The Real Agent Architecture

Contents

1 Overview **1**

2 The Real World Simulator **2**
2.1 Simulation Engine 2
2.2 RWS Class Library 2
2.3 RWS Procedure Library 4
2.3.1 Motion & Object Paths 4
2.3.2 Orientation, Rotation, & Kinematics 5
2.3.3 Collisions & Interactions 5
2.3.4 Detecting Contact 5
2.3.5 Sensing 6
2.4 RWS Streams 6
2.5 Visualization 7

3 Double Agent **8**
3.1 MESS 8
3.2 Defining Abstract Agents 8
3.2.1 Example Agent Definition 9
3.3 Reactive Agents 10
3.3.1 Example Reaction 10

4 Running a Simulation **11**
4.1 Simulating With RWS 11
4.2 Simulating With DA 11

1 Overview

The Real Agent Architecture (RAA) is a system for building realistic, continuous environments and AI agents to interact in those environments. Environmental simulation takes place in a package called the Real World Simulator (RWS), while the reasoning component of the RAA system is taken care of in the Double Agent (DA) package.

2 The Real World Simulator

RWS comprises an extensible set of mechanisms for conducting physical simulations. The primary set provided with the simulator include 2D kinematics, visual sensing, and collision detection for a small set of the platonic solids. From an implementation persepective, these mechanisms can be divided into two components: the class component and the procedural component. The former describes RWS entities by their interactional features, imparting on each entity a classification. The latter describes programmatically how the RWS classes behave in the physical simulation. As an implementor of RWS environments, the development cycle begins with a behaviour or feature not supported by RWS. Next, a class or set of classes is created to encompass entities that exhibit that behaviour or feature. Finally, the procedural component to the behaviour is constructed.

2.1 Simulation Engine

The RWS simulation engine provides a shell for advancing the simulation clock. The **advance** function advances the clock an arbitrary amount, and places a logical ordering on the updating of the simulation state. Figure 2.1 shows the logical operation of the **advance** function.[1]

Algorithm 2.1 ADVANCE-SIMULATION

```
ADVANCE(time)
while time< 0 do
    quantum ← granularity-of-simulation()
    paths ← compute-object-paths(objects)
    interactions ← compute-interactions(quantum, objects, paths)
    sort-by-time(interactions)
    quantum ← min(time,first(interactions),quantum)
    interact(first(interactions))
    update-object-locations(objects,paths,quantum)
    update-sensors()
    time ← time - quantum
```

2.2 RWS Class Library

Elements of a Real World environment are instantiations of particular classes of objects. For instance, if we wish to limit movement to a fixed area, we create an instance of a boundary, and place it in the environment. The concept ***boundary*** entails a class of objects that share common attributes: they prohibit movement and are visible, for example.

The Real World offers a basic library of building blocks for creating classes of objects with desirable interactional and observational features. They are listed below, with superclasses printed in italic.

`rw-object`

The root Real World Class. All classes which are instantiated should inherit from this class. Each object inheriting from this class has an ***id*** associated with it.

`physical-object-mixin`

An object with a ***location*** (a point) associated with it.

`path-associative-mixin`

An object that requires a ***path*** slot for motion or interaction detection.

[1] The actual advance function condenses all of the motion and interaction into a single function, `compute-gross-motion`.

`hittable-object-mixin` *path-associative-mixin*

An object of this class causes interactions when in physical contact with other objects of this type.

`oriented-object-mixin`

An object which has a virtual ***orientation*** associated with it. Not to be confused with ***heading***, which is direction of motion, this mixin assigns a distinct ***orientation*** to an object for use in forward kinematic chains and sensor computations.

`rotating-object-mixin`

An oriented object which can take on a rotational velocity em turn-rate is of this class.

`mobile-object-mixin` *path-associative-mixin*

A class of objects with the potential for self-propelled movement. Mobile objects have an instantaneous ***rate*** and ***direction*** associated with them.

`orientation-is-direction-mixin`

A class of objects in which the direction of movement is taken as the object's ***orientation*** (see class `orientation`).

`visible-object-mixin`

A class of objects which are detected by visual sensors. These objects have associated with them a ***color***, which can also be used for visualization. (see section 2.5)

`round-mixin`

One of a set of geometric attribute mixins used for collision and visibility computation. Possesses a ***radius*** slot.

`face`

A superclass for defining faces in 2space by defining two points on the face.

`plate-mixin` *face*

One of a set of geometric attribute mixins used for collision and visibility computation. A **PLATE** is a plane of finite dimension, and the 2 points of the face are it's 2 dimensional endpoints.

`plane-mixin` *face*

One of a set of geometric attribute mixins used for collision and visibility computation. The two points of the face superclass can be any 2 points on the plane.

`mesh-mixin`

One of a set of geometric attribute mixins used for collision and visibility computation. A mesh has associated with it a ***face-list***, which is a list of **FACE** objects.

`attached-mixin` *physical-object-mixin*

A class of objects whose location and orientation (if any) are specified relative to a "parent" object. Computations of absolute orientation and location are performed using the forward kinematics of the attached-object chain. The ***parent*** slot contains the kinematic parent of the attached object.

`sensor` *rw-object physical-object-mixin oriented-object-mixin*

A class of objects which sense things about the environment. Currently all sensors sense visual information.

`fixed-sensor` *sensor*

A class of sensors which remain in a fixed world location

`mobile-sensor` *sensor movable-object-mixin*

A class of sensors which move of their own volition.

`anchored-sensor` *sensor attached-mixin*

A class of sensors which move and rotate with their parent.

2.3 RWS Procedure Library

The RWS procedure library is the functional analog to the class library. It comprises a set of functions that produce the characteristic behaviours of entities in the class library. At the primitive level, an RWS simulation consists of perception, motion, and collision. The first distinction in the way entities perceive, move, and interact is made by their class. The second is based on the values of their features. The first of these distinctions is implemented in CLOS by method dispatching, and the second is implemented by the procedure deisnger.

Analogous to the class library, the procedural library is built up by defining a set of CLOS methods, each corresponding to a class of entities that adhere to a particular procedure. For example, the **`compute-simple-path`** procedure takes an entity as its argument. One procedure will accept objects of the type **`simple-mobile-object`** and compute its path according to some rules, and another procedure will accept objects of the type **`dual-track-mobile-object`** and compute its path according to another rule. The "rule" in this case corresponds to the body of the procedure. The major components to the procedure library are motion path computation, kinematics, interactions, and sensing. We now consider each in more detail.

2.3.1 Motion & Object Paths

Object motion is handled in the Real World by the tandem of methods **`compute-object-path`** and *move-object*. These methods specialize on the mobility mixins used to create the moving object instance. The method **`compute-object-path`** creates an object in the **`PATH`** hierarchy and stores it in the moving object's *path* slot. The **`move`** method updates the object's location based on the path information.

The bulk of the complexity of motion is in modelling objects with eccentric methods of locomotion. Consider an army tank; it's motion is controlled by the torque generated by each of its tank treads. The resulting path is a circular arc as shown in figure 1. The Real World has three types of paths built in.

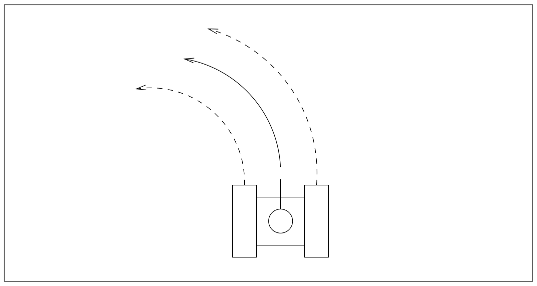

Figure 1: The motion path for an vehicle with two independently controlled treads. The path of the object is the "average" of the circular paths produced by the individual treads.

`null-path` *path*

A path that but produces no motion. The slot ***p1*** contains the location of the object, and slots ***t1*** and ***t2*** hold the start and finish times, respectively.

`simple-path` *path*

The class of linear paths. These paths indicate motion of uniform velocity from ***p1*** to ***p2*** starting at time ***t1*** and ending at time ***t2***.

`arc-path` *path*

The class of motion paths that lie along a circle. The path is defined by the beginning and ending points of the arc, ***p1*** and ***p2***, the circle's ***radius***, and the start and finish times ***t1*** and ***t2***. The tank in figure 1 is an **`arc-path`**.

The motion subsystem of the Real World allows for optimized performance using ***linear approximation*** of eccentric paths. Under the assumption that gross motion is performed over very fine gradations of time, setting the parameter **`*use-linear-approximation?*`** to **`t`** will cause the motion calculations to be performed by an object's **`compute-simple-path`** method rather than **`compute-object-path`**. The **`sompute-simple-path`** method should store the begin and end points of the complex path in a **SIMPLE-PATH**, rather than an **ARC-PATH** or otherwise. This allows for optimizations in the collision detection routines.

2.3.2 Orientation, Rotation, & Kinematics

2.3.3 Collisions & Interactions

When two objects of the class **HITTABLE-OBJECT** attempt to inhabit the same space, the real world signals a ***collision***, and attempts to resolve the situation by invoking an ***interaction*** appropriate to the bodies involved.

Collision detection occurs during a call to **`compute-gross-motion`**. After computing the ideal object paths of the moving objects, the function **`first-contact`** is called to determine the time at which the next collision will occur. If two objects are found to collide, a technique called ***binary path reduction*** is used to find the point of contact. The simulation update is then truncated to the time of first contact. The function **`handle-collision`** is then called, which in turn calls **`collide`** on the two colliding objects.

The **`collide`** function handles how the objects behave given the interaction. Subclasses of the class **HITTABLE-OBJECT-MIXIN** should be created to express how objects behave during collisions. One class, **ELASTIC-OBJECT-MIXIN**, is provided. An elastic object's rate remains constant, while it's direction is reflected over the normal to the point of contact.

2.3.4 Detecting Contact

Two Real World objects are said to be ***in contact*** if their geometries overlap or abutt each other. The method **`contact-internal`** computes whether two objects are in contact, and specializes on their geometry. Function definitions should return non-NIL if and only if the two geometries are found to be overlapping or abutting.

It is worth noting that this is the first instance of a method definition that is ***symmetric***. That is, calling **`contact-internal`** with a **ROUND-MIXIN** and a **PLATE-MIXIN** should perform the same computations as a call with the arguments reversed. Rather than defining mutliple, symmetric copies of the same function, RWS provides several macros. In contact detection, the macro **`defcontact-internal`** is appropriate. It's syntax is identical to **`defmethod`**.

Binary Path Reduction

Binary path reduction (BPR) is a simple method for resolving the initial point of contact between objects who are known to collide within some time period t_c. BPR is essentially binary search along both object's paths, simultaneously, to find the first point of contact.

BPR begins with two objects at time t_i, and a known contact at time t_c. The location of each object is then determined for the time $t_i + \frac{t_c - t_i}{2}$, dividing the paths into two segments. If the objects are in contact at this median point, then BPR is recursively run on the early path segments, otherwise it is run on the later segments. The recursive division continues until an ecceptable resolution has been reached. Resolution defaults to a distance specified in the variable **`*allowable-distance-gradation*`**, but also may be specified in terms of time by setting **`*use-time-not-distance*`** and specifying resoultion in **`*allowable-time-gradation*`**.

Detecting Collisions

The simple use of contact detection for collision detection can fail in a significant way. In the case where the granularity of time is mistakenly too coarse, it may be the case that objects that would truly collide will "pass through" each other in the simulation. Detection of such a situation requires checking for intersecting paths.

The function **intersect** can be used to indicate whether two paths intersect. Paths determined to be in intersection can then be passed through a variant of BPR that mixes path intersection and contact detection techniques to resolve to the point of contact.

Note: Path intersection is not implemented in RWS at this time.

2.3.5 Sensing

Sensing is carried out at the end of each simulation update. The function **compute-sensors** maps the function **sense** over all the sensors in the world. Sensors should be rooted at the class **SENSOR**, and each instantiable sensor type should have some primary **sense** method associated with it. Provided in the real world is one type of sensor, the **PINHOLE-CAMERA**.

Pinhole Cameras

Each pinhole camera has associated with it a number of pixels it can sense (***resoultion***) and a field of view (***spread***) in radians, in addition to orientation and location.

The Real World visual system uses raycasting to determine the visual sensor array. This procedure casts a ray through each pixel of the visual field and matches the ray against all of the visible objects in view. The color of the closest object in view is then used for that pixel of the visual array.

When implementing visible objects, it is then necessary to ensure that the object can in fact be sensed by the raycasting algorithm. The function **ray-instersect** works on a ray and an object to intersect with the ray. The method specializes on the geometry of the object being tested for intersection. Thus, each visible geometry must have a **ray-instersect** method associated with it. This function returns **NIL** if the object is not visible, or the parameter along the ray where the intersection occurs.

2.4 RWS Streams

The RWS provides an added layer of abstraction above objects and slot values. ***Streams*** can be thought of as access channels to real world values such as locations, orientations, and sensor values. Arbitrarily complex streams can be created and later queried for their current value by a simple function call with the stream name. The most typical use is as a liason between RWS and intelligent agents, who have limited access to real world state information, but streams are also useful for recording trial data and debugging.

Streams are created using the RWS macro **defstream**. The syntax for defining a RWS stream is

(**defstream** *name ((world-variable-name world) &key marshal? index) &body stream-generator*)

Once a stream has been defined, its value can be queried by calling **query-stream**, supplying the world and stream name as arguments. Additionally, the function **package-output** can be called to generate a complete list of stream values for an entire world. This function takes a world and the symbol **'streams** as its arguments.

As an example, consider a stream that is to reflect the location of an object called **:rover** in a world named **:my-world**.

```
(defstream :position ((world :my-world) :marshal? :point :index 1)
  (location (get-object world :rover)))
```

The above form will generate a stream called :position. The body of this form will be used to generate the value of this stream. The variable **world** is lexically bound to the world object named **:my-world** over the stream generation body. The **index** keyword allows streams to be ordered.

Marshalling

The **:marshal?** keyword for defining streams allows streams that contain structured data to be destructured from within a call to **package-output**. In the **:position** stream example above, the stream value at any given time will be a **POINT** – in the case of RWS, a two element dotted list. When dealing with this stream in isolation, it is convenient to get the whole point all at once, but when looking at all streams at once, via a call to **`package-output`**, it may be useful to treat each value in the point as a seperate stream, such that the return value is a flat, comprehensible list.

Marshalling is a technique for taking arbitrary data structures and putting them into a general purpose format.[2] For the purposes of RWS, when we *marshal* a stream value, we turn it into a list. Each element of this list is then treated as a stream for the purposes of **`package-output`**. This translation of an arbitrary type to a flat list is performed in the **`marshal`** function, a method which takes two arguments, the value to be marshaled, and a symbol signifying the type of marshalling to be done. In the case of the **`:position`** stream, the following method definition will handle the marshalling :

```
(defmethod marshal ((value cons) (type (eql :point)))
  (list (eksl-utils::point-x value) (eksl-utils::point-y value)))
```

2.5 Visualization

The Real World has a visualization interface based on CLIM. Users with access to CLIM can invoke the Real World GUI by calling the function **`show-world`** with the name of the world to view as it's argument. Figure *no figure* shows a screenshot of a typical Real World Visualization window.

info forthcoming

[2] The term "marshalling" comes from the distributed systems literature.

3 Double Agent

The Double Agent component to RAA provides an interface between objects in the Real World simulation and bodies of user-created intelligent control code. One begins by defining abstract agents and the real world objects, sensors, and streams with which they are associated, and completes the agent by defining its behaviour.

Double Agent provides a framework for implementing two kinds of behaviours: ***reactive*** and ***deliberative***. In the former, real world stimulus causes an instantaneous reaction by the agent. In the latter, an agent engages in ***thinking activities***, which have some duration associated with them. We now consider the implementation of agents and their behaviours in the Double Agent framework.

3.1 MESS

The difference between simulating agent control and simulating bodies in motion is that in the latter, activity is continuous and constant. Simulation is carried out by approximating motion over very fine grained snippets of time. The former is more abstract; an agent issues ***directives*** to its embodiment that allow it to behave to its specification. Agent based control can be viewed as a series of discrete events – a plan of directives to be carried out at specific times to accomplish a desired effect.

MESS, the Multiple Event Simulation Substrate, provides a framework for discrete-event simulation. In such a simulation, the world is divided into ***streams***, which roughly correspond to forces acting on the simulation. ***Events*** are scheduled within the streams and are processed in order of their specified time to maturity.

MESS and its event streams play three roles in Double Agent. First, each Double Agent simulation is allocated a ***synchronizing agent***. In its simplest form,[3] the synch agent merely generates a periodic ***update*** event to keep the simulation running smoothly when there is little or no agent activity.

MESS's second role is in processing so-called ***external events***. External events are possibly irregularly scheduled events which do not fall logically under the headings of "agents" or "simulation processes".

The final role event streams play is in the implementation of deliberative agents. Deliberative agents are those which spend time thinking before they act, and each deliberative agent is allocated one or more event streams into which it may schedule ***control events***, or events which change parameters of the simulation, such as the heading of a vehicle. Typically, a deliberative agent will engage in a period of thinking, which MESS schedules as a durative activity. Over the duration of thinking, the event stream is blocked. Only when the agent is done deliberating may it use the channel for scheduling control events.

3.2 Defining Abstract Agents

Agents are associated with a RWS world using the function **`proclaim-cognitive-component`**, with the name of the world as its argument. This creates the necessary structures for a world to support abstract agents, including the synch agent. The function **`renounce-cognitive-component`** deletes a cognitive component from existence.

Agents are defined using the **`defagent`** function:

(**`defagent`** *agent-name agent-type world &rest initargs*)

This function creates an agent and places it in the appropriate world. Agent types are created by mixing in predefined agent classes with user-defined ones. The predefined agent classes are:

`agent`

The root class of agents. Has a **`world`** slot.

`stream-associative-mixin`

An agent that gets a stream to schedule activities and events

`thinking-agent` *agent stream-associative-mixin*

An agent that performs operations in the world after deliberation.

[3] The simplest synch agent is of the type `stupid-synch-agent`.

reacting-agent *agent*

An agent that can perform operations in response to well-defined stimuli.

thinking-and-reacting-agent *thinking-agent reacting-agent*

An agent that can both react and think.

sensing-agent-mixin

An agent that can sense its environment. The **sensor-info** slot contains necessary information for more specific sensor processing. The functions **query-sensor** and **query-sensors** provide access to an agent's sensors.

stream-sensing-agent-mixin *sensing-agent-mixin*

An agent whose **sensor-info** slot contains a list of RWS stream names which it has access to.

Note – currently, double agent does not support association of real-world objects as embodiment, but the implemetation of an **embodied-agent-mixin**, with a slot for objects in the environment over which an agent has some control, is imminent.

3.2.1 Example Agent Definition

The following code sample implements a sphere, a sensor, and an agent associated with both.

```
(proclaim-cognitive-component :b-zone)

;;; objects

(defobject :b-zone :agent1-body 'special-ball :color (random-elt *ball-colors*)
  :location (eksl-utilities:point 0 0) :radius 1)

(defobject :b-zone :agent1-camera 'anchored-sensor :location #@(1.1 0)
  :orientation 0
  :spread (/ pi 10) :parent (get-object :b-zone :agent1-body) :resoultion 3)

;;; streams

(defstream :a1v ((world :b-zone) :marshal? :standard)
  (image-vector (get-object world :agent1-camera-1)) x)
(defstream :a1x ((world :b-zone) :marshal? :standard)
  (eksl-utils::point-x (location (get-object world :agent1-body))))

;;; agents

(defclass smart-ball-agent (reacting-agent stream-sensing-agent-mixin)
  ((body :initarg :body :accessor body)
   (sensor :initarg :sensor :accessor sensor)))

(defclass goal-seeking-agent () ())
(defclass static-goal-seeking-agent (goal-seeking-agent)
  ((goal-location :initarg :goal-location :accessor goal-location)))
(defclass dynamic-goal-seeking-agent (goal-seeking-agent)
  ((goal :initarg :goal-object :accessor goal-object)))

(defclass convoy-agent (dynamic-goal-seeking-agent smart-ball-agent)
  ((zero-range :initarg :zero-range :accessor zero-range)))
```

```
(defagent :agent2 'convoy-agent :b-zone
  :sensor-info '(:a2x :a2y :a2v :d)
  :body (get-object (get-world :b-zone) :agent2-body)
  :sensor :a2v :goal-object (get-object :b-zone :agent1-body))
```

The **stream-sensing-agent** and **embodied-agent**[4] provide clean boundaries for agent behaviour. Simply, an agent should only query the streams it has access to and issue directives to objects it embodies. However, there is nothing stopping an agent from subverting the DA API and doing system level RWS calls to access "illegal" streams and objects.

3.3 Reactive Agents

Implementing a reactive agent is accomplished in DA by defining the conditions under which a reaction is appropriate, and then describing the agent's reaction in Lisp terms. The macros **define-stimulus** and **define-response** facilitate these tasks, respectively.

(**define-stimulus** *name agent-binding &body condition-body*)
(**define-response** *name condition agent-binding &body response-body*)

3.3.1 Example Reaction

Below is the definition of a simple stimulus/response pair for the previously defined CONVOY-AGENT. The stimulus, **go-ahead**, always evaluates to **t**, and thus, the agent is always exerting some control. The reaction **aim** is always firing based on the **go-ahead** stimulus. The **aim** computations produce directives that orient the agent's body toward some "target" and move at a speed proportional to its distance from the target.

```
(define-stimulus go-ahead ((a chase+agent))
  t)

(define-reaction aim go-ahead ((a chase+agent))
  (let* ((target-angle (atan* (- (eksl-utils::point-y (goal-location a))
                                 (eksl-utils::point-y (location (body a))))
                              (- (eksl-utils::point-x (goal-location a))
                                 (eksl-utils::point-x (location (body a))))))
         (dtheta (pi-ize (- (pi-ize target-angle)
                            (pi-ize (direction (body a)))))))
    (setf (redirect-rate (body a)) (/ dtheta 3pi))
    (setf (rate (body a)) (* (min 1 (/ (- (distance (goal-location a)
                                                    (location (body a)))
                                          (zero-range a)) 20))
                             (max (- 1/3 (redirect-rate (body a))) 0)))))
```

3.4 External Events

Often it is useful to simulate events for which there is no corresponding entity in an RAA simulation. Consider, for example, a simulation in which there are varying weather conditions. Changing weather conditions do not fit into the RWS simulation engine anywhere, as the weather does not change in response to any physical entity. Likewise, the weather patterns are not agents. Quite simply, weather events are external events.

In DA, MESS streams are allocated to each distinct force that exists to create external events. Associated with each force is a scheduling mechanism for determining when the next event is to occur and a realizing mechanism which updates the simulation according to the actual event that occurs.

Currently, there is only a single scheduling mechanism provided that creates events on a regular schedule. Realization mechanisms are completely user-defined. The macro **make-regularly-scheduled-event** is used for creating an external force. Its useage is best described with an example.

[4] When it is implemented.

```
(make-regularly-scheduled-event (weather :my-world weather-event
 :period 20)
  (possibly-change-weather-conditions (get-world :my-world)))
```

In the above example, we create an event stream called **weather**. It works in the cognitive component of **:my-world**. It will automatically create an event of type **weather-event**, which will call **possibly-change-weather-conditi**
when such an event is realized. All of the arguments after **weather-event** in the call above are passed along as arguments to the event generator. In the example above, **:period 20** specifies that the time period between weather events should be 20 simulation ticks.

4 Running a Simulation

Running a simulation can be handled from either RWS or Double Agent. As you may have expected, those simulations that make use of Double Agent's reactive and deliberative control mechanisms should be handled by Double Agent. Those that do not can be driven using the RWS mechanisms, perhaps with some improvement in efficiency.

4.1 Simulating With RWS

The most simple way to simulate is using the **`advance`** function in the RWS system. It takes as arguments the name of the world to simulate and the a quantum of time to simulate. RWS will attempt to simulate the whole quantum in one chunk, if possible, so this value should be chosen wisely to express the proper amount of accuracy. One must chose a small enough quantum so that collisions will not be missed and linear approximations will not compromise the desired behaviours, but large enough so that the simulation will run quickly. Trial and error is the only recommended technique for deciding on the granularity of updates.

4.2 Simulating With DA

Double Agent adds extra layers of complexity to running a simulation. In addition to the regularly scheduled updates, DA simulations may add in external events, reaction events, and/or deliberative events.

Users only adding a reactive component to the simulation can simply load Double Agent on top of RWS, define the necessary stimuli and reactions, and call **`advance`** as if Double Agent had nothing to do with the simulation.

Users wanting to perform deliberative actions or external events must drive the simulation using MESS. All event scheduling, including regular synchronizing events, are handled using MESS streams, and the function **`da-advance`** should be used to update the simulation state. This function only requires the world's name as an argument; the granularity of the simulation is prespecified when the cognitive component of the simulation is defined. The **`da-advance`** function advances the simulation to the time of the next scheduled event.