In this project, you will be provided: (1) the Atlas middleware, (2) the Atlas sensor/actuator emulator, and (3) an Atlas Reactivity Engine (RE). You will use these components to design, develop and assess an RE Rule Optimizer. The goal of the optimization is to minimize communications between the applications and the sensors (messages), power consumption in the sensor network (energy consumed by the sensor platforms), and computation at the Atlas middleware hub.

The overall architecture of the project is shown in the Figure to the right. Provided components are in pink. You will develop the Optimizer from scratch and are expected to make any minor changes / additions to the Atlas RE as you see needed. Source code for the Atlas RE will be provided. A command interpreter will also be provided to allow you to create rules and enter a variety of commands.

You will first start the Atlas middleware within the OSGi environment. Then start up preconfigured emulators for the desired sensors that you will use. Then the Atlas RE should be started. Next the optimizer should be started before the command interpreter is launched.

By now, you should have already downloaded OSGi framework, installed Atlas middleware, and toyed with an Atlas Demo app.

The Atlas Emulator will allow you to emulate the Atlas platforms connected to a variety of sensors. It eliminates the requirement of having actual hardware nodes to work on the project. Among other things, the emulator will be covered during the Hands-on session. The emulator and manual can be downloaded from: http://www.icta.ufl.edu/atlas/emulator/

This is a group project (groups of 4). All developments must be done in Java.
Overview of the RE Engine

The engine follows an Event/Condition/Action paradigm in which a set of rules are defined by the programmer, and registered, maintained and appropriately triggered by the RE engine.

An RE command line interface for the programmer is available as an OSGi service. The interface facilitates input of commands from the programmer and output of prompts and rule trigger notifications to the programmers. The interface bundle currently communicates directly with the RE engine.

The RE engine is also a service bundle that registers new rules and manages all rules combined. It serves as storage and an evaluator of rules and their components. It also serves as a trigger mechanism for actions of “applicable” rules. The RE engine communicates with the Atlas middleware services on the one hand, and with the command line interface service on the other hand.

The RE engine implements the concepts of Events, Conditions, Actions, Rules and Commands. These concepts are defined as follows:

I. EVENTS

An event is a logical expression over available sensor values. Formally, an event, E, is defined as follows:

$$E = \text{Sensor}(\text{value}) \mid \text{Sensor}[a,b]$$

$$E = E + E \mid E*E \mid E*\text{seconds}^*E$$

Where value is a real number indicating the desired value of Sensor, and [a,b] is a desired range of Sensor values.

The operator + is intended as logical OR. The operator *, which has higher precedence than +, is intended as logical AND.

The operator *seconds* is a modified AND operator in which the concurrence requirement is relaxed. The right operand event is allowed to take place up to “seconds” number of seconds after the left operand event has occurred (is evaluated to true).

The same event may be part of several more complex events. An event could also belong to more than one rule.
II. CONDITIONS

Conditions are intended as logical expressions of variables local to the RE engine. In the provided RE version, all conditions are simple expressions consisting of one variable. The programmer is allowed to define as many conditions as needed. Programmer is also allowed to set and rest any conditions at any time from the command line interface. A condition, C, is therefore defined as:

\[ C = \{ \text{TRUE} | \text{FALSE} \} \]

The same condition may participate in more than one rule.

III. ACTION

An action is an invocation of one or more methods belonging to one or several application or device service bundles. The invocation logic is simple. It is a sequence of invocation of the methods. An action, A, is therefore defined as follows:

\[ A = \text{Service.method}; \]
\[ A = (A; A) \]

IV. RULE

A rule is a specific configuration of a predefined event E, condition C, and action A, and is therefore defined as:

\[ \text{RULE} = \\langle E, C, A \rangle \]

V. COMMANDS

The RE implements the following commands:

Command = LIST \{ event | condition | action | rule \}
Command = BASIC \{ event | action \}
Command = DEFINE \{ event event_name = E |
condition condition_name = C |
action action_name = A |
rule rule_name = RULE
\}
Command = SET condition_name = \{ TRUE | FALSE \}
Command = RUN
Command = STOP
Command = LOAD <file name>
The LIST command allows the programmer to “browse” and inspect available, user-defined events, conditions, actions and rules. The BASIC command allows the programmer to view basic events and actions that correspond to available service bundles in the Atlas middleware (not user-defined).

The intended use of the RE engine is for the programmer to view basic events and actions; then define conditions, and event; then form rules.

The RUN command is intended to run the engine, meaning to start evaluating rules and trigger actions if applicable. The only thing the programmer is able to change in RUN mode is resetting the value of conditions. The programmer may change any event or rule after the engine is stopped using the STOP command.

The LOAD command is intended to help the programmer save time in developing and testing an application. It allows the programmer to load a set of definitions for events, condition and rules. LOAD erases any available definitions prior to loading the new definition from <filename>

NOTATIONS

All lexical tokens and delimiters are colored in burgundy. The set of delimiters consists of:

\[ ( ) [ ] ; , \]

The set of tokens consists of:

<table>
<thead>
<tr>
<th>TRUE</th>
<th>FALSE</th>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>DEFINE</td>
<td>LIST</td>
<td>BASIC</td>
</tr>
<tr>
<td>RUN</td>
<td>STOP</td>
<td>LOAD</td>
<td>event</td>
</tr>
<tr>
<td>condition</td>
<td>action</td>
<td>rule</td>
<td>=</td>
</tr>
</tbody>
</table>

The RE Optimizer

How often should rules be evaluated? Continuously? Periodically? How can an RE engine scale in face of an increasing number of sensors, events, actions and/or rules? What is the computing and communication overhead of running the RE engine? And what are the power drain implications? These are all fair questions that motivate for Project II specific assignment – create a Rule Optimization layer into the Reactivity Engine, similar to the Query Optimization layer in the context of Database Query Processing.

The optimizer follows a simple principle: The Minimum Amount of Work principle. The optimizer should generate intelligent rule evaluation plans that generate the minimum amount of work for the engine and beneath (the sensor nodes). It
should use every opportunity possible to minimize the total work to be done by the engine and the sensor platforms. This includes sensing (by sensor platform), computing (by sensor platform) and communication (involving the sensor platform and the engine).

Your ideas could exploit and specialize on any number of the following general frameworks:

- **Push-Pull Envelop.** The engine could switch between push and pull modes for each sensor at any given point of time. That is, at any given time, t, the engine could seek the optimal push-pull envelop – a configuration of which sensors should participate, and if so, in which mode (push or pull, and if latter, at which frequency).
- **Caching.** Values of basic events known to the engine could be cached and reused for some period of time. Values could be brought via push or pull.
- **Short Cuts.** Some complex events may not need to be evaluated entirely given the logical nature of the event grammar.
- **Application Characteristics.** Analyzed applications could render knowledge that can help in optimizing the engine. This could be the case for fixed, predictable (also known as closed loop) applications, as well as dynamic, unpredictable (also known as open ended) applications. Such knowledge is important because it can help understand what is the minimum “sentience” needed to make decisions (sentience efficiency).

**PROJECT II:**

The following are the steps you need to go through to successfully complete this project.

1. **Understand all provided components.** This is easy but you must have all the components installed, read relevant manuals and simply use and play around with the components.
2. **Study source code of RE Engine.** This is crucial because you must understand “other people code”. This is not difficult but crucial and will take some time, so start as soon as possible.
3. **Think about and create your optimization algorithms.** Design your strategies and algorithms first. Analyze your design and create scenarios to get a sense of validation before you move on to implementation.
4. **Develop the Optimizer.** The command interpreter interface needs to connect to the Optimizer as well as the engine. The main implementation should result in an evaluation plan and associated schedule to interact with the engine. This is the main objective of the optimizer. You should add code that will assist you in assessing the scalability of your engine in presence of the optimizer.
5. **Assess the success/failure of your Optimizer.** You should design experiments to test the effectiveness of your optimizer and hence the engine scalability. Specific instructions will be provided to streamline the experimentation but you will be allowed to conduct your own experiments using test cases of your choosing.

**DELEVERABLES**

You will deliver source code with detailed documentation and about 5-10 pages report describing the following:

2. Your optimization ideas and strategies.
3. Formal description of your optimization algorithm
4. Your experiment results
5. Any additional features implemented and the rationale
6. Conclusions

All deliverable files should be emailed as a ZIP file to the TA by 11:59am Monday, November 30, 2010. There will be no extensions possible given the presentation schedule.