

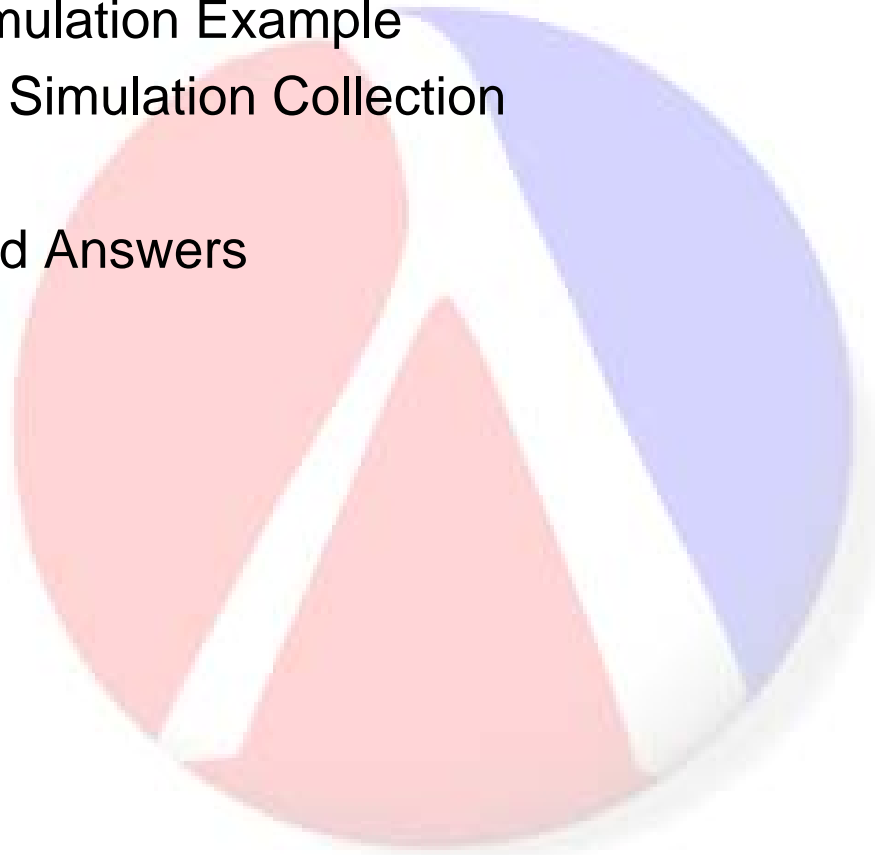


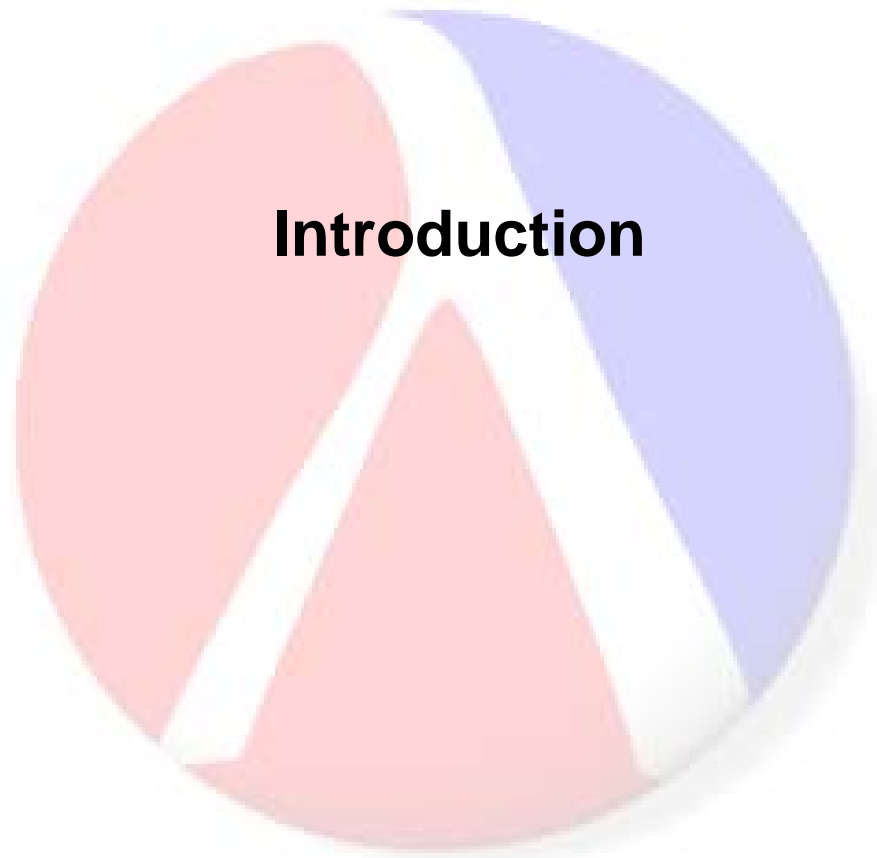
PLT Scheme **The Simulation Collection**

Dr. Doug Williams
m.douglas.williams@saic.com
February 16, 2006



- Introduction
- Simplified Simulation Example
- PLT Scheme Simulation Collection
- Future Plans
- Questions and Answers
- Workshop







- Recreate a previously available knowledge-based simulation environment capability
 - Previously implemented in Symbolics Common LISP
 - Re-implement in PLT Scheme
 - Availability – free download (www.drscheme.org)
 - Portability – Windows, Linux, UNIX, Mac OS X
- Extend previous work
 - Provide a better mathematical framework
 - Implement a process-based simulation engine
 - Support combined discrete-event and continuous simulations
 - Implement an efficient rule-based inference engine
- Provide a framework for implementing advanced knowledge-based simulations



PLT Scheme
Science Collection

- **PLT Scheme Science Collection**
 - Provides the mathematical and analysis framework
 - Previously part of the simulation collection, but provides functionality that is useful outside of simulations
 - Inspired by the GNU Scientific Library (GSL)
- **PLT Scheme Simulation Collection**
 - Provides a process-based, discrete-event simulation engine with automatic data collection
 - Supports combined discrete and continuous simulations
 - Designed to facilitate component-based simulation models
- **PLT Scheme Inference Collection**
 - Provides an efficient rule-based inference engine
 - Support both forward chaining (data-driven) and backward chaining (goal-driven) inferencing
 - Integrated with the simulation collection, i.e. inferencing can be done on simulation objects



PLT Scheme
Simulation Collection



PLT Scheme
Inference Collection



- Machine Constants
- Mathematical Constants and Functions
- Special Functions
- Random Number Generation
- Random Distributions
- Statistics
- Histograms
- Ordinary Differential Equations (Version 2.0)
- Chebyshev Approximations



- Simulation Environments (Basic)
- Simulation Control (Basic)
- Events
- Processes
- Resources
- Data Collection
 - Variables
 - Tally and Accumulate
- Sets
- Continuous Simulation Models
- Simulation Classes
- Simulation Control (Advanced)
- Simulation Environments (Hierarchical)
- Components



- Inference Environments (Basic)
- Inference Control (Basic)
 - Forward Chaining
 - Backward Chaining
- Assertions
 - Confidence
- Rule Sets
- Rules
- Inference Classes
- Inference Environments (Hierarchical)
 - States
 - State Space Search
- Inference Control (Advanced)
 - Truth Maintenance
 - Fuzzy Logic



- Development of all three collections is being moved to the Schematics project at SourceForge.
- Latest versions require PLT Scheme V301 or later
- PLT Scheme Science Collection
 - Version 2.0 Available via PLaneT
 - Reference Manual
- PLT Scheme Simulation Collection
 - Version 1.0 Available via PLaneT
 - Draft Reference Manual
- PLT Scheme Inference Collection
 - In progress



- PLT Scheme Science Collection
 - Release 1.0 October 2004 ✓
 - Release 2.0 December 2005 ✓
 - Ordinary Differential Equations (ODEs)
 - Release 2.1 February 2006
 - Additional CDFs
 - Beta, Incomplete Gamma, and Exponential Integral Special Functions
 - Additional ODE steppers
 - Bug fixes
- PLT Scheme Simulation Collection
 - Release 1.0 December 2005 ✓
 - Release 1.1 February 2006
 - Fixes to make models compatible with modules
 - Add linked events and renegeing
- PLT Scheme Inference Collection
 - Release 1.0 July 2006 (tentative)



Simplified Simulation Example



Simplified Simulation Example

- This Simplified Simulation Example condenses the basic discrete-event elements of the simulation collection and an example simulation model into a short, complete implementation with no dependencies.
- It will be used to examine the implementation of a continuation-based, discrete-event simulation engine.
- Basic elements
 - Event Definition and Scheduling
 - Simulation Control
 - Random Distributions (to remove external dependencies)
 - Example Simulation Model



- A continuation captures the current execution state of a computation. It defines how the computation will proceed with the value of the current expression.
- Continuations are first-class objects in Scheme.
- Consider the expression `(+ 3 (* 2 4))`
 - At the point where the subexpression `(* 2 4)` is being evaluated, the current continuation is `(+ 3 #)` [called from the top-level read-eval-print-loop (REPL)], where `#` is the value of the subexpression.
- `call-with-current-continuation` (or `call/cc`) allows the capture of the current continuation.
- The `let/cc` macro provides a more convenient syntax for the most common usage of `call/cc`.



- `(+ 3 (call/cc (lambda (cc) (* 2 4))))`
 - Evaluates `(* 2 4)` with the current continuation bound to the lambda variable `cc`. But, it doesn't do anything with the continuation.
- `(+ 3 (let/cc cc (* 2 4)))`
 - This expands into the previous expression. `let/cc` provides a convenient form for the most common usage of `call/cc`.
- `(+ 3 (let/cc cc (* (cc 2) 4)))`
 - Here, we actually call the continuation `cc` with a value of 2.
 - Remember that the continuation is `(+ 3 #)` [called from the top-level read-eval-print-loop (REPL)].
 - `(cc 2)` passes its argument, 2 in this case, to the continuation, which continues its execution with that value. In this case, returning 5 to the REPL, which prints it and loops back to get another expression.



- Remember that continuations are first-class objects.
- ```
(define cc-save #f)
(+ 3 (let/cc cc (set! cc-save cc)
 (* 2 4)))
```

  - Saves the continuation `(+ 3 #)` in a global variable `cc-save`.
- ```
(cc-save 2) → 5
```

 - Calls the saved continuation `(+ 3 #)`, which returns 5 to the REPL.
- ```
(+ 4 (cc-save 2)) → 5
```

  - Calls the saved continuation `(+ 3 #)`, which returns 5 to the REPL. The continuation `(+ 4 #)`, which was waiting to get the value of the subexpression `(cc-save 2)`, is abandoned.
- ```
(printf "value = ~a~n" (cc-save -3)) → 0
```

 - More of the same, returning 0 to the REPL. Again, the waiting continuation is abandoned.



- ```
(define cc-save #f)
(printf "value = ~a~n"
 (+ 3 (let/cc cc
 (set! cc-save cc) (* (cc 2) 4))))
```

  - Prints "value = 5" and returns to the REPL.
- ```
(cc-save 4)
```

 - Prints "value = 7" and returns to the REPL.
- ```
(let loop ((i 0))
 (if (= i 10)
 (cc-save i)
 (loop (+ i 1))))
```

  - On the 11<sup>th</sup> iteration of the loop, calls the continuation `cc-save`, which prints "value = 13" and returns to the REPL.



# Event Definition and Scheduling

- Defines the event structure, event list, and scheduling functions.
- The event structure has three fields:
  - time                               - time the event is to occur
  - function                           - function to apply
  - arguments                       - arguments to the function
- The event list is implemented as the global variable `*event-list*`. It is ordered by ascending time values.
- The `schedule` function adds an event to the event list.
- Event scheduling uses a simple recursive function to add the event at the appropriate place in the event list.



# Event Definition and Scheduling Code

```
(define *event-list* `())

(define-struct event (time function arguments))

(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))))))

(define (schedule event)
 (set! *event-list* (event-schedule event *event-list*)))
```



- Simulation control implements the main simulation loop and associated simulation control routines.
- The main simulation loop is implemented by the `start-simulation` function.
- The `stop-simulation` function allows user simulation code to exit the main simulation loop.
- The `wait/work` function allows user simulation code to simulate the passage of time.
- These routines make heavy use of continuations in their implementations
  - Thus the term continuation-based simulation



```
(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

(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*)))

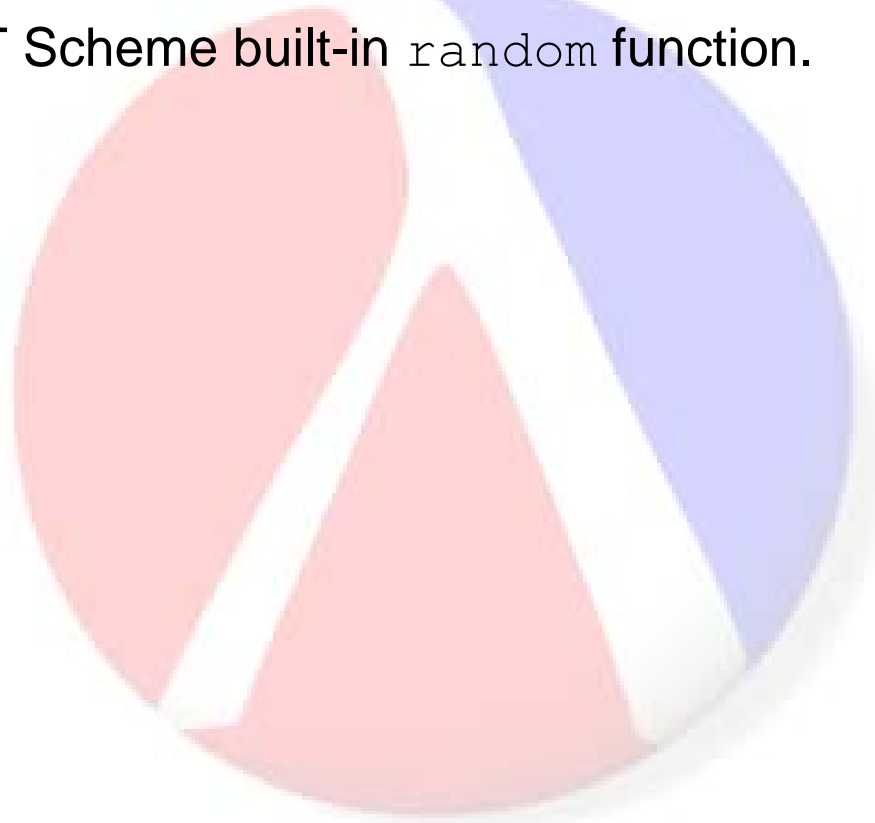
(define (stop-simulation)
 (*loop-exit*))
```



```
(define (start-simulation)
 (let/ec exit
 ;; Save the main loop exit continuation
 (set! *loop-exit* exit)
 ;; Main loop
 (let loop ()
 (let/cc next
 ;; Save the main loop next continuation
 (set! *loop-next* next)
 ;; Exit if no more events
 (if (null? *event-list*)
 (exit))
 ;; 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))))
```



- Simple implementations of `random-flat` and `random-exponential` to remove external dependencies.
- Uses the PLT Scheme built-in `random` function.





## Example Simulation Model

- This is the same simple simulation model that will be used (and extended) in the PLT Scheme Simulation Collection examples.
- `(generator n)` – generates  $n$  customers arriving into the system with arrival times that are exponentially distributed with a mean of 4.0.
- `(customer i)` – the  $i^{\text{th}}$  customer. The time each customer is in the system is uniformly distributed between 2.0 and 10.0.
- `(run-simulation n)` – resets and runs the simulation of  $n$  customers, unless terminated by a call to `stop-simulation`.



# Example Simulation Model Code

```
(define (generator n)
 (do ((i 0 (+ i 1)))
 ((= i n) (void))
 (wait/work (random-exponential 4.0))
 (schedule (make-event *time* customer (list i)))))

(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))

(define (run-simulation n)
 (set! *time* 0.0)
 (set! *event-list* '())
 (schedule (make-event 0.0 generator (list n)))
 ;(schedule (make-event 50.0 stop-simulation '()))
 (start-simulation))
```



# Example Simulation Model Output

```
>(run-simulation 10)
1.9492232981483808: customer 0 enters
6.174559533164965: customer 0 leaves
8.203725663933895: customer 1 enters
13.17036109975352: customer 1 leaves
14.146934654400557: customer 2 enters
16.435199558120686: customer 3 enters
18.93973929046175: customer 2 leaves
20.149969364833552: customer 3 leaves
20.44348015441581: customer 4 enters
26.21577096163317: customer 4 leaves
31.877101770738783: customer 5 enters
35.18128225872916: customer 5 leaves
36.11418547608223: customer 6 enters
38.63170173146095: customer 6 leaves
38.89736291069112: customer 7 enters
41.059028743352386: customer 8 enters
43.42485411162204: customer 9 enters
43.55302784829924: customer 8 leaves
46.92743758905254: customer 7 leaves
50.70005940715498: customer 9 leaves
```



# PLT Scheme Simulation Collection



- Simulation Environments (Basic)
- Simulation Control (Basic)
- Events
- Processes
- Resources
- Data Collection
  - Variables
  - Tally and Accumulate
- Sets
- Continuous Simulation Models
- Simulation Classes
- Simulation Control (Advanced)
- Simulation Environments (Hierarchical)
- Components



## Simulation Environment (Basic)

---

- A simulation environment encapsulates the state of a simulation.
  - Time
  - Event lists (now and future)
  - Loop and exit continuations
  - Process and event being executed
- Multiple simulation environments may exist at the same time.
  - Nested simulation environments are useful for data collection across multiple simulation runs (refer to the Open Loop and Closed Loop examples).
  - Nested simulation environments might be used to allow a low-fidelity model to reach steady state before kicking off a high-fidelity model.
  - Multiple, independent (or cooperating) models may exist as part of a larger system.
  - Note that these usages are different than hierarchical simulation environments, which are discussed later.



- Fields in a (basic) simulation environment:
  - `running?` #*t* if the main loop is running
  - `time` simulation time
  - `now-event-list` events to be executed now
  - `future-event-list` events to be executed in the future
  - `loop-next` continuation to return to the main loop
  - `loop-exit` continuation to exit the main loop
  - `event` executing event or #*f*
  - `process` executing process or #*f*



- The parameter `current-simulation-environment` represents the current simulation environment.
  - Defaults to `default-simulation-environment`
- Routines are provided to get and set fields in the `current-simulation-environment`.
  - `(current-simulation-running? [boolean])`
  - `(current-simulation-time [real])`
  - `(current-simulation-now-event-list [event-list])`
  - `(current-simulation-future-event-list [event-list])`
  - `(current-simulation-loop-next [continuation])`
  - `(current-simulation-loop-exit [continuation])`
  - `(current-simulation-event [event])`
  - `(current-simulation-process [process])`



- The `with-simulation-environment` macro evaluates its body with `current-simulation-environment` set to the specified simulation environment.
  - `(with-simulation-environment simulation-environment body ...)`
- The `with-new-simulation-environment` macro evaluates its body with `current-simulation-environment` set to a new simulation environment.
  - `(with-new-simulation-environment body ...)`



- The `schedule` macro schedules an event or process for execution.
  - `(schedule time (function . arguments))`
  - `(schedule now (function . arguments))`
  - `(schedule (at time) (function . arguments))`
  - `(schedule (in duration) (function . arguments))`
  - If *function* is the name of a process, a process instance is created and scheduled for execution. Otherwise, *function* must be a procedural object and an event is scheduled for execution.
- The `start-simulation` function implements the main simulation loop. It executes events until there are no more or the loop is explicitly exited via a call to `stop-simulation`.
- The `stop-simulation` function exits the current main simulation loop.
- The `wait/work` function simulates the passage of simulated time.
  - `(wait/work duration)`
  - `wait` and `work` are other names for the same function



- In a simulation model, an event represents an action that will take place in the future.
- In the simulation collection, an event represents the future application of a procedural object to a list of objects.
- Fields in the `event` structure:
  - `time` Time the event is to occur
  - `process` Process owning the event, or `#f`
  - `function` Function to be applied
  - `arguments` Arguments to the function
- Because events can represent the application of any functional objects, including continuations, events can also call the `wait/work` function. (In this case, the event object is reused.)
  - This is a slight extension to the definition in the first bullet. An event may represent a sequence of actions.



---

; 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)))
```



```
(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
```



- In a simulation model, a process represents an entity that actively progresses through time.
- In the simulation collection, a process encapsulates an event object that executes the body of the process; provides state information; and, most importantly, provides a handle allowing the process to interact with other simulation objects (e.g. resources or other processes).
- A process is defined using the `define-process` macro.
  - `(define-process (name . arguments)  
body ...)`
- Process instances are created via the `schedule` macro by specifying the name of the process as the `function` argument.



- Fields for the `process` structure include:
  - `event`
  - `state`
- The states of a process are:
  - `PROCESS-TERMINATED`
  - `PROCESS-CREATED`
  - `PROCESS-ACTIVE`
  - `PROCESS-WORKING/WAITING`
  - `PROCESS-WORKING-CONTINUOUSLY`
  - `PROCESS-DELAYED`
  - `PROCESS-INTERRUPTED`
  - `PROCESS-SUSPENDED`



```
; 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)))
```



```
(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
```



- In a simulation model, a resource is an entity (or entities) that is/are shared among processes.
- A resource is created using the `make-resource` function.
  - `(make-resource [units])`
- The fields of a resource are:
  - `units` Total number of units
  - `units-available` Number of units not allocated
  - `units-allocated` Number of units allocated
  - `satisfied` Set of processes satisfied
  - `queue` Set of processes waited
- The following functions request or relinquish resources:
  - `(resource-request resource [units])`
  - `(resource-relinquish resource [units])`



- There are two short-cut functions to variables within the queue and satisfied sets. They are to simplify data collection.
  - `(resource-queue-variable-n resource)`
  - `(resource-satisfied-variable-n resource)`
- Since the construct  
`(resource-request resource units)`  
`... ; use the resource`  
`(resource-relinquish resource units)`  
is used so much – virtually all resource usage has this form – the `with-resource` macro is provided.
  - `(with-resource (resource [units])  
body ...)`



```
; 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 (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)))
```



```
(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
```



- In general, the purpose for developing a simulation model is to collect data to analyze to gain insights into the system.
- In the simulation collection, data subject to automatic collection is stored in variable structures (i.e. variables).
- Data collection for a variable is initiated using either the `accumulate` or `tally` macro.
  - The `accumulate` macro initiates collection of time-dependant data.
  - The `tally` macro initiates collection of data that is not time-dependant.
- There are two categories of data collectors currently implemented in the simulation collection:
  - Statistics
  - History
- By default, statistics are accumulated for each variable.



- The data to be (automatically) collected in a simulation model is stored in variables (i.e. variable structures).
- A variable instance (i.e. a variable) is created using the `make-variable` function.
  - `(make-variable [initial-value])`
- The fields of interest in the variable structure are:
  - `value`                   The value of the variable
  - `statistics`               The statistics object for the variable, or `#f`
  - `history`                   The history object for the variable, or `#f`
- Data collectors (i.e. statistics or history objects) are automatically invoked, for example when a variable's value changes.
- Some simulation objects we've already used have fields implemented as variables.
  - `resource-queue-variable-n`
  - `resource-satisfied-variable-n`



- The `accumulate` macro initiates the collection of time-dependant data for a variable (i.e. an instance of the variable structure).
  - `(accumulate (variable-statistics variable))`
  - `(accumulate (variable-history variable))`
- For time-dependant data, the value for each data point is weighted by the duration it had that value.
  - For example, if you accumulated a variable that had values of 1 for 2 units of time, 2 for 1 units of time, 3 for 2 units of time, and 4 for 3 units of time, its average would be  $22/8 = 2.75$ .
- The accumulators are synchronized (updated) whenever the variable value changes or an accumulator for the variable is referenced.
  - Note that synchronization is performed on the variable and all accumulators for the variable are synchronized.
  - Note that a zero duration for a value does not result in synchronization.



- The `tally` macro initiates the collection of data that is not time-dependant for a variable (i.e. an instance of the variable structure).
  - `(tally (variable-statistics variable))`
  - `(tally (variable-history variable))`
- For data that is not time-dependant, the value for each data point has a unit weight (i.e., 1).
  - For example, if you tallied a variable that had values of 1, 2, 3, and 4, the average would be 2.5, regardless of the durations of each value.
- The talliers are updated whenever the value of a variable changes.



# Tally and Accumulate Example

```
(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)))))
```

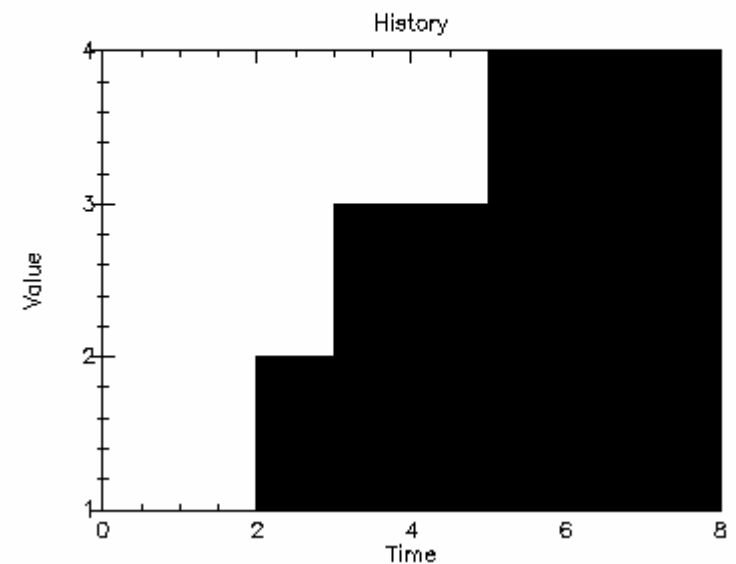
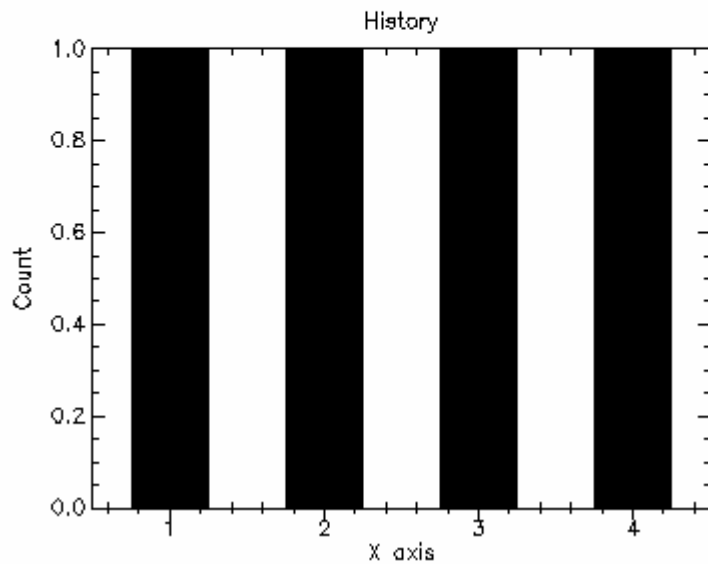


# Tally and Accumulate Example - Output

```
> (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
Sum = 22.0
Mean = 2.75
```





- The following statistics are provided:
  - `statistics-n`
  - `statistics-sum`
  - `statistics-mean`
  - `statistics-sum-of-squares`
  - `statistics-mean-square`
  - `statistics-variance`
  - `statistics-standard-deviation`
  - `statistics-maximum`
  - `statistics-minimum`
- The `variable-statistics` function returns the statistics data collector for a variable or `#f` if there isn't one defined.
- Shortcut functions (e.g. `variable-n`) are provided to access each statistic for a variable.
- The table on the next slide shows the computations performed for accumulating or tallying the statistics for a variable.



| statistic          | accumulate                                                     | tally                           |
|--------------------|----------------------------------------------------------------|---------------------------------|
| n                  | $\text{time}_{\text{current}} - \text{time}_0$                 | number of samples of X          |
| sum                | $\Sigma(X * (\text{time}_{\text{current}} - \text{time}_L))$   | $\Sigma X$                      |
| mean               | sum/n                                                          | sum/n                           |
| sum-of-squares     | $\Sigma(X^2 * (\text{time}_{\text{current}} - \text{time}_L))$ | $\Sigma X^2$                    |
| 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 (variance)                                                | sqrt (variance)                 |
| maximum            | maximum X for all X                                            | maximum X for all X             |
| minimum            | minimum X for all X                                            | minimum X for all X             |

$\text{time}_{\text{current}}$  = current simulated time

$\text{time}_L$  = simulated time variable was set to its current value

$\text{time}_0$  = simulated time variable was created, initially assigned a value, or last reset

X = variable value before change occurs



- A history maintains a record of the value of a variable (and durations for an accumulated history).
- The fields for a history include:
  - `time-dependent?` # $t$  if the history is accumulated
  - `initial-time` Time of the first value
  - `n` Number of entries
  - `values` List of values
  - `durations` List of durations or `()`
- Durations are used, as opposed to times, to simplify the use of the weighted statistics functions in the science collection.



- The `history-plot` function provides graphical output of a history using the PLOT Package and histogram from the science collection.
- Time-dependant (i.e. accumulated) histories are plotted as value versus time.
- Histories that are not time-dependant (i.e., tallied) are plotted as histograms.
  - If all of the values are discrete, a discrete histogram is used.
  - Otherwise, a histogram covering the entire range of values of the history with 40 bins is used.



## Example 3 – 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 (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))))
```



## Example 3 – Data Collection (cont'd)

---

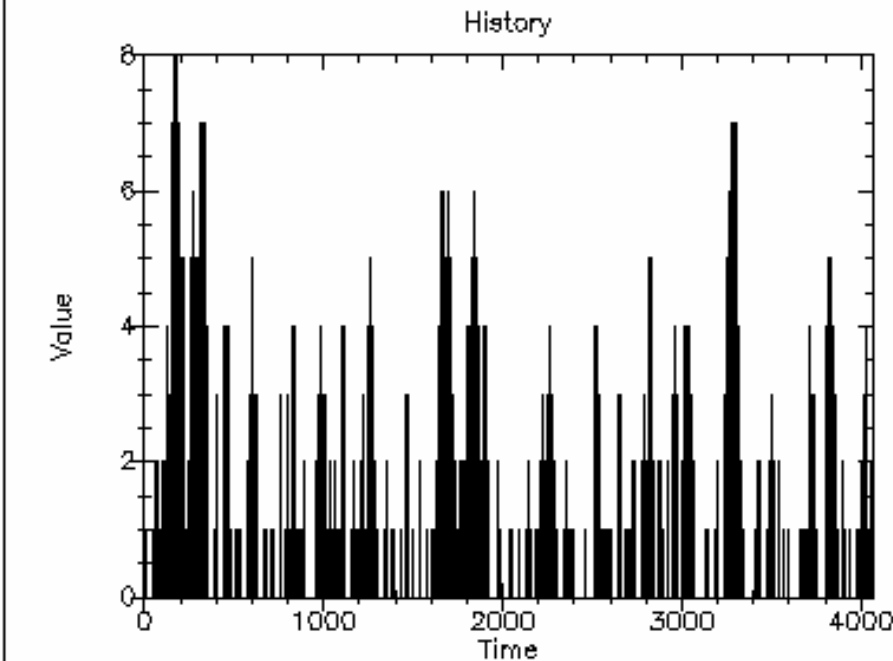
```
(define (run-simulation n)
 (with-new-simulation-environment
 (set! attendant (make-resource n-attendants))
 (schedule (at 0.0) (generator n))
 (accumulate (variable-statistics (resource-queue-variable-n attendant)))
 (accumulate (variable-history (resource-queue-variable-n attendant)))
 (start-simulation)
 (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)))
 (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))))))
```



# Example 3 – Output

```
> (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
```





## More Data Collection Examples

---

- As trivial as the example system we have been modeling is, there are still some simulation techniques we can demonstrate using it.
- Up to now, we have been running one run of a simulation model. This is useful when building the model. But, in general, we want to run the simulation model many times and look at the distributions of the outputs.
- The next two examples use the same basic model as Example 2 (and 3), but run it multiple times and generate the distribution of some output variable of interest.
  - Open Loop Example – Running a simulation model open loop means that whenever a resource request is made, it is immediately granted – that is, there are infinite resources. In the simulation collection, we do this by setting the number of resources to `+inf.0` (positive infinity). We look at the distribution of the maximum attendants required.
  - Closed Loop Example – Runs the simulation model multiple times (with a fixed number of attendants). We look at the distribution of average queue length.



```
; 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))))
```



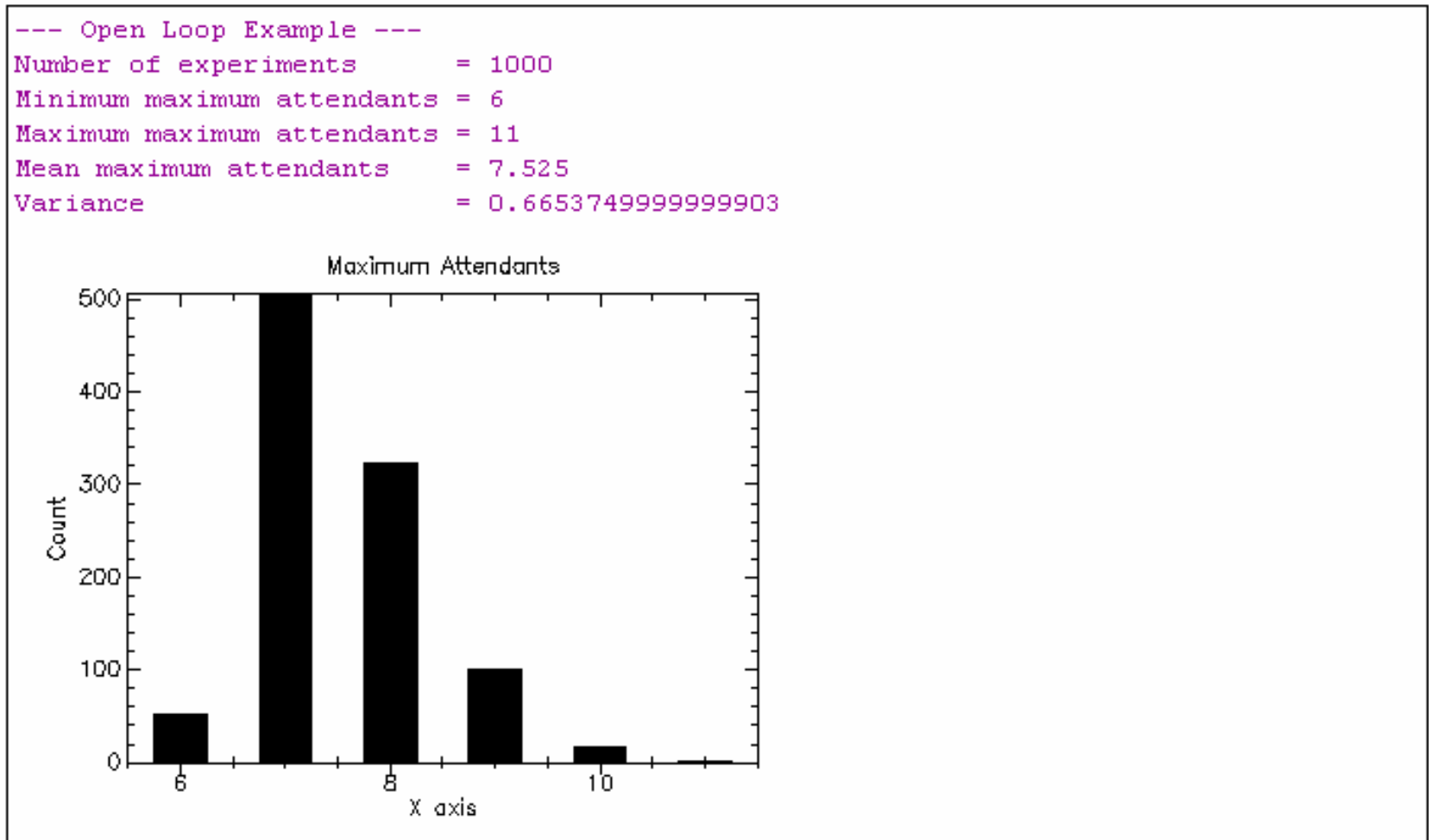
```
(define (run-simulation n1 n2)
 (with-new-simulation-environment
 (let ((max-attendants (make-variable)))
 (tally (variable-statistics max-attendants))
 (tally (variable-history max-attendants))
 (do ((i 1 (+ i 1)))
 ((> i n1) (void))
 (with-new-simulation-environment
 (set! attendant (make-resource +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))
 (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))))
```



# Open Loop Example – Output

```
> (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
```





```
; 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 (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))))
```



## Closed Loop Example (cont'd)

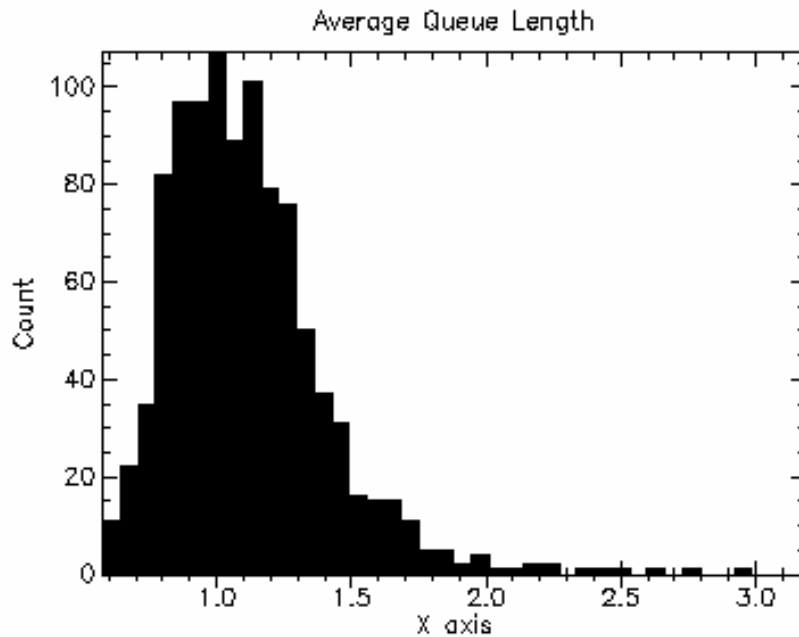
```
(define (run-simulation n1 n2)
 (let ((avg-queue-length (make-variable)))
 (tally (variable-statistics avg-queue-length))
 (tally (variable-history avg-queue-length))
 (do ((i 1 (+ i 1)))
 ((> i n1) (void))
 (with-new-simulation-environment
 (set! attendant (make-resource n-attendants))
 (schedule (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)))
```



# Closed Loop Example – Output

```
> (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
```





- A set is a general structure that maintains a list of its elements.
- In the simulation collection, a set is implemented as a doubly-linked list (for efficient insertion and deletion). Also, the number of elements in the set is implemented as a variable for data collection.
- Sets are created using the `make-set` function. The type of set, `fifo` or `lifo`, can be specified. The default type is `fifo`.
  - `(make-set [type])`
- The number of elements in a set is available using the function `set-n`. The associated variable is available using the function `set-variable-n`.
- The `set-empty?` predicate function is `#t` if the specified set is empty, and `#f` otherwise.
- The functions `set-first` and `set-last` return the first or the last element of the specified set, respectively.



- There are many operations defined on sets, including:
  - `(set-insert! set item)`
  - `(set-insert-first! set item)`
  - `(set-insert-last! set item)`
  - `(set-remove! set [item])`
  - `(set-remove-first! set [error-thunk])`
  - `(set-remove-last! set [error-thunk])`
- There are also iterators defined:
  - `(set-for-each-cell set proc)`
  - `(set-for-each set proc)`
- The `set-find-cell` function returns the cell containing the specified element, or `#f`.
- Note that the names of the set field mutators look a bit strange, e.g. `set-set-n!`, but they should never be seen in user code.



- This is a model of an industrial furnace that heats ingots for some industrial process. Subsequent versions of this model will be used to demonstrate the continuous simulation capability, but this one is strictly a discrete-event simulation.
- The model uses a set, `furnace`, to keep track of ingots that are in the furnace.
  - `(set-variable-n furnace)` is used for data collection.
- Note the use of the `self` variable within the `ingot` process. It refers to that specific instance of the process.
- Note the use of `make-random-source-vector` to create a vector of random sources for the simulation model.
- The code is provided in a separate handout.



```
> (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
```

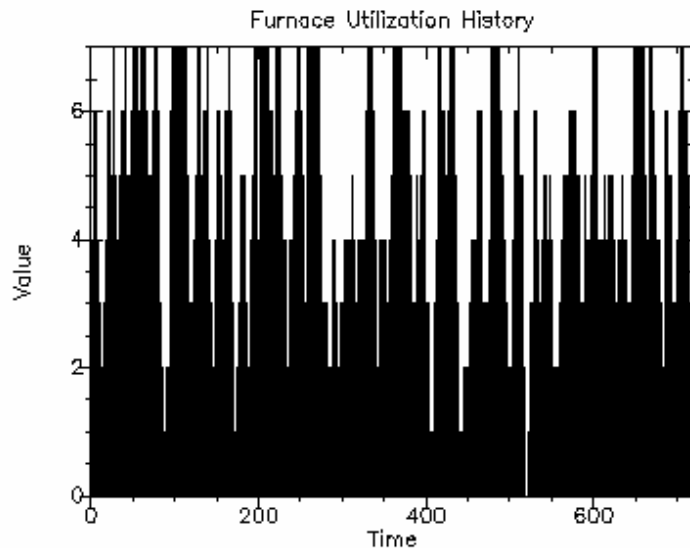
```
-- Furnace Utilization Statistics --
```

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

```
Variance = 3.2693449366238347
```

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

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





- The simulation collection provides both discrete-event and continuous simulation capabilities.
- Continuous variables contain the state data for a continuous model.
- A continuous model is defined using the `work/continuously` special form.
- The ordinary differential equation routines for initial value problems provided by the science collection (and ported from the GNU Scientific Library) are used for integrating the differential equations that define a continuous model.
  - Any of the ODE steppers that don't require a Jacobian matrix can be used.
  - The standard ODE control is used by default, but its parameters can be changed; an alternative can be used; or you can specify there is no control function (i.e. fixed step size) by setting it to `#f`.
  - The ODE evolver is used and the step-size can be limited.



## Continuous Simulation Models (cont'd)

---

- The set of differential equations being evaluated changes as processes enter or leave working/continuously statements.
- All of the continuous variables for the processes that are currently working continuously are stored a state vector representing the state of the (continuous) system.
  - For continuous variables that are in processes that are currently working continuously, variable value works on the state vector.
- All of the differential equations for all of the processes that are currently working continuously are evaluated at the same time under the control of the ODE stepper.



- Continuous variables are used to implement continuous models.
- The `make-continuous-variable` function creates a continuous variable.
  - `(make-continuous-variable [initial-value])`
- When a continuous variable is used outside the context of a process that is currently working continuously, it works like a normal variable.
- In the context of a process that is currently working continuously, `variable-value` works on the value in the state vector.
- Continuous variables have an additional (pseudo-)field, `variable-dt`, that is the computed derivative of the variable.
  - The derivative value is computed in the continuous models as specified in a `work/continuously` statement.
- A continuous variable is a variable and all of the corresponding data collection capabilities are available.



## Simulation Control (Continuous)

- The main change to the simulation control routines from a user's perspective is the addition of the `work/continuously` macro.
- The `work/continuously` macro defines a continuous model.
  - ```
(work-continuously  
  [until condition]  
  body ...)
```
- The `condition` specifies the terminating condition, if any, for the continuous model.
- The `body` expressions should compute the derivatives for any continuous variables in the continuous model.
 - The body expressions shouldn't have any side effects other than setting the derivatives.
- When a `work/continuously` form is evaluated, it adds an event to the continuous event list.



- The `start-simulation` function (i.e. the main simulation loop) has been enhanced to support continuous models.
 - When there are events on the continuous event list and the main loop needs to advance the time (i.e. execute a future event), rather than just jumping right to that time, it evaluates the continuous models on the continuous event list and advances time in small (controlled) steps.
 - At the end of each step, the terminating are evaluated. If any are true, the corresponding process is resumed (i.e. added to the now event list) to continue execution past the `work/continuously` form.
 - Also, at the end of each step, the continuous variables are set from the values in the state vector. This allows any data collectors to run.



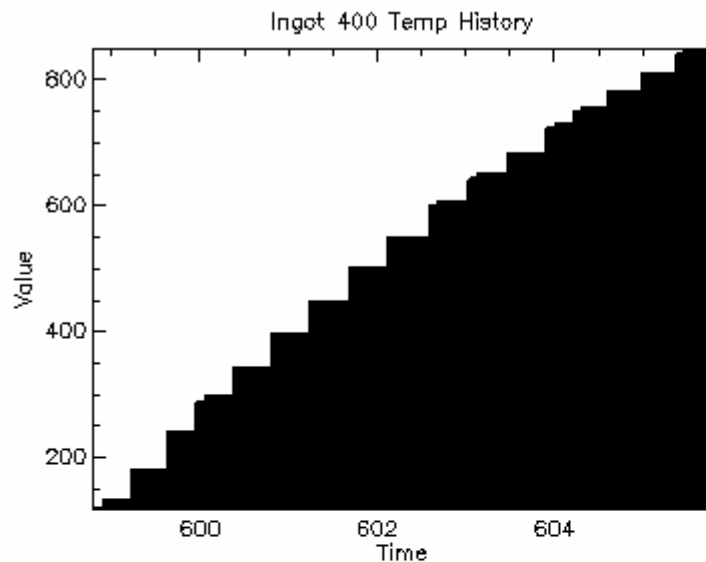
Should there be a `wait/continuously`?

- When I was preparing this talk, I thought making a flippant remark like, “I should have implemented a `wait/continuously`,” would be funny. Unfortunately, it started me thinking and I’ve pretty much convinced myself that it would actually be useful.
- If we have a `work/continuously` call that has a terminating condition but no body expressions, we essentially are waiting continuously.
- An example with the furnace model might be a thermostat that waits until the temperature exceeds some desired temperature. A `wait/continuously` call to do this might look like:

```
(wait/continuously  
  until (>= (variable-value furnace-temp)  
         desired-temp))
```
- Should there be a `wait/continuously`?



- We extend the furnace model to include a continuous model of the ingot heating.
- The current ingot temperature is stored in the continuous variable `current-temp`.
- The continuous model of the ingot heating is:
 - ```
(work/continuously
 until (>= (variable-value current-temp)
 final-temp)
 (set-variable-dt! current-temp
 (* (- furnace-temp (variable-value current-temp))
 heat-coeff)))
```
- There is also more data collection.
  - Every 100<sup>th</sup> ingot we plot the history of `current-temp`.
- The code is provided in a separate handout.



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

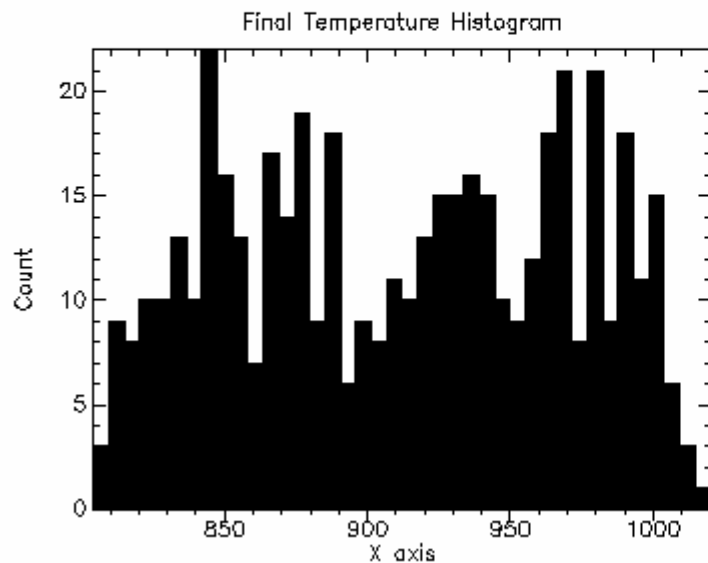
Maximum Heat Time = 10.88442744599513

Minimum Heat Time = 4.867835442878146



-- Final Temperature Statistics --

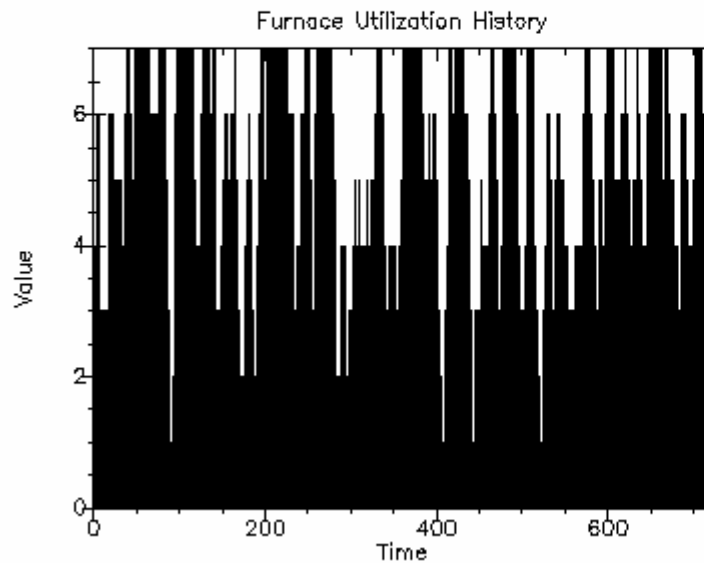
Mean Leave Temp = 912.2921783644567  
Variance = 3300.2864186657825  
Maximum Leave Temp = 1020.1172721368804  
Minimum Leave Temp = 804.2299580285976





# Furnace Model 2 – Output (cont'd)

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

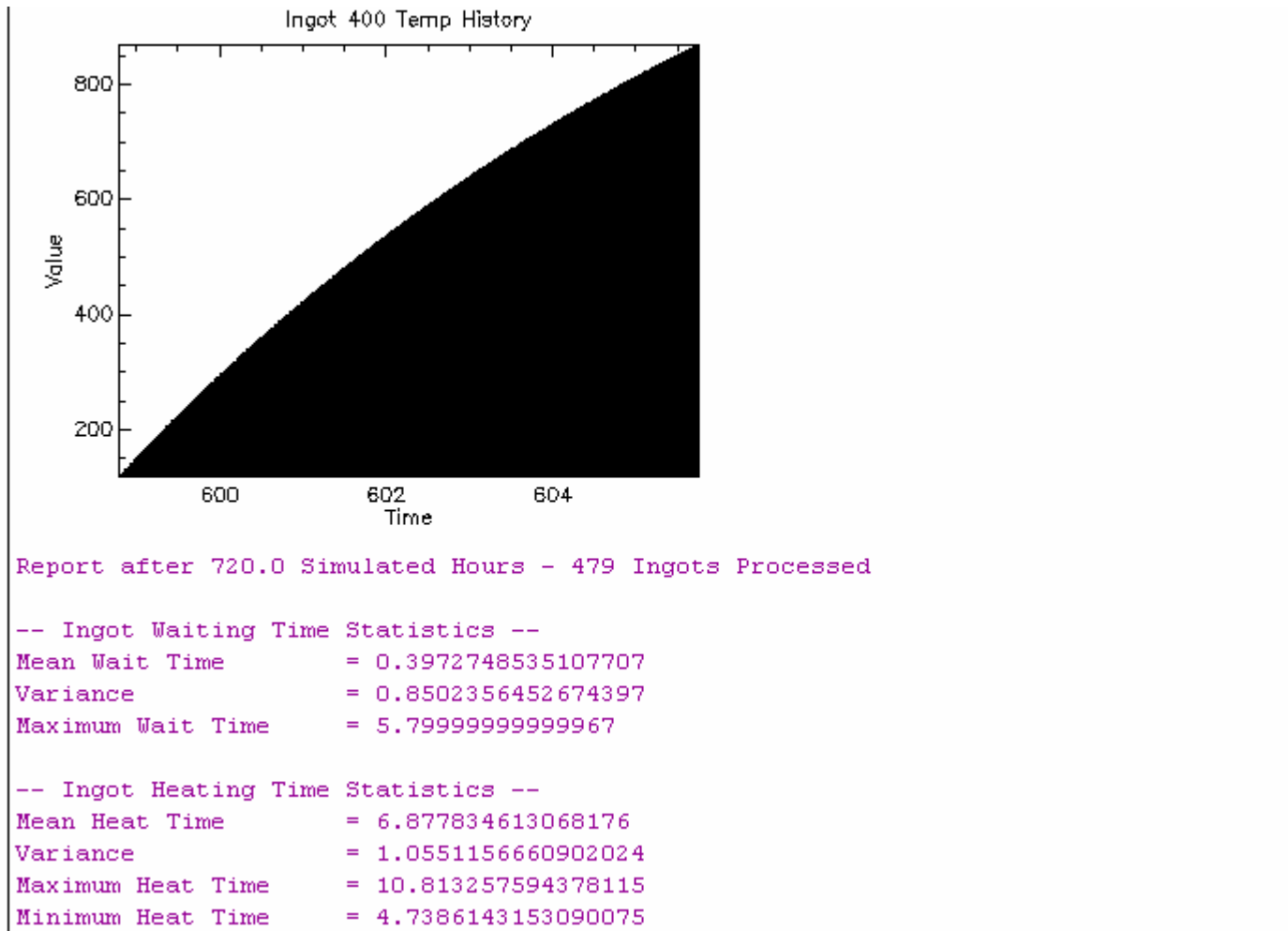




- Looking at the history plot of the ingot heating, we see a step function. Looking at the plot of final ingot temperatures, we see that we overshoot the 1000 degree mark (which was supposed to be our upper limit) by almost 20 degrees in some cases. What gives?
  - First, the model is doing exactly what we told it to do. (Or at least what I designed it to do.)
  - The differential equations are very well behaved – in fact, they are almost linear. The default ODE controller can set the step size rather high and still have an error estimate **AT THE POINTS THAT ARE EVALUATED** within our specified tolerance (which defaults to  $1.0e-6$ ).
    - This would be great if we had, for example, known exactly how long each ingot was to be heated. For example, we would know rather precisely the ingot temperature after, say, 2.5 hours.
- We will modify the model to provide a fixed step size of 1 (simulated) minute (e.g.  $1/60$  hour).



- The following code is added to the initialize routine.
  - `(current-simulation-step-size (/ 1.0 60.0))`  
`(current-simulation-control #f)`
- The first line sets the step size to 1 minute, since the basic time unit is hours.
- The second lines removes the default ODE control that changes the step size. With no ODE control, we have a fixed step size.
- You can look at the ODE section of the PLT Scheme Science Collection Reference Manual for more details.
- The code is provided in a separate handout.

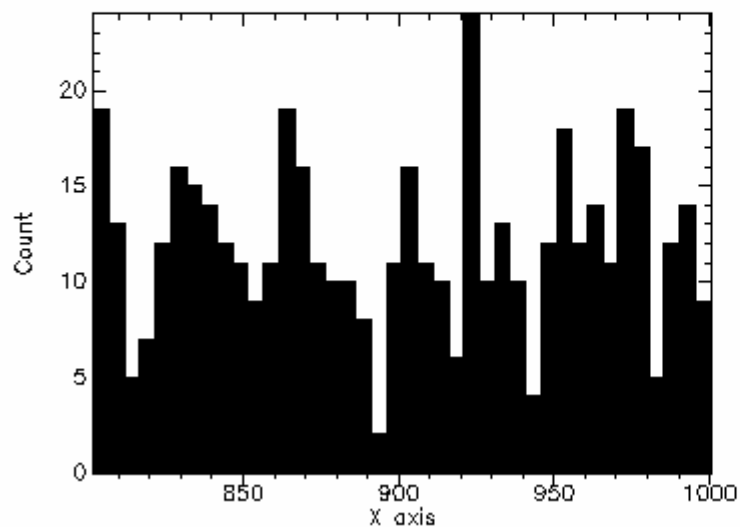




-- Final Temperature Statistics --

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

Final Temperature Histogram





- An alternative is to keep the ODE control, but limit the step size.
  - This allows the step size to float, subject to the upper limit.
  - Still maintains the accuracy control for step sizes below the upper limit. (The accuracy should also be there at the limit.)
- The following code is added to the initialize routine.
  - `(current-simulation-max-step-size (/ 1.0 60.0))`
- The code is provided in a separate handout.
- The results are almost identical to Furnace Model 2b and are also provided in a separate handout.



- Furnace Model 3 extends the model to include a continuous model of the furnace itself.
- The furnace model is an example of a continuous model with no terminating condition. It will run forever if not explicitly stopped.
- It also shows how continuous variables from outside of the process can be used in the differential equations.
- The output is similar to that of Furnace Model 2a and 2b and is not included here. It is provided in a separate handout.
- The code is provided in a separate handout.



- The simulation collection also provide an object oriented interface to user defined simulation objects.
  - Currently just processes and resources.
- Uses the class collection provided with PLT Scheme.
- It is not integrated into the rest of the simulation collection as it might be,
  - The good news about that is that there are no dependencies on the PLT Scheme class collection.
    - Alternate object-oriented mechanisms can be used.
  - The bad news is that none of the convenience macros (e.g., `with-resource`) know about the classes.
- The `process%` class does provide an abstraction that is lacking in standard processes.
  - User specified state information can be encapsulated and shared.



## Discussion: Simulation Classes

---

- One of the problems with Scheme in general is the lack of a standard class system for the language. As a result, there are a myriad of alternative class packages floating around.
  - PLT Scheme does have a standard class collection. However, the definition of the ‘standard’ class collection has changed over time. There is no reason to believe it won’t again.
  - If I do embrace the use of classes throughout the simulation collection, it will use the PLT Scheme class collection.
- Is an object-oriented (i.e. class-based) interface important for languages embedded in Scheme?
- Is the ability to use alternative class implementations important?



- A process class is an object-oriented representation of a process.
  - It encapsulates a process object.
  - Allows the sharing of user-defined process state information via fields in the process class.
- The `define-process-class` macro defines a new process class.
  - ```
(define-process-class (name [superclass-expr])  
  class-clause  
  ...  
  body-expr)
```
- For the specification of class clauses, see the documentation for the PLT Scheme class collection in the MzLib documentation.
- The `body-expr` is a single expression that is the body of the encapsulated process.
 - Don't forget to use `begin` if there are multiple expressions for the process body – which is the normal case.



- The encapsulated process is scheduled immediately (i.e. `now`) when an instance of a process class is created.
 - These are different semantics than processes, which are created using the `schedule` macro.
- The `process%` class provides the following methods:
 - `get-state`
 - `get-time`
 - `set-time`
 - `interrupt`
 - `resume`



- A resource class is an object oriented representation of a resource.
 - It encapsulates a resource.
- The `define-resource-class` macro defines a resource class.
 - ```
(define-resource-class (name [superclass-expr])
 class-clause
 ...)
```
- For the specification of class clauses, see the documentation for the PLT Scheme class collection in the MzLib documentation.
- The `resource%` class has a `units` init-field that is used to specify the number of units for an instance of a resource class.
- The `resource%` class provides the following methods:
  - `request`
  - `relinquish`
  - `satisfied-variable-n`
  - `queue-variable-n`



## Example 4 – 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)))
```



## Example 4 – Simulation Classes (cont'd)

---

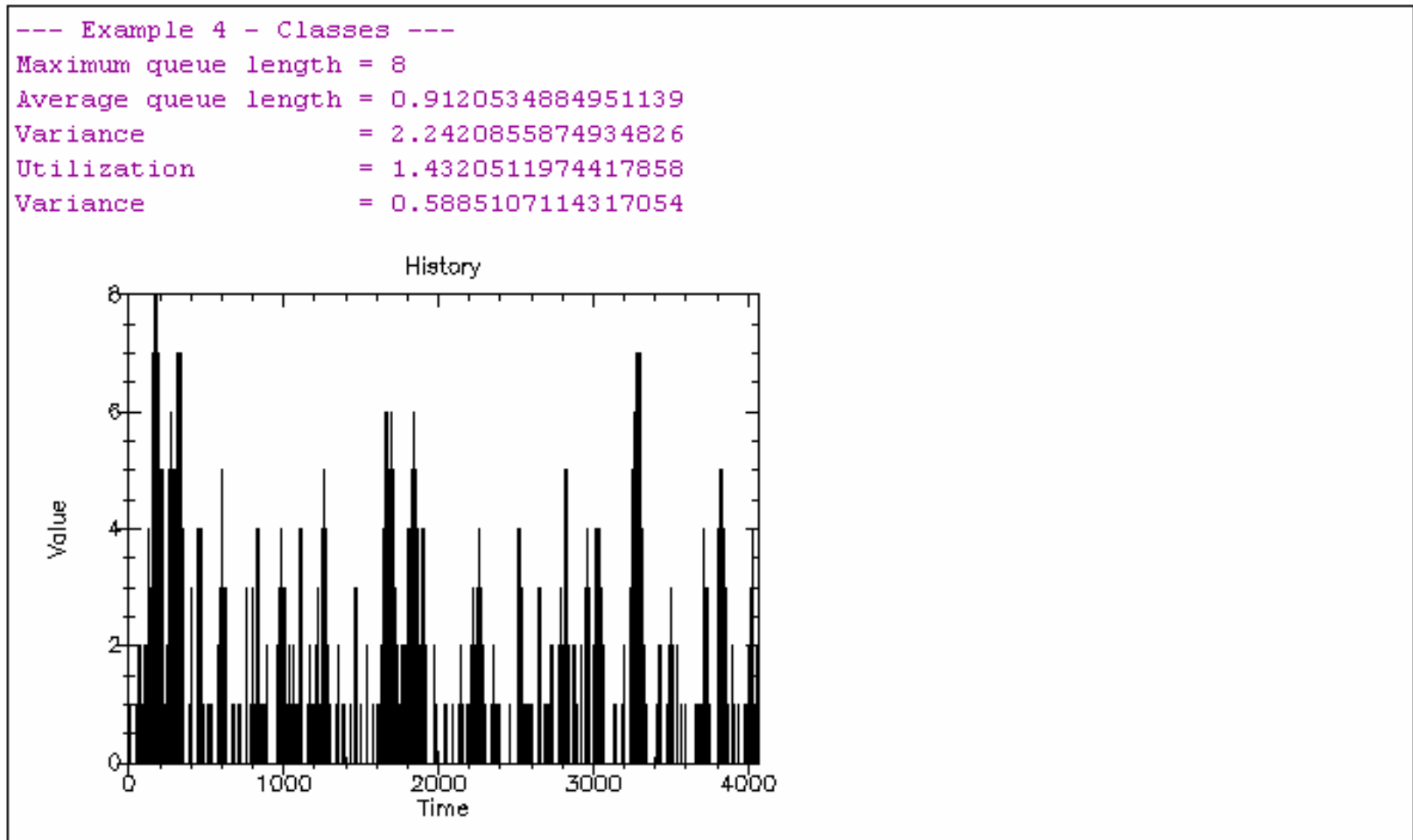
```
(define (run-simulation n)
 (with-new-simulation-environment
 (set! attendant (make-object resource% n-attendants))
 (make-object generator% n)
 (accumulate (variable-statistics (send attendant queue-variable-n)))
 (accumulate (variable-history (send attendant queue-variable-n)))
 (start-simulation)
 (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))))))
```



# Example 4 – Output

```
> (run-simulation 1000)
```

```
--- Example 4 - Classes ---
Maximum queue length = 8
Average queue length = 0.9120534884951139
Variance = 2.2420855874934826
Utilization = 1.4320511974417858
Variance = 0.5885107114317054
```





## Simulation Control (Advanced)

---

- The advanced simulation control functions allow process to suspend themselves or for processes to interrupt or resume other processes.
- These can be used to implement inter-process control strategies that are more complex than, for example, resources.
- The `suspend-process` function allows a process to suspend itself.
  - `(suspend-process)`
- The `interrupt-process` allows one process to interrupt another process. The interrupted process must currently be in a `wait/work`.
  - `(interrupt-process process)`
  - The `event-time` field of the event for the process is set to the amount of time remaining in the `wait/work`.



- The `resume-process` allows an interrupted process to be resumed.
  - `(resume-process process)`
  - The `event-time` field of the event for the process specifies the time remaining in the `wait/work`.
- Note that `suspend-process` sets the `event-time` field of the event for the process to zero. Therefore, a `resume-process` can be used to resume a suspended process also.

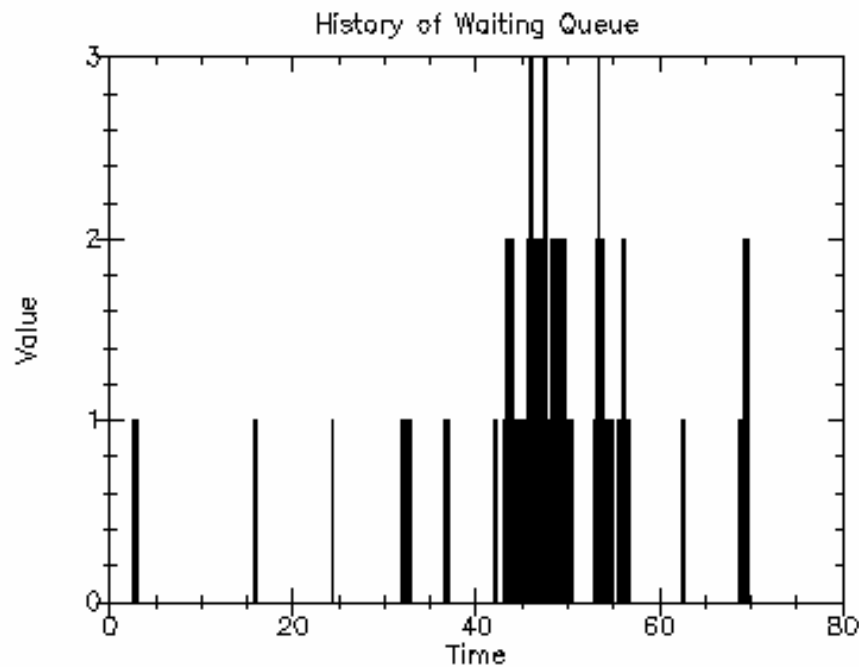


- The Harbor Model demonstrates the use of advanced simulation control.
- Ships are modeled using a process class, `ship%`. The `ship%` class has an `unloading-time` field that is accessible outside of the process.
- The dock contains up to two ships. A single ship can be unloaded twice as fast as two ships.
- The harbor master is called whenever a ship arrives or leaves. It allocated ships to the dock; adjusts the unloading time based on the number of ships in the dock; and removes ships from the queue as needed.
  - The harbor master is a procedure, not a process. It's actions are instantaneous and no timing is required.
- The code is provided in a separate handout.



```
> (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
```





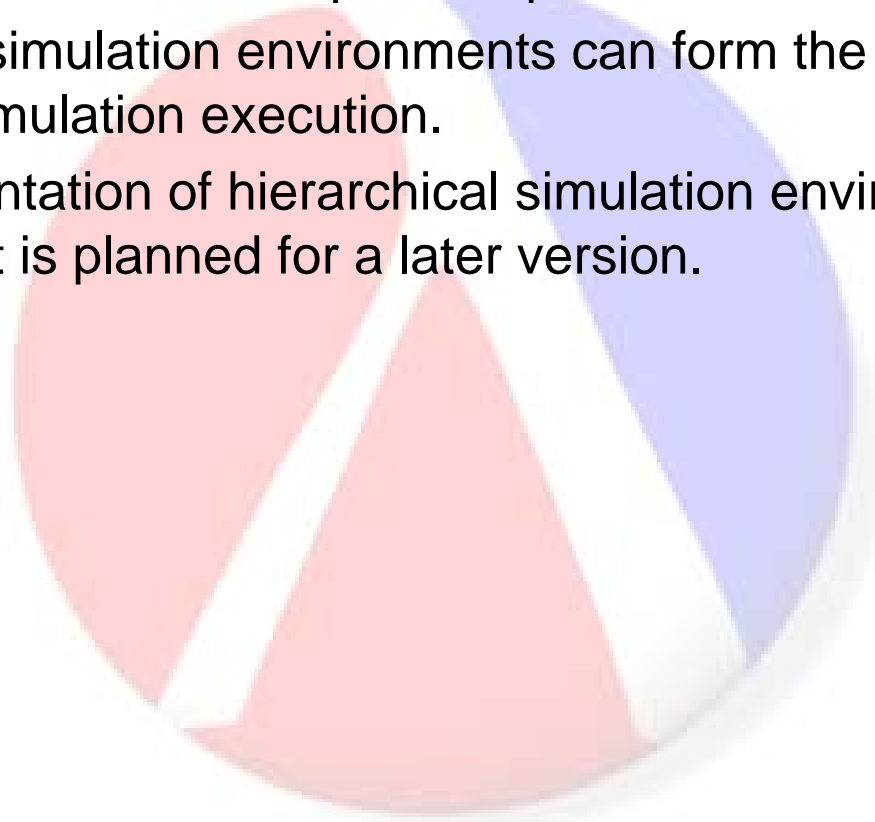
## Simulation Environments (Hierarchical)

---

- Hierarchical simulation environments allow simulation objects and events to be distributed across a tree of simulation environments.
- Each simulation environment has a unique parent simulation environment – except a root simulation environment that has no parent.
- Each simulation environment has a, possibly empty, list of children simulation environments.
- A single simulation main loop controls the execution of an entire tree of simulation environments from its root.
- A single event in the parent simulation environment represents a child simulation environment.
  - This event may be on the now event list or the future event list depending on the next event to be executed in the child simulation environment.
- All continuous events are rolled up to the root level.

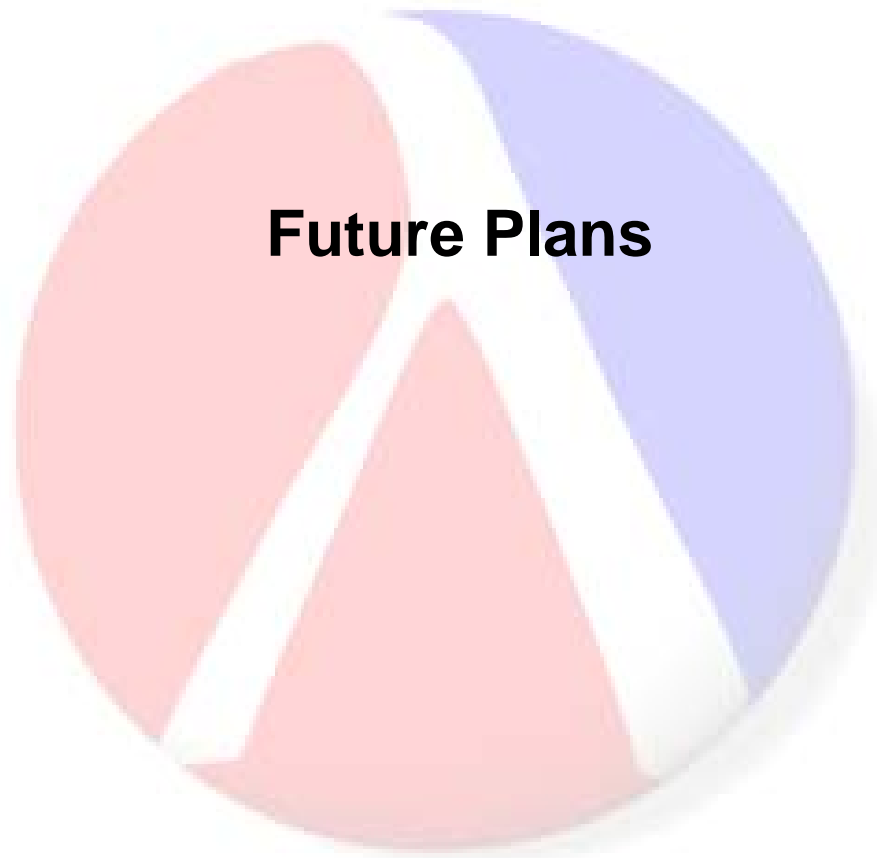


- Note that this is different than independent simulation environments such as were used in the Open-Loop and Closed-Loop examples.
- Hierarchical simulation environments can form the basis for distributed simulation execution.
- The implementation of hierarchical simulation environment is not complete, but is planned for a later version.





- A component encapsulates a child simulation environment and a set of simulation objects.
- For example, the Furnace Model may be extended to include a plant component.
  - The plant component would, in turn, include the soaking pits and furnaces at a location.
  - The plant component could then be instantiated multiple times to represent multiple plant instances.
- It is likely, that all components will be implemented as component classes.
- Components and component classes have not yet been implemented, but are planned for a later version.



A large, semi-transparent version of the PLT Scheme logo is centered on the page. It consists of a circle divided into three segments: a red segment on the left, a purple segment on the right, and a white segment at the bottom. The segments are separated by thin white lines.

# Questions and Answers





- The purpose of the workshop is to help anyone interested in getting the PLT Scheme Simulation Collection, and therefore the PLT Scheme Science Collection, up and running.
- Install PLT Scheme (also known as DrScheme)
  - The current versions of the science and simulation collections require PLT Scheme Version 301.
    - There are source code differences that preclude running them on Version 209.
    - We need Version 301 or later. Earlier 299/300 versions had a bug in PPlaneT that prevented proper compilation as well as a bug in the PLoT package that caused an error. (2D plots still error, but a fix is available.)
    - There are version for Windows (95 and up), Mac OS X, Mac OS Darwin, Linux (various flavors), UNIX (various flavors), or from source code.
  - Use the ‘Pretty Big (includes MrEd and Advanced)’ language option.
- Install the PLT Scheme Science Collection and PLT Scheme Simulation Collection using the PPlaneT command line.



- Test the installation by running the examples.
  - PLT Scheme Science Collection examples
  - PLT Scheme Simulation Collection examples
- Reference manuals are available as pdf files.
  - The PLT Scheme Science Collection Reference Manual, Version 2.0 is complete.
  - The PLT Scheme Simulation Collection Reference Manual, Version 1.0 is still a draft, but it largely complete.
- Workshop Examples
  - These are from SimPy – a Python simulation package that some in the Denver LISP User's Group have used in the past
  - Jackson - A simulation of messages passing through a network of queues
  - Cellphone – A cell phone system



- M. Douglas Williams, PhD  
Sr. Scientist  
Science Applications International Corporation  
Denver, CO  
[m\\_douglas\\_williams@saic.com](mailto:m_douglas_williams@saic.com)  
(303) 229-0315

