Objective:
   How do we achieve easy smart environment programming?

Idea:
   What if implementing new service is just as easy as configuring existing modules?

How does it work?
   1. Break down services into composition of smaller, atomic services.
   2. Analyze the structure of these atomic services, and distill the common paradigms behind these basic services.
   3. Create basic utility bundles for enabling each of the basic paradigms. The previous presentation identified four paradigms from the sample services in the Gator Tech House article.
      a. Trigger – Action paradigm
      b. Target tracking paradigm
      c. Inventory paradigm
      d. Profile initiated paradigm
   4. Tweak these basic bundles so they can be flexible to some degree to be useful for a large portion of normal services.
   5. The flexibility of this approach comes from 3 places:
      a. For simple, straightforward services that can exactly fit into one of the basic paradigms. The flexibility comes from the defining the configuration for the service, for instance, what are the sensors associated with this service, which actuators, the signal-response mapping, etc.
      b. For slightly more complex services that can be breakdown into cascading composition of simpler services, or recursively hierarchical stacked services, this approach allows user to specify how they would like to cascade/stack the basic services/bundles together to for a new service.
      c. For novel or extremely complex service control, there is always the possibility of writing new controller module. These new modules can also be implemented in a flexible way so future services with similar paradigm can reuse these new controller modules.
   6. One of the basic assumptions is that the vendors providing sensors/actuators would take care of all implementation of functionalities, following a set of standard API, and create a list of possible readings/signals/actions associated with their unit.
Demonstration:

Simply creates two configuration files allows us to create two services in seconds, provided that the sensor and actuator bundles were already in place.

In this demo, the two configuration files are for ‘smart mailbox’ service and ‘smartwave’ service. The first one display a text message when new mail arrives in the mailbox and the second one read RFID tag on microwavable dinners, and decide the setting of microwave for cooking/defrosting them.
<smart mail box service>
  <sensor>mailbox</sensor>
  <actuator>notifier</actuator>
  <sensor signal>
    <signal type number>000</signal type number>
    <action mapping>
      <signal value>true</signal value>
      <actuator action>
        <actuator action number>100</actuator action number>
        <actuator action parameter>You've got mail.</actuator action parameter>
        <actuator action>On</actuator action>
      </actuator action>
    </action mapping>
  </sensor signal>
</smart mail box service>

<smartwave service>
  <sensor>microwaverfid</sensor>
  <actuator>microwave</actuator>
  <sensor signal>
    <signal type number>000</signal type number>
    <action mapping>
      <signal value>21</signal value>
      <actuator action>
        <actuator action number>101</actuator action number>
        <actuator action parameter>1.30 10</actuator action parameter>
        <actuator action>On</actuator action>
      </actuator action>
    </action mapping>
    <action mapping>
      <signal value>221</signal value>
      <actuator action>
        <actuator action number>102</actuator action number>
        <actuator action parameter>2.30 6</actuator action parameter>
        <actuator action>On</actuator action>
      </actuator action>
    </action mapping>
  </sensor signal>
</smartwave service>
ActuatorInterface:

/**
 * This interface method allows controller to issue command to actuator asynchronously.
 * actionItem is used to specify which functionality of actuator is to be triggered, parameter
 * is used to supply the parameters needed for triggering the functionality, and action
 * is a boolean that either turns on or off the specified functionality.
 */
public void issueCommand(int actionItem, Object parameter, boolean action);

/**
 * This interface method allows controller to issue command to actuator expecting immediate
 * effects. actionItem is used to specify which functionality of actuator is to be
 * triggered, parameter is used to supply the parameters needed for triggering the
 * functionality, and action is a boolean that either turns on or off the specified functionality.
 */
public void activate(int actionItem, Object parameter, boolean action);

/**
 * This interface method allows controller to access current status of a particular
 * functionality provided by the actuator. The return value is either on or off.
 */
public boolean getStatus(int actionItem);

/**
 * This interface method allows the access to the list of functionalities
 * implemented by the actuator. Probably used at the time of service composition.
 */
public Vector getFunctionalityList();
SensorInterface

/**
 * This interface method allows other modules to retrieve the reading of the sensor
 * using poll model.
 */
public Object retrieveReading();

/**
 * This interface method allows other modules to register for proactive pushing of
 * readings for further operations.
 */
public void register(ActuatorInterface actuator, Vector parameters);

/**
 * This interface method allows other modules to change the frequency, granularity, etc
 * or other parameter of proactive sensor update.
 */
public void setRetreivingParameters(ActuatorInterface actuator, Vector parameters);

/**
 * This interface method allows other modules to remove themselves from registry so
 * they will no longer receive proactively pushed readings.
 */
public void unregister(ActuatorInterface actuator);

/**
 * This interface method allows the access to the list of possible signal types
 * implemented by the sensor. Probably used at the time of service composition.
 */
public Vector getSignalTypeList();
000  Pushing Data to Registered Service
100  Text Notification
101  Microwave Cooking
102  Microwave Defrosting
Tracking paradigm:

Sensor Reading should track Profile Target via Actuator

Example:

Climate Control Service

Paradigm: Tracking
Description: Sensor(Temperature) tracks Profile(Temperature) via Actuator(Temperature)
Restriction: ContextManager(Temperature)
Specialized controller: None
Push/Poll model: Poll
Frequency: 3 seconds
Record: current Actuator(Temperature) state

Sensor(Temperature)
  Domain: All
  Preprocess: Average
  Scale: Fahrenheit

Profile(Temperature)
  Default Upper Bound: 70 (or give a default subject name)
  Default Lower Bound: 66 (or give a default subject name)
  Scale: Fahrenheit
  Subject: QUERY[Smart Room Status Service(Occupant)]

Actuator(Temperature)
  Domain: All
  Selection: None

Output:
  Command for Actuator, ON/OFF

Continuous Tracking paradigm (poll model)
1. Service controller setup targets from profile
2. Service controller gets hold of all sensors of interest
3. Service controller sets up poll frequency and other parameters
4. Service controller polls the readings from sensors
5. Readings are forwarded to preprocessor (possibly context classification)
6. Runs decision making algorithm
7. Issues command to actuator.

* A list of entries in SRS needs to be selectable
Smart Room Status Service

Paradigm: Inventory
Description: Register Sensor(Person), Sensor(Appliance), …
Restriction: ContextManager(Person), ContextManager(Appliance), …
Specialized controller: None
Inventory file: Smart Room
Mapping rule: Person <-> Occupant, …
Smart Mailbox

Paradigm: Trigger – Pull
Description: Sensor(Mailbox) triggers Actuator(Notification)
Restriction: None
Specialized controller: None

Sensor(Mailbox)

Actuator(Notification)
  Domain: All
  Selection: 1. Audio 2. Video

Output:
  Message number, ON/OFF

Trigger – Pull paradigm
1. Service controller register itself to sensors of interest
2. After trigger arrives,
   Service controller finds actuators of interest
3. Send command to Actuator

* Can specify whether to wait for acknowledgement from actuator
To Do:
1. Cascading service / Hierarchical service
2. Definition of sensor / actuator profile
3. ** Must haves and prefers
Smart Blind – User Request

Paradigm: Trigger – Push
Description: Sensor(User Request) triggers Actuator(Blind)
Restriction: None
Specialized controller: None

Sensor(User Request)

Actuator(Blind)

Output:
  Command for Actuator, ON/OFF

Trigger – Push paradigm
1. Once a particular user request is activated, find service controller/actuator by using user request item as search criteria
2. The found controller/actuator is then put in local registry
3. Proceed as in Trigger – Pull paradigm.

? Considering the 3-value output than 2-value ON/OFF, use ON/OFF for now