A Multithreaded Software Simulation of An Electrical Circuit

Jenny Y. Huang
Advisor and project supervisor
Prof. Furrukh Khan

The Ohio State University

The system in this report is a “real-time” software simulation of an electrical circuit. The system is based on a layered architecture, and uses the Java event model to eliminate the coupling of lower (generic) layers to the higher (specialized) layers. The system achieves its responsiveness by exploiting the multithreaded environment offered by the Java platform. The system is designed as a collection of reusable Java Beans components. These components can be wired together in an IDE (Integrated Development Environment) to make the final application. The aim is that inexperienced programmers, can drop these beans into the pallet of an IDE, at design time, and set the properties of these components, as well as wire them together, to make the resulting application, without writing code which needs a deeper understanding of the complexities of multithreading and event handling. Customized editors are provided for the assembling of these beans, so that the assembling together of the components in an IDE can be accomplished even by novice programmers. The components in this project are written in 100% Java. JBuilder 3.5 was used for all the code generation, and development of the Beans. Helpful wizards of JBuilder, like the “BeanExpress”, were used to facilitate the development of BeanInfo Classes, and the generation of custom Editors. Rational Rose 98 was used to generate the static class diagrams, and dynamic interaction diagrams.

The system models a circuit which consists of electronic components. These electronic components are sources (current, or voltage), and resistors. Each component of the circuit can be connected to an Alarm. The alarm gets current or voltage value of a component through a current or voltage sensor. Whenever the current or voltage goes beyond the limit set in the alarm, the alarm will react, the clock will stop and the simulation terminates itself. A number of plotters are connected to a number of circuit components to plot current or voltage of the components. Same as alarms, plotters get component current or voltage through current or voltage sensors.

The simulation starts when a clock starts running. Interval Informers, with user defined time intervals, register as listeners of every tick of the clock. Sources, circuit calculators, alarms, plotters etc. are not interested in being informed of every click of the clock, instead they are interested in getting informed periodically, with the period of a “time interval”, specific to that component. Each of these components, therefore, registers itself with an interval informer, and gets informed whenever its specified time interval has passed. When informed by its associated interval informer, each registered component’s event handling function is invoked. This function would generate voltage or current for a source component, calculate the circuit voltages or currents for a circuit calculator component, plot the appropriate values for plotter component, etc.

To demonstrate this architecture we use a circuit that consists of a single current or voltage source with a number of resistances connected in series. The source generates either voltage or
current as a function of time. A circuit calculator, specific to this series circuit, calculates the current going through the circuit.

Some details dealing with multi threading issues in this project.

The clock and each interval informer run on its own thread of execution. When the clock starts running, all the interval informer threads start running. When the clock stops, all the interval informer threads stop and the simulation ends.

Each object of the class "IntervalInformer" does its own event notification of its registered components, in its own “notification” thread of execution which is separate from the thread of the clock. This “notification” thread spends all its time in an infinite loop while waiting to be notified. It runs, or waits, based on the value of a property "notifyType" of the class "IntervalInformer". The value of the “notifyType” property is set by the clock, and also by a thread manager object “ThreadsManager”, whose job is to notify the lower priority threads, once the higher priority threads are done with their work. Between waits, during the running state, invocation of the event handling function of each component registered with the "IntervalInformer" object is done in this “notification” thread. Threads associated with the highest event priority "IntervalInformer" object handle events first while threads associated with lower priority "IntervalInformer" objects wait. Those lower priority threads are notified after all the higher priority threads finish event handling, by the “ThreadsManager” object, that sets the “notifyType” property of the “IntervalInformer” with the next lower priority.

There are two types of “notifyType” properties in this project. One is the notification caused by the tick of the clock, and the other is the notification caused by the “ThreadsManager” telling the lower priority “IntervalInformer” that the higher priority thread is done with its work. "notifyType" is initially set to 1. The tick of the clock changes this property from 1 to 0. The “ThreadsManager” object sets this property from 0 to 1 in a lower priority thread, after making sure that the higher priority threads have all finished their event notification methods.

The clock sleeps for a pre-determined period of time. The amount of this sleep constitutes one unit of “tick” in the simulation. We map each unit of this tick into one “second”. Thus one second is the smallest unit of time in this simulation. In each cycle of the event handling, the clock listener "IntervalInformer" keeps track of the time when the event is being handled.

Threads associated with each "IntervalInformer" can be either “lagable” or “non-lagable”. “Non-lagable” threads are those for which event handling should be complete within the time interval defined in "IntervalInformer". Naturally, a non-lag able "IntervalInformer" thread should determine only once during its specified time period, that its time period is over. This is handled by a counter in the “Interval Informer” which is incremented each time “IntervalInformers” determines that its time period is over, after listening to the clock. At the end of each cycle this counter is set to zero. So if a “non-lagable” IntervalInformer thread sees this counter to be more than one, then it throws a "ClockSleepNotLongEnoughException", the clock will stop and all the "IntervalInformers” threads will stop. This means that the system is too slow to keep up with the current speed of the clock (controlled by how long the clock sleeps between each time tick). A “lagable” IntervalInformer does not check this counter, since it is permitted for this thread to lag behind the rest of the simulation, to do its task, e.g. a printer, printing the results of a simulation, can “take its time” to do its job.
As mentioned above, a class named "ThreadsManager" is used to organize all Clock Listener "IntervalInformer" objects based on their event priorities. All "IntervalInformer" objects registered in the clock are registered in "ThreadsManager" before the clock’s time loop starts. An “ArrayList” container class inside the "ThreadsManager" is used to store all the "IntervalInformer" objects. This list is sorted according to each "IntervalInformer" object's event priority. Each element of the list, corresponding to an integer index, is a “set” of "IntervalInformer" objects, all with the same priority corresponding to the index of the ArrayList.

The clock listener "IntervalInformer" handles event by informing its own listeners which are the sources, plotters, alarms and circuit calculators, etc. Each electronic component, such as a voltage or current source, resistance, etc. does not merely store the latest value (voltage and current), but instead stores a list of its current and previous voltage and current values for every tick of the clock, along with the value of this tick. Plotters and alarms get circuit component values (voltages or currents) from this list for the time they are interested in, by looking into this list associated with each electronic component. This storage of past values is important for “lag able” threads since they are allowed to lag behind the simulation and “take their time” to do their job. So even as new values of voltages and currents arrive in the list kept by each electronic component, the “lag able” components read the older values to do their job, at a pace commensurate with the amount of processing required by their task.
Circuit Simulation Bean Assembly Example

1. Bean Installation

install following beans in the bean palette of the IDE:

Clock.class, IntervalInformer.class, ThreadsManager.class;
WaveAttributes.class, WaveSine.class, WaveCosine.class, VoltageSource.class,
Resistance.class, VoltageSensor.class, CurrentSensor.class;
AlarmBell.class;
PlotterFile.class;
SeriesCircuit.class, SeriesCircuitCalculator.class

2. Bean Instantiation and Property Setting

Select “ThreadsManager” bean from the palette and drop it into the application you are building. In the IDE property editor for this bean, set the bean object name to “manager”.

Select “Clock” bean and drop it into the application. In the IDE property editor, set the bean object name to “clock”, then set the bean properties as follows:

lifeTimeInSeconds: 10
sleepingTime: 1000
threadsManager: manager

Select “IntervalInformer” bean and drop it into the application. In the IDE property editor, set the bean object name to “informer_v1”, then set the bean properties as follows:

secondsInterval: 1.0
eventPriority: high
itsClock: clock
itsName: informer_v1

Select “IntervalInformer” bean and drop it into the application. In the IDE property editor, set the bean object name to “informer_cirCalculator”, then set the bean properties as follows:

secondsInterval: 1.0
eventPriority: medium
itsClock: clock
itsName: informer_cirCalculator

Select “IntervalInformer” bean and drop it into the application. In the IDE property editor, set the bean object name to “informer_alarm”, then set the bean properties as follows:

secondsInterval: 2.0
eventPriority: low
itsClock: clock
itsName: informer_alarm

Select “IntervalInformer” bean and drop it into the application. In the IDE property editor, set the bean object name to “informer_plotter”, then set the bean properties as follows:

secondsInterval: 2.0
eventPriority: lowest
itsClock: clock
itsName: informer_plotter
Select “WaveAttributes” bean and drop it into the application. In the IDE property editor, set the bean object name to “waveAttrib”, then set the bean properties as follows:

- dcValue: 11.8
- frequency: 5.0
- phase: 45.8
- amplitude: 2.0

Select “WaveSine” bean and drop it into the application. In the IDE property editor, set the bean object name to “waveSine”, then set the bean properties as follows:

- itsWaveAttributes: waveAttrib

Select “VoltageSource” bean and drop it into the application. In the IDE property editor, set the bean object name to “v1”, then set the bean properties as follows:

- ItsIntervalInformer: informer_v1
- itsWave: waveSine
- itsName: V1

Select “Resistance” bean and drop it into the application. In the IDE property editor, set the bean object name to “r1”, then set the bean properties as follows:

- resistance: 2.4
- itsName: R1

Select “Resistance” bean and drop it into the application. In the IDE property editor, set the bean object name to “r2”, then set the bean properties as follows:

- resistance: 2.0
- itsName: R2

Select “VoltageSensor” bean and drop it into the application. In the IDE property editor, set the bean object name to “sv_v1”, then set the bean properties as follows:

- itsEcomponent: v1
- itsName: SV_V1

Select “CurrentSensor” bean and drop it into the application. In the IDE property editor, set the bean object name to “sc_v1”, then set the bean properties as follows:

- itsEcomponent: v1
- itsName: SC_V1

Select “VoltageSensor” bean and drop it into the application. In the IDE property editor, set the bean object name to “sv_r1”, then set the bean properties as follows:

- itsEcomponent: r1
- itsName: SV_R1

Select “CurrentSensor” bean and drop it into the application. In the IDE property editor, set the bean object name to “sc_r1”, then set the bean properties as follows:

- itsEcomponent: r1
- itsName: SC_R1

Select “VoltageSensor” bean and drop it into the application. In the IDE property editor, set the bean object name to “sv_r2”, then set the bean properties as follows:

- itsEcomponent: r2
- itsName: SV_R2
Select “CurrentSensor” bean and drop it into the application. In the IDE property editor, set the bean object name to “sc_r1”, then set the bean properties as follows:
itsEcomponent: r2 
itsName: SC_R2

Select “AlarmBell” bean and drop it into the application. In the IDE property editor, set the bean object name to “alarmB_sc_r1”, then set the bean properties as follows:
itsIntervalInformer: informer_alarm 
itsSensor: sc_r1 
valueLimit: 5.5

Select “SeriesCircuit” bean and drop it into the application. In the IDE property editor, set the bean object name to “sCircuit”, then set the bean properties in the custom editor as follows:
ItsEcomCollect: v1, r1, r2 (the IDE invokes our custom editor for this property).

Select “SeriesCircuitCalculator” bean and drop it into the application. In the IDE property editor, set the bean object name to “sCircuitCalculator”, then set the bean properties as follows:
itsIntervalInformer: informer_cirCalculator
itsCircuit: sCircuit

Select “PlotterFile” bean and drop it into the application. In the IDE property editor, set the bean object name to “plotterFile”, then set the bean properties as follows:
itsIntervalInformer: informer_plotter
itsSensorCollect: sv_v1, sc_v1, sv_r1, sc_r1, sv_r2, sc_r2 (the IDE invokes our custom editor for this property).

The program generated by the IDE reflecting step 2 above is attached.

3. The bean assembler needs to input the following lines of code to start the simulation.

```
Application app = new Application();  // This line generated by the IDE
app.clock.start();  // This line typed in by bean assembler
app.clock.join();  // this line typed in by bean assembler
```

4. Use IDE to run the program and the result is printed in a file called “CircuitOutputFile.doc” as set in bean “plotterFile”. The output file is attached.
public class Application {
  Clock clock = new Clock();
  IntervalInformer informer_v1 = new IntervalInformer();
  IntervalInformer informer_alarm = new IntervalInformer();
  IntervalInformer informer_cirCalculator = new IntervalInformer();
  IntervalInformer informer_plotter = new IntervalInformer();
  WaveAttributes waveAttrib = new WaveAttributes();
  WaveSine waveSine = new WaveSine();
  VoltageSource v1 = new VoltageSource();
  Resistance r1 = new Resistance();
  Resistance r2 = new Resistance();
  VoltageSensor sv_v1 = new VoltageSensor();
  CurrentSensor sc_v1 = new CurrentSensor();
  VoltageSensor sv_r1 = new VoltageSensor();
  CurrentSensor sc_r1 = new CurrentSensor();
  VoltageSensor sv_r2 = new VoltageSensor();
  CurrentSensor sc_r2 = new CurrentSensor();
  AlarmBell alarmB_sc_r1 = new AlarmBell();
  SeriesCircuitCalculator sCirCalculator = new SeriesCircuitCalculator();
  SeriesCircuit sCircuit = new SeriesCircuit();
  ThreadsManager manager = new ThreadsManager();
  PlotterFile plotterFile = new PlotterFile();

  public Application() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

  private void jbInit() throws Exception {
    sCircuit.setItsEcomCollect(new E_ComponentCollection(new E_Component[] {v1, r1, r2}, new String[] {"v1", "r1", "r2"}));
    plotterFile.setItsIntervalInformer(informer_plotter);
    plotterFile.setItsSensorCollect(new SensorCollection(new Sensor[] {sv_v1, sc_v1, sv_r1, sc_r1, sv_r2, sc_r2}, new String[] {"sv_v1", "sc_v1", "sv_r1", "sc_r1", "sv_r2", "sc_r2"}));
    sCirCalculator.setItsCircuit(sCircuit);
    sCirCalculator.setItsIntervalInformer(informer_cirCalculator);
    alarmB_sc_r1.setValueLimit(5.5);
    alarmB_sc_r1.setItsSensor(sc_r1);
    alarmB_sc_r1.setItsIntervalInformer(informer_alarm);
    sc_r2.setItsEcomponent(r2);
    sc_r2.setItsName("SC_R2");
    sv_r2.setItsEcomponent(r2);
    sv_r2.setItsName("SV_R2");
    sc_r1.setItsEcomponent(r1);
    sc_r1.setItsName("SC_R1");
    sv_r1.setItsEcomponent(r1);
  }
}
sv_r1.setItsName("SV_R1");
sc_v1.setItsEcomponent(v1);
sc_v1.setItsName("SC_V1");
sv_v1.setItsEcomponent(v1);
sv_v1.setItsName("SV_V1");
r2.setResistance(2.0);
r2.setItsName("R2");
r1.setResistance(2.4);
r1.setItsName("R1");
v1.setItsIntervalInformer(informer_v1);
v1.setItsWave(waveSine);
v1.setItsName("V1");
waveSine.setItsWaveAttributes(waveAttrib);
waveAttrib.setAmplitude(2.0);
waveAttrib.setFrequency(5.0);
waveAttrib.setPhase(45.8);
waveAttrib.setDcValue(11.8);
informer_plotter.setSecondsInterval(2.0);
informer_plotter.setItsClock(clock);
informer_plotter.setEventPriority("lowest");
informer_plotter.setItsName("informer_plotter");
informer_plotter.setLagable(true);
informer_cirCalculator.setSecondsInterval(1.0);
informer_cirCalculator.setItsClock(clock);
informer_cirCalculator.setEventPriority("medium");
informer_cirCalculator.setItsName("informer_cirCalculator");
informer_alarm.setSecondsInterval(2.0);
informer_alarm.setItsClock(clock);
informer_alarm.setItsName("informer_alarm");
informer_v1.setSecondsInterval(1.0);
informer_v1.setItsClock(clock);
informer_v1.setEventPriority("high");
informer_v1.setItsName("informer_v1");
clock.setLifeTimeInSeconds(10);
clock.setSleepingTime(1000);
clock.setThreadsManager(manager);
## Circuit Simulation Bean Assembly Example

**Output File**

Printed at: 6:17:38 PM, Sep 10, 2000

<table>
<thead>
<tr>
<th>Time</th>
<th>SensorType</th>
<th>SensorName</th>
<th>E_CompType</th>
<th>E_CompName</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2s</td>
<td>Voltage Sensor</td>
<td>SV_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>10.438673252785744</td>
</tr>
<tr>
<td>2s</td>
<td>Current Sensor</td>
<td>SC_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>2.372425739269487</td>
</tr>
<tr>
<td>2s</td>
<td>Voltage Sensor</td>
<td>SV_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>5.693821774246769</td>
</tr>
<tr>
<td>2s</td>
<td>Current Sensor</td>
<td>SC_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>2.372425739269487</td>
</tr>
<tr>
<td>2s</td>
<td>Voltage Sensor</td>
<td>SV_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>4.744851478538974</td>
</tr>
<tr>
<td>2s</td>
<td>Current Sensor</td>
<td>SC_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>2.372425739269487</td>
</tr>
<tr>
<td>4s</td>
<td>Voltage Sensor</td>
<td>SV_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>12.145154784891638</td>
</tr>
<tr>
<td>4s</td>
<td>Current Sensor</td>
<td>SC_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>2.760262451117355</td>
</tr>
<tr>
<td>4s</td>
<td>Voltage Sensor</td>
<td>SV_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>6.624629882668165</td>
</tr>
<tr>
<td>4s</td>
<td>Current Sensor</td>
<td>SC_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>2.760262451117355</td>
</tr>
<tr>
<td>4s</td>
<td>Voltage Sensor</td>
<td>SV_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>5.52052490223471</td>
</tr>
<tr>
<td>4s</td>
<td>Current Sensor</td>
<td>SC_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>2.760262451117355</td>
</tr>
<tr>
<td>6s</td>
<td>Voltage Sensor</td>
<td>SV_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>12.582107640960098</td>
</tr>
<tr>
<td>6s</td>
<td>Current Sensor</td>
<td>SC_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>2.859569918400022</td>
</tr>
<tr>
<td>6s</td>
<td>Voltage Sensor</td>
<td>SV_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>6.862967804160053</td>
</tr>
<tr>
<td>6s</td>
<td>Current Sensor</td>
<td>SC_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>2.859569918400022</td>
</tr>
<tr>
<td>6s</td>
<td>Voltage Sensor</td>
<td>SV_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>5.71939836800044</td>
</tr>
<tr>
<td>6s</td>
<td>Current Sensor</td>
<td>SC_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>2.859569918400022</td>
</tr>
<tr>
<td>8s</td>
<td>Voltage Sensor</td>
<td>SV_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>10.142356706702833</td>
</tr>
<tr>
<td>8s</td>
<td>Current Sensor</td>
<td>SC_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>2.305081069705189</td>
</tr>
<tr>
<td>8s</td>
<td>Voltage Sensor</td>
<td>SV_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>5.532194567292453</td>
</tr>
<tr>
<td>8s</td>
<td>Current Sensor</td>
<td>SC_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>2.305081069705189</td>
</tr>
<tr>
<td>8s</td>
<td>Voltage Sensor</td>
<td>SV_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>4.610162139410378</td>
</tr>
<tr>
<td>8s</td>
<td>Current Sensor</td>
<td>SC_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>2.305081069705189</td>
</tr>
<tr>
<td>10s</td>
<td>Voltage Sensor</td>
<td>SV_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>13.799654944580265</td>
</tr>
<tr>
<td>10s</td>
<td>Current Sensor</td>
<td>SC_V1</td>
<td>Voltage Source</td>
<td>V1</td>
<td>3.136285214677333</td>
</tr>
<tr>
<td>10s</td>
<td>Voltage Sensor</td>
<td>SV_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>7.527084152255985</td>
</tr>
<tr>
<td>10s</td>
<td>Current Sensor</td>
<td>SC_R1</td>
<td>Resistance R1</td>
<td>R1</td>
<td>3.136285214677333</td>
</tr>
<tr>
<td>10s</td>
<td>Voltage Sensor</td>
<td>SV_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>6.272570429354666</td>
</tr>
<tr>
<td>10s</td>
<td>Current Sensor</td>
<td>SC_R2</td>
<td>Resistance R2</td>
<td>R2</td>
<td>3.136285214677333</td>
</tr>
</tbody>
</table>
Clock

sleepingTime: int

IntervalInformer

interval: int = 1
notifyType: int = 1

IntervalInformer

interval: int = 2
notifyType: int = 1

Low Event Priority

High Event Priority

Note:

............... Thread Waiting

.......... Thread Running

Time = 0

Time = 1

Time = 2

Time = 3

Time = 4

Figure 1: Threads Wait and Notification Sequence
Figure 2: “ClockSleepNotLongEnough” Exception Handling
A Multi-Threaded Software Simulation of An Electrical Circuit
1. Element of ArrayList "clockListenersArray" is a set of ClockListeners w/ the same priority. The list is ordered based on the listener’s priority and the list’s capacity is hardwired to match the no. of priorities defined in class EventPriority.

2. ArrayList "cycleCompletedThreadsCount" is to keep track the number of finished threads which have the same priority. When the last thread finishes its cycle, the next priority threads will be notified.

3. Function init() is to initially set up the capacity and null elements for above two lists to match the no. of priorities defined in class EventPriority.

4. Function initThreads() is to create and start a thread to each listener. Listener’s function setItsThread() is called here.

Users are allowed to set all attributes except “totalIntervalInSeconds”、“totalIntervalInMinutes” is updated every time when ‘hourInterval’, ‘minuteInterval’ and ‘secondInterval’ are set.

4. Function initThreads() is to create and start a thread to each listener.

above two lists to match the no. of priorities defined in class EventPriority.

3. Function init() is to initially set up the capacity and null elements for finished threads which have the same priority. When the last thread finishes
public void intervalPassed(IntervalEvent e) throws IntervalException {
    IntervalInformer informer = e.getSource();
    int time = find the tick time from "e";
    double interval = find the interval from "e";

    double value;
    if(!informer.isLagable()) {
        value = itsSensor.getEcomponentValue(time);
    }
    else {
        int t = (int) (alarmCounter+1) * interval;
        if(itsSensor/valueExist(t)) {
            value = itsSensor.getEcomponentValue(t);
            ++alarmCounter;
        }
    }

    if(overLimitAlarm) {
        if(value > valueLimit) ringAlarm()
    }
    else {
        if(value < valueLimit) ringAlarm()
    }
}

Package: alarm
public void plot(IntervalEvent e) {
    // Getting value sensed by its sensors similar to Alarm, refer to Alarm diagram, function intervalPassed(IntervalEvent e)
}

public void intervalPassed(IntervalEvent e) {
    plot(e);
}

<<Abstract>>
Plotter
plotCounter : int = 0
ItsSensorCollect : SensorCollection

<<Abstract>> plot(e : IntervalEvent) : void
intervalPassed() : void

<<Interface>>
IntervalListener
(from clock)
intervalPassed()

SensorCollection
(from sensorCollectionEditor)
sensorIndexed : Sensor[]
refs : String[]
SensorCollection(sensors : Sensor[])
<<Interface>>
IntervalListener
(from clock)

intervalPassed() 0..*

<<Interface>>
IntervalInformer
(from clock)

intervalPassed(intervalEvent : IntervalEvent)

<<Interface>>
IntervalListener
(from clock)

intervalPassed() 0..*

<<Interface>>
Circuit

current() : double
getItsEcomCollect() : E_ComponentCollection

SeriesCircuit
itsEcompCollect : E_ComponentCollection

ParalleCircuit
itsEcompCollect : E_ComponentCollection

E_ComponentCollection
from ecomCollectionEditor

ecomIndexed : E_Component[]
refs : String[]
E_ComponentCollection(eoms : E_Component[])
With bean editor, we can assemble beans as following:

```java
E_Component ecom1 = new E_Component();
E_Component ecom2 = new E_Component();
Circuit circuit = new Circuit();

public void jbInit() {
    circuit.setItsEcompCollect(new E_ComponentCollection(new E_Component[] {s1, s2}));
}
```

```
Package: ecomCollectionEditor
```
With bean editor, we can assemble beans as following:

```java
Sensor s1 = new CurrentSensor();
Sensor s2 = new VoltageSensor();
Plotter plotter = new Plotter();;

public void jbInit() {
    plotter.setItsSensorCollect(new SensorCollection(new Sensor[]{s1, s2}));
}
```
Sequence 1: Threading Initialization and Clock Event Handling

Clock sleeps for a pre-determined period of time. The Threads Manager's method `initThreads()` shall be complete within the period of clock sleeping time, otherwise the incompletion of threads initialization will cause unstable program execution and will cause a later exception throwing as described below.

Clock sleeps for a pre-determined period of time. Non-lagable threads shall complete event handling within the time interval defined in "IntervalInformer". Otherwise a "ClockSleepNotLongEnough" exception will be thrown, the clock will stop and all the "IntervalInformer" threads will stop. See sequence 3 for threads termination.
**Sequence 2: IntervalInformer Event Handling and Event Priority Sequence**

**High Priority**

- `fireIntervalEvent (IntervalEvent)`
- `setNotifyType( 1 )`
- `Loop through all Interval Listeners to start their event handling in run( ) function`
- `checkClockSleepingTime( )`
- `addCycleCompletedThreadsCount( priority)`
- `threadsCycleCompleted(ClockListener)`
- `setNotifyType( 1 )`
- `notifyAll( )`
- `Loop through all next priority Interval Informers`
- `wait( )`

**Low Priority**

An "ArrayList" container class inside the "Threads Manager" stores Integers and each Integer corresponds the number of "IntervalInformer" threads which have finished event handling. This list is sorted according to each threads' event priority. When a thread finishes its event handling, it increments its associated integer in the list. When the integer equals the total number of threads registered in the Threads Manager, it means all threads with the same priority have finished event handling and it is time for the Threads Manager to notify the lower priority threads.

Thread checks the clock sleeping time after every major method execution to make sure that the thread execution finishes before the clock starts the next looping, otherwise an exception will be thrown.
Sequence 3: Threading Termination

Clock gets out of loop

setShouldRun(false)

StopThreads()

Loop through all Interval Informers.

In run() function

This thread of execution throws an exception and gets out of its while loop in function run().

setShouldRun(false)

notifyAll()

To notify associated thread to get out of its while loop in function run().

File: D:\JBuilder35Projects\CircuitSimulatorUsingThreads\myproject\circuit_simulator\Doc\CircuitSimulator.mdl 10:48:12 PM Sunday, September 10, 2000  Sequence Diagram: Logical View / Threading Termination Page 1