

# **PLT Scheme Simulation Collection**

Reference Manual  
Edition 1.0 for Version 1.0  
Draft

M. Douglas Williams

January 2006

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Routines Available in the Simulation Collection . . . . .	1
1.2	The Simulation Collection is Free Software . . . . .	2
1.3	Obtaining the Simulation Collection . . . . .	2
1.4	No Warranty . . . . .	2
<b>2</b>	<b>Using the Simulation Collection</b>	<b>3</b>
2.1	An Example . . . . .	3
2.2	Loading the Simulation Collection . . . . .	3
2.3	Graphics Modules . . . . .	3
<b>3</b>	<b>Simulation Environments (Basic)</b>	<b>4</b>
3.1	The <code>simulation-environment</code> Structure . . . . .	5
3.2	Current Simulation Environment . . . . .	6
<b>4</b>	<b>Simulation Control (Basic)</b>	<b>9</b>
4.1	Scheduling Events and Processes . . . . .	9
4.2	Controlling the Simulation Main Loop . . . . .	10
4.3	Simulating Waiting and Working . . . . .	10
<b>5</b>	<b>Events</b>	<b>11</b>
5.1	The <code>event</code> Structure . . . . .	11
5.2	Event Lists . . . . .	12
5.3	Example – Functions as Events . . . . .	13
<b>6</b>	<b>Processes</b>	<b>15</b>
6.1	Defining a Process . . . . .	15
6.2	Creating and Accessing Processes . . . . .	15
6.2.1	The <code>process</code> Structure . . . . .	15
6.2.2	Creating Processes . . . . .	16
6.2.3	Process States . . . . .	17
6.3	Example – Processes . . . . .	17
<b>7</b>	<b>Resources</b>	<b>19</b>
7.1	Example – Resources . . . . .	19

<b>8 Data Collection</b>	<b>22</b>
8.1 Variables	22
8.2 Tally and Accumulate	22
8.2.1 Example – Tally and Accumulate Example	22
8.3 Statistics	25
8.4 History	25
8.4.1 History Graphics	25
8.5 Example – Data Collection	25
8.6 Data Collection Across Multiple Simulation Runs	27
8.6.1 Open Loop Processing	27
8.6.2 Closed Loop Processing	28
<b>9 Sets</b>	<b>31</b>
9.1 Example – Furnace Model 1	31
<b>10 Continuous Simulation Models</b>	<b>34</b>
10.1 Continuous Variables	34
10.2 Simulation Control (Continuous)	34
10.2.1 Example – Furnace Model 2	34
10.3 Simulation Environment (Continuous)	40
10.3.1 Fixed Step Size	40
10.3.2 Limiting Step Size	43
10.4 Furnace Model 3	45
<b>11 Simulation Classes</b>	<b>46</b>
11.1 Process Classes	46
11.2 Resource Classes	46
11.3 Example – Simulation Classes	46
<b>12 Simulation Control (Advanced)</b>	<b>49</b>
12.1 Example – Harbor Model	49
<b>13 Simulation Environments (Hierarchical)</b>	<b>53</b>
<b>14 Components</b>	<b>54</b>
<b>A GNU Lesser General Public License (LGPL)</b>	<b>55</b>
<b>B Simplified Simulation System</b>	<b>66</b>
<b>Bibliography</b>	<b>70</b>
<b>Index</b>	<b>71</b>

# List of Figures

# List of Tables

8.1 Statistics Computations . . . . .	24
---------------------------------------	----

# Chapter 1

## Introduction

The PLT Scheme Simulation Collection implements a combined discrete-event and continuous simulation engine for developing simulation models in PLT Scheme. The simulation engine:

- Provides a process-based simulation engine
- Provides combined discrete-event and continuous simulation models
- Provides automatic data collection

The source code is distributed with the simulation collection and licensed under the GNU Lesser General Public License (LGPL)[?].

The motivation behind the PLT Scheme Simulation Collection is to provide the simulation engine for developing knowledge-based simulations in PLT Scheme. It is based on work originally done in Symbolics Common Lisp. This is not as much a port of the earlier work as a complete re-engineering of the system into PLT Scheme. In particular, it makes extensive use of continuations to provide a process-based approach to building simulation models. It also adds support for building continuous simulation models, which were not part of the original work.

### 1.1 Routines Available in the Simulation Collection

The PLT Scheme Simulation Collection provides the following functionality for building and executing simulation models:

- Simulation Environments (Basic)
- Simulation Control (Basic)
- Events

- Processes
- Resources
- Data Collection
- Sets
- Continuous Simulation Models
- Simulation Classes
- Simulation Control (Advanced)
- Simulation Environments (Hierarchical)

The use of these functions is described in this manual. Each chapter provides detailed definitions of the functions, with example code.

## 1.2 The Simulation Collection is Free Software

The PLT Scheme Simulation Collection is free software – this means that anyone is free to use it and to redistribute it in other free programs. The simulation collection is not in the public domain – it is copyrighted and there are conditions on its distribution. Specifically, the PLT Scheme Simulation Collection is distributed under the GNU Lesser General Public License (LGPL)[?]. A copy of the LGPL is provided as Appendix A of this document.

## 1.3 Obtaining the Simulation Collection

The preferred method for obtaining the PLT Scheme Simulation Collection is via the PLaneT Package Repository (PLaneT), PLT Scheme’s centralized package distribution system[?]. The PLaneT identifier for the PLT Scheme Simulation Collection, Version 1.0 (or later) is ("williams" "simulation.plt" 1 0). PLT Scheme will automatically download and install the simulation collection from the PLaneT server. See Chapter 2 for an example.

Note that Version 1.0 of the PLT Scheme Simulation Collection requires PLT Scheme Version 300 or higher.

## 1.4 No Warranty

The PLT Scheme Simulation Collection is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. It is your responsibility to validate the behavior of the software and their accuracy using the source code provided. See the GNU Lesser General Public License (LGPL)[?] for more details.

## Chapter 2

# Using the Simulation Collection

This chapter describes how to use the PLT Scheme Simulation Collection and introduces its conventions.

The simulation collection requires the PLT Scheme Science Collections and automatically loads required portions it using PLaneT. Code using the simulation collection often also requires the science collection and must load it as specified in the PLT Scheme Science Collection Reference Manual.

### 2.1 An Example

### 2.2 Loading the Simulation Collection

### 2.3 Graphics Modules

## Chapter 3

# Simulation Environments (Basic)

A simulation environment encapsulates the state of a simulation model. The basic state information includes the following:

- Time
- Now and future event lists
- Main loop and exit continuations
- Process and event being executed

Multiple simulation environments may exist at the same time. This is useful for implementing various simulation techniques.

Nested simulation environments are useful for data collection across multiple simulation model runs. Typically, an outer simulation model defines the global simulation data to be collected across the model runs. It then executes the inner simulation model multiple times and updates the global simulation data with the results of each run. See the Open Loop and Closed Loop examples in Chapter ?? for examples.

Nested simulation environments can be used to allow a lower fidelity (but faster) simulation model to reach a steady state before starting a higher fidelity simulation model.

Multiple simulation environments can facilitate the development of multiple, independent (or cooperating) simulation models as part of a larger system.

Note that these usages of multiple simulation environments is different than hierarchical simulation environments, which allow partitioning of simulation elements within a simulation model. Hierarchical simulation models are described in Chapter ??.

This section describes the basic features of simulation environments.

### 3.1 The simulation-environment Structure

simulation-environment

Structure: simulation-environment

Contract: (struct simulation-environment  
 ((running boolean?)  
 (time (>=/c 0.0))  
 (now-event-list event-list?)  
 (future-event-list event-list?)  
 (loop-next (union/c procedure? #f))  
 (loop-exit (union/c procedure? #f))  
 (event (union/c event? #f))  
 (process (union/c process? #f))))

This structure defines a simulation environment.

**running?** – #t if the simulation main loop is running in this simulation environment, #f otherwise.

**time** – The simulation clock for the simulation model.

**now-event-list** – An event list containing the events to be executed now, i.e. before the simulation clock is advanced.

**future-event-list** – An event list containing the events to be executed in the (simulated) future, i.e., the simulation clock is advanced between these events.

**loop-next** – The continuation to return to the simulation main loop running in this simulation environment. Calling this continuation signifies the end of processing of the current event. This field is #f if the simulation main loop is not executing in this simulation environment.

**loop-exit** – The continuation to exit the simulation main loop running in this simulation environment. Calling this continuation signifies the end of the execution of the simulation model. This field is #f if the simulation main loop is not executing in this simulation environment.

**event** – The event currently executing in the simulation environment, or #f.

**process** – The process currently executing in the simulation environment, or #f.

make-simulation-environment

Function: (make-simulation-environment)

Contract: (-> simulation-environment)

This function returns a new simulation environment with `time = 0.0`, the event lists are empty, and the simulation main loop is not running (i.e. `running? = #f`, `loop-next = #f`, `loop-exit = #f`, `event = #f`, and `process = #f`).

`default-simulation-environment`

Variable: `default-simulation-environment`

This variable contains the simulation environment that is used as the default value for the current simulation environment (see Section 3.2).

## 3.2 Current Simulation Environment

`current-simulation-environment`

Function: `(current-simulation-environment env)`

Function: `(current-simulation-environment)`

Contract: `(case-> (-> simulation-environment? any)  
(-> simulation-environment?))`

Gets or sets the current simulation environment.

`current-simulation-running?`

Function: `(current-simulation-running? running?)`

Function: `(current-simulation-running?)`

Contract: `(case-> (-> boolean? any)  
(-> boolean?))`

Sets or gets the `running?` field of the current simulation environment.

`current-simulation-time`

Function: `(current-simulation-time time)`

Function: `(current-simulation-time)`

Contract: `(case-> (-> (>=/c 0.0) any)  
(-> (>=/c 0.0)))`

Sets or gets the `time` field of the current simulation environment.

`current-simulation-now-event-list`

Function: `(current-simulation-now-event-list now-event-list)`

Function: `(current-simulation-now-event-list)`

Contract: `(case-> (-> (union/c event-list? #f) any)  
(-> (union/c event-list? #f)))`

Sets or gets the `now-event-list` field of the current simulation environment.

`current-simulation-future-event-list`

Function: (`current-simulation-future-event-list future-event-list`)

Function: (`current-simulation-future-event-list`)

Contract: (`case-> (-> (union/c event-list? #f) any)`  
                   (`-> (union/c event-list? #f))`)

Sets or gets the `future-event-list` field of the current simulation environment.

`current-simulation-loop-next`

Function: (`current-simulation-loop-next loop-next`)

Function: (`current-simulation-loop-next`)

Contract: (`case-> (-> (union/c procedure? #f) any)`  
                   (`-> (union/c procedure? #f))`)

Sets or gets the `loop-next` field of the current simulation environment.

`current-simulation-loop-exit`

Function: (`current-simulation-loop-exit loop-exit`)

Function: (`current-simulation-loop-exit`)

Contract: (`case-> (-> (union/c procedure? #f) any)`  
                   (`-> (union/c procedure? #f))`)

Sets or gets the `loop-exit` field of the current simulation environment.

`current-simulation-event`

Function: (`current-simulation-event event`)

Function: (`current-simulation-event`)

Contract: (`case-> (-> (union/c event? #f) any)`  
                   (`-> (union/c event? #f))`)

Sets or gets the `event` field of the current simulation environment.

`current-simulation-process`

Function: (`current-simulation-process process`)

Function: (`current-simulation-process`)

Contract: (`case-> (-> (union/c process? #f) any)`  
                   (`-> (union/c process? #f))`)

Sets or gets the `process` field of the current simulation environment.

`with-simulation-environment`

Macro: (`with-simulation-environment simulation-environment`  
                   `body ...`)

This macro evaluates its body with the current simulation environment bound to the value of `simulation-environment`.

`with new-simulation-environment`

Macro: (`with-new-simulation-environment`  
body ...)

This macro evaluates its body with the current simulation environment bound to a newly created `simulation-environment`. This is most common way to create and use a new simulation environment.

## Chapter 4

# Simulation Control (Basic)

The PLT Scheme Simulation Collection provides functionality for basic simulation control. Basic simulation control includes the simulation main loop for controlling the execution of discrete events and processes, scheduling events and processes, and simulating passing of (simulated) time (i.e. waiting and working).

### 4.1 Scheduling Events and Processes

One of the main characteristics of discrete simulation models is the ability to schedule procedural simulation elements (e.g. events and processes) to occur at specified (simulated) times. Typically, an event list is maintained that contains entries for each of the scheduled elements.

In the simulation collection, each simulation environment has two such event lists: the now event list and the future event list. The now event list contains entries for elements that are to be executed before the simulation clock is advanced, i.e. now. The future event list contains entries for elements that are to be executed in the (simulated) future. (There is also a continuous event list used for continuous simulation models. It is described in Chapter 10.)

`schedule`

Macro: (`schedule now` (*function* . *arguments*)  
*simulation-environment*)

Macro: (`schedule (at time)` (*function* . *arguments*)  
*simulation-environment*)

Macro: (`schedule (in duration)` (*function* . *arguments*)  
*simulation-environment*)

Macro: (`schedule time` (*function* . *arguments*)  
*simulation-environment*)

Macro: (`schedule time-spec` (*function* . *arguments*))

This macro schedules events and processes for execution. If *function* is the name of a process, then a process instance is created and scheduled for execution

using the specified *arguments* at the specified time. Otherwise, *function* must evaluate to a procedural object and the function is scheduled to be applied to the specified *arguments* at the specified time. If *simulation-environment* is not supplied, (`current-simulation-environment`) is used.

The time is specified using one of the following forms:

- `now` – The event or process is scheduled to be executed before the simulation clock is advanced. That is, it is added to the now event list.
- (`at time`) – The event or process is scheduled to be executed at the specified *time*. An error is signaled if *time* is in the (simulated) past. That is, *time* must be greater than or equal to (`simulation-environment-time simulation-environment`).
- (`in duration`) – The event or process is scheduled to be executed after the specified *duration* has elapsed. This is equivalent to (`at (+ duration (simulation-environment-time simulation-environment))`). An error is signaled if *duration* is negative.
- *time* – This is equivalent to (`at time`).

## 4.2 Controlling the Simulation Main Loop

`start-simulation`

Function: (`start-simulation`)

Contract: (`-> any`)

Begin execution of the simulation main loop in the current simulation environment. It saves the `loop-next` and `loop-exit` continuations in the current simulation environment. It then executes events and processes from the event list(s) until either there are no more to be executed, or the simulation main loop is explicitly exited.

`stop-simulation`

Function: (`stop-simulation`)

Contract: (`-> any`)

Terminates execution of the simulation main loop running in the current simulation environment. It calls (`current-simulation-loop-exit`).

## 4.3 Simulating Waiting and Working

`wait/work`

`wait`

`work`

# Chapter 5

## Events

In a simulation model, an *event* represents an action that will take place in the (simulated) future.

In the PLT Scheme Simulation Collection, an event represents the (simulated) future application of a procedural object to a list of arguments.

### 5.1 The event Structure

event

Structure: event

Contract: (struct event  
((time (<=/c 0.0))  
(process (union/c process? #f))  
(function (union/c procedure? #f))  
(argument list?)))

This structure defines an event. Note that since the `function` field can contain any procedural object, including continuations, any event can call the `wait/work` function. This is a slight extension to the definition of event given above in that an event may represent a sequence of actions.

`time` – The (simulated) time the event is to occur.

`process` – The process owning the event or `#f`.

`function` – The procedural object containing the code that represents the event or `#f`.

`arguments` – A list of the arguments for the `function`.

## 5.2 Event Lists

An *event list* stores a list of events. Events are stored in order of their `time` values.

`event-list`

Structure: `event-list`

Contract: `(struct event-list  
                  (events list?))`

This structure defines an event list. The current implementation just encapsulates a list that stores the events.

`events` – A list of the events on the event list.

`make-event-list`

Function: `(make-event-list)`

Contract: `(-> event-list?)`

This function returns a new event list that is empty.

`event-list-empty?`

Function: `(event-list-empty? event-list)`

Contract: `(-> event-list? boolean?)`

This function returns `true`, `#t`, if the `event-list` is empty, and `false`, `#f` otherwise.

`event-list-add!`

Function: `(event-list-add! event-list event)`

Contract: `(-> event-list? event? any)`

This function adds an event to an event list. Currently, the event list is ordered by `time` only.

`event-list-remove!`

Function: `(event-list-remove! event-list event)`

Contract: `(-> event-list? event? any)`

This function removes an event to an event list. No error is signaled if the specified `event` is not on the event-list.

`event-list-pop!`

Function: `(event-list-pop! event-list)`

Contract: `(-> event-list? event?)`

This function removes the first event from the event list and returns it.

### 5.3 Example – Functions as Events

This example is a simulation model of a simple system. Customer arrivals are exponentially distributed with an interarrival time of 4.0 minutes. The time a customer remains in the system is uniformly distributed between 2.0 and 10.0 minutes. The output is a simple trace of customer arrivals and departures.

The PLT Scheme Science Collection provides the random distribution functions.

```

; Example 0 - Functions as Events

(require (planet "simulation.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

(define (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (schedule now (customer i))))

(define (customer i)
  (printf "~a: customer ~a enters~n"
          (current-simulation-time) i)
  (work (random-flat 2.0 10.0))
  (printf "~a: customer ~a leaves~n"
          (current-simulation-time) i))

(define (run-simulation n)
  (with-new-simulation-environment
    (schedule (at 0.0) (generator n))
    (start-simulation)))

```

The simulation model is executed by calling `(run-simulation n)`, where `n` is the number of customer to simulate through the system. For example, `(run-simulation 10)` produces the following output:

```

>(run-simulation 10)
0.6153910608822503: customer 0 enters
5.599485116393393: customer 1 enters
6.411843645405005: customer 2 enters
8.48917994426752: customer 0 leaves
10.275428842274628: customer 1 leaves
14.749397986170655: customer 2 leaves
23.525886616767437: customer 3 enters

```

```
27.18604340910279: customer 3 leaves
32.1644631797164: customer 4 enters
33.14558760001698: customer 5 enters
39.67682614849173: customer 4 leaves
40.486553934113665: customer 6 enters
41.168084930967424: customer 5 leaves
45.72670063299798: customer 6 leaves
46.747675912143016: customer 7 enters
49.212327970772435: customer 8 enters
50.556538752352886: customer 9 enters
51.46738784004611: customer 8 leaves
52.514846525674855: customer 7 leaves
56.11635302397275: customer 9 leaves
>
```

# Chapter 6

## Processes

In a simulation model, a *process* represents an entity in the simulation that actively progresses through time.

In the PLT Scheme Simulation Collection, a process encapsulates an event that that executes the body of the process; provides state information for the process; and, most importantly, provides a handle that allows the process to interact with other simulation objects (e.g. resources or other processes).

### 6.1 Defining a Process

A process is defined using the `define-process` macro.

`define-process`

Macro: (`define-process` (`name` . `arguments`)  
          `body` ...)

This macro defines a new process with the specified `name`. Syntactically, the `define-process` macro is the same as `define` for a function – indeed, an unnamed procedural object is created and associated with the process. However, the simulation collection maintains references to process object, thus allowing them to interact with each other.

The variable `self` is bound to the process instance during the execution of the process.

### 6.2 Creating and Accessing Processes

#### 6.2.1 The process Structure

`process`

Structure: `process`

Contract: `(struct process  
                  ((process-def process-def?)  
                  (event event?)  
                  (state integer?)))`

This structure defines a process. The `process-def` field points to a structure that contains information from the process definition needed internally by the simulation collection. The `event` field contains the event object that represents the execution of the body of the process. The `state` field contains the state of the process. Note that there are other fields used for continuous processes. These are described in Chapter 10, Continuous Simulation Models.

There are a few short-cut functions that return information from the other structures pointed to by a process.

`process-name`

Function: `(process-name process)`

Contract: `(-> process? symbol?)`

This function returns the name of the specified `process`.

`process-time`

Function: `(process-time process)`

Contract: `(-> process? real?)`

This function returns the next event time of the specified `process`. This is the value of the `time` field of the `event` field of the process. This is useful in advanced simulations using the `interrupt` and `resume` simulation control functions.

`set-process-time!`

Function: `(set-process-time! process time)`

Contract: `(-> process? real? any)`

This function sets the next event time of the specified `process`. It sets the value of the `time` field of the `event` field of the process. This is useful in advanced simulations using the `interrupt` and `resume` simulation control functions.

## 6.2.2 Creating Processes

The normal way to create a process is using the `schedule` macro as described in Section 4.1, Scheduling Events and Processes.

There is also a primitive `make-process` function that creates a process instance without scheduling its initial execution.

`make-process`

### 6.2.3 Process States

## 6.3 Example – Processes

This example is the same as the simulation model in Chapter 5. Indeed, the only syntactic difference is the use of `define-process` instead of `define` for the generator and customer processes.

```

; Example 1 - Processes

(require (planet "simulation.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

(define-process (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (schedule now (customer i))))

(define-process (customer i)
  (printf "~a: customer ~a enters~n"
          (current-simulation-time) i)
  (work (random-flat 2.0 10.0))
  (printf "~a: customer ~a leaves~n"
          (current-simulation-time) i))

(define (run-simulation n)
  (with-new-simulation-environment
    (schedule (at 0.0) (generator n))
    (start-simulation)))

```

The output is identical to that of the simulation model in Chapter 5.

```

>(run-simulation 10)
0.6153910608822503: customer 0 enters
5.599485116393393: customer 1 enters
6.411843645405005: customer 2 enters
8.48917994426752: customer 0 leaves
10.275428842274628: customer 1 leaves
14.749397986170655: customer 2 leaves
23.525886616767437: customer 3 enters
27.18604340910279: customer 3 leaves
32.1644631797164: customer 4 enters
33.14558760001698: customer 5 enters

```

```
39.67682614849173: customer 4 leaves
40.486553934113665: customer 6 enters
41.168084930967424: customer 5 leaves
45.72670063299798: customer 6 leaves
46.747675912143016: customer 7 enters
49.212327970772435: customer 8 enters
50.556538752352886: customer 9 enters
51.46738784004611: customer 8 leaves
52.514846525674855: customer 7 leaves
56.11635302397275: customer 9 leaves
>
```

# Chapter 7

## Resources

In a simulation model, a *resource* represents an entity (or entities) that is/are shared among processes.

### 7.1 Example – Resources

This example extends the simple system of Chapters 5 and 6 by adding attendants to the system. Before leaving the system, each customer must interact with an attendant. The two attendants are modeled using a single resource with two units. This means there is a single queue for the two attendants. Again the output is a simple trace of customer arrivals, attendant acquisitions, and departures.

```
; Example 2 - Resources

(require (planet "simulation.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

(define n-attendants 2)
(define attendant #f)

(define-process (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (schedule now (customer i))))

(define-process (customer i)
  (printf "~a: customer ~a enters~n"
```

```

        (current-simulation-time) i)
(resource-request attendant)
(printf "~a: customer ~a gets an attendant~n"
        (current-simulation-time) i)
(work (random-flat 2.0 10.0))
(resource-relinquish attendant)
(printf "~a: customer ~a leaves~n"
        (current-simulation-time) i))

(define (run-simulation n)
  (with-new-simulation-environment
    (set! attendant (make-resource n-attendants))
    (schedule (at 0.0) (generator n))
    (start-simulation)))

```

The simulation model is executed by calling `(run-simulation n)`, where `n` is the number of customer to simulate through the system. For example, `(run-simulation 10)` produces the following output:

```

>(run-simulation 10)
0.6153910608822503: customer 0 enters
0.6153910608822503: customer 0 gets an attendant
5.599485116393393: customer 1 enters
5.599485116393393: customer 1 gets an attendant
6.411843645405005: customer 2 enters
8.48917994426752: customer 0 leaves
8.48917994426752: customer 2 gets an attendant
10.275428842274628: customer 1 leaves
16.82673428503317: customer 2 leaves
23.525886616767437: customer 3 enters
23.525886616767437: customer 3 gets an attendant
27.18604340910279: customer 3 leaves
32.1644631797164: customer 4 enters
32.1644631797164: customer 4 gets an attendant
33.14558760001698: customer 5 enters
33.14558760001698: customer 5 gets an attendant
39.67682614849173: customer 4 leaves
40.486553934113665: customer 6 enters
40.486553934113665: customer 6 gets an attendant
41.168084930967424: customer 5 leaves
45.72670063299798: customer 6 leaves
46.747675912143016: customer 7 enters
46.747675912143016: customer 7 gets an attendant
49.212327970772435: customer 8 enters
49.212327970772435: customer 8 gets an attendant
50.556538752352886: customer 9 enters

```

51.46738784004611: customer 8 leaves  
51.46738784004611: customer 9 gets an attendant  
52.514846525674855: customer 7 leaves  
57.02720211166597: customer 9 leaves  
>

## Chapter 8

# Data Collection

### 8.1 Variables

### 8.2 Tally and Accumulate

#### 8.2.1 Example – Tally and Accumulate Example

```
;; Test Tally and Accumulate
(require (planet "simulation-with-graphics.ss"
               ("williams" "simulation.plt" 1 0)))

(define tallied #f)
(define accumulated #f)

(define-process (test-process value-duration-list)
  (let loop ((vdl value-duration-list))
    (when (not (null? vdl))
      (let ((value (caar vdl))
            (duration (cadar vdl)))
        (set-variable-value! tallied value)
        (set-variable-value! accumulated value)
        (wait duration)
        (loop (cdr vdl))))))

(define (main value-duration-list)
  (with-new-simulation-environment
    (set! tallied (make-variable))
    (tally (variable-statistics tallied))
    (tally (variable-history tallied))
    (set! accumulated (make-variable))
    (accumulate (variable-statistics accumulated))))
```

```

(accumulate (variable-history accumulated))
(schedule (at 0.0) (test-process value-duration-list))
(start-simulation)
(printf "--- Test Tally and Accumulate ---~n")
(printf "~n--- Tally ---~n")
(printf "N    = ~a~n" (variable-n tallied))
(printf "Sum  = ~a~n" (variable-sum tallied))
(printf "Mean = ~a~n" (variable-mean tallied))
(printf "~a~n" (history-plot (variable-history tallied)))
(printf "~n--- Accumulate ---~n")
(printf "N    = ~a~n" (variable-n accumulated))
(printf "Sum  = ~a~n" (variable-sum accumulated))
(printf "Mean = ~a~n" (variable-mean accumulated))
(printf "~a~n" (history-plot
              (variable-history accumulated))))

```

```

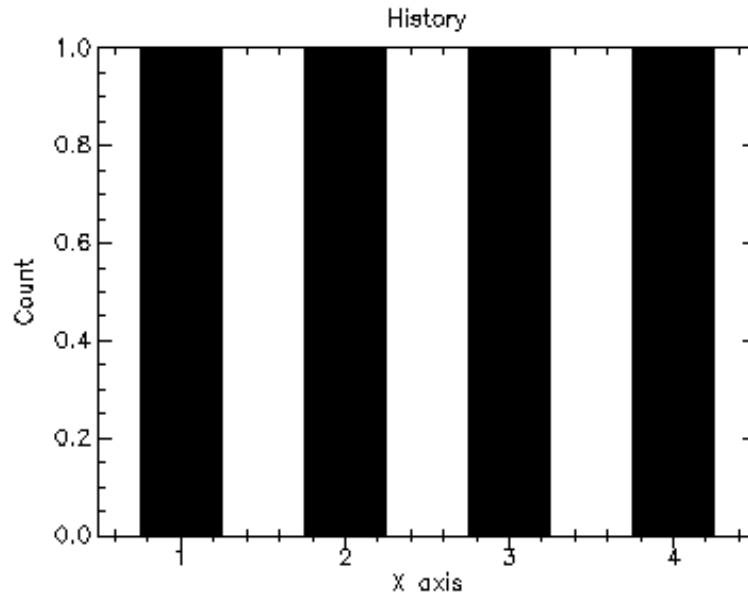
>(main '((1 2)(2 1)(3 2)(4 3)))
--- Test Tally and Accumulate ---

```

```

--- Tally ---
N    = 4
Sum  = 10.0
Mean = 2.5

```



```

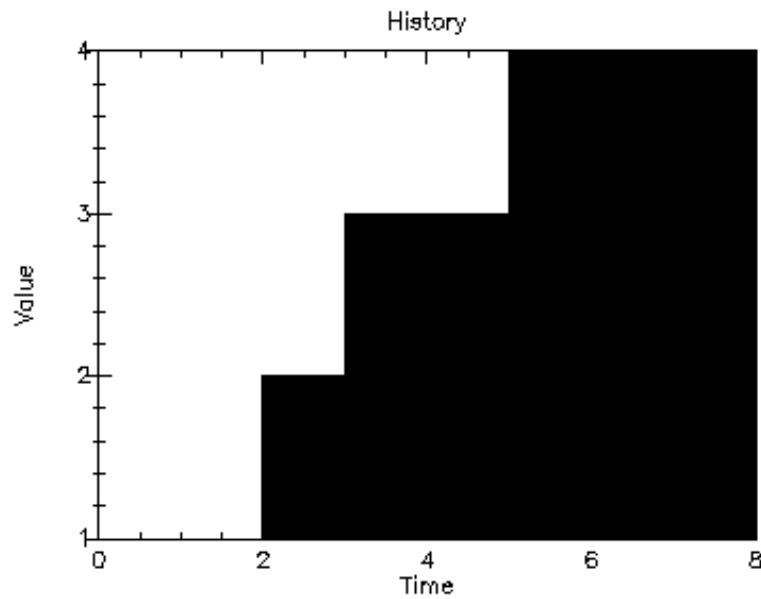
--- Accumulate ---
N    = 8.0

```

statistic	accumulate	tally
n	$time_C - time_0$	# of samples of $X$
sum	$\sum(X * (time_C - time_L))$	$\sum X$
mean	sum/n	sum/n
sum-of-squares	$\sum(X^2 * (time_C - time_L))$	$\sum X^2$
mean-square	sum-of-squares/n	sum-of-squares/n
variance	mean-square - mean <sup>2</sup>	mean-square - mean <sup>2</sup>
standard-deviation	$\sqrt{\text{variance}}$	$\sqrt{\text{variance}}$
maximum	maximum $X$ for all $X$	maximum $X$ for all $X$
minimum	minimum $X$ for all $X$	minimum $X$ for all $X$

Table 8.1: Statistics Computations

Sum = 22.0  
Mean = 2.75



>

## 8.3 Statistics

## 8.4 History

### 8.4.1 History Graphics

## 8.5 Example – Data Collection

```

; Example 3 - Data Collection

(require (planet "simulation-with-graphics.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

(define n-attendants 2)
(define attendant #f)

(define-process (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (schedule now (customer i))))

(define-process (customer i)
  (with-resource (attendant)
    (work (random-flat 2.0 10.0))))

(define (run-simulation n)
  (with-new-simulation-environment
    (set! attendant (make-resource n-attendants))
    (schedule (at 0.0) (generator n))
    (accumulate (variable-statistics
                 (resource-queue-variable-n attendant)))
    (accumulate (variable-history
                 (resource-queue-variable-n attendant)))
    (start-simulation)
    (printf "--- Example 3 - Data Collection ---~n")
    (printf "Maximum queue length = ~a~n"
            (variable-maximum
             (resource-queue-variable-n attendant)))
    (printf "Average queue length = ~a~n"
            (variable-mean
             (resource-queue-variable-n attendant)))
    (printf "Variance                = ~a~n"
            (variable-variance
             (resource-queue-variable-n attendant))))

```

```

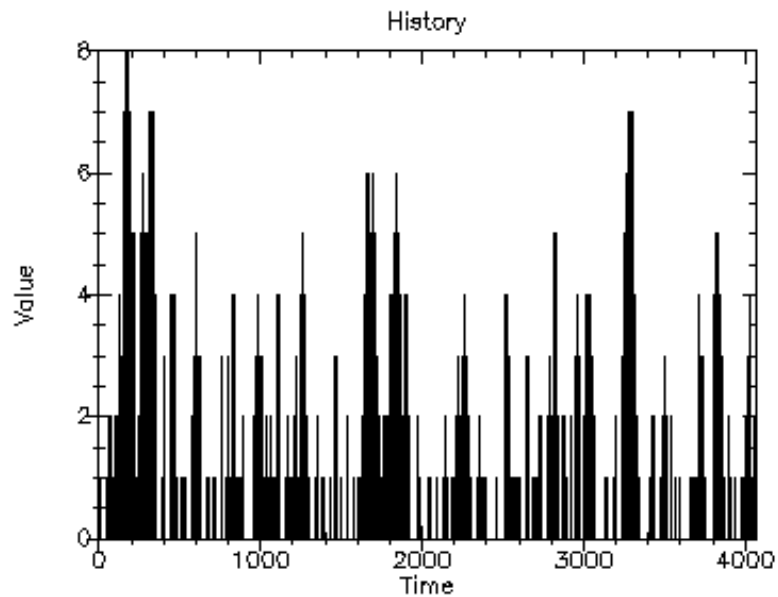
      (variable-variance
      (resource-queue-variable-n attendant)))
(printf "Utilization      = ~a~n"
      (variable-mean
      (resource-satisfied-variable-n attendant)))
(printf "Variance        = ~a~n"
      (variable-variance
      (resource-satisfied-variable-n attendant)))
(print (history-plot
      (variable-history
      (resource-queue-variable-n attendant))))))

```

```

>(run-simulation 1000)
--- Example 3 - Data Collection ---
Maximum queue length = 8
Average queue length = 0.9120534884951139
Variance              = 2.2420855874934826
Utilization           = 1.4320511974417858
Variance              = 0.5885107114317054

```



```

>

```

## 8.6 Data Collection Across Multiple Simulation Runs

### 8.6.1 Open Loop Processing

#### Example - Open Loop Processing

```

; Open Loop Example

(require (planet "simulation-with-graphics.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

(define attendant #f)

(define (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (schedule now (customer i))))

(define-process (customer i)
  (with-resource (attendant)
    (wait/work (random-flat 2.0 10.0))))

(define (run-simulation n1 n2)
  (with-new-simulation-environment
    (let ((max-attendants (make-variable)))
      (tally (variable-statistics max-attendants))
      (tally (variable-history max-attendants))
      (do ((i 1 (+ i 1)))
          ((> i n1) (void))
          (with-new-simulation-environment
            (set! attendant (make-resource +inf.0))
            (schedule (at 0.0) (generator n2))
            (start-simulation)
            (set-variable-value! max-attendants
              (variable-maximum
                (resource-satisfied-variable-n attendant))))))
      (printf "--- Open Loop Example ---~n")
      (printf "Number of experiments      = ~a~n"
              (variable-n max-attendants))
      (printf "Minimum maximum attendants = ~a~n"
              (variable-minimum max-attendants))
      (printf "Maximum maximum attendants = ~a~n"
              (variable-maximum max-attendants))))

```

```

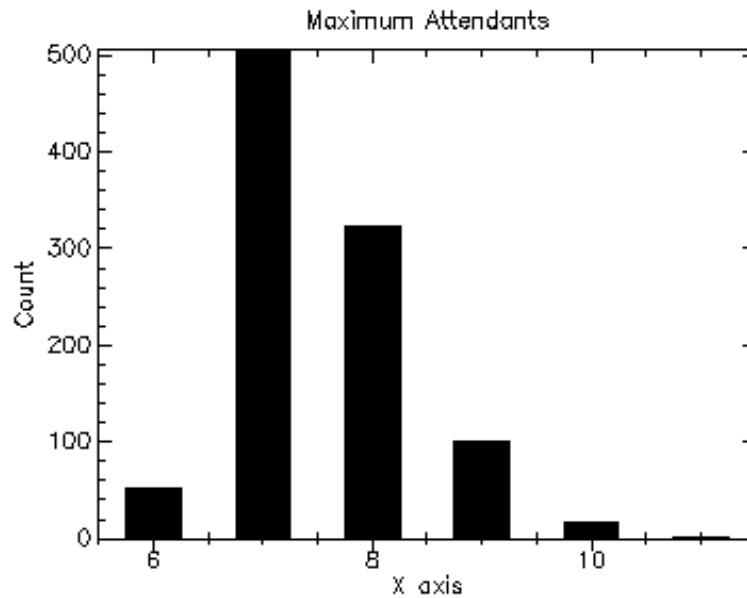
(variable-maximum max-attendants))
(printf "Mean maximum attendants   = ~a~n"
(variable-mean max-attendants))
(printf "Variance                   = ~a~n"
(variable-variance max-attendants))
(print (history-plot (variable-history max-attendants)
                    "Maximum Attendants"))
(newline)))

```

```

>(run-simulation 1000 1000)
--- Open Loop Example ---
Number of experiments      = 1000
Minimum maximum attendants = 6
Maximum maximum attendants = 11
Mean maximum attendants    = 7.525
Variance                   = 0.6653749999999903

```



>

## 8.6.2 Closed Loop Processing

### Example – Closed Loop Processing

```
; Closed Loop Example
```

```

(require (planet "simulation-with-graphics.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

(define n-attendants 2)
(define attendant #f)

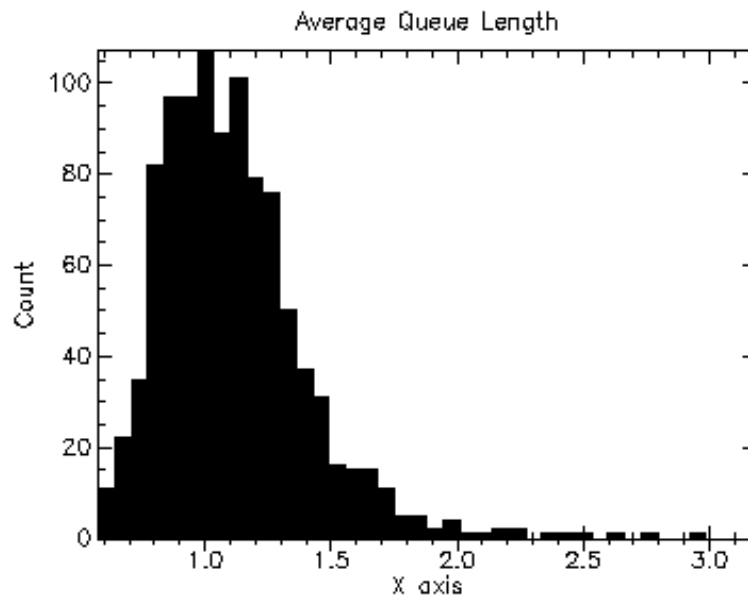
(define-process (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (schedule now (customer i))))

(define-process (customer i)
  (with-resource (attendant)
    (work (random-flat 2.0 10.0))))

(define (run-simulation n1 n2)
  (let ((avg-queue-length (make-variable)))
    (tally (variable-statistics avg-queue-length))
    (tally (variable-history avg-queue-length))
    (do ((i 1 (+ i 1)))
        ((> i n1) (void))
        (with-new-simulation-environment
          (set! attendant (make-resource n-attendants))
          (schedule (at 0.0) (generator n2))
          (start-simulation)
          (set-variable-value! avg-queue-length
                               (variable-mean (resource-queue-variable-n attendant))))))
    (printf "--- Closed Loop Example ---~n")
    (printf "Number of attendants      = ~a~n" n-attendants)
    (printf "Number of experiments      = ~a~n"
           (variable-n avg-queue-length))
    (printf "Minimum average queue length = ~a~n"
           (variable-minimum avg-queue-length))
    (printf "Maximum average queue length = ~a~n"
           (variable-maximum avg-queue-length))
    (printf "Mean average queue length    = ~a~n"
           (variable-mean avg-queue-length))
    (printf "Variance                      = ~a~n"
           (variable-variance avg-queue-length))
    (print (history-plot (variable-history avg-queue-length)
                        "Average Queue Length"))
    (newline)))

```

```
>(run-simulation 1000 1000)
--- Closed Loop Example ---
Number of attendants          = 2
Number of experiments         = 1000
Minimum average queue length = 0.5792057912006373
Maximum average queue length = 3.182757214703683
Mean average queue length    = 1.1123279920475524
Variance                      = 0.08869696318792064
```



```
>
```

# Chapter 9

## Sets

### 9.1 Example – Furnace Model 1

```
;;; Model 1 - Discrete Event Model
(require (planet "simulation-with-graphics.ss"
               ("williams" "simulation.plt" 1 0)))
(require (planet "random-source.ss" ("williams" "science.plt")))
(require (planet "random-distributions.ss"
               ("williams" "science.plt")))

;;; Simulation Parameters
(define end-time 720.0)
(define n-pits 7)

;;; Data collection variables
(define total-ingots 0)
(define wait-time (make-variable))

;;; Model Definition
(define random-sources (make-random-source-vector 2))

(define furnace #f)
(define pit #f)

(define (scheduler)
  (let loop ()
    (schedule now (ingot))
    (wait (random-exponential (vector-ref random-sources 0) 1.5))
    (loop)))

(define-process (ingot)
```

```

(let ((arrive-time (current-simulation-time)))
  (with-resource (pit)
    (set-variable-value!
      wait-time (- (current-simulation-time) arrive-time))
    (set-insert! furnace self)
    (work (random-flat (vector-ref random-sources 1) 4.0 8.0))
    (set-remove! furnace self))
    (set! total-ingots (+ total-ingots 1))))

(define (stop-sim)
  (printf
    "Report after ~a Simulated Hours - ~a Ingots Processed~n"
    (current-simulation-time) total-ingots)
  (printf "~n-- Ingot Waiting Time Statistics --~n")
  (printf "Mean Wait Time      = ~a~n"
    (variable-mean wait-time))
  (printf "Variance                = ~a~n"
    (variable-variance wait-time))
  (printf "Maximum Wait Time       = ~a~n"
    (variable-maximum wait-time))
  (printf "~n-- Furnace Utilization Statistics --~n")
  (printf "Mean No. of Ingots      = ~a~n"
    (variable-mean (set-variable-n furnace)))
  (printf "Variance                  = ~a~n"
    (variable-variance (set-variable-n furnace)))
  (printf "Maximum No. of Ingots = ~a~n"
    (variable-maximum (set-variable-n furnace)))
  (printf "Minimum No. of Ingots = ~a~n"
    (variable-minimum (set-variable-n furnace)))
  (printf "~a~n"
    (history-plot
      (variable-history (set-variable-n furnace))
      "Furnace Utilization History")))
  (stop-simulation))

(define (initialize)
  (set! total-ingots 0)
  (set! wait-time (make-variable))
  (set! pit (make-resource n-pits))
  (set! furnace (make-set))
  (accumulate (variable-history (set-variable-n furnace)))
  (tally (variable-statistics wait-time))
  (schedule (at end-time) (stop-sim))
  (schedule (at 0.0) (scheduler)))

(define (run-simulation)

```

```
(with-new-simulation-environment  
  (initialize)  
  (start-simulation)))
```

```
>(run-simulation)
```

```
Report after 720.0 Simulated Hours - 479 Ingots Processed
```

```
-- Ingot Waiting Time Statistics --
```

```
Mean Wait Time      = 0.1482393804317038
```

```
Variance            = 0.24601817483957691
```

```
Maximum Wait Time   = 3.593058032365832
```

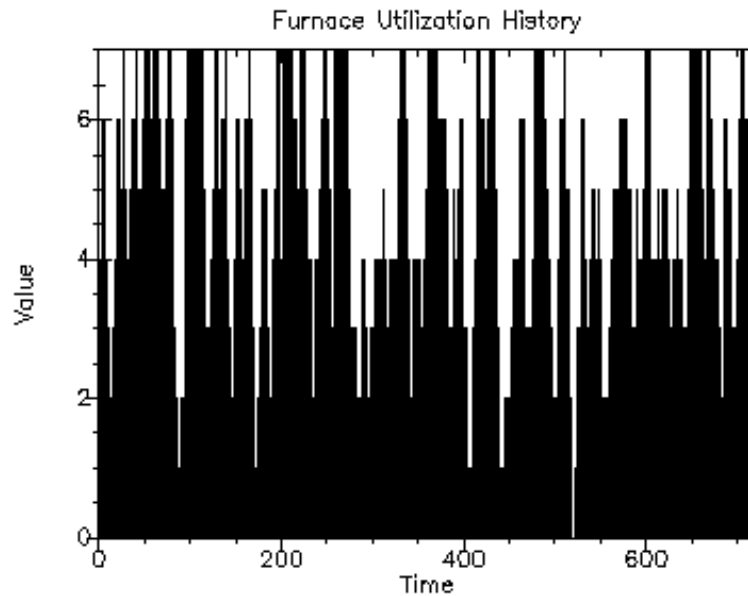
```
-- Furnace Utilization Statistics --
```

```
Mean No. of Ingots  = 4.0063959874393795
```

```
Variance            = 3.2693449366238347
```

```
Maximum No. of Ingots = 7
```

```
Minimum No. of Ingots = 0
```



```
>
```

## Chapter 10

# Continuous Simulation Models

### 10.1 Continuous Variables

### 10.2 Simulation Control (Continuous)

#### 10.2.1 Example – Furnace Model 2

```
; Model 2 - Continuous Simulation Model
(require (planet "simulation-with-graphics.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-source.ss" ("williams" "science.plt")))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

;; Simulation Parameters
(define end-time 720.0)
(define n-pits 7)

; Data collection variables
(define total-ingots 0)
(define wait-time #f)
(define heat-time #f)
(define leave-temp #f)

;;; Model Definition
(define random-sources (make-random-source-vector 4))

(define furnace-set #f)
(define furnace-temp 1500.0)
```

```

(define pit #f)

(define (scheduler)
  (let loop ((i 0))
    (schedule now (ingot i))
    (wait (random-exponential (vector-ref random-sources 0) 1.5))
    (loop (+ i 1))))

(define-process (ingot i)
  (let* ((initial-temp (random-flat (vector-ref random-sources 1)
                                   100.0 200.0))
        (heat-coeff (+ (random-gaussian
                       (vector-ref random-sources 2)
                       0.05 0.01)
                       0.07))
        (final-temp (random-flat (vector-ref random-sources 3)
                                 800.0 1000.0))
        (current-temp (make-continuous-variable initial-temp))
        (arrive-time (current-simulation-time))
        (start-time #f))
    (with-resource (pit)
      (if (= (modulo i 100) 0)
          (accumulate (variable-history current-temp)))
      (set-variable-value!
       wait-time (- (current-simulation-time) arrive-time))
      (set-insert! furnace-set self)
      (set! start-time (current-simulation-time))
      (work/continuously
       until (>= (variable-value current-temp) final-temp)
       (set-variable-dt!
        current-temp
        (* (- furnace-temp (variable-value current-temp))
           heat-coeff)))
      (set-variable-value!
       heat-time (- (current-simulation-time) start-time))
      (set-variable-value!
       leave-temp (variable-value current-temp))
      (set-remove! furnace-set self))
    (if (variable-history current-temp)
        (printf
         "~a~n"
         (history-plot (variable-history current-temp)
                       (format "Ingot ~a Temp History" i))))
    (set! total-ingots (+ total-ingots 1))))

(define (stop-sim)

```

```

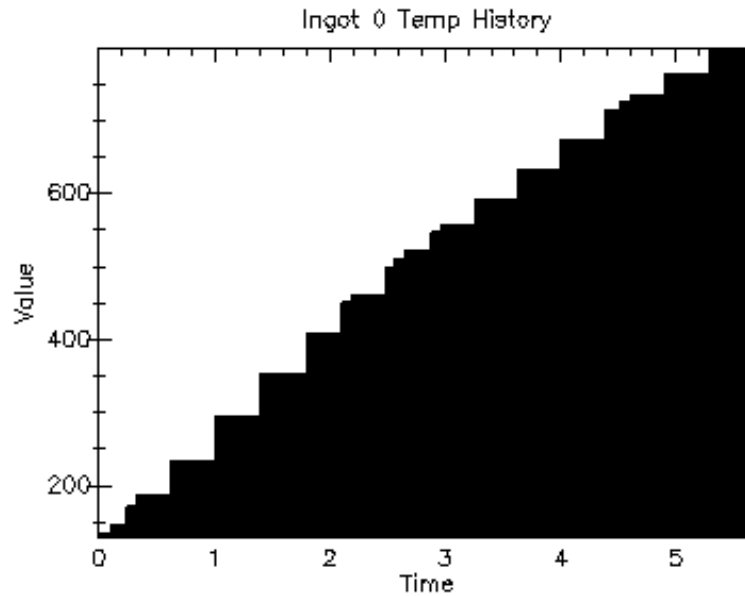
(printf
  "Report after ~a Simulated Hours - ~a Ingots Processed~n"
  (current-simulation-time) total-ingots)
(printf "~n-- Ingot Waiting Time Statistics --~n")
(printf "Mean Wait Time      = ~a~n"
  (variable-mean wait-time))
(printf "Variance            = ~a~n"
  (variable-variance wait-time))
(printf "Maximum Wait Time    = ~a~n"
  (variable-maximum wait-time))
(printf "~n-- Ingot Heating Time Statistics --~n")
(printf "Mean Heat Time      = ~a~n"
  (variable-mean heat-time))
(printf "Variance            = ~a~n"
  (variable-variance heat-time))
(printf "Maximum Heat Time    = ~a~n"
  (variable-maximum heat-time))
(printf "Minimum Heat Time    = ~a~n"
  (variable-minimum heat-time))
(printf "~n-- Final Temperature Statistics --~n")
(printf "Mean Leave Temp      = ~a~n"
  (variable-mean leave-temp))
(printf "Variance            = ~a~n"
  (variable-variance leave-temp))
(printf "Maximum Leave Temp    = ~a~n"
  (variable-maximum leave-temp))
(printf "Minimum Leave Temp    = ~a~n"
  (variable-minimum leave-temp))
(printf "~a~n"
  (history-plot (variable-history leave-temp)
    "Final Temperature Histogram"))
(printf "~n-- Furnace Utilization Statistics --~n")
(printf "Mean No. of Ingots    = ~a~n"
  (variable-mean (set-variable-n furnace-set)))
(printf "Variance            = ~a~n"
  (variable-variance (set-variable-n furnace-set)))
(printf "Maximum No. of Ingots = ~a~n"
  (variable-maximum (set-variable-n furnace-set)))
(printf "Minimum No. of Ingots = ~a~n"
  (variable-minimum (set-variable-n furnace-set)))
(printf
  "~a~n"
  (history-plot (variable-history (set-variable-n furnace-set))
    "Furnace Utilization History"))
(stop-simulation))

```

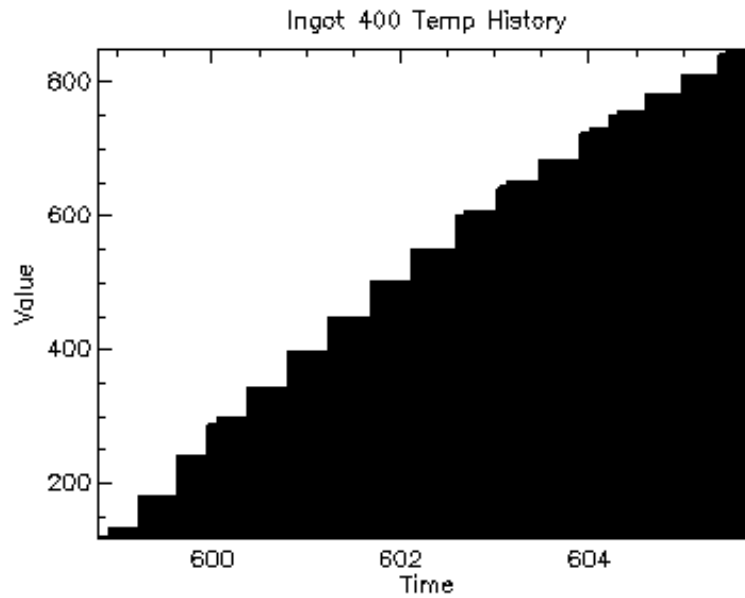
```
(define (initialize)
  (set! total-ingots 0)
  (set! wait-time (make-variable))
  (set! heat-time (make-variable))
  (set! leave-temp (make-variable))
  (set! pit (make-resource n-pits))
  (set! furnace-set (make-set))
  (accumulate (variable-history (set-variable-n furnace-set)))
  (tally (variable-statistics wait-time))
  (tally (variable-statistics heat-time))
  (tally (variable-statistics leave-temp))
  (tally (variable-history leave-temp))
  (schedule (at end-time) (stop-sim))
  (schedule (at 0.0) (scheduler)))

(define (run-simulation)
  (with-new-simulation-environment
    (initialize)
    (start-simulation)))
```

```
>(run-simulation)
```



...



Report after 720.0 Simulated Hours - 479 Ingots Processed

-- Ingot Waiting Time Statistics --

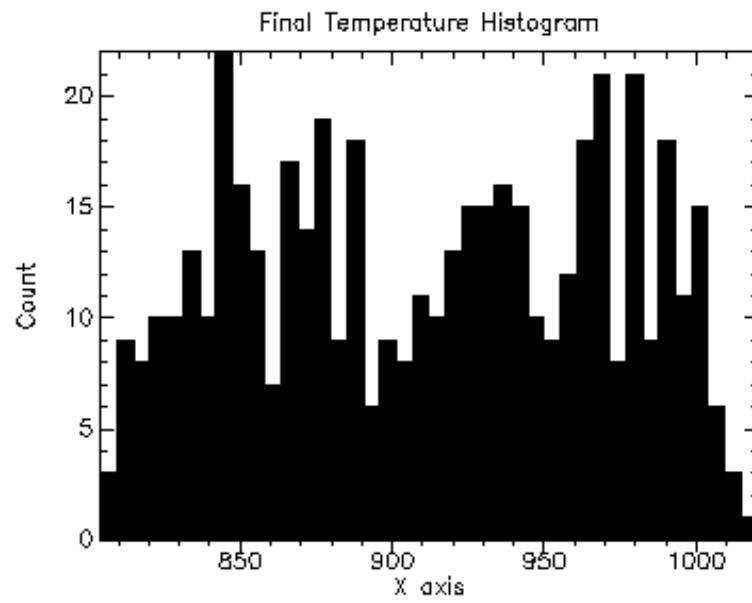
Mean Wait Time = 0.44596828009621853  
 Variance = 1.004363939227031  
 Maximum Wait Time = 6.380898029191428

-- Ingot Heating Time Statistics --

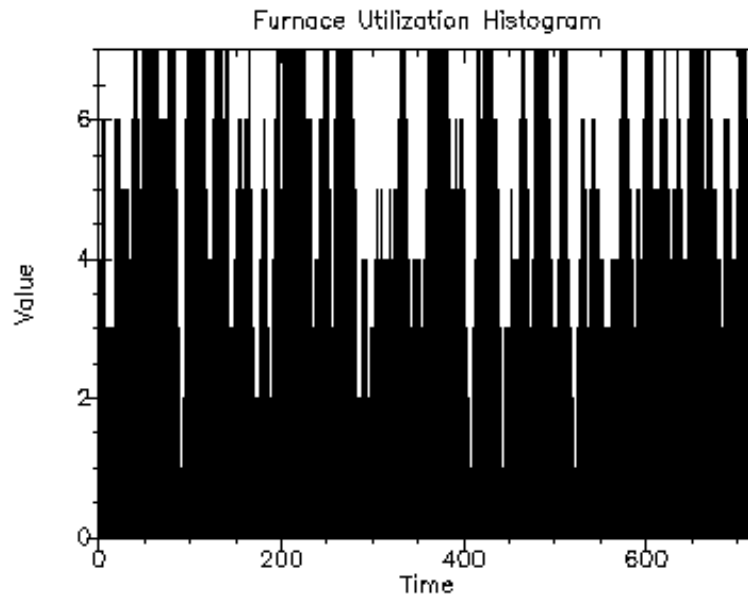
Mean Heat Time = 7.032338489600859  
 Variance = 1.0511669721776045  
 Maximum Heat Time = 10.88442744599513  
 Minimum Heat Time = 4.867835442878146

-- Final Temperature Statistics --

Mean Leave Temp = 912.2921783644567  
 Variance = 3300.2864186657825  
 Maximum Leave Temp = 1020.1172721368804  
 Minimum Leave Temp = 804.2299580285976



```
-- Furnace Utilization Statistics --  
Mean No. of Ingots = 4.687638930905194  
Variance = 3.321384522948101  
Maximum No. of Ingots = 7  
Minimum No. of Ingots = 0
```



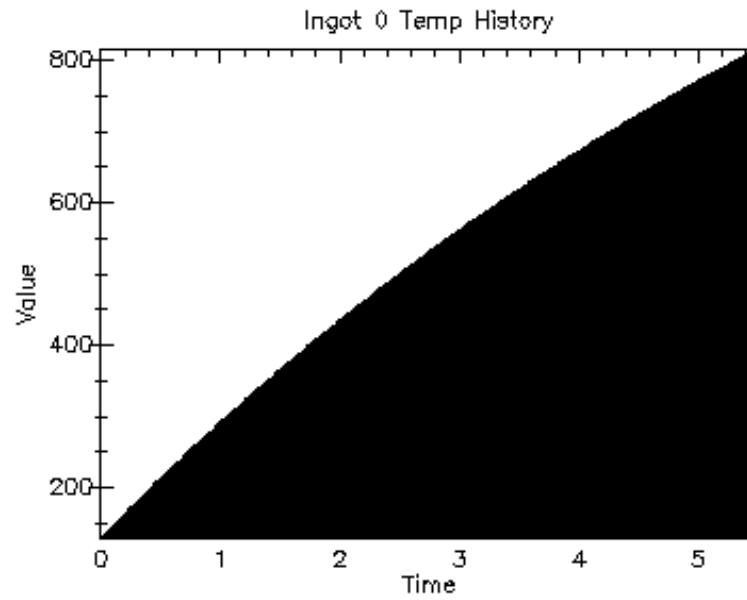
## 10.3 Simulation Environment (Continuous)

### 10.3.1 Fixed Step Size

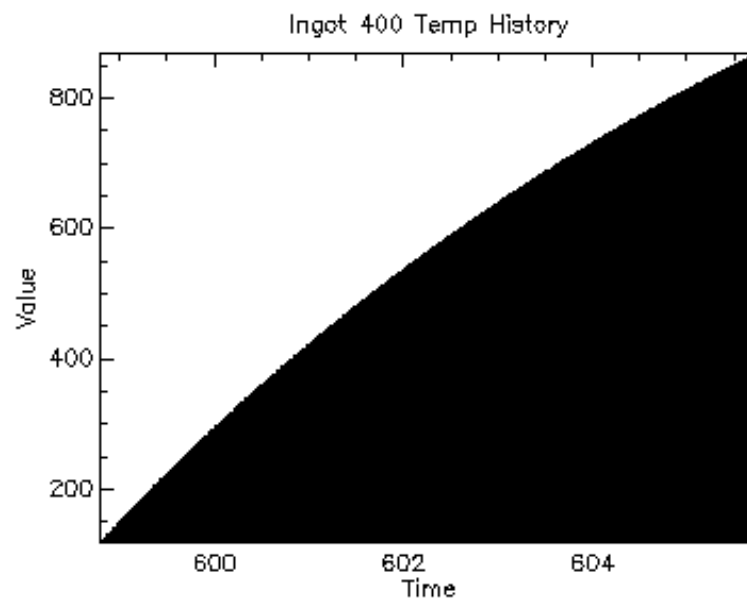
#### Example – Furnace Model 2a

```
(define (initialize)
  ;; Set continuous simulation settings
  (current-simulation-step-size (/ 1.0 60.0))
  (current-simulation-control #f)
  ;; ---
  (set! total-ingots 0)
  (set! wait-time (make-variable))
  (set! heat-time (make-variable))
  (set! leave-temp (make-variable))
  (set! pit (make-resource n-pits))
  (set! furnace-set (make-set))
  (accumulate (variable-history (set-variable-n furnace-set)))
  (tally (variable-statistics wait-time))
  (tally (variable-statistics heat-time))
  (tally (variable-statistics leave-temp))
  (tally (variable-history leave-temp))
  (schedule (at end-time) (stop-sim))
  (schedule (at 0.0) (scheduler)))
```

```
\end{verbatim}  
\begin{Verbatim}  
>(run-simulation)
```



...



Report after 720.0 Simulated Hours - 479 Ingots Processed

-- Ingot Waiting Time Statistics --

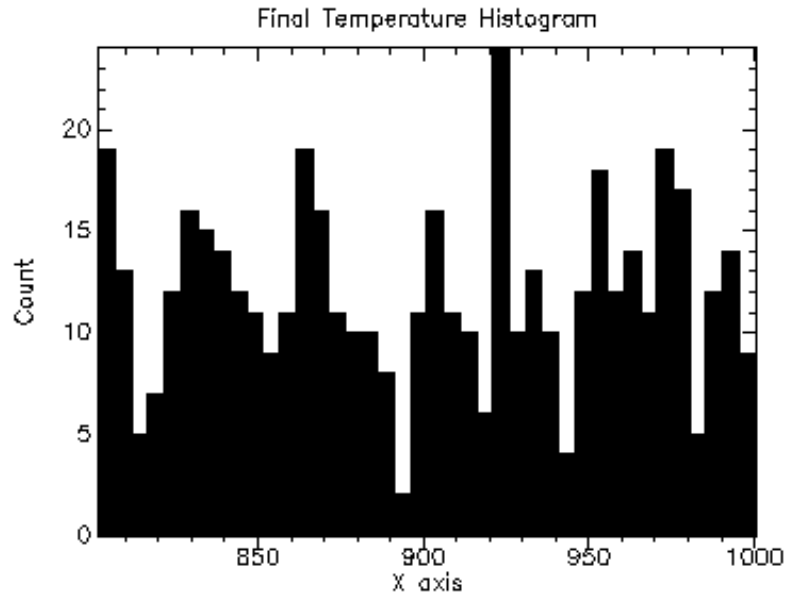
Mean Wait Time = 0.3972748535107707  
 Variance = 0.8502356452674397  
 Maximum Wait Time = 5.79999999999967

-- Ingot Heating Time Statistics --

Mean Heat Time = 6.877834613068176  
 Variance = 1.0551156660902024  
 Maximum Heat Time = 10.813257594378115  
 Minimum Heat Time = 4.7386143153090075

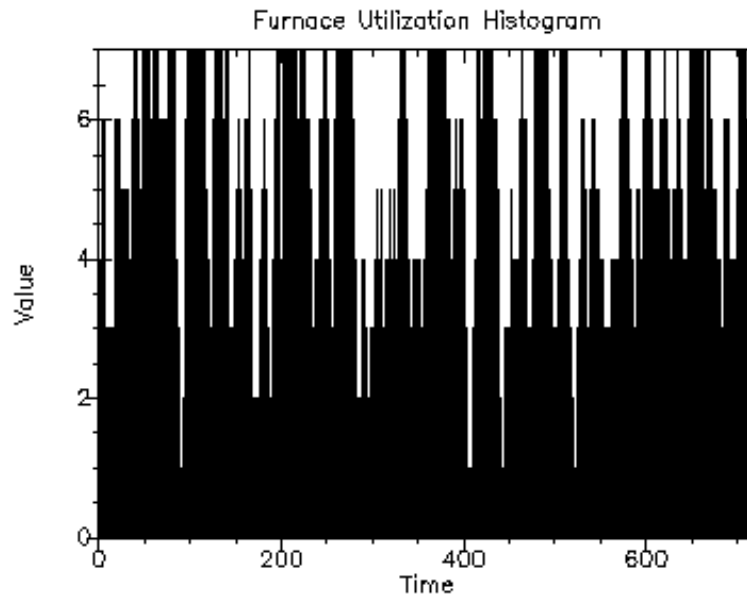
-- Final Temperature Statistics --

Mean Leave Temp = 901.3208708693385  
 Variance = 3389.992335826624  
 Maximum Leave Temp = 1000.3203098942319  
 Minimum Leave Temp = 801.994264590762



-- Furnace Utilization Statistics --

Mean No. of Ingots = 4.584850935267482  
 Variance = 3.382482163082148  
 Maximum No. of Ingots = 7  
 Minimum No. of Ingots = 0



### 10.3.2 Limiting Step Size

#### Example – Furnace Model 2b

```
(define (initialize)
  ;; Set continuous simulation settings
  (current-simulation-max-step-size (/ 1.0 60.0))
  ;; ---
  (set! total-ingots 0)
  (set! wait-time (make-variable))
  (set! heat-time (make-variable))
  (set! leave-temp (make-variable))
  (set! pit (make-resource n-pits))
  (set! furnace-set (make-set))
  (accumulate (variable-history (set-variable-n furnace-set)))
  (tally (variable-statistics wait-time))
  (tally (variable-statistics heat-time))
  (tally (variable-statistics leave-temp))
  (tally (variable-history leave-temp))
  (schedule (at end-time) (stop-sim))
  (schedule (at 0.0) (scheduler)))
\end{verbatim}
\begin{verbatim}
>(run-simulation)
```

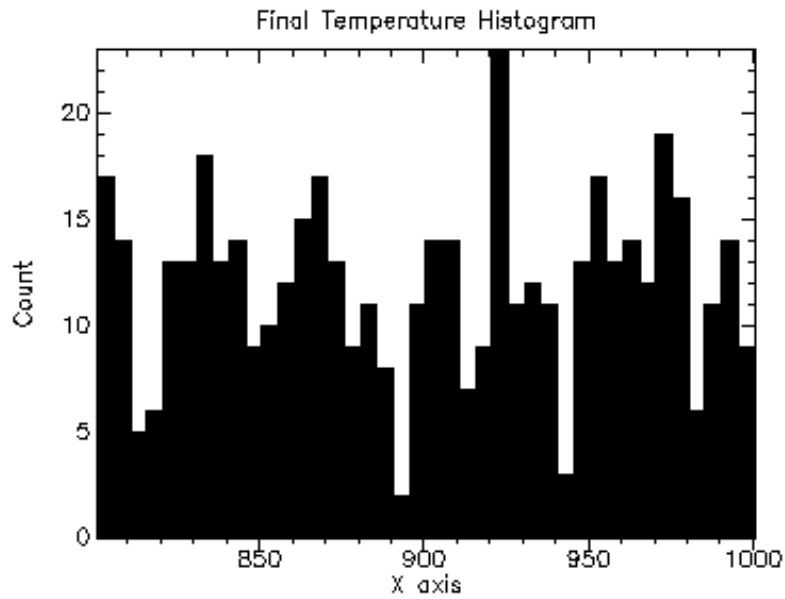
...

Report after 720.0 Simulated Hours - 479 Ingots Processed

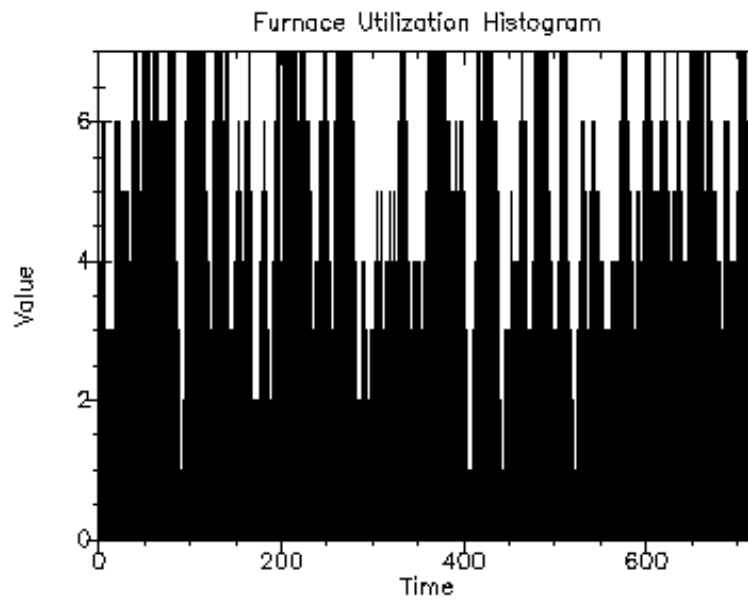
```
-- Ingot Waiting Time Statistics --
Mean Wait Time      = 0.39728613689438286
Variance            = 0.8495464450741739
Maximum Wait Time   = 5.8033836666662495
```

```
-- Ingot Heating Time Statistics --
Mean Heat Time      = 6.8777413568066175
Variance            = 1.0551344984370417
Maximum Heat Time   = 10.816121927711492
Minimum Heat Time   = 4.7386143153090075
```

```
-- Final Temperature Statistics --
Mean Leave Temp     = 901.3138280211458
Variance            = 3391.192401219392
Maximum Leave Temp  = 1000.4429841574237
Minimum Leave Temp  = 800.9274096651966
```



```
-- Furnace Utilization Statistics --
Mean No. of Ingots  = 4.584788893949024
Variance            = 3.382567115580496
Maximum No. of Ingots = 7
Minimum No. of Ingots = 0
```



#### 10.4 Furnace Model 3

# Chapter 11

## Simulation Classes

### 11.1 Process Classes

### 11.2 Resource Classes

### 11.3 Example – Simulation Classes

```
; Example 4 - Classes

(require (planet "simulation-with-graphics.ss"
               ("williams" "simulation.plt" 1 0)))
(require (planet "random-distributions.ss"
               ("williams" "science.plt")))

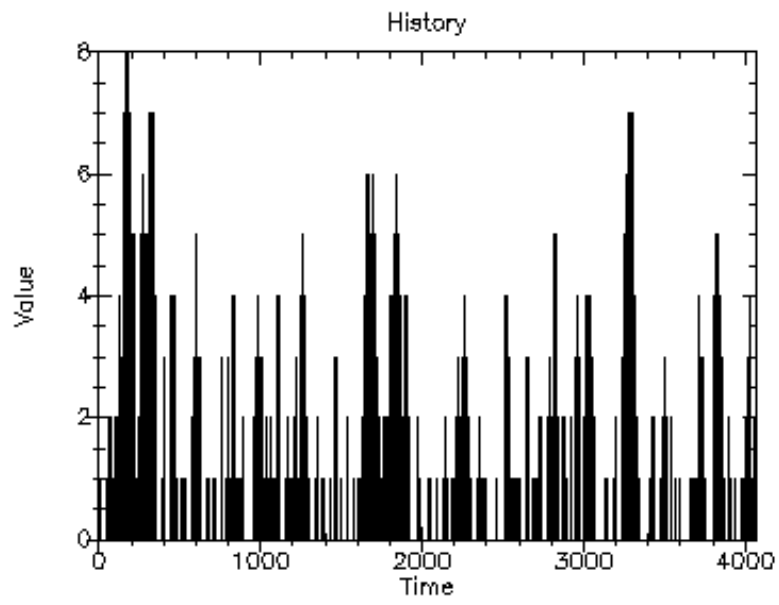
(define n-attendants 2)
(define attendant #f)

(define-process-class generator%
  (init-field (n 1000))
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait (random-exponential 4.0))
      (make-object customer% i)))

(define-process-class customer%
  (init-field i)
  (begin
    (send attendant request)
    (work (random-flat 2.0 10.0))
    (send attendant relinquish)))
```

```
(define (run-simulation n)
  (with-new-simulation-environment
    (set! attendant (make-object resource% n-attendants))
    (make-object generator% n)
    (accumulate (variable-statistics
                 (send attendant queue-variable-n)))
    (accumulate (variable-history
                 (send attendant queue-variable-n)))
    (start-simulation)
    (printf "--- Example 4 - Classes ---~n")
    (printf "Maximum queue length = ~a~n"
            (variable-maximum
              (send attendant queue-variable-n)))
    (printf "Average queue length = ~a~n"
            (variable-mean
              (send attendant queue-variable-n)))
    (printf "Variance           = ~a~n"
            (variable-variance
              (send attendant queue-variable-n)))
    (printf "Utilization         = ~a~n"
            (variable-mean
              (send attendant satisfied-variable-n)))
    (printf "Variance           = ~a~n"
            (variable-variance
              (send attendant satisfied-variable-n)))
    (print (history-plot
            (variable-history
              (send attendant queue-variable-n))))))
```

```
>(run-simulation 1000)
--- Example 4 - Classes ---
Maximum queue length = 8
Average queue length = 0.9120534884951139
Variance             = 2.2420855874934826
Utilization          = 1.4320511974417858
Variance             = 0.5885107114317054
```



## Chapter 12

# Simulation Control (Advanced)

### 12.1 Example – Harbor Model

```
;;; Harbor Model

(require (planet "simulation-with-graphics.ss"
                ("williams" "simulation.plt" 1 0)))
(require (planet "random-source.ss" ("williams" "science.plt")))
(require (planet "random-distributions.ss"
                ("williams" "science.plt")))

;;; Data collection variables
(define cycle-time #f)

;;; Model definition
(define random-sources (make-random-source-vector 2))

(define dock #f)
(define queue #f)

(define (scheduler)
  (let loop ()
    (make-object ship%)
    (wait (random-exponential (vector-ref random-sources 0)
                              (/ 4.0 3.0)))
    (loop)))

(define-process-class ship%
  (field (unloading-time (random-flat
```

```

                                (vector-ref random-sources 1)
                                1.0 2.5)))
(let ((arrival-time (current-simulation-time)))
  (when (not (harbor-master this 'arriving))
    (set-insert! queue this)
    (suspend-process)
    (work unloading-time)
    (set-remove! dock this)
    (set-variable-value!
     cycle-time (- (current-simulation-time) arrival-time))
    (harbor-master this 'leaving)))

(define ship-unloading-time
  (class-field-accessor ship% unloading-time))

(define set-ship-unloading-time!
  (class-field-mutator ship% unloading-time))

(define (harbor-master ship action)
  (case action
    ((arriving)
     (if (< (set-n dock) 2)
         ;; Dock is not full
         (begin
          (if (set-empty? dock)
              (set-ship-unloading-time!
               ship (/ (ship-unloading-time ship) 2.0))
              (let ((other-ship (set-first dock)))
                (send other-ship interrupt)
                (send other-ship set-time
                 (* (send other-ship get-time) 2.0))
                (send other-ship resume)))
            (set-insert! dock ship)
            #t)
          ;; Dock is full
          #f))
    ((leaving)
     (if (set-empty? queue)
         (if (not (set-empty? dock))
             (let ((other-ship (set-first dock)))
               (send other-ship interrupt)
               (send other-ship set-time
                (/ (send other-ship get-time) 2.0))
               (send other-ship resume)
               #t))
         (let ((next-ship (set-remove-first! queue)))

```

```

        (set-insert! dock next-ship)
        (send next-ship resume)
        #t)))
    (else
     (error 'harbor-master "illegal action value ~a" action))))

(define (stop-sim)
  (printf "Harbor Model - report after ~a simulated days - ~a ships processed~n"
         (current-simulation-time) (variable-n cycle-time))
  (printf "Minimum unload time was ~a~n"
         (variable-minimum cycle-time))
  (printf "Maximum unload time was ~a~n"
         (variable-maximum cycle-time))
  (printf "Average queue of ships waiting to be unloaded was ~a~n"
         (variable-mean (set-variable-n queue)))
  (printf "Maximum queue was ~a~n"
         (variable-maximum (set-variable-n queue)))
  (printf "~a~n"
         (history-plot (variable-history (set-variable-n queue))
                       "History of Waiting Queue"))
  (stop-simulation))

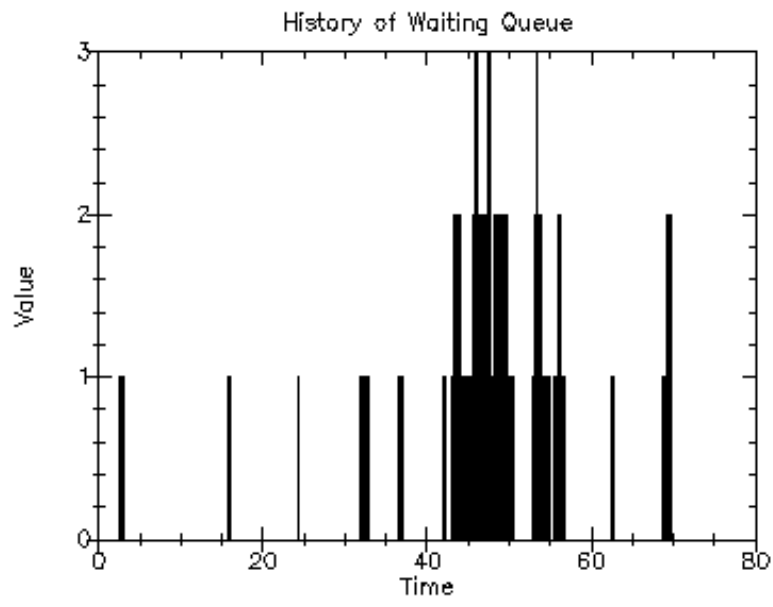
(define (run-simulation)
  (with-new-simulation-environment
   (set! cycle-time (make-variable))
   (tally (variable-statistics cycle-time))
   (set! dock (make-set))
   (set! queue (make-set))
   (accumulate (variable-history (set-variable-n queue)))
   (schedule now (scheduler))
   (schedule (at 80.0) (stop-sim))
   (start-simulation)))

```

```

>(run-simulation)
Harbor Model - report after 80.0 simulated days - 65 ships processed
Minimum unload time was 0.5656279138989291
Maximum unload time was 3.893379568241123
Average queue of ships waiting to be unloaded was 0.24532233055969996
Maximum queue was 3

```



## Chapter 13

# Simulation Environments (Hierarchical)

# Chapter 14

## Components

# Appendix A

## GNU Lesser General Public License (LGPL)

GNU LESSER GENERAL PUBLIC LICENSE  
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts  
as the successor of the GNU Library Public License, version 2, hence  
the version number 2.1.]

### Preamble

The licenses for most software are designed to take away your  
freedom to share and change it. By contrast, the GNU General Public  
Licenses are intended to guarantee your freedom to share and change  
free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some  
specially designated software packages--typically libraries--of the  
Free Software Foundation and other authors who decide to use it. You  
can use it too, but we suggest you first think carefully about whether  
this license or the ordinary General Public License is the better  
strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use,  
not price. Our General Public Licenses are designed to make sure that  
you have the freedom to distribute copies of free software (and charge  
for this service if you wish); that you receive source code or can get  
it if you want it; that you can change the software and use pieces of

it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it

does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE  
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a

portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a

medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the

copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent

license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status

of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>
```

```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.
```

```
This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.
```

```
You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library 'Frob' (a library for tweaking knobs) written by James Random Hacker.
```

```
<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

## Appendix B

# Simplified Simulation System

```
;; Simplified Simulation System

;; Event definition and scheduling

;; The event list is maintained in ascending order - the first item of
;; the list is the next event to be executed.
(define *event-list* '())

;; Each event has a time the event is to be executed, the function to
;; be executed, and the (evaluated) arguments to the function.
(define-struct event (time function arguments))

;; schedule: event -> void
;; Add an event to the event list.
(define (schedule event)
  (set! *event-list* (event-schedule event *event-list*)))

;; event-schedule: event x list of events -> list of events
;; Return a new list of events corresponding to the given event added
;; to the given list of events.
(define (event-schedule event event-list)
  (cond ((null? event-list)
        (list event))
        ((< (event-time event)
            (event-time (car event-list)))
         (cons event event-list))
        (else
         (cons (car event-list)
               (event-schedule event (cdr event-list))))))

;; Simulation control routines
```

```

;; Global simulation control variables
(define *time* 0.0)           ; current simulation time
(define *event* #f)          ; currently executing event
(define *loop-exit* #f)      ; main loop exit continuation
(define *loop-next* #f)     ; main loop next continuation

;; wait/work: real -> void
;; Simulate the delay while work is being done. Add an event to
;; execute the current continuation to the event list.
(define (wait/work delay)
  (let/cc continue
    ;; Reuse the current event - it would become garbage anyway
    (set-event-time! *event* (+ *time* delay))
    (set-event-function! *event* continue)
    (set-event-arguments! *event* '())
    (schedule *event*)
    ;; Done with this event
    (set! *event* #f)
    ;; Return to the main loop
    (*loop-next*)))

;; start-simulation: -> void
;; This is the main simulation loop. As long as there are events to
;; be executed (or the simulation is explicitly stopped): remove the
;; next event from the event list, advance the clock to the time of
;; the event, and apply the event's functions to it's arguments.
(define (start-simulation)
  (let/ec exit
    ;; Save the main loop exit continuation
    (set! *loop-exit* exit)
    ;; Main loop
    (let loop ()
      ;; Exit if no more events
      (if (null? *event-list*)
          (exit))
      (let/cc next
        ;; Save the main loop next continuation
        (set! *loop-next* next)
        ;; Execute the next event
        (set! *event* (car *event-list*))
        (set! *event-list* (cdr *event-list*))
        (set! *time* (event-time *event*))
        (apply (event-function *event*)
                (event-arguments *event*)))
      (loop))))

;; stop-simulation: -> void
;; Stop the execution of the current simulation (by jumping to its
;; exit continuation).

```

```

(define (stop-simulation)
  (*loop-exit*))

;; Random Distributions (to remove external dependencies)

;; random-float -> real
;; Returns a random real in (0.0, 1.0].
(define (random-float)
  (/ (exact->inexact (random 2147483647))
     2147483648.0))

;; random-flat: real x real -> real
;; Returns a random real number from a uniform distribution between a
;; and b.
(define (random-flat a b)
  (+ a (* (random-float) (- b a))))

;; random-exponential: real -> real
;; Returns a random real number from an exponential distribution with
;; mean mu.
(define (random-exponential mu)
  (* (- mu) (log (random-float))))

;; Example Simulation Model

;; generator: integer -> void
;; Process to generate n customers arriving into the system.
(define (generator n)
  (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (wait/work (random-exponential 4.0))
      (schedule (make-event *time* customer (list i)))))

;; customer: integer -> void
;; The ith customer into the system. The customer is in the system
;; 2 to 10 minutes and then leaves.
(define (customer i)
  (printf "~a: customer ~a enters~n" *time* i)
  (wait/work (random-flat 2.0 10.0))
  (printf "~a: customer ~a leaves~n" *time* i))

;; run-simulation: integer
;; Run the simulation for n customers (or until explicitly stopped at
;; some specified time).
(define (run-simulation n)
  ;; Reset the time and the event list
  (set! *time* 0.0)
  (set! *event-list* '())
  ;; Schedule the customer generator
  (schedule (make-event 0.0 generator (list n)))

```

```
;; Stop the simulation at the specified time (optional)
;;(schedule (make-event 50.0 stop-simulation '()))
;; Start the simulation main loop
(start-simulation)

;; Run the simulation for 10 customers.
(run-simulation 10)
```

# Bibliography

- [1] Williams, M. Douglas, *PLT Scheme Science Collection Reference Manual*, Edition 2.0 for Version 2.0, December 2005

# Index

current-simulation-environment, 6  
current-simulation-event, 7  
current-simulation-future-event-list,  
7  
current-simulation-loop-exit, 7  
current-simulation-loop-next, 7  
current-simulation-now-event-list,  
6  
current-simulation-process, 7  
current-simulation-running?, 6  
current-simulation-time, 6  
  
default-simulation-environment, 6  
define-process, 15  
  
event, 11  
event-list, 12  
event-list-add!, 12  
event-list-empty?, 12  
event-list-pop!, 12  
event-list-remove!, 12  
  
make-event-list, 12  
make-process, 16  
make-simulation-environment, 5  
  
process, 15  
process-name, 16  
process-time, 16  
  
schedule, 9  
set-process-time!, 16  
simulation-environment, 5  
start-simulation, 10  
stop-simulation, 10  
  
wait, 10  
wait/work, 10  
with new-simulation-environment, 8  
with-simulation-environment, 7  
work, 10