Schema from settings and description 1

wrapper

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | RunWrapper() |  |  |  |  |
|  | SimState |  |  |  |  |
|  | RunWrapper |  |  |  |  |
|  | FinancialModel |  |  |  |  |
|  |  | From this moment the path is known |  |  |  |
|  | simState | Creates a SimState with a new random number generator initialized to the given seed, plus a new, empty schedule. |  |  |  |
|  | MersanTwister | Constructor using a given seed. Though you pass this seed in as a long |  |  |  |
|  |  |  |  |  |  |
|  |  | Initialize the pseudo random number generator. … |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | 1. Schedule **defines a threadsafe scheduling queue** in which **events can be**   **scheduled to occur at future time**. The **time of the most recent event which**  **has already occurred is given by the** <b>**getTime()</**b> method. If the current  time is <tt>BEFORE\_SIMULATION</tt> (defined to be <tt>EPOCH - 1</tt>), then  the schedule is set to the "time before time" (the schedule hasn't started  running yet). If the current time is <tt>AFTER\_SIMULATION</tt> (positive  infinity), then the schedule has run out of time. <tt>EPOCH</tt> (0.0) is  defined as the first timestep for which you can legally schedule a value.  <tt>EPOCH\_PLUS\_ESPILON</tt> is defined as the smallest possible second  Time step for which you can legally schedule a value. If you're scheduling  events to occur on integer time steps, you may want to ensure that your  simulation does not run beyond <tt>MAXIMUM\_INTEGER</tt> (9007199254740992L or  9.007199254740992E15). For values of a double d >= <tt>MAXIMUM\_INTEGER</tt>,  d + 1 == d ! | 1. An **event** is defined as a <b>**Steppable**</b> **object.** You can **schedule events to**   **either occur a single time or to occur repeatedly at some interval**. If the  event occurs **repeatedly**, the schedule will provide you with a <b>Stoppable</b> **object on which** you can call <b>**stop()</**b> **to cancel all**  **future repeats of the event**. If instead you **wish to "stop" a single-time**  **even**t from occurring before its time has come, you should do so through the  use of a <b>**TentativeSte**p</b> object. At present **you cannot delete objects**  **from the Schedule -- just stop them and let them drop out in due course.** | 1. The **schedule is pulsed** **by** calling its <b>**step(...)**</b> method. **Each pulse**,   the **schedule finds** the **minimum time** at which **events are scheduled**, **moves**  ahead **to that time**, **and then calls all the events scheduled at that time.**  Multiple events may be scheduled for the same time. **No event** may be **scheduled**  **for a time earlier than getTime().** **If** at time getTime() you schedule a new  **event for time getTime()**, then actually this **event will occur at time getTime()+epsilon**, that is, the smallest possible slice of time greater than getTime(). |  |
|  |  | 1. **Events at a step** are **further subdivided and scheduled according to their**   <i>**ordering**</i>, an integer. **Objects for scheduled for lower orderings for a**  **given time will be executed before objects with higher orderings for the same**  **time.**  **If** objects are scheduled for **the same time** and have **the same ordering**  value, their **execution will be randomly ordered** with respect to one another. | 1. A schedule may be <i>**sealed**</i> meaning **that it will refuse to accept any**   **further scheduled events even if its time is not yet AFTER\_SIMULATION.** This  is largely done internally by MASON code: you probably will never want to do  this. **Once a schedule is sealed it cannot be unsealed until it is reset().** | 1. **You can** **clear out** the entire Schedule, unseal it, and restart it to BEFORE\_SIMULATION by calling **reset().**   **However, this does not prevent**  **AsynchronousSteppables from** suddenly **rescheduling** themselves **in the queue.**  **Stopping** the simulation **from** within a Steppable object's **step() method** is **best done** by **calling SimState.kill().**  **From the main thread**, the most straightforward **way to stop** a simulation is to **just stop calling schedule.step(...),** **and** proceed **directly to SimState.finish().** |  |
|  |  | 1. **You can get the number of times that step(...)** has been called on the   schedule by **calling the getSteps() method.** This value is incremented just as  the Schedule exits its step(...) method and only if the method returned true.  Additionally, you can get a string version of the current time with the  getTimestamp(...) method. | 1. <b>Note on Synchronization</b>. In order to maximize the ability for threads to access the Schedule at any time, **Schedule uses two locks for synchronization.**   **First**, the <b>**step() method synchronizes on the Schedule</b> itself**. This **prevents step() from being called simultaneously from different thread**s; **also step()** tests to **make sure that it's not called reentrantly** **from**  within **the same thread**  **Second**, <b>many methods **synchronize on an internal**  **lock**</b>, **including step().** **This allows step() to synchronize on the lock only to suck out the relevant Steppables from the Heap and to advance the timestep; ;** all other portions of step() are outside of the lock.  Thus when step() actually steps the Steppables, even in different threads (like  AsynchronousSteppable or ParallelSequence), they can turn around and submit  step-requests to the Schedule even while it's still in its step() method. | 1. One downside to this flexibility is that it's very inefficient to check, at   each step of a Steppable, whether the Schedule has been reset or not. **Thus**  **now if you call** reset() or [better] **SimState.kill()**, the **Schedule will**  **continue to step Steppables until it has exhausted ones scheduled for the**  **current timestep**. Only at that point will it cease. |  |
|  | Scheadule |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | Heap |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | … |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | Bag |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | FinancialModel |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Now wrapper = false  but later we will change it to true |  |  |  |
|  |  |  |  |  |  |
|  | Reporter | Now we have to establish the **file for the reports** (what and where will be saved) |  |  |  |
|  | FinancialModel  AND  ModelFactory | Now we have to establish our **model based on information from files** |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | RunWrapper |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | RunWrapper  AND  Mersenne Twister |  |  |  |  |
|  |  | Let’s **start simulation** |  |  |  |
|  | FinancialModel | From this moment we know what will be next |  |  |  |
|  | simState | start() method  Called **immediately prior to starting the simulation**, **or in-between simulation runs**.  This gives you a **chance to set up initially, or reset from the last simulation run.**  The **default** version simply **replaces the Schedule with a completely new one (simulation run).**  **---**  **We are still in the main thread** |  |  |  |
|  |  | Random Generator |  |  |  |
|  |  | Calls all the registered Asynchronnous. During this period, any methods which attempt to register things for the schedule will simply be ignored. |  |  |  |
|  |  |  |  |  |  |
|  |  | Returns all the AsynchronousSteppable items presently in the registry. |  |  |  |
|  |  | // reset schedule |  |  |  |
|  |  | **Empties out the schedule and resets it to a pristine state BEFORE\_SIMULATION**, with **steps = 0**. If you're looking for a way **to kill your simulation** from a Steppable, **use SimState.kill()** instead. |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | In the super.start() (where super is SimState) there is also scheadule.reste()  Here (FinancialModel) there is next scheadule.reste() with this difference that this time we do the method on the this.scheadule. reste() so on the FinancialModel |  |  |  |
|  |  |  |  |  |  |
|  | FinMod buildAgents() |  |  |  |  |
|  | ModelFactory |  |  |  |  |
|  | ModelFactory | … |  |  |  |
|  | ModelFactory |  |  |  |  |
|  | ModelFactory | initialize traders and add them to the list |  |  |  |
|  | ModelFactory  ContPlayer | create precisely specified player which in this case is ContPlayer  Each player extends GenericPlayer and generic player extends Steppable |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | GenericPlayer  Setup()  an  a  gent |  |  |  |  |
|  |  |  |  |  |  |
|  | Schedule  a  Player/Agent |  |  |  |  |
|  | Schedule  the  target  reapitingly | **Schedules the event to recur at the specified interval starting at getTime() + interval, and at the provided ordering**.  If this is a valid interval (must be > 0) and event, schedules the event and returns a Stoppable, else returns null.  The recurrence will continue until getTime() >= AFTER\_SIMULATION, the Schedule is cleared out, or the  Stoppable's stop() method is called, whichever happens first.  Note that **calling stop()** on the Stoppable will not only **stop the**  **repeating**, but will <i>**also**</i> **make the Schedule completely forget** (lose  the pointer to) the Steppable scheduled here. This is particularly useful  if you need to make the Schedule NOT serialize certain Steppable objects.  // synchronized so getting the time can be atomic with the subsidiary  // scheduleRepeating function call |  |  |  |
|  |  |  |  |  |  |
|  |  | **Schedules the event to recur at the specified interval starting at the** **provided time, and in the ordering provided**. **If** the getTime() **==** the  provided time, then **the first event is instead scheduled to occur at getTime() + epsilon** (the minimum possible next timestamp).  **If this is a valid time, ordering, interval (must be > 0),** and **event, schedules the event and returns a Stoppable.** The recurrence will continue until getTime() >= AFTER\_SIMULATION, the Schedule is cleared out, or the Stoppable's stop() method is called, whichever happens first.  Note that **calling stop()** on the Stoppable will not only **stop the**  **repeating**, but will <i>**also**</i> **make the Schedule completely forget** (lose  the pointer to) the Steppable scheduled here. This is particularly useful  if you need to make the Schedule NOT serialize certain Steppable objects. |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | this.ordering was = 0. It is being changed here |  |  |  |
|  |  |  |  |  |  |
|  |  | **Handles repeated steps**. This is done by **wrapping the Steppable with a IterativeRepeat object which is itself Steppable**, and **on its step calls its subsidiary Steppable, then reschedules itself.** IterativeRepeat **is stopped by**  **setting its subsidiary to null**, and so the next time it's scheduled it won't reschedule itself (or call the subsidiary). A private class for Schedule.  We've moved it out of being an **inner class of Schedule** (**class inside a class**) and will ultimately make it a separate class in the package. (Java treats is as separate class in the package). |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | **Schedules an item**. You **must synchronize on this.lock before calling this method**. This **allows us to avoid synchronizing twice, and incurring and overhead** |  |  |  |
|  |  |  |  |  |  |
|  | Heap | Adds an element to the heap with the given key.  Created agent is being added to the heap with the given key |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | GenericPlayer |  |  |  |  |
|  | ModelFactory | Manually changed the number of players from 1500 to 3 |  |  |  |
|  |  | Now we add 2 more agents |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | initialize market agent  After adding agents/Players we add market agent |  |  |  |
|  | Market |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | target.myMarket.orderBooks.add((OrderBook) Class.*forName*("model.market.books." + target.optionsMap.get("orderBookClass")).newInstance()); |  |  |  |
|  | ContBook |  |  |  |  |
|  | ContBook |  |  |  |  |
|  | ContBook |  |  |  |  |
|  |  |  |  |  |  |
|  |  | After setting the market we set the world (the boundaries conditions for our market and agents) |  |  |  |
|  |  |  |  |  |  |
|  | ContBook |  |  |  |  |
|  | ContBook |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Now schedule repeating market |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | … |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | scheadule |  |  |  |  |
|  |  |  |  |  |  |
|  | heap |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | RunsWrapper |  |  |  |  |
|  |  |  |  |  |  |
|  |  | **Steps the schedule**, **gathering** and **ordering all** the items to step on the next time step (skipping blank time steps), and then **stepping all of them in the decided order**. Returns FALSE if nothing was stepped – the schedule is exhausted or time has run out. |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Change the time |  |  |  |
|  |  | Return the value of time of the current min element key[0] so the time is being changed form ‘-1’ to 0 |  |  |  |
|  |  |  |  |  |  |
|  |  | The whole process of // grab all of the steppables in the right order. To do this, we employ two Bags:  1. Each iteration of the while-loop, we grab all the steppables of the next ordering, put into the substeps Bag  2. Next we either shuffle or reverse the substeps. 3. Next we add them all to the end of the currentSteps Bag 4. Then we clear the substeps bag, but we don't let them GC yet  5. Last, out of the while-loop, we clear the substeps bag "for real", allowing them to GC  is described in different file |  |  |  |
|  |  |  |  |  |  |
|  |  | Second while loop |  |  |  |
|  |  |  |  |  |  |
|  |  | Third while loop |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Break loop |  |  |  |
|  |  | clear out the substeps for real |  |  |  |
|  |  |  |  |  |  |
|  | bag |  |  |  |  |
|  |  | execute |  |  |  |
|  |  | Loop x5 |  |  |  |
|  |  | … |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Go back to bigger while loop (not jet ‘for loop’)  The while loop is set to true and is set to 50 because the maxTime is set to 50 and in each loop time is incremented by one |  |  |  |
|  |  |  |  |  |  |
|  |  | In the last while loop the situation is changing. Step = 49, time = 48 and we start next (last) loop |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Before calculationafter calculation |  |  |  |
|  |  | 2nd loop |  |  |  |
|  |  | 3rd loop  It will be the last ‘big while loop’ and we will have one more this smaller while loop. Tiem 50 and later after execution stem will be 51. (if on the some of this pictures are ‘not normal’ data it is possible becouse some was modified moanually so the execution go faster) |  |  |  |
|  |  | 4th loop !!! last one  before   after  queue is empty numElem = 0 |  |  |  |
|  |  | So now we have currentKey = 0 and this would be the condition letting us go further  currentKey.time != time  was the condition that let us did last loop so we could clear the queue |  |  |  |
|  |  | execute |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Here we have true and this is why our while loop will like to go again  but our queue is empty and we have numElem=0; so next what we get is: return false which ends while loop ride |  |  |  |
|  |  | Than we go back to our big loop and create completely new environment and loop next 9 more times |  |  |  |
|  | Simstaet  Kill() | Before we go into our for loop 9 more times we have to finish what has been starter (the first for loop) hhw.finish(); | \* Called either at the proper or a premature end to the simulation. If the  \* user quits the program, this function may not be called. It is possible  \* for this method to be called multiple times. If you need to check for  \* this possibility, the easiest way is to set a flag in start() and clear  \* it in the first finish(). |  |  |
|  |  | \* A Steppable on the schedule can call this method to cancel the simulation | \* A Steppable on the schedule can call this method to cancel the  \* simulation. All existing AsynchronousSteppables are stopped, and then the  \* schedule is reset. AsynchronousSteppables, ParallelSequences, and  \* non-main threads should not call this method directly -- it will  \* deadlock. Instead, they may kill the simulation by scheduling a Steppable  \* for the next timestep which calls state.kill(). |  |  |
|  |  | … |  |  |  |
|  |  | … |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | **Empties out** the schedule but **does not reset the time or steps.**  If you're looking for a way to kill your simulation from a Steppable, use SimState.kill() instead.  Note that any agents presently at THIS TIME STEP will STILL be stepped -- including possibly reinserting themselves in the schedule. |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Seals the schedule: after a schedule is sealed, no further Steppables may be scheduled on it |  |  |  |
|  |  |  |  |  |  |
|  | RunWerapper | Now we can go back and create new world |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |