User Guide for KCDCC AW Agent Package

Greg Smith and Mary Lou Maher
Key Centre of Design Computing and Cognition
University of Sydney

November 6, 2002

Contents

1 Overview

This document describes the KCDCC AW Agent Package using java sensors and effectors and the Jess language for coding agent reasoning as rules.

The aim of the package is a flexible, object oriented framework in which agents become the basis for all elements of an AW world. By using agent models we aim to design intelligent, interactive virtual worlds that exhibit rational behaviours. It is based on a set of abstract classes that form the generic architecture for constructing a society of agents. Central to the framework are the `Society` and the `Agent` (Figure 1).

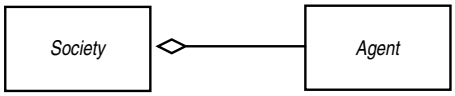

Figure 1: A `Society` is an aggregation of `Agents`.

A Society is an aggregation of Agents that share a common connection with a virtual world. Normally these Agents would share some ontological connection, such as a room agent plus a set of wall agents that collectively comprise a virtual conference room. The Society manages computational resources, such as the connection to the virtual world, on behalf of the Agents.

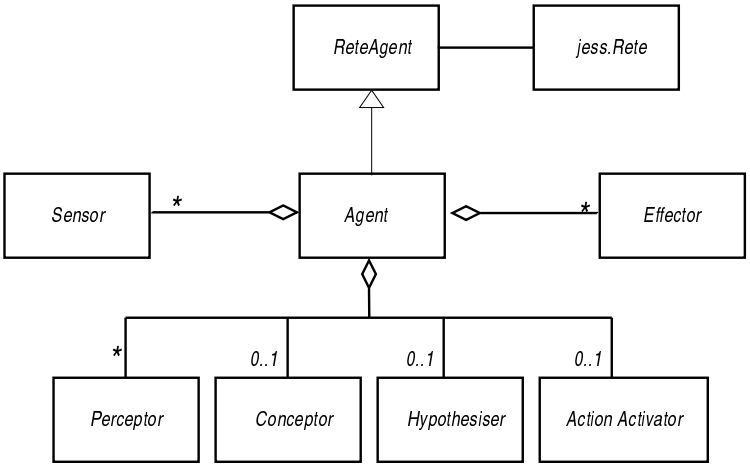

Figure 2: Components comprising an `Agent` or a `ReteAgent`.

Each entity capable of reflexive, reactive or reflective behaviours is modeled as an agent. The Agent has these generic behaviours and an optional 3D representation, and can both dynamically change the 3D representation plus produce non-visual behaviours.

The procedure to insert a 3D object into Active Worlds is to copy an existing object, move it to a desired location, and edit a dialog box to specify its properties. The procedure to create an agent in the world is to configure the agent as a set of sensors and a rule base. Our agents are configured using an XML file that is loaded via a validating parser. This is instead of being statically linked at compile time. This flexibility will eventually provide for the reconfiguration of Agents running in a world without having to recompile and restart the server.

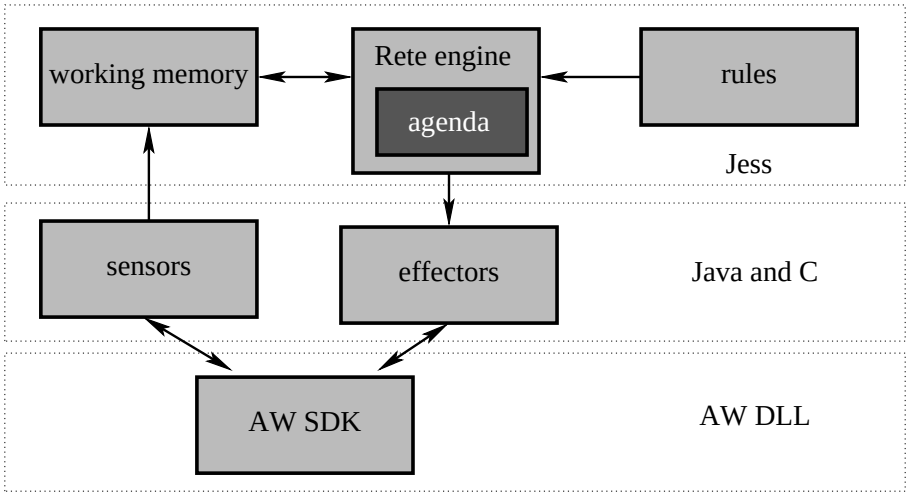

Figure 3: Architecture of `ReteAgent`.

`Sensors` are java objects that sense an agent's environment and `Effectors` act on that environment. These employ a Java Native Interface (JNI) to the Active Worlds (AW) SDK (Figure 4). An effector is a function call from the right-hand side of a rule. `ReteAgent` is an implementation of an agent that uses Jess for everything except sensors and effectors. A `Society` can contain one or more agents that are instances of `ReteAgent`). The creator of a `ReteAgent` configures the agent by specifiying a set of `Sensors`. Sensor data is stored in the Jess working memory. Messages from other agents are also recorded in the Jess working memory by the sensors. That is, each time new sense-data is received a new fact (actually, a java bean) is asserted into Jess working memory. See Sections 4 and 5.

`Effector` and `Sensor` classes are written in java. AW sensors and effectors additionally employ a java native interface to the AW SDK. Developers can also write their own sensors and/or effectors by extending the classes in the `kcdcc.awa.base` java package.

2 Files included in this release

The following files comprise the package. The files need to be saved into one directory, called demos2 by default. The jar files contain compiled java[1]. You do not need to unzip

[1] A jar file is a zip format file with an extra manifest added for use by the java VM.

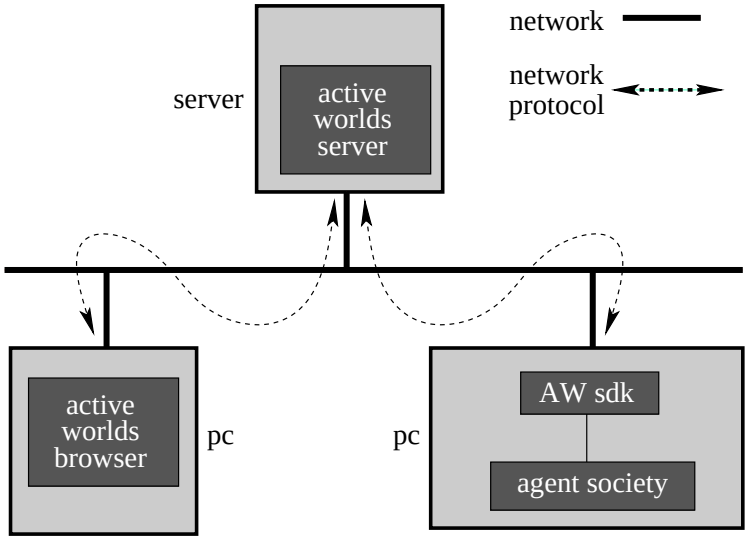

Figure 4: A `Society` communicates with the AW server via the AW SDK.

these.

- Files needed by all agents (You will need to download the Jess files yourself, we don't distribute this with the AW Agent package)
  - `base.jar` is the compiled java for the base classes. They are in the documentation as package `kcdcc.awa.base`. You don't need to recompile these.
  - `xerces.jar` is the Xerces 1 XML parser from www.apache.org.
    is used to search strings using regular expressions. Consequently it will soon be replaced with new classes from the core java 1.4 sdk.
  - `jess60.jar` containing Jess v6.0. Go to `http://herzberg.ca.sandia.gov/jess/` to download version 6.0. You can get a full version for free for academic use but you need to complete a form saying what it is for etc. There is a trial version that has an expiry period.
    AW SDK build number must match that used by `awa.dll`. Currently this is build 24. You can get `Aw.dll` from http://www.activeworlds.com/sdk/. We assume in this readme that you use an AW server version 3.3 or later.
  - `aw.dll` is a java native interface (JNI) library for the Active Worlds SDK.
  - `awa.dll` is the SDK from Active Worlds.
  - `concurrent.jar` contains compiled java classes for multithreading. It is used by `base.jar`.
- Files specific to each agent
  - `streetlights.bat` is an MS-DOS batch file that runs a society of agents with only one agent: SAM. Edit this as per Section 3 in order to run the streetlights demo, or copy and edit to create your own agent.

- `streetlights.properties` is an optional file containing java system properties for Sam. Edit this as per Section 3 to run the streetlights demo, or copy and edit to create your own agent.
- `streetlights.clp` contains the Jess rules for agent Sam.
- `streetlights.xml` contains the configuration for society Sam.

3 Details of files that need to be modified

These changes are described assuming that you are running the streetlights demo. To create your own agent you will copy each of the 4 streetlights files and rename the files to be the name of your agent. Then you will edit these new agent files to have your agents configuration, properties, rules, etc.

3.1 Edit bat file

The first thing you need to do is to edit the bat file. Assuming that you are running society Streelights (agent Sam), edit `streetlights.bat`.

The first line of `streetlights.bat` sets the path environment variable:

```
set PATH=\demos2;C:\J2SDK1~1.0_0\bin;c:\WINNT\system32;c:\WINNT
```

If at MIT:

```
set PATH=\demos2;C:\PROGRA~1\JDK13~1.1\bin;c:\WINNT\system32;c:\WINNT
```

If you put the files from Section 2 into a directory other than `\demos2` then you will need to change this path. The above path also assumes that you downloaded the Sun java 1.4.0 SDK (from http://java.sun.com) into its default directory, or if at MIT, the 1.3.1 version. Change the path accordingly if this is not the case (using MS-DOS 8+3 character pathnames!).

The second line of `streetlights.bat` is the classpath. This should contain a list of the jar files in your directory.

The last line of `streetlights.bat` runs the program. For example,

```
java -Dkcdcc.awa.base.Society.properties="streetlights.properties"
kcdcc.awa.base.Society
```

runs the society with the system property `kcdcc.awa.base.Society.properties` set to file `streetlights.properties` in the current directory.

3.2 Edit properties file

Change the following properties in the property file:

- `-Dkcdcc.awa.base.SocietyParser.inFile` to the full pathname of the XML file containing the configuration of your society

- `-Dkcdcc.citizen` to your citizen number
- `-Dkcdcc.privilegePassword` to your privilege password

3.3 Edit XML file

Change the following parameters in `streetlights.xml` (see http://www.arch.usyd.edu.au/ g_smith for the grammar of the XML)

- In element `session` you need to set `domain`, `port` and `worldName` according to your AW server, set `avatarType` to the number of an avatar, and `avatarname` to any name you like.
- In element `reteagent`, set `name` to any name you like your agent to have and set the parameter with name `name` to have as `value` the URL of the clp file containing your Jess rules (such as " file:///demos2/streetlights.clp"). A `session` element can contain any number of `Agent` and/or `reteagent` elements.
- Set the `location` element to the desired location of your society. The x value is positive West direction and 1W is the same as x=1000. The y value is the up direction. The z value is positive North and 1N is the same as z=1000. Obviously, if a number of people are concurrently running this agent in the same world then you should use distinct locations (and agent names). Additionally, try not to use location $(0, 0, 0)$ or your avatar may appear inside citizens at ground zero.
- Set one `Sensor` element for each sensor required in your agent. For example,

  ```
  <sensor class="kcdcc.awa.base.AWAvatarSensor"/>
  ```

  configures an avatar sensor,

  ```
  <sensor class="kcdcc.awa.base.AW3DObjectSensor">
      <parameter name="width" value="1000"/>
      <parameter name="height" value="1000"/>
  </sensor>
  ```

  configures a sensor that looks for 3D objects within a region 1000×1000 around the location of the agent society, and

  ```
  <sensor class="kcdcc.awa.base.AWChatSensor">
      <parameter name="recognisedCitizens" value="Greg"/>
  </sensor>
  ```

  configures a chat sensor that only recognises chat from citizen "Greg". Change "Greg" to your citizen name or do not enter a value at all.

4 Sensors available in this release

Sense-data pushed into Jess working memory is given a fact name derived from its java class name and package name. For example, java class `Added3DObjectSenseData` in package `kcdcc.awa.base` becomes fact `kcdcc_awa_base_Added3DObjectSenseData`. The following AW sensors are available in the main package `kcdcc.awa.base`:

`AWChatSensor` — Receive chat from world.
Provides `TextMessageSenseData` with the following Jess definstance slots:
- `text`, which is a String
- `sender`, which is a String
- `msgType`, which is an integer (0=said, 1=broadcast, 2=whisper)

The optional parameter `recognisedCitizens` restricts chat text to only those citizens. The parameter is a set of citizen names. The default is to recognise chat from all citizens. The parameter is intended for use where agents receive commands via chat (as does the agent in section 7.4.3) or for efficiency reasons.

`AWAvatarSensor` — Sense avatar adds, changes, deletes and clicks.
For avatar delete provides `AvatarSenseData` with the following Jess definstance slots:
- `name`, which is a String
- `session`, which is an integer
- `avatarDeleted`, which is TRUE

For avatar add/change provides `LocatedAvatarSenseData` with the following Jess definstance slots:
- `name`, which is a String
- `session`, which is an integer
- `avatarDeleted`, which is FALSE
- `locn`, which is a kcdcc.awa.base.Location6DF, containing slots for `x`, `y`,`z`,`roll`,`yaw`,`tilt`.

For avatar clicks: not tested yet

`AW3DObjectSensor` — Sense objects within configured region.
For object add/change provide `AddedObject3DSenseData` with the following Jess definstance slots:
- `objectNo`, which is an integer
- `model`, which is a String

▶ `objectLocn`, which is an integer
▶ `ownerNo`, which is an integer
▶ `buildTimestamp`, which is an integer
▶ `description`, which is a String
▶ `action`, which is a String
▶ `objectNo`, which is an integer

For object delete provide `DeletedObject3DSenseData` with the following Jess definstance slots:
▶ `objectNo`, which is an integer

This sensor looks for objects within the configured region about its initialised location. To change locations, shift the location of the agent (the `location` slot of the `kcdcc_awa_base_ReteAgent` fact), and use the `reset` method of the sensor.

The following non-AW sensors are available:

- `AgentChatSensor` to receive chat directly from another agent in the same society.
- `TimeSensor` to receive periodic time sense-data so as to drive polled tasks. It has one configuration parameter

  `period`, being the time period in seconds.
- `VRTTimeSensor`, which extends `TimeSensor` to provide virtual reality time. With regard to time,
  - Time in the java class `java.util.Date` is the number of milliseconds since midnight Jan 1st 1970 GMT
  - The Jess `(time)` function is the number of seconds since midnight Jan 1st 1970
  - The AW SDK only provides time in the universe attributes event as attribute.
    It provides the number of seconds since midnight Jan 1st 1970 in timezone GMT-2hrs (called VRT). VRT is the time shown on the client browser.
    This event is only provided at login time, so VRTTimeSensor keeps its own time.
- `SQLSensor` to receive SQL results from a database via JDBC.
- `DOMSensor` to receive an arbitrary XML message from another agent.
- `ACLSensor` to send an ACL message to another agent. This will be explained in document “Advanced User Guide for KCDCC AW Agent package”.

5 Effectors available in this release

All effectors have the same architecture. Effectors are created as needed - they do not need to be loaded at configuration time. To use them, follow these steps:

1. Create and initialise an effector. For example, to create an AW chat effector from Jess,

```
(defglobal ?*awchat* = nil)
(defclass awchateffector AWChatEffector)
(bind ?*awchat* (new  AWChatEffector))
(?*awchat* init ?a nil))
```

2. Some effectors will have a `reset` method that should be used for each new effect-data.

3. For each effector property there will be a set method. Again, for the AW chat effector from Jess,

```
(?*awchat* setMsg "Hi there! I like to look.")
(?*awchat* setMsgType (get-member Society CHAT_SAID))
```

4. Once all of the properties are set, activate the effector.

```
(?*awchat* activate)
```

   The `activate` function will return an object. For many effectors this object will be `nil`.

The following AW effectors are available in the main package `kcdcc.awa.base`:

`AWChatEffector`	Send chat to AW, returning `nil`. The following properties exist: ▶ `msg`, which is a String ▶ `msgType`, which is an integer (0=said, 1=broadcast, 2=whisper)
`AWMoveEffector`	Move the avatar for the `Society` to a new orientation by a defined distance, returning `nil`. The following properties exist: ▶ `deltaX`, which is an integer ▶ `deltaY`, which is an integer ▶ `deltaZ`, which is an integer ▶ `roll`, which is an integer ▶ `yaw`, which is an integer

► `tilt`, which is an integer

This effector is best used indirectly via the `moveto` method of the `Society`. See `light-move-action-1` in Section 7.4.6 for an example.

`AW3DObjectEffector` Add, change or delete a 3D object to/in/from the world, returning an integer AWobject number.
The following properties exist:
► `command`, which is 1 for add, 2 for change, 3 for delete
► `x`, which is an integer
► `y`, which is an integer
► `z`, which is an integer
► `yaw`, which is an integer (roll and tilt to be added)
► `model`, which is a String
► `description`, which is a String
► `action`, which is a String
► `objNo`, which is an integer

The following non-AW effectors are available:

- `DOMEffector` sends an arbitrary XML message to another agent or broadcasts to all agents in the society.
- `ACLEffector` to send an ACL message to another agent. This will be explained in document

  "Advanced User Guide for KCDCC AW Agent package".

6 Procedure to run the streetlights demo

1. Put all of the files listed in Section 2 into one directory. The supplied files assume that this directory is called `\demos2`.
2. Make the changed described in Sectionmods.
3. Open an MS-DOS command shell (this is strongly recommended because it prints messages that help with debugging), change to that directory and type

   ```
   streetlights
   ```

   This will run the streetlights.bat file. If you have created your own agent and have another bat file, the general procedure is to type the name of the bat file.

4. Output from the agent society: Jess rules that use `AWChatEffector` will print messages to the AW browser chat window. Jess rules that use the Jess function `(printout)` or print to the java standard output stream will print to the MS-DOS command shell window. By moving your citizen's avatar around the world you

   can find locations with greater resolution that is shown on the AW

   browser using the printout effector to print the location of the avatar.

5. To end the session you can either enter control-C in the MS-DOS command shell, or alternatively, the chat sensor rules look for a chat message from a recognised citizen that contains

   ```
   command die
   ```

   When you type this in the AW chat window, the message is detected by the agent and the society dies.

7 Demonstration

In this section we describe a demonstration single-agent society named 'Streetlights" with an agent named "Sam" that builds a 3D representation of itself in an AW world and then maintains itself according to citizens needs. This demo is still being developed (which is code for "not fully tested and debugged"). It nevertheless provides an example of how to use the package.

7.1 Configuration for `Streetlights`

This society "Streetlights" has a single agent "Sam". This configuration file has the agent society entering the agent world and uses the streetlights.clp file for the jess rules. There are sensors for chat text, avatars, 3D objects (within a configured region of the agent), and virtual reality time.

streetlights.xml

```
1  <?xml version="1.0" ?>
2  <!DOCTYPE catalog
3             SYSTEM "http://www.arch.usyd.edu.au/~g_smith/Society.dtd">
4  <society domain="auth.activeworlds.com"
5           port="5696"
6           avatarType="26"
7           avatarname="Streetlights"
8           worldname="agent">
9  <reteagent name="Sam">
10 <parameter name="jess" value="file:///demos2/streetlight.clp"/>
11 <location x="-916" y="0" z="-114"/>
12 <behaviours codebase="file:///demos2">
13 <sensor class="kcdcc.awa.base.AW3DObjectSensor">
```

```
    <parameter name="width" value="350"/>
    <parameter name="height" value="350"/>
  </sensor>
  <sensor class="kcdcc.awa.base.AWAvatarSensor"/>
  <sensor class="kcdcc.awa.base.AWChatSensor">
    <parameter name="recognisedCitizens" value="Greg"/>
  </sensor>
  <sensor class="kcdcc.awa.base.VRTTimeSensor">
    <parameter name="period" value="60"/>
  </sensor>
  </behaviours>
  </reteagent>
  </society>
```

To run the demo, type streetlights in the MS-DOS command window to execute the streetlights.bat file. You will need to type `<control-C>` from the MS-DOS command window to kill the agent.

The Jess rules are loaded from URL file:///demos2/streetlights.clp, which is the local directory \demos2. Any URL could be used, for example a HTTP protocol URL.

7.2 Annotated Jess fact base for agent `Sam`

The sensors are written in java, so the first thing to do is to specify where to find the relevant java classes for the sensor data.

```
(import jess.*)
(import kcdcc.awa.base.*)
```

The sensors use java Beans to store the sense data. Jess provides the `defclass` construct to automatically generate a template that represents a class of java Beans. Each java bean property is seen by Jess rules as a slot in an unordered fact. In addition, three slots are always defined:

- slot `OBJECT` is a reference to the java bean instance
- slot `class` is the object returned by the `getClass` method that every java instance has
- slot `name` is the name of the bean.

`ReteAgent` takes advantage of `defclass` by pushing `SenseData`, as well as a reference to the `kcdcc.awa.base.ReteAgent` instance, into Jess working memory as java beans. `ReteAgent` only pushes beans into working memory for which `defclass` templates exist.

This means you need a `defclass` for those beans, that is, for the sense data facts, that are required for your rules. Below is the list of `defclass` statements for the sense data used in `Sam`. In the `defclass` statement, the name of the fact in Jess is first (the long name) and the name of the java Bean is second. In your rules you will use the long name. The slots for these facts are defined in this guide in the section 4.

```
1  (clear)
2
3  (defclass kcdcc_awa_base_Agent Agent)
4  (defclass kcdcc_awa_base_ReteAgent ReteAgent
5      extends kcdcc_awa_base_Agent)
6
7  (defclass kcdcc_awa_base_SenseData SenseData)
8  (defclass kcdcc_awa_base_TimeSenseData TimeSenseData
9      extends kcdcc_awa_base_SenseData)
10 (defclass kcdcc_awa_base_Object3DSenseData Object3DSenseData
11     extends kcdcc_awa_base_SenseData)
12 (defclass kcdcc_awa_base_AddedObject3DSenseData AddedObject3DSenseData
13     extends kcdcc_awa_base_Object3DSenseData)
14 (defclass kcdcc_awa_base_DeletedObject3DSenseData DeletedObject3DSenseData
15     extends kcdcc_awa_base_Object3DSenseData)
16 (defclass kcdcc_awa_base_AvatarSenseData AvatarSenseData
17     extends kcdcc_awa_base_SenseData)
18 (defclass kcdcc_awa_base_LocatedAvatarSenseData  LocatedAvatarSenseData
19     extends kcdcc_awa_base_AvatarSenseData)
20 (defclass kcdcc_awa_base_TextMessageSenseData TextMessageSenseData
21     extends kcdcc_awa_base_SenseData)
```

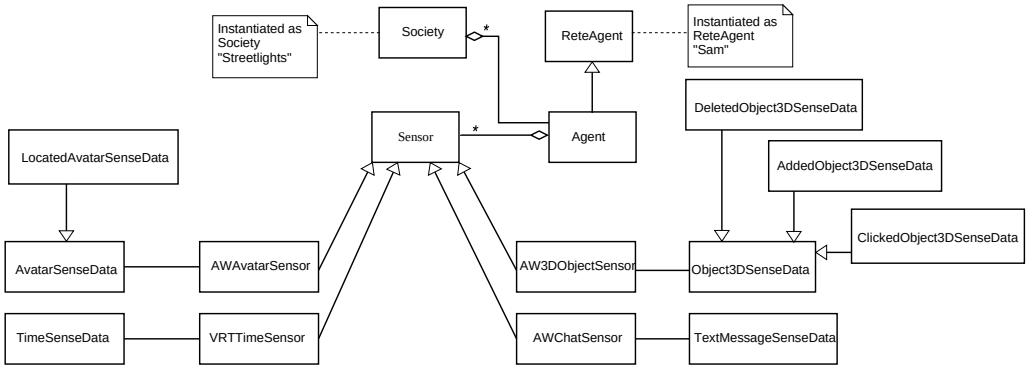

Figure 5: UML diagram showing the sensors and sense-data of “Sam”.

The `extends` construct tells Jess that one bean is a specialisation of another, allowing for inheritance of properties. If you want to know what the slots of a defclass fact are without looking through java code, do something like this:

```
(printout t (ppdeftemplate kcdcc_awa_base_ReteAgent) crlf)
```

Figure 5 shows the inheritance in the sense-data used by Sam, and hence further illustrates the need for the `extends` declarations in the `defclass` constructs.

The following `deftemplate` statements define the names and slots of the unordered facts in Sam that are not sense data.

Each 3D object in the world is represented in the structure, model, component, and Object3D facts below. The information in the structure, model, and component facts are essentially from the sense data and the information in the Object3D fact is determined by jess rules.

Each avatar in the world is represented in the avatar and citizen facts.

The schedule fact keeps track of which module is the current focus for the selection and execution of rules.

The intention fact stores information about goals.

```
(deftemplate MAIN::structure
    "One structure that can be built from 3D components"
    (slot structureId (type INTEGER) (default 0))
    (slot description (type STRING) (default nil))
)

(deftemplate MAIN::model
    "One 3D model for a 3D component"
    (slot modelId (type INTEGER) (default 0))
    (slot name (type STRING) (default nil))
)

(deftemplate MAIN::component
    "One 3D component of a structure"
    (slot structureId (type INTEGER) (default 0))
    (slot awNumber (type INTEGER) (default 0))
    (slot xOffset (type INTEGER) (default 0))
    (slot yOffset (type INTEGER) (default 0))
    (slot zOffset (type INTEGER) (default 0))
    (slot yaw (type INTEGER) (default 0))
    (slot modelId (type INTEGER) (default 0))
    (slot description (type STRING) (default ""))
    (slot action (type STRING) (default ""))
)

(deftemplate MAIN::Object3D
    "A perceived 3D object"
    (slot object-type (type LEXEME))
    (slot location (type OBJECT))
    (slot state (type LEXEME) (default NONE))
    (slot object-id (type INTEGER))
    (slot owner)
    (slot model (type STRING))
    (slot timestamp (type INTEGER))
```

```
35  )
36
37
38  (deftemplate MAIN::avatar
39      "A perceived avatar"
40      (slot location (type OBJECT))
41      (slot session (type INTEGER))
42      (slot entrytime (type INTEGER))
43      (slot citizen-name (type STRING))
44  )
45
46
47  (deftemplate MAIN::citizen
48      "A citizen is a memory of an avatar"
49      (slot expirytime) ;AW time at which memory expires
50      (slot name)       ;name of citizen
51      (slot entrytime)  ;AW time that citizen entered region
52                        ;near streetlight
53      (slot exittime))  ;AW time that citizen left region
54                        ;near streetlight
55
56
57  (deftemplate MAIN::schedule (slot collection (type
58  OBJECT)))
59
60  (deftemplate MAIN::intention (slot id) (slot label))
61
```

7.3 Annotated Jess rules for agent `Sam`

In Sam, there are rules to determine percepts for time, a command from chat text, certain 3D objects, and of citizens presence. Additionally there is a memory of citizens near the agent. These facts persist for some number of days and then are forgotten.

Conception uses truth maintenance rules to determine what is the most interesting nearby thing (citizen or object), plus very simple learning of when citizens have been present in the past.

For the hypothesiser there are fixed sets of intentions, each of which is represented as a set of rules: maintenance of lights on/off, and the relocation of lights - move nearer to citizen, or nearer to an interesting object. This demo shows one possible way to use modules to partition the rules. This makes it easier to understand the code, but mainly it is done for the scheduler. All facts are in module MAIN, the default module. In most cases facts are explicity asserted with the `MAIN::` module label. Facts asserted without an explicit module label are asserted with the current focus module.

This is only a demonstration agent. There are lots of things that it could do but doesn't, or that it could do better, or just do differently. It also undoubtably still has bugs; for example, the MEMORY module hasn't been fully tested yet. Further, we would prefer that the avatar was not present but it is required for the current SDK for the

events that our sensors receive. You should pretend that it is not there (this will be fixed in a later version of the package). As a demo, though, it does show how the agent package can be used.

Below you will find the following in the startup rule:

```
(modify ?a (traceEnabled TRUE))
```

This enables a debugging trace. Comment out this line to disable it. Be aware that the trace prints out heaps of stuff that is invaluable for debugging.

7.3.1 MAIN module

The MAIN module has a start-up rule and a scheduler.

```
1
2
3   (defrule MAIN::start-up-rule
4       (declare (salience 10))
5       ?a <- (kcdcc_awa_base_ReteAgent)
6   =>
7       (set-multithreaded-io TRUE)
8
9       (modify ?a (traceEnabled TRUE))
10
11      (assert (MAIN::reset-intentions))
12      (assert (MAIN::last-action 0))
13
14      (assert (MAIN::structure (structureId 1) (description "Streetlight")))
15      (assert (MAIN::model (modelId 1) (name "stlamp1")))
16
17      (assert (MAIN::component
18                  (structureId 1)
19                  (xOffset 50) (yOffset 0) (zOffset 50)
20                  (modelId 1)
21                  (description "Intelligent street light")
22                  (action "")))
23      (assert (MAIN::build))
24
25      ;See below for description of the next fact.
26      (assert (MAIN::schedule (collection
27                   (new java.util.Vector
28                      (call java.util.Arrays
29                         asList
30                         (create$ REFLECTIVE HYPOTHESISER CONCEPTION MEMORY
31                                  PERCEPTION REFLEXIVE))))
32      ))
33      (printout t "Agent initialised" crlf)
```

```
34  )
35
```

The scheduler requires that you use `extends kcdcc_awa_base_SenseData` on each sense-data defclass or else it won't trigger the scheduler rule when new sense-data arrives.

We use the focus stack to restrict rules to firing within one module at a time. This will be especially useful for the hypothesiser, which can use multiple sets of intentions by putting rules for an intention in a separate module and then scheduling it (and not other intentions). The decision by the hypothesiser on which intention to enable will be goal based, such as via some measure of utility from current concepts. We implement the scheduler with the Jess focus stack. The reason is to ensure that the control mechanism is isolated from any knowledge: all rules on the agenda from the current focus module run, then all rules on the agenda from the next scheduled module run, and so on until we get back to the MAIN module (which will then trigger off the next sense-data). The fact `(schedule (collection ?s))` is a collection of module names to schedule IN REVERSE ORDER. Make ?s an instance of a java collection class that provides an iterator method.

The next rule controls scheduling. Hypothesiser can alter `(schedule ...)` to change which set of intentions are current. We could partition action activation into REFLEXIVE, REACTIVE and so on to prevent reflective rules from firing if there are reflexive ones, if that is what is required. Developers could also change this to other data structures and rules if so desired.

```
1   (deffunction scheduler-iterate (?s)
2       (printout t "SCHEDULE IS: " (?s toString) crlf)
3       (bind ?it (?s iterator))
4       (while (?it hasNext)
5           (bind ?m (?it next))
6           (focus ?m))
7   )
8
9   (defrule MAIN::scheduler-rule-1
10      "Schedule all modules everytime something changes"
11      (declare (salience 5))
12      (kcdcc_awa_base_SenseData)
13      (MAIN::schedule (collection ?s))
14  =>
15      (scheduler-iterate ?s)
16  )
17
```

7.3.2 MEMORY module

In PERCEPTION there will only ever be one 3D object with a particular object-id (x/z?). That is, a table of Object3D facts could be keyed off object-id (x/z?). In MEMORY this is not the case; rather, facts remain until they decay. For now we maintain an explicit expiry time in every MEMORY fact. An alternative is to use a JessListener and just maintain fact id versus expiry time. Or use a database via JDBC. Or use a java data structure instead of facts. Or ...

So far there is only a memory of avatar presence. After a persistence mechanism is implemented for this agent we may add more memories.

```
1  (defmodule MEMORY)
2
3  ;1209600 = 14days * 24hours/day * 60 minutes/hour * 60 seconds/hour
4  (defglobal ?*decay-time* = 1209600)
5
6  ;Poll interval is 60 seconds - should match XML configuration
7  ; (next release will contain a method on ReteAgent to get this)
8  (defglobal ?*poll-interval* = 60)
9
10 (deftemplate MAIN::citizen
11     "A citizen is a memory of an avatar"
12     (slot expirytime) ;AW time at which memory expires
13     (slot name)       ;name of citizen
14     (slot entrytime)  ;AW time that citizen entered region near streetlight
15     (slot exittime))  ;AW time that citizen left region near streetlight
16
17
18
19 (deffunction MEMORY::remember-citizen (?expiryt ?n ?entryt ?exitt)
20     "Add a memory of a citizen"
21     (assert (MAIN::citizen (expirytime ?expiryt)
22                     (name ?n)
23                     (entrytime ?entryt)
24                     (exittime ?exitt)))
25 )
26
27 (defrule MEMORY::forget-memory-rule-1
28     "When currenttime changes we check for expired memories"
29     (MAIN::currenttime ?t)
30     ?f <- (MAIN::citizen (expirytime ?et&:(<= ?et ?t)))
31 =>
32     (retract ?f)
33 )
```

7.3.3 PERCEPTION module

In perception we find patterns from sense-data.

```
(focus PERCEPTION)

(defrule PERCEPTION::time-rule-1
    "1st time sense-data - assert current time"
    (declare (salience 0))
    ?f1 <- (MAIN::kcdcc_awa_base_TimeSenseData (interval ?t))
    (not (MAIN::currenttime ?))
=>
    (retract ?f1)
    (assert (MAIN::currenttime ?t))
)

(defrule MAIN::time-rule-2
    "Time sense-data update - modify current time"
    (declare (salience 0))
    ?f1 <- (MAIN::kcdcc_awa_base_TimeSenseData (interval ?t))
    ?f2 <- (MAIN::currenttime ?)
=>
    (retract ?f1 ?f2)
    (assert (MAIN::currenttime ?t))
)

(deftemplate MAIN::Object3D
    "A perceived 3D object"
    (slot object-type (type LEXEME))
    (slot location (type OBJECT))
    (slot state (type LEXEME) (default NONE))
    (slot object-id (type INTEGER))
    (slot owner)
    (slot model (type STRING))
    (slot timestamp (type INTEGER))
)


(defrule PERCEPTION::look-perception-rule-1
    "Perceive a new 3D object"
    (declare (salience 0))
    ?f <- (MAIN::kcdcc_awa_base_AddedObject3DSenseData (objectNo ?no)
                                                       (model ?m&:(neq ?m "stlamp1"))
                                                       (objectLocn ?locn)
                                                       (ownerNo ?own)
                                                       (buildTimestamp ?t))
    (not (MAIN::Object3D (object-id ?no)))
=>
    (assert (MAIN::Object3D (object-type NULL)
                      (location ?locn)
```

```
                        (object-id ?no)
                        (owner ?own)
                        (model ?m)
                        (timestamp ?t)))
    (retract ?f)
)

(defrule PERCEPTION::look-perception-rule-2
    "Perceive a changed 3D object"
    (declare (salience 0))
    ?f1 <- (MAIN::kcdcc_awa_base_AddedObject3DSenseData (objectNo ?no)
                                                        (model ?m)
                                                        (objectLocn ?locn)
                                                        (ownerNo ?own)
                                                        (buildTimestamp ?t))
    ?f2 <- (MAIN::Object3D (object-id ?no))
=>
    (modify ?f2 (location ?locn)
                (object-id ?no)
                (owner ?own)
                (model ?m)
                (timestamp ?t))
    (retract ?f1)
)

(defrule PERCEPTION::look-perception-rule-3
    "Is the new 3D object a picture?"
    (declare (salience 0))
    ?f <- (MAIN::Object3D (object-type NULL) (model ?m))
    (test (eq (str-index "pict" (lowcase ?m)) 1))
                                        ;test is equivalent to regular
                                        ;expression "^[pP][iI][cC][tT].*"
=>
    (modify ?f (object-type PICTURE))
)

(defrule PERCEPTION::look-perception-rule-4
    "Is the new 3D object a panel?"
    (declare (salience 0))
    ?f <- (MAIN::Object3D (object-type NULL) (model ?m))
    (test (eq (str-index "pan" (lowcase ?m)) 1))
                                        ;test is equivalent to regular
                                        ;expression "^[pP][aA][nN].*"
=>
    (modify ?f (object-type PANEL))
)

(defrule PERCEPTION::look-perception-rule-5
    "Is the new 3D object something else? Then forget about it"
    (declare (salience -1))
    ?f <- (MAIN::Object3D (object-type NULL) (model ?m))
```

```
98      (test (neq (str-index "pan" (lowcase ?m)) 1))
99      (test (neq (str-index "pict" (lowcase ?m)) 1))
100 =>
101     (retract ?f)
102 )
103
104 (defrule PERCEPTION::look-perception-rule-6
105     "Perceive a deleted known 3D object"
106     (declare (salience 0))
107     ?f1 <- (MAIN::kcdcc_awa_base_DeletedObject3DSenseData (objectNo ?no))
108     ?f2 <- (MAIN::Object3D (object-id ?no))
109 =>
110     (retract ?f1 ?f2)
111 )
112
113 (defrule PERCEPTION::look-perception-rule-7
114     "Perceive a deleted unknown 3D object"
115     (declare (salience 0))
116     ?f1 <- (MAIN::kcdcc_awa_base_DeletedObject3DSenseData (objectNo ?no))
117     (not (MAIN::Object3D (object-id ?no)))
118 =>
119     (retract ?f1)
120 )
121
122
123
124 (defrule PERCEPTION::avatar-perception-rule-1
125     "Perceive a new avatar"
126     (declare (salience 0))
127     ?f <- (MAIN::kcdcc_awa_base_LocatedAvatarSenseData
128           (name ?n)
129           (avatarDeleted FALSE)
130           (locn ?l)
131           (session ?s))
132     (not (MAIN::avatar (citizen-name ?n)))
133     (MAIN::currenttime ?t)
134 =>
135     ;(printout t ?n " is at " (?l toString) crlf)
136     (assert (MAIN::avatar (location ?l)
137                           (entrytime ?t)
138                           (session ?s)
139                           (citizen-name ?n)))
140     (retract ?f)
141 )
142
143 (defrule PERCEPTION::avatar-perception-rule-2
144     "Perceive a changed avatar"
145     (declare (salience 0))
146     ?f1 <- (MAIN::kcdcc_awa_base_LocatedAvatarSenseData
147           (name ?n)
148           (avatarDeleted FALSE)
```

```
            (locn ?l)
            (session ?s))
    ?f2 <- (MAIN::avatar (citizen-name ?n))
=>
    ;(printout t ?n " is now at " (?l toString) crlf)
    (modify ?f2 (location ?l))
    (retract ?f1)
)

(defrule PERCEPTION::avatar-perception-rule-3
    "Perceive a deleted known avatar"
    (declare (salience 0))
    ?f1 <- (MAIN::kcdcc_awa_base_AvatarSenseData (session ?s)
                                                 (avatarDeleted TRUE))
    ?f2 <- (MAIN::avatar (session ?s) (citizen-name ?n) (entrytime ?t1))
    (MAIN::currenttime ?t2)
=>
    (bind ?expiryt (+ ?t1 ?*decay-time*))
    (MEMORY::remember-citizen ?expiryt ?n ?t1 ?t2)
    (retract ?f1 ?f2)
)

(defrule PERCEPTION::avatar-perception-rule-4
    "Perceive a deleted unknown avatar"
    (declare (salience 0))
    ?f1 <- (MAIN::kcdcc_awa_base_AvatarSenseData (session ?s)
                                                 (avatarDeleted TRUE))
    (not (MAIN::avatar (session ?s)))
=>
    (retract ?f1)
)

(defrule PERCEPTION::text-perception-rule-1
    "Perceive text command to turn on light"
    (declare (salience 0))
    ?f <- (MAIN::kcdcc_awa_base_TextMessageSenseData (text ?t))
    (test (neq (str-index "command lights on" ?t) FALSE))
=>
    (assert (MAIN::command ON))
    (retract ?f)
)

(defrule PERCEPTION::text-perception-rule-2
    "Perceive text command to turn on light"
    (declare (salience 0))
    ?f <- (MAIN::kcdcc_awa_base_TextMessageSenseData (text ?t))
    (test (neq (str-index "command lights off" ?t) FALSE))
=>
    (assert (MAIN::command OFF))
    (retract ?f)
)
```

```
200
201  (defrule PERCEPTION::text-perception-rule-3
202      "Default text perception"
203      (declare (salience -10))
204      ?f <- (MAIN::kcdcc_awa_base_TextMessageSenseData (text ?t))
205  =>
206      (retract ?f)
207  )
```

7.3.4 CONCEPTION module

Some of the concepts here are truth maintenance facts, some are patterns from MEMORY. The first set of rules here are to find a concept (`CONCEPTION:pattern ?hourno`). They induce a logical conjunction over a space of 24*7 boolean dimensions, where each dimension is TRUE if at least one citizen visited during the MEMORY decay period for that hour. Hour number has a range of `0 .. ?*maxhour*`.

We represent the conjunction as a set of facts (`pattern ?hourno`) where `?hourno` is one TRUE dimension. A logical conjunction is far from the best way to represent this, and a boolean is not the best feature to use either. But it does demonstrate that the MEMORY facts can be used to find CONCEPT level patterns. Better representations include a stochastic model (the expected number of citizens at any particular time) and a disjunction over a 2 dimensional space (time of day versus day). The first thing that could be done is to change from (`pattern ?hourno`) to (`pattern (lower ?lowerhourno) (upper ?upperhourno)`). That is, make each pattern is an interval (a feature subspace) and not a point, thus enabling generalisation to unseen data.

The algorithm used is inspired by the Incremental Specific to General (ISG) algorithm from Pat Langley's "Elements of Machine Learning". It is breadth first, so it fits with the Rete engine. ISG requires both positive and negative training instances or we risk overfitting the data. Positive instances are the citizen facts, but finding negative instances is problem. We could randomly sample the space of citizens not covered by positive training instances. We haven't done this. We could do; we just haven't yet. For this demo we just take negative instances from the set difference between any current hypotheses and the current positive training instances. This works because the citizens memory decays, so that citizen facts that once were positive may no longer be. This set of rules are not the best way to find this particular representation, but it does allow for future modification.

This approach works over the short term but strictly speaking is problematic. ISG works on a lattice of hypotheses, with the most specific generalisation at one end and the most general at the other end. So if we take particular citizen fact to be a positive instance one day we really shouldn't take it to be negative later. To correct for this we need one additional integrity checking rule: at midnight every day we reset the set of hypotheses to a single most specific generalisation (that is, the empty list). You can think of this as being like a file server running a daily background file system integrity check.

```
(defmodule CONCEPTION)

(defglobal ?*maxhour* = (* 7 24))

;Not only does the mod in the next rule force a number 0..?*maxhour*,
;it also forces an integer return value
(deffunction CONCEPTION::timeToHourNo (?t)
    (bind ?h (mod (/ ?t 3600) ?*maxhour*))
    (return ?h)
)



(defrule CONCEPTION::positive-instance-rule-1
    (MAIN::citizen (entrytime ?t1) (exittime ?t2))
=>
    (bind ?h1 (CONCEPTION::timeToHourNo ?t1))
    (bind ?h2 (CONCEPTION::timeToHourNo ?t2 ))
    (if (< ?h2 ?h1) then (bind ?max (+ ?*maxhour* ?h2))
                    else (bind ?max ?h2))
    (bind ?i ?h1)
    (while (<= ?i ?max) do
       (bind ?val (mod ?i ?*maxhour*))
       (assert (MAIN::confirm-positive ?val))
       (bind ?i (+ ?i 1))
    )
)

(defrule CONCEPTION::positive-instance-rule-2
    ?f <- (MAIN::confirm-positive ?h)
    (not (MAIN::pattern ?h))
=>
    (assert (MAIN::pattern ?h))
    (retract ?f)
)

(defrule CONCEPTION::positive-instance-rule-3
    ?f <- (MAIN::confirm-positive ?h)
    (MAIN::pattern ?h)
=>
    (retract ?f)
)

;return TRUE if ?h is inside the interval [?h1,?h2] when ?h2 wraps around
;zero (as hour numbers do)
(deffunction CONCEPTION::insideInterval (?h ?h1 ?h2)
    (if (< ?h2 ?h1)
     then
       (if (<= ?h1 ?h)
```

```
         then
            (bind ?max (+ ?*maxhour* ?h2))
            (bind ?htemp ?h)
         else
            (bind ?max (+ ?*maxhour* ?h2))
            (bind ?htemp (+ ?*maxhour* ?h)))
        (return (and (<= ?h1 ?htemp) (<= ?htemp ?max)))
      else
        (return (and (<= ?h1 ?h) (<= ?h ?h2)))
    )
)
(deffunction CONCEPTION::insideTimeInterval (?h ?t1 ?t2)
    (bind ?h1 (CONCEPTION::timeToHourNo ?t1 ))
    (bind ?h2 (CONCEPTION::timeToHourNo ?t2 ))
    (return (CONCEPTION::insideInterval ?h ?h1 ?h2))
)

(defrule CONCEPTION::negative-instance-rule-1
    ?f <- (MAIN::pattern ?h)
    (not (MAIN::citizen (entrytime ?t1)
                    (exittime ?t2&:(CONCEPTION::insideTimeInterval ?h ?t1
?t2))))
=>
    (retract ?f)
)


;ADD rule CONCEPTION::midnight-daemon-rule here
```

The second set of rules here use truth maintenance to maintain a concept of what is nearby the streetlight The first rule only asserts a new TM node if the distance is less than a threshold of 10000. `?la` and `?ll` are both instances of the java class `kcdcc.awa.base.Location`. As can be seen from the javadoc documentation, this provides a method

`public double distance(Location ref)`

that finds the distance between two locations.

```
(defrule CONCEPTION::occupied-rule-1
    "Is there a citizen nearby?"
    (logical (MAIN::Object3D (object-id ?id) (object-type LIGHT) (location
?ll)))
    (logical (MAIN::avatar (citizen-name ?n) (location ?la)))
    (test (< (?ll distance ?la) 10000))
=>
    (assert (MAIN::nearby CITIZEN ?n (?ll distance ?la)))
)


```

```
(defrule CONCEPTION::occupied-rule-2
    "Is there a PANEL nearby?"
    (logical (MAIN::Object3D (object-id ?lid)
                             (object-type LIGHT)
                             (location ?ll)))
    (logical (MAIN::Object3D (object-type PANEL)
                             (object-id ?id)
                             (location ?lp)))
    (test (< (?ll distance ?lp) 10000))
=>
    (assert (MAIN::nearby PANEL ?id (?ll distance ?lp)))
)

(defrule CONCEPTION::occupied-rule-3
    "What is the nearest CITIZEN?"
    (logical (MAIN::nearby CITIZEN ?n1 ?d1))
    (logical (not (MAIN::nearby CITIZEN ?n2&:(neq ?n1 ?n2)
                                        ?d2&:(< ?d2 ?d1))))
=>
    (assert (MAIN::nearest CITIZEN ?n1 ?d1))
)

(defrule CONCEPTION::occupied-rule-4
    "What is the nearest PANEL?"
    (logical (MAIN::nearby PANEL ?n1 ?d1))
    (logical (not (MAIN::nearby CITIZEN ? ?)))
    (logical (not (MAIN::nearby ? ? ?d2&:(< ?d2 ?d1))))
=>
    (assert (MAIN::nearest PANEL ?n1 ?d1))
)
```

7.3.5 HYPOTHESISER module

The hypothesiser generates intentions (partial plans) from beliefs (concepts in WM) and goals. It should do this using backward chaining, or a planning algorithm, or implicitly from a reinforcement learning model of utility, or ...

For this demo, however, we encode two sample sets of intentions:

- Maintenance of lights on/off
- Relocation of lights - move nearer to citizen, or nearer to an interesting object

These intentions are too simple to really be called plans but they do for the purposes of this demo. Examples of more complex intentions are the protocol an agent uses to negotiate with another, and an ordered set of actions needed to construct a complex 3D structure in the world.

We have encoded the intentions. To demonstrate that Jess can write its own rules, and thusi that it could we can generate plans, we have one rule write the intention rules. The rules could also have been written from Java if desired.

```
(defmodule HYPOTHESISER)

(deftemplate MAIN::intention (slot id) (slot label))

(defrule HYPOTHESISER::initialise
    ?f <- (MAIN::reset-intentions)
    ;(MAIN::currenttime ?t)
=>
    (retract ?f)

    ;Intentions #1
    (bind ?mod (gensym*))
    (assert (MAIN::intention (id ?mod) (label PLAN1)))
    (call (engine) addDefmodule ?mod "")

    (build (str-cat "(defrule " ?mod "::" (gensym*)
       " ?o <- (MAIN::Object3D (object-type LIGHT) (object-id ?id) "
       "           (state HOLD-ON))"
       "(MAIN::currenttime ?t)"
       "(MAIN::last-action ?tlast&:(>= ?t (+ ?tlast 300)))"
       "=> (modify ?o (state ON)))"))

    (build (str-cat "(defrule " ?mod "::" (gensym*)
       " ?o <- (MAIN::Object3D (object-type LIGHT) (object-id ?id) "
       "           (state HOLD-OFF))"
       "(MAIN::currenttime ?t)"
       "(MAIN::last-action ?tlast&:(>= ?t (+ ?tlast 300)))"
       "=> (modify ?o (state OFF)))"))

    (build (str-cat "(defrule " ?mod "::" (gensym*)
       " (MAIN::pattern ?hr)"
       "(MAIN::currenttime ?t&:(=  (CONCEPTION::timeToHourNo ?t) ?hr))"
       "?o <- (MAIN::Object3D (object-type LIGHT) (object-id ?id) "
       "          (state OFF))"
       "=> (assert (MAIN::command ON))"))

    (build (str-cat "(defrule " ?mod "::" (gensym*)
       " (MAIN::currenttime ?t)"
       "(not (MAIN::pattern ?hr&:(=  (CONCEPTION::timeToHourNo ?t) ?hr)))"
       "?o <- (MAIN::Object3D (object-type LIGHT) (object-id ?id) "
       "          (state ON))"
       "=> (assert (MAIN::command OFF))"))


    ;Intentions #2
    (bind ?mod (gensym*))
    (assert (MAIN::intention (id ?mod) (label PLAN2)))
    (call (engine) addDefmodule ?mod "")

```

```
    (build (str-cat "(defrule " ?mod "::" (gensym*)
       " (MAIN::nearest PANEL ?n ?)"
       "(MAIN::Object3D (object-type LIGHT))"
       "(MAIN::Object3D (object-type PANEL) (object-id ?id) (location ?l))"
       "=> (bind ?x (+ (get-member ?l x) 90))"
       "(bind ?z (+ (get-member ?l z) 90))"
       "(assert (MAIN::light-move-action PANEL ?n ?x ?z)))"))

    (build (str-cat "(defrule " ?mod "::" (gensym*)
       " (MAIN::nearest CITIZEN ?n ?d&:(> ?d 1000))"
       "(MAIN::Object3D (object-type LIGHT))"
       "(MAIN::avatar (location ?l) (citizen-name ?n))"
       "=> (bind ?x (+ (get-member ?l x) 90))"
       "(bind ?z (+ (get-member ?l z) 90))"
       "(assert (MAIN::light-move-action CITIZEN ?n ?x ?z)))"))

    (assert (MAIN::current-intention nil))
)
```

The hypothesiser needs to choose between candidate sets of intentions according to goals and beliefs. Once again in this demo we have encoded rules to do this. The goals are:

- Provide lighting for nearby citizens
- Don't provide lighting unnecessarily
- Locate lighting where it is most useful

```
(defrule HYPOTHESISER::trigger-intentions-rule-1
    "Intentions 1 trigger when nearest target is already nearby"
    (declare (salience 0))
    (MAIN::intention (id ?mod) (label PLAN1))
    (MAIN::Object3D (object-type LIGHT))
    (MAIN::schedule (collection ?sched))
    (MAIN::nearest ?t ?n ?distance)
    (test (< ?distance 1000))
    ?i <- (MAIN::current-intention nil|PLAN2)
=>
    (?sched clear)
    (?sched addAll
           (call java.util.Arrays
                 asList
                 (create$ REFLECTIVE ?mod HYPOTHESISER CONCEPTION MEMORY
                          PERCEPTION REFLEXIVE)))
    ;The above could change to use a Jess list directly. It shows
    ;using Java containers
```

```
19
20      (retract ?i)
21      (assert (MAIN::current-intention PLAN1))
22      (pop-focus)
23  )
24
25  (defrule HYPOTHESISER::trigger-intentions-rule-2
26      "Intentions 2 trigger when nearest target is too far away"
27      (declare (salience 0))
28      (MAIN::intention (id ?mod) (label PLAN2))
29      (MAIN::Object3D (object-type LIGHT))
30      (MAIN::schedule (collection ?sched))
31      (MAIN::nearest ?t ?n ?distance)
32      (test (> ?distance 1100)) ;hysteresis - test not of 1000
33      ?i <- (MAIN::current-intention nil|PLAN1)
34  =>
35      (?sched clear)
36      (?sched addAll
37              (call java.util.Arrays
38                    asList
39                    (create$ REFLECTIVE ?mod HYPOTHESISER CONCEPTION MEMORY
40                             PERCEPTION REFLEXIVE)))
41      (retract ?i)
42      (assert (MAIN::current-intention PLAN2))
43      (pop-focus)
44  )
```

The hypothesiser should do some planning or backward chaining from desires, current intentions and beliefs to new intentions. An example that is left as an exercise is to replace the current HYPOTHESISER module with one that uses the backward chaining mechanism of Jess.

7.3.6 Action Activator modules

We don't have a distinct reactive module. Both reactive and reflective behaviours operate from the reflective module. The REFLEXIVE module implements actions that trigger off sense-data: behaviours that don't require beliefs. Reactive behaviour involve actions that trigger off percepts, and possibly concepts: they require beliefs but not intentions. Reflective behaviours require desires and intentions, or goals and partial plans to achieve them. For both reactive and reflective behaviours, however, the same actions are triggered. It is the antecedents of the rules that trigger them that change. This is the reason that there is no separate REACTIVE module in the rules that follow.

Also note that each action will involve one or more effector activations, as actions are a higher level construct than effect-data.

```
1   (defmodule REFLEXIVE)
2
```

```
;Currently we command a LIGHT to be on via chat as a reactive behaviour.
;When we write a click sensor there will be rules here for reflexive
;behaviours to switch the light on and off

(defrule REFLEXIVE::build-rule
    "Build the 3D street light"
    (declare (salience 0))
    (MAIN::kcdcc_awa_base_ReteAgent (OBJECT ?a) (location ?locn))
    ?f <- (MAIN::build)
    (MAIN::structure (structureId ?sid))
    ?c <- (MAIN::component (structureId ?sid)
                    (awNumber 0)
                    (xOffset ?xo)
                    (yOffset ?yo)
                    (zOffset ?zo)
                    (modelId ?mid)
                    (description ?d)
                    (action ?act))
    (MAIN::model (modelId ?mid) (name ?n))
    (MAIN::currenttime ?t)
    ?lt <- (MAIN::last-action ?)
=>
    (bind ?llocn (new Location6DF ?locn))
    (set-member ?llocn x (+ ?xo (get-member ?locn x)))
    (set-member ?llocn y (+ ?yo (get-member ?locn y)))
    (set-member ?llocn z (+ ?zo (get-member ?locn z)))

    (printout t "BUILDING" crlf)
    (bind ?eff (new AW3DObjectEffector))
    (?eff init ?a nil)
    (?eff reset)
    (?eff setCommand (get-member kcdcc.awa.base.AW3DObjectEffector ADD))
    (?eff setX (get-member ?llocn x))
    (?eff setY (get-member ?llocn y))
    (?eff setZ (get-member ?llocn z))
    (?eff setModel ?n)
    (?eff setDescription ?d)
    (?eff setAction ?act)
    (bind ?no (?eff activate))
    (if (neq ?no nil) then
       (bind ?no (?no intValue))
       (modify ?c (awNumber ?no))

       (assert (MAIN::Object3D (object-type LIGHT)
                        (location ?llocn)
                        (object-id ?no)
                        (owner ?a)
                        (model ?n)
                        (timestamp ?t)))
    )
    (retract ?f ?lt)
```

```
    (assert (MAIN::last-action ?t))
)

(defmodule REFLECTIVE)
(defrule REFLECTIVE::lightswitch-action-1
    (declare (salience 0))
    ?f <- (MAIN::command ON)
    (MAIN::kcdcc_awa_base_ReteAgent (OBJECT ?a))
    ?o <- (MAIN::Object3D (object-type LIGHT) (location ?llocn) (object-id
?id))
    ?c <- (MAIN::component (awNumber ?id)
                    (modelId ?mid)
                    (description ?d)
                    (action ?act))
    (MAIN::model (modelId ?mid) (name ?n))
    (MAIN::currenttime ?t)
    ?lt <- (MAIN::last-action ?)
=>
    (printout t "SWITCHING ON STREETLIGHT" crlf)
    (bind ?alocn (get-member ?a location))
    (bind ?newaction "create light type=point color=blue brightness=1")
    (modify ?c (action ?newaction))
    (bind ?x (get-member ?llocn x))
    (bind ?z (get-member ?llocn z))

    (bind ?eff (new AW3DObjectEffector))
    (?eff init ?a nil)
    (?eff reset)
    (?eff setCommand (get-member kcdcc.awa.base.AW3DObjectEffector CHANGE))
    (?eff setObjNo ?id)
    (?eff setOldx ?x)
    (?eff setX ?x)
    (?eff setY (get-member ?llocn y))
    (?eff setOldz ?z)
    (?eff setZ ?z)
    (?eff setModel ?n)
    (?eff setDescription ?d)
    (?eff setAction ?newaction)
    (bind ?no (?eff activate))
    (if (neq ?no nil)
     then
          (bind ?no (?no intValue))
          (modify ?c (awNumber ?no))
          (modify ?o (object-id ?no)))
    (modify ?o (state HOLD-ON))
    (modify ?c (action ?newaction))
    (retract ?lt ?f)
    (assert (MAIN::last-action ?t))
)

(defrule REFLECTIVE::lightswitch-action-2
```

```
105     (declare (salience 0))
106     ?f <- (MAIN::command OFF)
107     (MAIN::kcdcc_awa_base_ReteAgent (OBJECT ?a))
108     ?o <- (MAIN::Object3D (object-type LIGHT) (location ?llocn) (object-id
109   ?id))
110     ?c <- (MAIN::component (awNumber ?id)
111                     (modelId ?mid)
112                     (description ?d)
113                     (action ?act))
114     (MAIN::model (modelId ?mid) (name ?n))
115     (MAIN::currenttime ?t)
116     ?lt <- (MAIN::last-action ?)
117   =>
118     (printout t "SWITCHING OFF STREETLIGHT" crlf)
119     (bind ?alocn (get-member ?a location))
120     (bind ?newaction "create light brightness=0")
121     (modify ?c (action ?newaction))
122     (bind ?x (get-member ?llocn x))
123     (bind ?z (get-member ?llocn z))
124
125     (bind ?eff (new AW3DObjectEffector))
126     (?eff init ?a nil)
127     (?eff reset)
128     (?eff setCommand (get-member kcdcc.awa.base.AW3DObjectEffector CHANGE))
129     (?eff setObjNo ?id)
130     (?eff setOldx ?x)
131     (?eff setX ?x)
132     (?eff setY (get-member ?llocn y))
133     (?eff setOldz ?z)
134     (?eff setZ ?z)
135     (?eff setModel ?n)
136     (?eff setDescription ?d)
137     (?eff setAction ?newaction)
138     (bind ?no (?eff activate))
139     (if (neq ?no nil)
140      then
141           (bind ?no (?no intValue))
142           (modify ?c (awNumber ?no))
143           (modify ?o (object-id ?no)))
144     (modify ?o (state HOLD-OFF))
145     (modify ?c (action ?newaction))
146     (retract ?lt ?f)
147     (assert (MAIN::last-action ?t))
148   )
```

This next action contains 2 effector signals, demonstrating that ;actions are at a higher level than effects. The first effector is driven by method moveTo on the Society. Note the flaw with this: there are no rules to reset the AW3DObjectSensor to the new location

```
(defrule REFLECTIVE::light-move-action-1
    (declare (salience 0))
    ?f <- (MAIN::light-move-action ?trigger ?triggerid ?x ?z)
    (MAIN::kcdcc_awa_base_ReteAgent (OBJECT ?a))
    ?o <- (MAIN::Object3D (object-type LIGHT)
                          (location ?llocn)
                         (object-id ?id))
    ?c <- (MAIN::component (awNumber ?id)
                     (modelId ?mid)
                     (xOffset ?xo)
                     (zOffset ?zo)
                     (description ?d)
                     (action ?act))
    (MAIN::model (modelId ?mid) (name ?n))
    (MAIN::currenttime ?t)
    ?lt <- (MAIN::last-action ?)
=>
    (printout t "MOVING STREETLIGHT" crlf)
    (bind ?oldx (get-member ?llocn x))
    (bind ?oldz (get-member ?llocn z))

    (bind ?alocn (get-member ?a location))
    (set-member ?alocn x ?x)
    (set-member ?alocn z ?z)
    ((?a getSociety) moveTo ?alocn)

    (bind ?x (+ ?x ?xo))
    (bind ?z (+ ?z ?zo))

    (bind ?eff (new AW3DObjectEffector))
    (?eff init ?a nil)
    (?eff reset)
    (?eff setCommand (get-member kcdcc.awa.base.AW3DObjectEffector CHANGE))
    (?eff setObjNo ?id)
    (?eff setOldx ?oldx)
    (?eff setX ?x)
    (?eff setY (get-member ?llocn y))
    (?eff setOldz ?oldz)
    (?eff setZ ?z)
    (?eff setModel ?n)
    (?eff setDescription ?d)
    (?eff setAction ?act)
    (bind ?no (?eff activate))
    (if (neq ?no nil)
     then
          (bind ?no (?no intValue))
          (modify ?c (awNumber ?no))
          (modify ?o (object-id ?no)))

```

```
50      (retract ?lt ?f)
51      (assert (MAIN::last-action ?t))
52   )
```

7.4 Testing `Streetlights`

This is a kind of test schedule that both illustrates the usage of the agent and, by the things not tested, shows what has not been debugged yet. It assumes that only your citizen is logged into the world. Adapt the tests accordingly if that is not the case.

1. Start your browser, move near to where you configured the agent location, and look in that direction.

2. From an MS-DOS command window, run the BAT file for the society. (It is a good idea to set the ecreen buffer height property of the window to a large number, such as 500.)

3. Wait for the first time sense-data (about 1 minute) until something like the following appears

   `(MAIN::currenttime 10355125180)`

   You should also see in the trace the sense-data from your avatar and nearby 3D objects. The streetlight should then appear.

4. From the chat window of the browser, type

   `command lights on`

   The light will turn on. (When the click sensor is written you will also be able to click on the light.) Type

   `command lights off`

   to turn it off. From the trace you should be able to see what rules fired, and what facts were asserted and retracted.

5. From the browser, moveaway from the light. You should see a fact something like

   `(MAIN::nearest CITIZEN "Greg" 819.20...)`

   with the number (the distance) increasing as you move away. Keep moving until the distance is more than least 1100.

6. Wait for the next time period (the next time sense-data). Then

   `(MAIN::current-intention PLAN1)`

   changes to

   `(MAIN::current-intention PLAN2)`

   Wait for the next time period and the light will move near to your avatar (you may need to turn to see it).

Note that these rules are still fragile. That is, if you keep moving whilst the light is relocating it may get confused (PLAN2 needs more work).

7. Move back to your original locationi and wait for the light to move back.

8. Insert a new 3D object `pan1101` next to the light. A fact like the following will be asserted

   ```
   (MAIN::nearby PANEL 1318005020 50.0)
   ```

9. Exit the browser. A memory fact like

   ```
   (MAIN::pattern 26)
   ```

   will be asserted.

10. Run the browser again and command the lights off (even though they are already off - we are changing state from `OFF` to `HOLD-OFF`). Look for a fact like

    ```
    => (MAIN::last-action 1035523896)
    ```

    Add 300 (seconds) to the number at the end of this asserted fact. Wait for the next `currenttime` fact that is asserted with at least this value. At the next time period

    ```
    (MAIN::Object3D (object-type LIGHT) ...  (state HOLD-OFF) ..)
    ```

    will change to

    ```
    (MAIN::Object3D (object-type LIGHT) ...  (state OFF) ..)
    ```

    The PLAN1 intentions will then command the lights on (because a sufficient time has passed since the chat command OFF and the pattern of past usage had the lights on at this time of day).

    Note that this last test will not work unless at least 1 citizen has logged on *and out* with the same hour of day as is the current time. This is important here because we haven't implemented persistence yet for this agent (for example, we have demo agents that access a MySQL relational database but this agent does not).

8 Release Notes

8.1 Changes in v0.2.2

then no avatar is set for the society. Instead, it goes into global mode. This only works for AW servers at version 3.3 or later.

Rationalised `Object3DSenseData`. It is now abstract, with new sense-data `DeletedObject3DSenseData` and `ClickedObject3DSenseData`. The same thing will soon be done to the avatar sense-data.

another location, and possibly another world. The owner of the agent must be a caretaker or have the EJECT right to use this effector. The agent would also need to have an `AWAvatarSensor` to use this sensibly.

8.2 Changes in v0.2.1

Parameter "recognisedCitizens" in sensor `AWChatSensor` is now optional. If present it should be a set of names of citizen that are allowed to chat with the agent. If not present then all citizens are recognised (it is assumed that recognition and filtering will occur in perception in this case). Filtering recognised citizens at the sensor stage is useful for (i) performance reasons (ii) limiting who can issue text commands.

The properties file is optional. You can now alternatively put the same property definitions on the java command line in the BAT file.

There has been a change in `TimeSensor`. There is a parameter `period` which is the interval in seconds at which sense-data should be generated. There is new sensor `VRTTimeSensor` that extends `TimeSensor`. This provides VRT (virtual reality time, or time in seconds since midnight Jan 1st 1970 in timezone GMT-2hrs).

Change in `awa.lib` to correct bug in AW3DObjectEffector JNI implementation.

When calling `init` on an effector, you can now specify `null` on the `params` parameter if there are no properties.

There is a new property on `ReteAgent` called traceEnabled. Set this to `TRUE` to get a trace on the standard output that is like the Jess `(watch all)`. For example, add `(modify ?a (traceEnabled TRUE))` to the RHS of an initialisation rule, where `?a` is a fact `kcdcc_awa_base_ReteAgent`.

In the XML configuration, element $\langle session \rangle$ is now $\langle society \rangle$.

The names of some sensors and effectors has changed: see Sections 4 and 5.

Package `jakarta-oro-2.0.3.jar` is no longer required.

8.3 Changes in v0.2

Avatar click and delete in `AWAvatarSensor` now work. `AWLookSensor` and `AWLookEffector` have been written: see the demo agent `voyeur` for how to use them.

`Location6DF` is a new class that extends `Location`. It is a six degrees of freedom location: (x,y,z) position plus (roll,yaw,tilt). Other classes have been updated to match.

As the conference room demo is still in flux, it hasn't been included.

8.4 Changes in v0.1.1

This has been compiled and tested using the java 1.4.0 SDK from Sun. It is a partly tested version of v0.2. The same set of java classes exist as in v0.1 but there is an additional class `ReteAgent`. This loads a configured set of sensors, using exactly the same code as is used by `agent`, but it pushes the `SenseData` into Jess working memory instead of pushing into `Perceptors` and having the `Perceptors` push into Jess working memory. A `Society` can contain both the `ReteAgents` and the other kinds of `Agent`.

I have rewritten the existing AW sensors and effectors to use AW SDK build 24 directly. That is, the agents no longer use the wrapper from INSEAD. This is cleaner, it does not depend on old unmaintained code from somewhere else, and the INSEAD wrapper was only for build 15 of the SDK. This means that the SDK methods to get and set SDK attributes are hidden inside java native interface (JNI) methods. Users of the agent will only see `SenseData` and effect-data corresponding to `Sensors` and `Effectors`. Note

that the JNI needs to be written in C and you need to understand multithreading, deadlock etc. So the base java package will contain a standard set of configurable sensors and effectors; everything else can be done from Jess if desired. Users should be able to use any methods or events in the current SDK provided a `Sensor` or `Effector` has been written for it.

Each sensor constructs `senseData` that is a java bean and is pushed into Jess working memory as a static definstance. The Jess perception rules will have `SenseData` java beans in their antecedents, and will remove the sense-data from WM as soon as new percept facts are asserted. There will also be an `Agent` definstance but it will be dynamic. The names of these definstances are derived from the java package name. For example, a `kcdcc.awa.base.ReteAgent` agent will have a Jess fact tag `kcdcc_awa_base_ReteAgent`, and a `kcdcc.awa.base.TextMessageSenseData` java bean will have a Jess fact tag `kcdcc_awa_base_TextMessageSenseData`. So, as these java beans are what end up as facts in Jess WM, deciding what data from the SDK looks like as sense-data is important.

The sensors and effectors are now multithreaded (previously all SDK input and output had to be chanelled through the one thread because of a limitation in the JNI from INSEAD). The infostation demo is included in the zip file but the conference room demo has not. Once the conference room demo is updated an example of inter-agent communication will bve available. Likewise with the look sensor. Note that the demo infostation uses a java helper classs ShedHelper. This isn't because it has to, it is just because I already had this code (but organised differently). It could be all done in Jess (a useful execrcise for someone?).

The `AWLookSensor` and `AWLookEFFector` do not work yet, nor do avatar click or avatar delete. These (when they work) use the SDK query mechanism to look at what objects are in the world near the agent.

The documentation is yet to be updated.

8.5 Initial version 0.1

If you go to the documentation, click on package `kcdcc.awa.demos` in the right hand frame and click on `ShedActivator`. You will find there links for the xml and properties file. You will need to edit the properties file to add your citizen number and privilege password. You can rename it but change it in the batch file above to match. Similarly you can copy and/or edit the xml - you will need to change any pathnames.

You can run the infostation demo without the database: remove sensor `SQLSensor` and perceptor `SQLStructurePerceptor` from the xml configuration, and don't commands "command build" or "command demolish" (see to the documentation for `kcdcc.awa.demos.ShedActivator`). If you do this the agent will only move the avatar around until you use command "command die", at which point it will die.

The other demo, the (only partly finished) conference station does not use an external database but it does demonstrate multiple agents communicating.

Finally, the xml configuration has a `codebase` attribute. Set this to the path to where you put your files. This is still not tested, however, so don't expect too much of the jar loading. If someone wants to write an all bells and whistles jar loader ...