

PLT Scheme Knowledge-Based Simulation Part 2 – The Simulation Collection

Dr. Doug Williams m.douglas.williams@saic.com November 16, 2005



Agenda

- 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 Simulation Collection



PLT Scheme

•

- 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



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



PLT Scheme Simulation Collection

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



PLT Scheme Inference Collection

- Inference Environment
- Inference Control
- Rule Sets
- Rules
- Assertions







- Development of all three collections is being moved to the Schematics project at SourceForge.
- PLT Scheme Science Collection
 - Version 2.0 Stable
- PLT Scheme Simulation Collection
 - Version 0.9 Stable
 - Features remaining for Version 1.0 release
 - Hierarchical environments
 - Components
 - Working on reference manual for Version 1.0 release
- PLT Scheme Inference Collection
 - Code for a naïve forward-chaining inference engine has been ported to PLT Scheme



Schedule

- PLT Scheme Science Collection
 - − Release 1.0 October 2004 ✓
 - Release 2.0 November 2005
 - Ordinary Differential Equations (ODEs)
- PLT Scheme Simulation Collection
 - Release 0.9 November 2005
 - Release 1.0 December 2005
- PLT Scheme Inference Collection
 - Release 1.0 July 2006 (tentative)



Simplified Simulation Example



- This Simplified Simulation Example condenses the basic discreteevent 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 continuationbased, 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)

- Saves the continuation (+ 3 #) in a global variable cc-save.
- (cc-save 2) \rightarrow 5
 - Calls the saved continuation (+ 3 #), which returns 5 to the REPL.
- (+ 4 (cc-save 2)) \rightarrow 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 = $\sim a \sim n''$ (cc-save -3)) $\rightarrow 0$
 - 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 11th iteration of the loop, calls the continuation cc-save, which

prints "value = 13" and returns to the REPL.



- 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 *eventlist*. 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 startsimulation 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*))
```



Simulation Control Code (cont'd)

```
(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 randomexponential to remove external dependencies.
- Uses the PLT Scheme built-in random function.





- 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 ith 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



PLT Scheme Simulation Collection

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



- 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?
 - time
 - now-event-list
 - future-eve<mark>nt-list</mark>
 - loop-next
 - loop-exit
 - event
 - process

#t if the main loop is running simulation time events to be executed now events to be executed in the future continuation to return to the main loop continuation to exit the main loop executing event or #f 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 currentsimulation-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.
- 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



f

- 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 #
- 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 (lib "simulation.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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)))
```



Example 0 – 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



- 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 (lib "simulation.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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)))
```



Example 1 – 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



- 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
 - units-ava<mark>ilable</mark>
 - units-allocated
 - satisfied
 - queue

Total number of units Number of units not allocated Number of units allocated Set of processes satisfied 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 (lib "simulation.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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-relinguish 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)))
```



Example 2 – 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



- 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-dependent data.
 - The tally macro initiates collection of data that is not time-dependent.
- 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 makevariable 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-dependent data for a variable (i.e. an instance of the variable structure).
 - (accumulate (variable-statistics variable))
 - (accumulate (variable-history variable))
- For time-dependent 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 timedependent 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.



- The following statistics are provided:
 - statistics-n
 - statistics-sum
 - statistics-mean
 - statistics-sum-of-squares
 - statistics-mean-square
 - statistic<mark>s-variance</mark>
 - statistics-standard-deviation
 - statistic<mark>s-maximum</mark>
 - statistic<mark>s-minimum</mark>
- 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	time _{current} - time ₀	number of samples of X
sum	Σ(X*(time _{current} - time _L))	ΣΧ
mean	sum/n	sum/n
sum-of-squares	Σ(X ² *(time _{current} - time _L))	ΣX^2
mean-square	<pre>sum-of-squares/n</pre>	<pre>sum-of-squares/n</pre>
variance	<pre>mean-square - mean²</pre>	mean-square-mean ²
standard-deviati <mark>on</mark>	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

time_{current} = current simulated time

time_L = simulated time variable was set to its current value

time₀ = 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?
 - initial-time
 - n
 - values
 - durations

- #t if the history is accumulated Time of the first value Number of entries List of values 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

```
(require (lib "simulation-with-graphics.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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)))
```



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







- 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 (lib "simulation-with-graphics.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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





; Closed Loop Example

```
(require (lib "simulation-with-graphics.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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)))
```



```
(define (run-simulation n1 n2)
 (let ((avg-gueue-length (make-variable)))
   (tally (variable-statistics avg-gueue-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
                                       = \sim a \sim n'' n-attendants)
                                        = ~a~n"
    (printf "Number of experiments
            (variable-n avg-queue-length))
    (printf "Minimum average queue length = ~a~n"
            (variable-minimum avg-gueue-length))
    (printf "Maximum average queue length = ~a~n"
            (variable-maximum avg-gueue-length))
    (printf "Mean average queue length = ~a~n"
            (variable-mean avg-gueue-length))
    (printf "Variance
                                           = \sim a \sim n''
            (variable-variance avg-queue-length))
    (print (history-plot (variable-history avg-queue-length) "Average Queue Length"))
    (newline)))
```



Closed Loop Example – Output





- 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.



Sets (cont'd)

- The 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.



Furnace Model 1 – Output





- 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.



- 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, variabledt, 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.



- 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.



- 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 is useful.
- If we have a work/continuously call that has a terminating condition but no body expressions, we essentially are waiting continuously.
- 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 100th ingot we plot the history of current-temp.
- The code is provided in a separate handout.


Furnace Model 2 – Output





Furnace Model 2 – Output (cont'd)





Furnace Model 2 – Output (cont'd)





- 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.



Furnace Model 2a – Output





Furnace Model 2a – Output (cont'd)





- 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., withresource) 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.



- 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 fiends in the process class.
- The define-process-class macro defines a new process class.

```
- (define-process-class (name [superclass-expr]) class-clause
```

```
body-ex<mark>pr)</mark>
```

- 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 - Classes
```

```
(require (lib "simulation-with-graphics.ss" "simulation"))
(require (lib "random-distributions.ss" "science"))
(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 relinguish)))
```



```
(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 gueue-variable-n)))
  (printf "Variance
                                 = ~a~n"
           (variable-variance (send attendant queue-variable-n)))
                               = \sim a \sim n''
  (printf "Utilization
           (variable-mean (send attendant satisfied-variable-n)))
                                 = ~a~n"
  (printf "Variance
           (variable-variance (send attendant satisfied-variable-n)))
  (print (history-plot (variable-history
                         (send attendant queue-variable-n))))))
```







- 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.







PLT Scheme Simulation Collection

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, possible 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 affect independent simulation environments such as were used in the Open-Loop and Closed-Loop examples.
- Hierarchical simulation environments can also form the basis for distributed simulation execution.
- The implementation of hierarchical simulation environment is not complete, but is planned for Version 1.0.





- 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 Version 1.0.







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 299.
 - There are source code differences that preclude running them on Version 209.
 - We need Version 299.400 or later. Earlier 299 versions had a bug in the PLoT package that caused some graphics routines to fail.
 - 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 in a local collects directory.



- 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 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
 <u>m_douglas_williams@saic.com</u> (303) 229-0315