCollection new.
points add: 1@1. points add: 2@3. ...
aGraph data: points.
aGraph plotPoints display
```

Figure 3.3




```
aGraph ← ScatterGraph title: 'A ScatterGraph' xAxis: 'x' yAxis: 'y'.
aGraph extent: 500@400.
aGraph origin: 120@300 xLength: 350 yLength: 200.
aGraph xLower: 0 upper: 10 step: 1.
aGraph yLower: 0 upper: 10 step: 1.
points ← OrderedCollection new.
points add: 0@0. points add: 1@1. ...
aGraph data: points.
aGraph joinPoints display
```

Figure 3.4




```
aGraph    Regression new.
aGraph updateAt: 0 store: 1. ....
(aGraph graphForm: 'regression' xAxis: 'x' yAxis: 'y') display
```

Figure 3.5

4.0 COMPARISON WITH DEMOS

4.1 Features

This section gives a comparison of features available to the user in DEMOS with features in the completed Smalltalk simulation system.

Distributions Below is a list of the names of DEMOS distributions, with Smalltalk counterparts.

DEMOS	Smalltalk
CONSTANT	Constant
ERLANG	Erlang
EMPIRICAL	Empirical
NEGEXP	Exponential
NORMAL	Normal
POISSON	Poisson
RANDINT	RandInt
DRAW	Bernoulli

In addition, Smalltalk provides the following distributions:

Smalltalk
SampleSpace
Binomial
Geometric
Gamma

File Input DEMOS uses READDIST to allow file input to EMPIRICAL. In Smalltalk this class is unnecessary, since streams already provide access to disk files if desired.

Antithetics Antithetic variates are available in DEMOS. This technique attempts to reduce the variance of values sampled from a distribution. Two identical runs are executed; the first with ANTITHETIC set to false, and the second with ANTITHETIC true. The first run generates random variates via the inverse distribution method from a stream of seeds, U. The second run uses the same stream, but changes U to 1- U. The values produced in each run are expected to be inversely correlated.

In the Smalltalk system, the seeds for the inverse distribution method are drawn from an instance of class `Random`. This random number generator has its own initial seed genaration, so sequences of random numbers are not reproduceable without modifying this feature. Because of the usefulness of automatic seeds, antithetics have not been implemented.

Entities

The discussion here does not cover resource and conditional queueing protocols. These are dealt with in the final section.

Scheduling

DEMOS uses the `SCHEDULE` and `HOLD` procedures to produce arrivals of entities into the simulation. Smalltalk provides the more intuitive methods:

```
schedule:at
scheduleArrivalOf:At
scheduleArrivalOf:accordingTo:
scheduleArrivalOf:accordingTo:startingAt:
```

The DEMOS procedure `INTERRUPT` removes an entity from whatever queue it is in and schedules it immediately. It has not been added to the Blue Book System. In DEMOS, actual entities wait in queues and so can be removed and otherwise processed. In the Smalltalk simulator, queues contain `DelayedEvents`, and entities await the return of the method which created the `DelayedEvent`. Inspection of the `DelayedEvents` in the queue qives no clue as to which entities are waiting on them. The consequence is that `INTERRUPT` cannot be implemented in any straightforward way.

DEMOS's `HOLD` is equivalent to `delayFor:`. Smalltalk also has `delayUntil:`, to allow referencing by absolute time as well as relative time.

Simulation Control

The duration of a simulation run (in simulated time) is determined in DEMOS by a `HOLD` or by passivating the main program just before it terminates and having an entity reschedule it when the simulation should end.

The Smalltalk simulator is slightly more flexible. If a simulation runs out of events to execute, it terminates. Earlier termination may be effected by scheduling `self finishUp` in `defineArrivalSchedule`, or by any entity sending `ActiveSimulation finishUp` at any time. A further technique is for the user to cease sending `proceed` to a simulation.

Resources

`RES` and `BIN` differ only because `BIN`s have no upper limit. In the Blue Book System, both are handled by `ResourceProvider`. An upper limit may be imposed by entities only releasing what they have acquired.

Queues

In Smalltalk, queues are treated as resources. `WAITQ` is equivalent to `ResourceCoordinator`, and `CONDQ` is the parent of `ConditionQueue`. The non-specific utility `QUEUE` can be replaced by standard `SortedCollections` in Smalltalk.

Data Collection

Below is a list af data collectors available in both systems:

DEMOS	Smalltalk
COUNT	Count
TALLY	Tally
HISTOGRAM	Histogram
ACCUMULATE	Accumulate
REGRESSION	Regression
TRACE	EventMonitor
SNAPSQS	----
SNAPQUEUES	----

The last two DEMOS features, `SNAPSQS` and `SNAPQUEUES` cannot easily be implemented in the Blue Book System for exactly the same reason that `INTERRUPT` cannot.

4.2 Worked Example

This section lists a DEMOS example program, as given in the DEMOS implementation guide (program 8), and a similar Smalltalk program.

DEMOS

```
BEGIN EXTERNAL CLASS DEMOS;
DEMOS
        BEGIN REF(RES)TUGS, JETTIES;
        REF(CONDQ)DOCKQ; BOOLEAN LOWTIDE;
        REF(RDIST)NEXT, DISCHARGE;

        ENTITY CLASS BOAT;
        BEGIN
                NEW BOAT("BOAT").SCHEDULE(NEXT.SAMPLE);
        DOCK:
                JETTIES.ACQUIRE(1);
                DOCKQ.WAITUNTIL(TUGS.AVAIL >= 2 AND
                                                NOT LOWTIDE);
                TUGS.ACQUIRE(2);
                HOLD(2.0);
                TUGS.RELEASE(2);
                DOCKQ.SIGNAL;
        UNLOAD:
                HOLD(DISCHARGE.SAMPLE);
        LEAVE:
                TUGS.ACQUIRE(1);
                HOLD(2.0);
                TUGS.RELEASE(1); JETTIES.RELEASE(1);
                DOCKQ.SIGNAL;
        END***BOAT***;

        ENTITY CLASS TIDE;
        BEGIN
        LOWTIDEON:
                LOWTIDE := TRUE;
                HOLD(4.0);
```

```
            DOCKQ.SIGNAL;
        LOWTIDEOFF:
            HOLD(9.0);
            REPEAT;
        END***TIDE***;

        OUTF :- NEW OUTFILE("P8");
        OUTF.OPEN(BLANKS(70));
        TRACE;
        TUGS          :- NEW RES("TUGS", 3);
        JETTIES       :- NEW RES("JETTIES", 2);
        DOCKQ         :- NEW CONDQ("DOCKQ");
        NEXT          :- NEW NEGEXP("NEXT BOAT", 0.1);
        DISCHARGE     :- NEW RES("DISCHARGE", 14.0, 3.0);
        NEW TIDE("TIDE").SCHEDULE(1.0);
        NEW BOAT("BOAT").SCHEDULE(0.0);
        HOLD(50.0);
        NOTRACE;
        HOLD(28.0*24.0-50.0);
        END;
END;
```

Smalltalk

```
Simulation subclass WharfSimulator
instance variables    lowTide
instance methods
initialize
        super initialize.
        lowtide ← true

defineArrivalSchedule
        self scheduleArrivalOf: Boat accordingTo:
             (Exponential named:'Next Boat' parameter:0.1).
        self scheduleArrivalOf: (Tide new) at: 1.0.
        self schedule: [self finishUp] at: 28*24-50
```

```
defineResources
        self produce: 3 of: 'Tugs'.
        self produce: 2 of: 'Jetties'.
        self makeCondQueue: 'Dockq'

lowTide: aBoolean
        lowTide ← aBoolean

lowTide
        ↑lowTide
```

SimulationObject subclass Boat

instance methods

```
tasks
        | myJetty myTugs |
        myJetty ← self acquire: 1 ofResource: 'Jetties'.
        self joinCondQueue: 'Dockq' until:
                [(self inquireFor: 2 ofResource: 'Tugs')
                        & (ActiveSimulation lowTide) not]
        myTugs ← self acquire: 2 ofResource: 'Tugs'.
        self holdFor: 2.0.
        self release: myTugs.
        self testCondQueue: 'Dockq'.
        self holdFor: (Normal named: 'Discharge'
                                mean: 14.0 deviation: 3.0)next.
        myTugs ← self acquire: 1 ofResource: 'Tugs'.
        self holdFor: 2.0.
        self release: myTugs.
        self release: myJetty.
        self testCondQueue: 'Dockq'
```

```
SimulationObject subclass Tide
instance methods
tasks
        [ActiveSimulation lowTide: true.
        self holdFor: 4.0.
        ActiveSimulation lowTide: false.
        self holdFor: 9.0] whileTrue
```

5.0 BIBLIOGRAPHY

(1) Adele GoldBerg and David Robson.
SMALLTALK-80 THE LANGUAGE AND ITS IMPLEMENTATION
(Addison-Wesley, Reading, MA, 1983)

(2) Graham Birtwistle.
DEMOS IMPLEMENTATION GUIDE AND REFERENCE MANUAL
Research Report no. 81/70/22
(University of Calgary, 1981)

(1) Donald E. Knuth.
THE ART OF COMPUTER PROGRAMMING, VOL 2
Seminumerical Algorithms
(Addison-Wesley, Reading, MA, 1969)

APPENDIX A

1.1.3 Smalltalk Basics

The purpose of this section is to give the reader unfamiliar with Smalltalk enough information to understand most of the discussion occurring later in the report. Smalltalk is a large and powerful programming environment based on a small number of universally applied principles. These ideas are unconventional and have correspondingly unusual terminology, but once they are grasped the language becomes simple and rewarding to use. Because of the extent of the Smalltalk system, any treatment here must be rudimentary. A full description is given in the Blue Book.

Objects

Smalltalk is the archetypal object-oriented programming language. Everything is done using objects. An object is any component of the programming system. For example, the following are all objects:

```
3, 'a string', aRectangle, Resource, Compiler...
```

Variables and Methods

Objects have their own private variables and a number of methods which define the functions they can perform. Methods can be invoked by sending a message to an object. Messages consist of the name of the method to be executed and any parameters required. For example, `aRectangle` can be defined as a 50 by 50 square with its top left corner at the point (0,0) on the screen by:

```
aRectangle origin: 0@0 extent: 50@50.
```

This piece of code sends the message `origin:extent:` to `aRectangle` with the appropriate parameters. `aRectangle` responds by executing the method:

```
origin: topLeft extent: distancePoint
        origin ← topLeft.
        corner ← origin + distancePoint
```

This method sets the variables `origin` and `corner` to the required values. The variables are internal to `aRectangle` and may only be accessed by its methods.

Methods always return a value. For example, `aRectangle center` will execute:

```
center
        ↑self topLeft + self bottomRight // 2
```

The uparrow signifies the value to be returned. It is the lowest precedence operator. If a method does not explicitly return a result, the object containing the method is returned.

Methods

Methods always consist of sending more messages to other objects, except at the most basic level, where a primitive is called to actually do something. The user never needs to be aware of this level.

Classes

All objects belong to a class. They are said to be instances of that class. Classes define the variables and methods which will be used by their instances. Thus each instance has the same set of instance variables, although these variables may have different values.

Classes are themselves objects, and so may have their own variables and methods. Class variables occur only once and their values are global to all instances of the class. Class methods are normally used to create new instances and to initialize class variables.

Inheritance

Classes form a hierarchical structure of inheritance. A subclass inherits all the variables and methods of it's superclass. If a message is sent to an object which does not have the corresponding method defined in it's class, then the message is passed up to the superclass. This process is repeated until the method is found, or the root class `object` is reached, in which case a `doesNotUnderstand` error message results.

Processes

Smalltalk supports (conceptually) concurrent processes. Two or more objects may be apparently executing methods at the same time. Communication between these processes is normally via message passing, but when synchronisation is required, semaphores provide the necessary mechanism.

Simulation in Smalltalk

Smalltalk is a particularly elegant language for event-driven simulation. Independently functioning objects form a very natural representation of simulation entities. In addition, Smalltalk provides graphics facilities for bitmapped screens, allowing pleasing result presentation.

Restrictions

The version of Smalltalk on which this system has been implemented is Apple's level0 image. This is a reduced system, with many standard classes removed. The absence of these classes, and the frequency with which the system fails, imposed some restrictions on the development of additions to the simulator. In some cases, the methods used are more complex than would be possible on a full system because the most straightforward methods are missing. The result, however, should run on a standard Smalltalk system.

Internal memory space on a Macintosh Plus under level0 is also a limiting factor. With the additional code of the simulator included, remaining space is too small to allow anything other than small example simulations of a few classes to be written. Only those simulator classes required for a particular example should be loaded into the system. They should be filed-out and removedwhenever possible.