Creating a New Virtual Human with the FLoReS Dialogue Manager

Disclaimer

Although this module is not part of the vhtoolkit distribution, it is compatible with it and can be used when more flexibility is desired for natural language understanding, generation and dialog management. The module is available open source from this github repository.

Getting started

1. Clone the jmnl github repository.
2. Install Eclipse.
3. Install Java at least 8.
4. In eclipse, import the existing project defined in the github repository.
5. Run the main in `edu.usc.ict.nl.ui.chat.ChatInterface` with arguments `-s chatInterface.xml`.

Overview

the process of creating a character is iterative in nature. The following steps create an initial character that will probably need to be refined by repeating the steps as required to obtain the desired behavior.

In general, three actions are required to create the content necessary to drive the natural language component:

1. Create an initial set of system and user utterances. Each utterance needs to be associated to an identifying string (like a name), usually referred to as speech act. The natural language understanding (NLU), dialog management (DM) and natural language generation (NLG) modules will use these identifiers instead of the real utterances.
2. Create a dialogue policy. This policy consists of:
    a. the set of variables that constitute the dialogue state (the dialogue manager is an information based one. That is it decides what to say based on what the user said and the current state of the conversation as represented in the dialogue state (also called information state)).
    The information state can also contain: forward inference rules (e.g. `brother(x,y)=>sibling(x,y)`) and predications like `brother(Person1,Person2).`
    b. the set of sub-dialogue networks. These sub-dialogues are similar to planning operators, with preconditions and effects. The dialogue manager (DM), will select one based on what the user said and the current dialogue state. Each sub-dialogue typically carries out a short portion of an entire dialogue (e.g. the greeting phase, or answering a question). sub-dialogues roughly corresponds to functions in programs, if a set of dialogue steps often happen together and multiple times, it's worth grouping them into a sub-dialogue.
3. Train the natural language understanding module to map a given utterance to one of the known identifying strings (speech acts) defined in the first step.

The entire information that defines a character, e.g. CakeVendor, for the FLoReS system is defined in a set of files sitting in the directory resources /characters/CakeVendor/.

This directory contains three sub-directories that parallel the 3 steps defined above:

- **content**: this directory contains the files that define the user and system utterances and their identifying strings.
- **dm**: this directory contains the dialogue policy
- **nlu**: this directory contains the natural language understanding model learned by the corresponding module from the data in the data found in the content directory

Step 1: authoring the content

Authoring the content consists of editing 2 files. One for the user utterances and one for the system utterances. The file that contains the user utterances is basically the training data for the natural language understanding (NLU) module. The system utterance file instead contains the utterances that the character can say.

User utterances:

The NLU module given an utterance returns the most probably identifying strings. It's based on a maximum entropy multiclass classifier and therefore the user utterance file should list utterances maintaining their natural frequency. That is, the best way to obtain these utterances is by running wizard of oz experiments or role plays. Then annotate the data by assigning to each utterance said by a user during these experiments an identifying string. These identifying strings are sometime called speech acts or dialogue acts (in case more domain specific semantic is attached to the basic speech act). Examples of dialogue acts are: question.age to mark all utterances in which the user is asking about the age of the addressee.

When we design a dialogue policy for the character using this content, whenever we want to wait for the user to say a certain utterance, we will use the string identifier (speech act) associated to that utterance.

System utterances:

These utterances are the one the system can say. Similarly to the user utterances, each utterance has a specific string identifier. When designing the dialogue policy, if we want to say a certain system utterance, we will use the corresponding identifier (also here we call the identifier speech act).

File format:

The user and system utterances files use the same Excel spreadsheet format. These files have a number of columns (these are the initial 2 rows of the system utterances file for the character used in the example below):

TTERANCE_ID	VERSION	CHARACTER	STATE	SPEECH_ACT	TEXT
				statement.not-understand	I'm sorry, I didn't understand what you said. Please try to rephrase it.
				greeting.hello	Hello

The only 2 columns of relevance are SPEECH_ACT and TEXT.

SPEECH_ACT contains the string identifier for the corresponding utterance found in the TEXT column.

The user utterance file needs to be called user-utterances.xlsx and the system utterance file must be called system-utterances.xlsx (these names can be configured, but the default configuration looks for those names in each character available).

Step 2: Authoring the dialogue policy

Overview

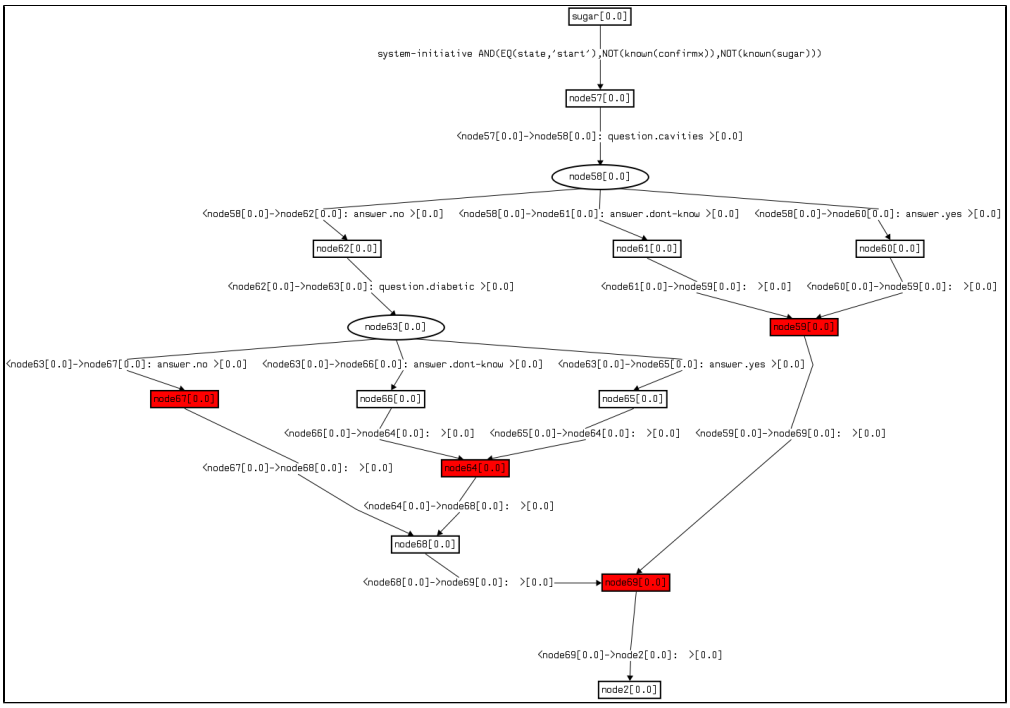
The FLoReS (Forward Locking Reward Seeking) dialogue manager is an information state and event driven dialogue manager. That is, it does nothing unless an event is received. When an event is received it searches for the best action (i.e. sub-dialogue) that can be executed in the current information state and that achieves the highest expected reward. Once the best action is found it starts executing it. Unless:

- the current action in execution is the same as the best action found. In that case the dialogue manager simply continues to execute the action.
- there is no best action. That can happen in 2 cases:
    - there are no actions that can be executed given the current event and information state.
    - all executable actions have negative expected reward

As mentioned earlier the dialogue manager searches for the best available action every time an event comes in. Events are generated by multiple sources, for example, the user typing, the system's timer, additional connected modules (e.g. a speech recognizer).

Operators (sub-dialogues or actions):

Actions are also called sub-dialogues and define dialogue trees. For example this is one sub-dialogue found in the CakeVendor example below:



These sub-dialogue trees define a small self-contained portion of conversation. the criteria to use to decide what should be a sub-dialogue is similar to the criteria used to decide what should be a function or method in a programming language: generality and reusability.

For example, the sub-dialogue above takes care of finding out whether the system can sell to the user a cake with normal sugar or with Xylitol based on collecting information about the user having many cavities or having diabetes. Because the utterances found in this sub-dialogue can happen only in that specific context, then it makes sense to keep them in the same sub-dialogue.

A sub-dialogue can be in 3 states: ACTIVE, INACTIVE and DORMANT.

At any time in the system there is at most 1 active sub-dialogue: the current action. As said above, in some cases there may be no active actions. All actions are normally inactive, unless they have been active and they have been substituted (swapped-out) by another action before their natural termination (that is, at some point the system found a better action and so changed the state of the current action to dormant and made the newly found best action as active). Not all actions that are active and are swapped out for a new best action can become dormant. Some will go back directly to the inactive state. An action, to be allowed to become dormant, must have special entry paths that allow for it to be awoken back to the active state in case it becomes again the best action.

Entry paths:

A sub-dialogue has multiple entry paths. The entry paths have a specific order (decided by the author) and each entry path has conditions to regulate when it can be taken and has also a start state. That is when the system during the search for the best available action considers a certain sub-dialogue, it'll considers all the possible entry paths in the order specified. The first that has satisfied conditions will be taken and it'll start the execution of the action at the specified start state in the sub-dialogue tree.

The possible types of entry paths are:

1. user event entry paths: a user event entry path defines an entry path that can be taken only when certain specified events are received. A user event entry path can also have an optional condition on the information state. This optional condition is a Boolean expression of information state variables.

2. system initiative entry paths: these entry paths have no events associated with them. They can have optional information state condition as for the user entry paths. These entry paths are used to give to a certain sub-dialogue the possibility of being initiated by the system no matter what event has been received. For example one can define a periodic timer event that wakes-up the dialogue manager periodically even if the user is inactive.
3. re-entry paths: these paths allow an action that becomes dormant to become active again. Each re-entry path like the other can have an optional information state condition. It can also have an optional system utterance identifier associated to it. This system utterance will be said when the associate re-entry path is taken. An example of a system utterance appropriate for a re-entry path is something like "coming back to where we were..." or "so, i mentioned earlier...".

We refer to the entry paths with their conditions also as preconditions as that is the name traditionally used by the planning community.

Nodes and edges:

The edges of a sub-dialogue tree are of three types:

1. user edges: these edges tell the system to wait for a certain event before traversing them. If a state has one outgoing edge that is a user edge, then all outgoing edges of that state will be user edge. this property make a state a user waiting state that blocks the execution of the action until the user says any of the events in the outgoing user edges.
2. system edges: these edge when traversed make the system say a particular utterance. System edges take time to be traversed: the time taken by the associated system utterances to be played (one can configure to ignore this waiting but the default is to wait for a system edge to finish playing the associated animation). System edges can be of three types:
    a. a normal speech act given as a constant string. The DM will simple send a request to the NLG to create surface text for the given speech act.
    b. an interruptible speech act. This is a system line for which we are ready to receive an interruption. That is, if the user says something we except to prioritize what the user says and interrupt if we have an interruption policy in place that generates an interrupt request.
    c. an evaluation system action: this action has as argument an expression that needs to be evaluated and its result must be a string to be handled like the normal case above.
3. condition edges: these edges are used to connect state when we don't want to wait for an event and we don't want the virtual human to say anything.
4. wait edges: an edge that does nothing but wait a specific amount of time.

Nodes contain effects. There are three types of effects:

1. an information state update. That is, changing the value of some variable in the information state when the node containing the effect is entered.
    a. like an assignment or an assertion.
2. a reward. A reward can be a numeric constant or an expression returning a number. When the state containing a reward is reached, the system achieves the associated reward. A sub-dialogue can have multiple rewards associated to multiple states.
3. swap out the current sub-dialog (force the sub-dialogue to go from ACTIVE to DORMANT state).
4. a request to interrupt the current system action
5. sending an internal message
6. sending a VH protocol message

In the example of sub-dialogue given here the red nodes are states with effects. These states can be inspected to display the particular effects associated with them. This graphical representation of a sub-dialogue is generated for debug purposes, it's not used to edit the sub-dialogue, just to check that the intended form is correctly generated from the provided information.

End node:

Each sub-dialogue is terminated when the execution path reaches a node that has no more outgoing edges.

Final sub-dialogue:

Each sub-dialogue can be marked final. That means that when the end node of a final sub-dialogue is reached, the conversation ends. When the conversation ends the DM will ignore all events and the user will not be able to interact with the virtual character anymore.

Daemon sub-dialogue:

A sub-dialogue can be also marked as a daemon sub-dialogue. These subdialogues are extensions of the event listeners, they can have system initiative entrance conditions and do more complex processing and information state updates.

Execution

Execution of a sub-dialogue consists of taking a certain entry path (the one that lead to the maximum expected reward) and then at every node, take the first outgoing edge (the order is from left to right and is specified by the author) that can be taken (that is has a satisfied condition) until we reach a waiting point: a user state (i.e. a state with user outgoing edges). At that point the dialogue manager terminates the execution and waits for the next event. If the incoming event is one of the expected events (i.e. the events specified in the user edges) then the execution continues along the first satisfied user edge. If the final node is reached, the sub-dialogue is terminated and becomes inactive and the system searches for a new optimal action to start executing.

Information state:

The information state is formed by variables and stores the current state of the conversation. Four things can update the information state:

1. the **dialogue manager** (DM) takes care of updating a set of special variables (e.g. the time since the last user action). These special variables can be found in a file called specialVariables.xml in the dm sub-directory. The file is automatically generated every time the DM starts.
2. **event listeners**: one can associate to certain events automatic updates that are executes every time a particular event is received. these updates are also called state less updates because they happen regardless of the current action or best selected action.
3. **effects**: as described in the effects section above, a sub-dialogue node can have a specific effect to update the value of a certain variable.

4. **forward inference rules**: one can specify an ordered list of implications. They are executed every time a change is made to the information state. when one is found in which the antecedent of the implication is true, the consequent is executed. For example, give the rule "if A then B else C" if A is true, then B is executed otherwise C is executed. The else part is optional. A is a Boolean expression. B and C are assignments.

Special variables:

The dialogue manager has a predefined set of special variables it updates automatically and that can be used in a dialogue policy, if needed.

This list of special variable can vary with each version of the dialogue manager. The list is printed out in a file called specialVariables.xml in the dm sub-directory. The content is for example:

```
<sv id="timeSinceLastUserAction" value="0" type="NUMBER" desc="Time in seconds since the last thing said by the
user."/>
<sv id="timeSinceLastSystemAction" value="0" type="NUMBER" desc="Time in seconds since the last thing said by
the system."/>
<sv id="consecutiveUnhandledUserActions" value="0" type="NUMBER" desc="Number of consecutive user actions for
which the system had no direct response (handler)."/>
<sv id="timeSinceLastAction" value="0" type="NUMBER" desc="Time in seconds since anyone said something (user or
system)."/>
<sv id="timeSinceLastResource" value="0" type="NUMBER" desc="Time in seconds since the last resource link/video
was given."/>
<sv id="event" value="null" type="TEXT" desc="Name of last speech act said by the user and processed by the
system."/>
<sv id="lastNonNullSubdialog" value="null" type="TEXT" desc="Name of last sub-dialog executed by the system."/>
<sv id="systemEvent" value="null" type="TEXT" desc="Name of the speech act last said by the system."/>
<sv id="timerInterval" value="1" type="NUMBER" desc="Time in seconds between 2 consecutive timer events."/>
<sv id="preferForms" value="true" type="BOOLEAN" desc="If true and a form is available for the current system
speech act, the form will be selected by the NLG."/>
<sv id="tmpEvent" value="null" type="TEXT" desc="Variable used to store the input event that generated one of
the internal events (e.g. unhandled, ignore and loop)."/>
```

Each line found in that file shows a special variable. `id` is the name of the variable to be used if you want to refer to that variable in your dialogue policies. v`alue` is the initial value given to that variable at start-up. `type` specifies the type of the variable (just to give you an idea, variables are untyped so you can change if want what that variable stores, but when the dialogue manager will automatically update that variable it'll write again something that belongs to the predefined type found in this description). `desc` contains a textual description of what that variable contains.

Dialogue policy execution

When an event is received, the dialogue manager (DM) checks to see if it is expected by the current action (i.e. the current action is at a user node and one of the user outgoing edges is waiting for the received event). If the current action is waiting for the received event the DM will continue the execution of the current action. Otherwise it'll execute a forward search to find the best action to execute. The forward search simulates possible future conversations. It's a breath first search and it's limited by time and depth (i.e. it'll always return quickly even if the search space is huge). Currently the limits are: 250ms or 10 levels maximum (i.e. the dialogue manager terminates the search for the optimal action after 250ms or if the search graph that represents the possible future conversations reaches a depth of 10 sub-dialogues, that is the search had enough time (i.e. within the 250ms timeout) to explore all possible conversations made up using a sequence of 10 sub-dialogues).

when an event is received, the forward search has two possible scenarios to consider:

1. Ignore the received event: here the dialogue manager searches for the most promising system initiative operators. Two sub-searches are executed:
    a. consider whether the best operator is to keep the current active operator as it is, and
    b. consider all other system initiative operators.
2. Handle the received event: here the dialogue manager searches for the most promising operator among those that are paused or inactive and that handle the received event.

The best action is the one that maximizes the expected reward. More precisely the formula is:

$$E[O_i, I] = \sum_{I_i \in I_r} \left(\alpha \cdot P(I_i) \cdot R(O_i, I_i) + \arg\max_{O} \left(E[O, I_i] \right) \right)$$

$E[O_i, I]$ is the expected reward of operator O_i in state I

α is a discount factor to penalize future actions

$P(I_i)$ is the probability of reaching the operator's final state I_i

$R(O_i, I_i)$ is the reward achieved by reaching the final state I_i

The preconditions are used to limit which sub-dialogues can be executed in a given state. Rewards are used to differentiate among a set of executable sub-dialogue.

The policy format

The dialogue policy is composed by several files. The main file that defines it is called policy.xml (also this name can be configured, but this is the default name).

A typical policy.xml file will look like the following:

policy.xml

```
<policy xmlns:xi="http://www.w3.org/2001/XInclude">
 <xi:include href="macros.xml"/>
 <xi:include href="initKB.xml"/>
 <xi:include href="goals.xml"/>
 <stepDiscount value="0.9"/>
 <include href="textFormat/policy.txt"/>
</policy>
```

line 2 specifies the file used to define macros used in effects (events and information state updates).

line 3 specifies the file used to define all the variables in the information state and to initialize them.

line 4 specifies the file that defines the basic value of the rewards available in this dialogue policy.

line 5 specifies the discount factor alpha mentioned in the expected reward formula. The line given above defines a discount factor of 0.9.

line 6 includes a file that specifies some operators (actions/sub-dialogues) in a particular text format. One could specify the sub-dialogue trees directly in a xml variant but it's harder and so we prefer to document how to design operators using this special text format. One can have any number of text format files included. When designing complex characters, it is helpful to organize the sub-dialogues in multiple files.

To include multiple files just duplicate line 5 for each different text format file that needs to be included.

The information state initialization file:

The following example shows the format of the information state initialization file:

Information state initialization file format

```
<informationState>
   <initialize expr="assign(lastNonNullSubdialog,null)"/>
   <initialize expr="assign(timeSinceLastAction,0)"/>
   <initialize expr="assign(alreadyAsked,false)"/>
   <initialize expr="imply(AND(>(delta-symptom_worried,0), deployed), ++(ptsd_counter, delta-symptom_worried))"
/>
 ...
</informationState>
```

It is a collection of lines to assign initial values to a set of variables. If a variable is used in a sub-dialogue, it must be defined in this file otherwise an error will be generated indicating which variable is undefined and where it was used.

For example, the line: `<initialize expr="assign(lastNonNullSubdialog,null)"/>` defines the variable `lastNonNullSubdialog` and assigns to it the value `null`. The syntax of the information state update is described in this section.

Normally all initialization entries are assignments (to give an initial value to a particular variable). However, one can use also another construct called an implication. This instead of defining and initializing a variable stores a forward inference rule in the knowledge base (as mentioned in the information state section). For example, the implication defined with `imply(AND(>(delta-symptom_worried,0), deployed), ++(ptsd_counter, delta-symptom_worried))` creates a forward rule that evaluates the increment `++(ptsd_counter,delta-symptom_worried)` every time there is an update to the information state and the condition `AND(>(delta-symptom_worried,0), deployed)` is satisfied.

A note about values:

Variables in the information state can have various types of values:

- A number like in `assign(timeSinceLastAction,0.3)`
- A string like in `assign(name,'John')`
- A Boolean like in `assign(notTrue,false)`
- No value like in `assign(unknown,null)`

- an hash table line in `assign(tmp,set(answer2questionmap,'answer.question.1','question.1'))` this assignment is equivalent to a java statement: `tmp=answer2questionmap.put('answer.question.1','question.1')`
- a list like in `assign(tmp,`topic(?)) here `tmp` will contain all arguments, `x`, for which `topic(x)` is true in the information state.
- A java object can be assigned but only programmatically

Syntax of conditions, formulas and information state updates:

Here we describe the syntax used to define conditions and effects. Typically we use a prefixed syntax where the operator or function is specified first followed by its arguments. For some operators an infix version is also available. when unsure, use the prefix version. Formulas and variable names are case insensitive. internally all names are lowercase. You can still use syntax like thisIsALongVarName to facilitate human reading, but internal it makes no difference and if one uses a version like thisisalongvarName it'll work fine.

Conditions

We call conditions all expressions that have as result a Boolean value. Conditions are used in the definition of entry paths or as the first argument of an implication. They are also used as second argument of assignments (to assign a variable a Boolean value).

For conditions we have the classic comparison operators >, <, >=, <=, == and the usual Boolean operators: AND, OR, NOT (also available as ! and ~). The comparison operators work as expected with strings and numbers (no special operator to compare strings).

For example, `AND(>(delta-symptom_worried,0), deployed)` defines a conjunction of 2 formulas: `>(delta-symptom_worried,0)` satisfied when the variable delta-symptom_worried is greater than 0 and `deployed` satisfied when the variable `deployed` contains the value `true`.

Effects

By effects here we mean just the information state updates. Effects in sub-dialogues can also be rewards. But here we are describing the syntax of the effects that update the information state. They can be of two types: assignments and implications. Implications are conditional assignments, that is they execute an assignment only if a particular condition is true (they have also an else portion).

Assignments

Assignments are used to update the value of a specified variable. They take 2 arguments: a variable being assigned and a formula returning a value to be assigned to the first argument. For example `assign(lastNonNullSubdialog,null)` assigns the value `null` to the variable `lastNonNullSubdialog` . The special `assign` operator is available also in infix form as =. The formula `assign(var1,2)` is equivalent to `var1=2`.

There are a couple of assignments for which a special syntax is available:

- **Increments**: assignments that increment the value of a number variable. These assignments can be done using the ++ operator. This operator can take 1 or 2 arguments. `++(var1)` increments the variable `var1` by 1. `++(var1,2)` increments `var1` by 2. The second argument, if present, can be a variable or complex expression that is evaluated to provide the increment value. The syntax `++(var1,var2)` is equivalent to `assign(var1,+(var1,var2))`.
- **Assertions**: this are assignments to a variable of a Boolean value. For example, if we want to make a certain variable `true` then instead of executing `assign(var1,true)` we can execute `assert(var1)`. If we want to make a variable false we execute `assert(!var1)` or `assert(NOT(var1))`.

Implications

Implications are used to define conditional assignments. An implication takes 2 or 3 arguments: a condition and 1 or 2 assignments. For example, `imply(==(var1,2),assign(var1,3),assign(var2,4))`executes `assign(var1,3)` if `==(var1,2)` is true, otherwise it executes `assign(var2,4)`. the third argument (the else part) is optional and can be omitted.

Special functions (aka Custom functions)

Special functions can be added by implementing the interface `edu.usc.ict.nl.kb.cf.CustomFunctionInterface`. Special functions are a way to define new functions by associating arbitrary Java code to a certain string. At the moment the following special functions are defined:

- Hash functions:
  - `newMap()`: this function creates a new hash table.
  - `clear(var)`: empties the hash table stored in the variable `var`.
  - `get(var1,var2)`: returns the value associated to the key `var2` in the hash table `var1`.
  - `set(var1,var2,var3)`: sets the value `var3` to the key `var2` in the hash table `var1`.
- List functions:
  - `get(var1,var2)`: returns the value associated to index `var2` in list `var1` (index can also be the string `"random"` in that case the function returns a random element of the list).
  - `exists(var1,var2,var3)`: returns true iff there exists an element of `var2` for which `var3` is true when substituted to the variable named `var1`.
  - `intersect(var1,var2)`: computes the intersection between the two given collections.
  - `len(var1)`: returns the length of the given list.
  - `removeIf(var1,var2,var3)`,`removeIfNot(var1,var2,var3)`: returns the list formed by the elements of the list `var2` for which the boolean expression `var3` is false (,true). `var1` is the loop variable.
  - `set(var1,var2,var3)`: sets the value `var3` at position `var2` in list `var1`.
  - `subtract(var1,var2)`: removes all the elements in the list `var2` from the list `var1`.
  - `union(var1,var2)`: computes the union of the two lists.
- String functions:
  - `match(var1,var2)`: maps to the String.matches(regexp) Java method. var1 must be a string or evaluate to one. var2 must be a string or evaluate to one. The content of var2 must be a valid Java regular expression.
  - `concatenate(var1,...,varn)`: concatenates the provided strings.

- Time functions:
  - currentTime(): returns the current time in milliseconds since 1/1/1970.
  - `getLastTimeMark(var1)`: returns the last time (in milliseconds since 1/1/1970) the current operator was in state `var1`, where `var1` can be either "DONE" or "ENTER".
  - `getLastTimeMark(var1,var2)`: returns the last time the current operator said the speech act `var2`. `Var1` must be "SAY".
- Ordering:
  - `follows(var1,var2)`: var1 is a string constant (or a variable with a string constant as value) and var2 is a boolean (or a variable with a boolean value). Var2 is optional, by default it's false. The function returns true if the operator named by var1 has already been executed. If var2 is true, then the function returns true only if the operator named by var1 has already been completed (that is, any final state in the operator has been executed (as opposed to being swapped out before completion)).
- Topic:
  - `isCurrentTopic(var)`: returns true if the provided string or variable containing a string matches one of the topics of the sub-dialogue currently active.
  - `isLastNonNullTopic(var)`: similar to `isCurrentTopic` but executes the match on the last non null topic. That is, if currently there are no active networks, this will match the value of var with the topic of the last active network.
- Numbers:
  - `min(var1,...,varn)`,`max(var1,...,varn)`: returns the min/max of the given list of numbers.
  - `random(var)`: generates a random number from 0 to the value in `var`-1. `var` doesn't have to be a variable but can also be a numeric constant.
  - `round(var)`: returns the output of java.lang.Math.round applied to the input argument when converted to a float value.
- Debug:
  - trace(var): prints a java stack trace when var is evaluated.
  - print(var): prints the value of var when the expression is evaluated by the system.
- Other:
  - `if(var1,var2,var3)`: return the evaluation of `var2` if `var1` evaluates to true, if `var1` evaluates to false it returns the evaluation of `var3`. null if `var1` returns null.
  - `known(expr)`: this returns true of the provided expression evaluates to anything but the NULL value.
  - `numToString(var)`: returns the string representation of the given number. For example, it returns "twenty three" for 23.
  - `hasBeenInterrupted(var)`: returns true if the current operator has been swapped out by an interruption.
  - `isInterruptible()`: returns true if the current transition being executed is interruptible (by the user).
  - `isQuestion(var)`: returns true if the provided var evaluates to a string that contains the string "question". This maps to the method `edu.usc.ict.nl.io.NLU.isQuestion` overwrite with your own specific NLU class if you want to customize or write a new custom function.

Quotation

Delayed evaluation is available using the special operator `quote`. For example, if we execute this assignment `assign(expr1,quote(+(var1,var2,3)))` we save in the variable `expr1` the expression that computes the sum of `var1`, `var2` and the constant 3. every time we use the variable `expr1` it's like if we use the entire expression it contains. If we later write the condition `>=(expr1,34)` it's equivalent to the condition `>=(+(var1,var2,3),34)`.

Macros

Macros can be defined to name complex expressions used in conditions and effects. The system also supports templates. For example,

```
<formulamacro left="isAvailable(topic)" right="exists(m3,question(topic,?),or(!known(answered('other',m3)),!known(answered('self',m3))))"/>
```

the above defines a template macro `isAvailable` that accepts one argument, for example, if the argument `topic` is the variable `tt`, the template generates the expression: `exists(m3,question(t,?),or(!known(answered('other',m3)),!known(answered('self',m3))))`

The system also supports event macros to provide a simple way to define random options for system actions:

```
<eventmacro left="OKAY" right="or(AI_alrightA,backchannel.okay_confirm,AI_mhmC,AI_alrightE,AI_mhmE,AI_uhhuhE)"/>
```

this macro defines a system speech act called "OKAY" that could be verbalized as any of the 5 speech act listed.

The reward definition file:

Reward definition file

```
<goals>
    <goal id="simple" desc="the basic reward" value= "10"/>
    <goal id="quick" desc="reward for something more important" value= "30"/>
...
</goals>
```

Each `<goal>` element defines a new reward (we refer to them also as goal to stick with the planning terminology even though they are not really goals).

For example, the line `<goal id="simple" desc="the basic reward" value= "10"/>` defines the reward named `simple` with description "`the basic reward`" and value `10`. This lines internally defines a variable named `valueFor_simple` with value `10`. This variable name is used if one wants to change the global value associated to a specific reward at run time (i.e. as an effect of a certain action).

The Text Format used by the files that defines the sub-dialogues (aka operators or actions):

This section describes the text format. As mentioned before one can include in the root policy file any number of files in text format containing the definition of sub-dialogues. This features allows the author to organize the sub-dialogues in some meaningful way.

All files needs to be in the format described here.

Examples will be used to illustrate the format. This text defines a sub-dialogue named `qamake` that has a structure typical of question-answering sub-dialogues:

Defining a sub-dialogue

```
Network qamake {
    #topic: qa
    #entrance condition: current NLU speech act = question.what.you-make

    system: answer.what.make
    #goal: simple
}
```

(sorry, one more way to call a sub-dialogue: `Network`)

Notice the use of the standard `xi:include` to include content from external xml files. Use this technique to divide the content of the policy file as you desire.

Flow of execution:

As mentioned earlier a sub-dialogue defines a small conversation as a tree with nodes and edges. Nodes contain effects, while edges are either user, system or condition edges.

Without entry paths, the remaining structure found in the sub-dialogue definition creates a tree with a single root. Unless otherwise specified the flow created is a single linear path of edges. To create a node with more than one outgoing edge one needs to use a special keyword (`OR` or `IF`).

to this basic tree, the entry path add ways to define when and where the execution can start. An entry path defines when it can be traversed and at which point of the sub-dialogue it'll start the execution if taken.

Topics:

First we list the topics associated with this sub-dialologue. This is done using line 2, that associates this network to the topic `qa`. We can associate a network to as many topics as we like by adding more lines like this one.

Topics use a hierarchical structure using the "." as a separator. The argument of the topic tag, `#topic:`, should be a complete path (i.e. from root to leaf) in this hierarchy.

Entrance conditions:

Line 3 defines a user entry path satisfied when the input event is `question.what.you-make`.

A system initiative entry path would be defined with a line like: `#entrance condition: system initiative`

A re-entry path is defined by a line like `#reentrance option: statement.back where statement.back is the line that the system will say when it takes that re-entry path.`

All entry paths can be followed by an optional condition tag that specifies further restriction on when the entry path can be taken based on the information state. For example, the following entrance condition:

System entry with condition

```
#entrance condition: system initiative
#condition: and(state=='start',!known(type),known(sugar))
```

specifies a system initiative entry path what can be taken when the variable `state` contains the string `'start'` and the variable `type` is `null` and the variable sugar is **not** `null`.

An entrance condition should be placed where we want to start the execution if it's traversed. Entrance conditions don't have to be in the initial part of the sub-dialogue even though that is the most common location. Then can be anywhere we could start the execution of the sub-dialogue. in particular, re-entry paths should be located where execution can restart after an interruption. Interruptions can happen anywhere but the most common places are while waiting for a user input (i.e. at the state before user edges).

System actions:

Line 5 defines a system action that makes the system say the line associated to the identifier `answer.what.make`.

That system action is translated into a system edge.

User and system actions can be followed by an optional condition that specifies further restrictions on when that edge can be taken. For system actions the semantics of the optional condition is slightly different: the edge will be traversed no matter what, but the line associated to the edge will not be said if the condition is false. So the system line with condition should be read as "say this if this is true". The user line with condition should be read as "wait for this event and this should also be true".

Effects:

Line 6 attaches the reward effect to the state reached after the system edge is executed. This reward effect says that the reward associated tot he defined goal named `simple` will be achieved when the execution reaches that point on the sub-dialogue tree.

Information state updates are defined using a line like `#action: state='exit'` in which the value of the variable `state` is set to be the string `'exit'`.

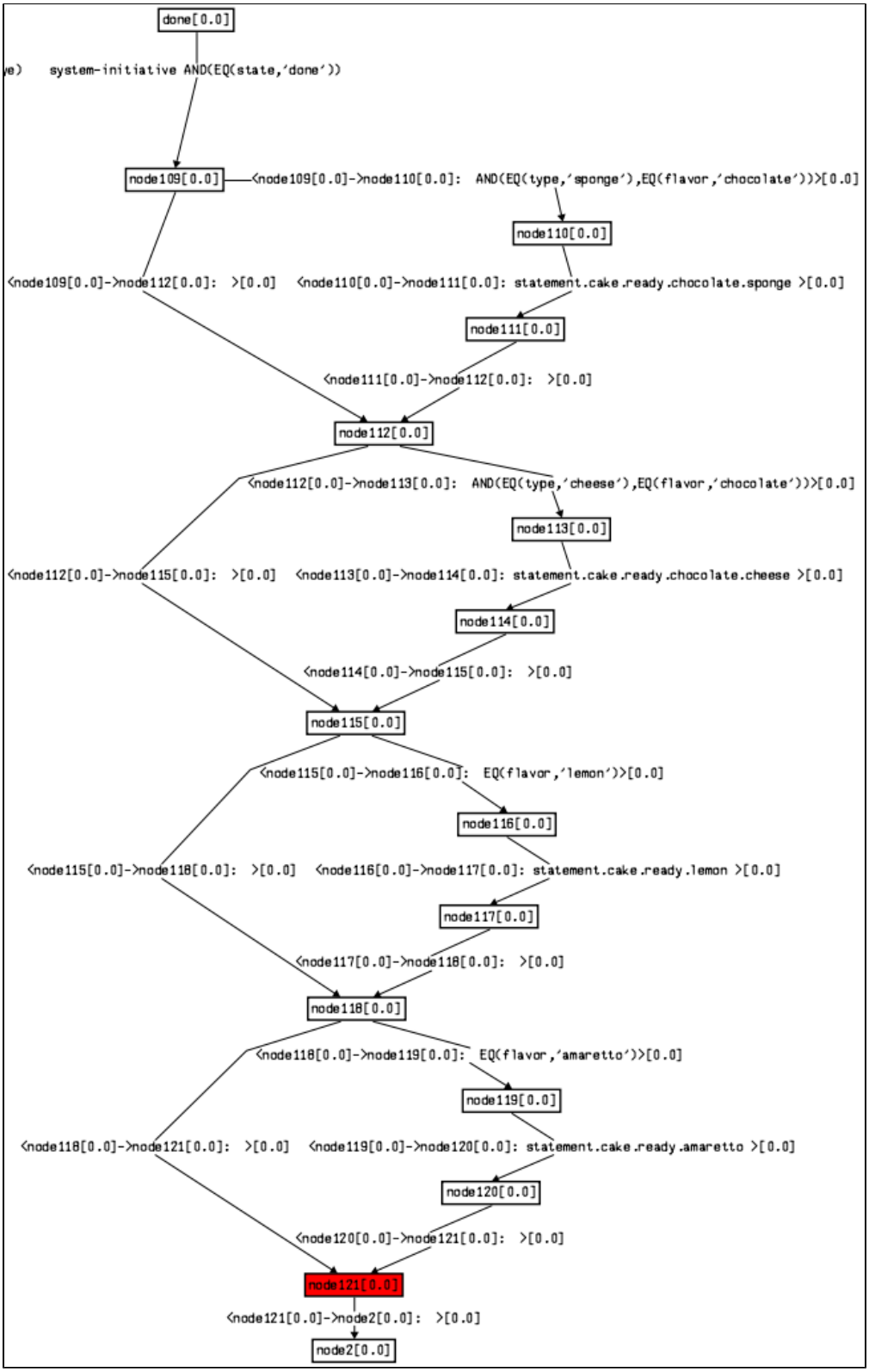
Example with conditional edges:

This sub-dialogue defines a confirmation dialogue tree that says different things depending of the value of the `type` and `flavor` information state variables.

Sub-dialogue with conditional edges

```
Network done {
    #topic: done
    #entrance condition: system initiative
    #condition: and(state=='done')
    {
        if (and(type=='sponge',flavor=='chocolate'))
            system: statement.cake.ready.chocolate.sponge
        if (and(type=='cheese',flavor=='chocolate'))
            system: statement.cake.ready.chocolate.cheese
        if (flavor=='lemon')
            system: statement.cake.ready.lemon
        if (flavor=='amaretto')
            system: statement.cake.ready.amaretto
    }
    #action: state='exit'
    #goal: simple
}
```

The automatically generated dialogue tree is:



In general the `IF` keyword has the following syntax:

IF syntax

```
IF (condition)
{
        //system/user actions
}
```

```
ELSE
{
        //system/user actions
}
```

The ELSE block is optional.

A more complex example with user actions, ORs, DO and SWAPOUT:

User actions, ORs, DO, SWAPOUT

```
Network flavorCheese {
    #topic: set.flavor
    #entrance condition: system initiative
    #condition: and(state=='start',type=='cheese',known(sugar))

    #reentrance option: statement.back

    system: question.cake.flavor
    {
        {
            user: statement.flavor.chocolate
            #action: flavor='chocolate'
        }
        OR
        {
            user: statement.flavor.lemon
            #action: flavor='lemon'
        }
        OR
        {
            user: statement.flavor.amaretto
            system: apology.flavor
            #action: clarifyFlavors=true
            #action: swapout
        }
    }
    DO
    #action: state='done'
    #goal: simple
}
```

Here we see a re-entry path with no optional condition (remember that an entry path can be taken only if the sub-dialogue containing it is paused, that is it was started earlier by a normal entry path and so it already passed one level of conditions. This is to say that it's typical for re-entry paths to not have conditions).

Line 8 is a system edge that brings as to a node with 3 outgoing edges defined by the 2 ORs. The use of OR should be read as follow this edge if you can, OR this, OR this.... For simple system actions that do not require different information state updates, an event macro would be easier to use.

If one needs to specify a complex sub-tree instead of just a simple edge with no effects, one should surround the block of text defining that sub-tree with curly brackets to define its scope unambiguously. You can see the use of curly brackets for scoping at line 9, and then for the three blocks of code that defines the three arguments of the ORs.

Line 26 closes the bracket opened at line 9. This generates empty edges that bring all execution paths open till then (3 in this case) to one single node.

The `DO` keyword creates a new node to which we can attach effects if we don't need to use the standard way to create nodes using user or system actions. In this case 2 effects are attached to that node, a reward and an information state update.

Line 11 shows a user action that defines an edge waiting for the event `statement.flavor.chocolate`.

The rest are normal information state updates. The remaining new statement introduced is line 24: `#action: swapout`. this effect forces, when executed, the sub-dialogue to become paused. No further content can come after a swapout action as it'll not be executed. One would need to put a re-entry path there to be able to re-start execution right after the swapout action. This action can be used as a way to emulate calling another sub-dialogue. For example, line 23 sets the variable `clarifyFlavors` to true enabling a different network to be executed. Then we swapout the current network. Given how we set the rewards, the other network will be selected and once completed (if again the rewards are set properly) execution will restart the `flavorCheeses` network.

Comments:

Comments can be inserted using the Java style. Single line comments are // and multi-line comments are /* */.

Debugging:

Syntax checks:

Syntax checks are executed at load time and if problems are encountered the policy is discarded and a message printed that says where the problem was encountered.

Other messages are printed that may require your attention. For example, if user or system actions use undefined string identifiers you may need to press ENTER to continue the execution.

Graph conversion:

Also, if one defines very complex sub-dialogues with many different variable updates a warning may be presented saying that too many possible final conversation states (a conversation state is a set of effects defined along each possible path of the sub-dialogue tree) are defined. This will make the search step impossibly slow and consequentially the rewards useless. One way to avoid the problem is to add `ignore` statements to tell the code that process the sub-dialogue to find the possible different final conversation states to avoid considering a specific variable. These `ignore` statements should be added at the beginning of the network. For example, the statement: `#ignore: var1` tells the code to ignore updates to the variable named `var1`.

Another check to do to a policy is to generate the graphical representation and check that it matches the desired design. To generate the graph representation one needs to set the debug level associated with the policy parser to DEBUG: modify in the file src/log4j.properties the line:

```
log4j.logger.edu.usc.ict.nl.dm.reward.model.RewardPolicy=warn
```

to

```
log4j.logger.edu.usc.ict.nl.dm.reward.model.RewardPolicy=debug
```

Then after the policy is loaded, you'll find a .gdl file named policy_complete_path_to_policy_file.gdl. Open this using the aiSee software.

Dialogue manager logs:

The system generates two log files in the logs directory. for each conversation, it generates a separate xml file with the following name: `chat-log-MACHINE-USER-[YEAR_MONTH_DAY]-[HOURS_MINUTES_SECONDS]-sid=999-pid=.xml`. This file should be seen in a browser that supports xsl style files (e.g. Firefox). It contains the record of the conversation with the NLU interpretation and the changes in the information state.

The second log file has the name: `system-logs-MACHINE-USER-[YEAR_MONTH_DAY]-[HOURS_MINUTES_SECONDS]-sid=999-pid=` with no extension. This contains the system messages generated and the content depends on the log level set for each component in the log4j configuration file: src/log4j.properties.

Event listeners:

As mentioned in the information state section one can also define event listeners that execute predefined updates to the information state every time a particular event is received irrespectively of the currently active sub-dialogue.

To do so you can a list of listeners to the policy file using this syntax:

Listeners

```
<listeners>
        <listen event="internal.timer" update="imply(questionnaire_flag==2, ++(break_timer,timerInterval))"/>
    <listen event="internal.timer" update="assign(smalltalk_pause_lock_auto,isQuestion(systemEvent))"/>
    <listen event="answer.observable.*" update="++(symptom_said)"/>
</listeners>
```

The above defines 3 listeners. The first two fire when the event `internal.timer` is received. The third fire when any event that is prefixed by `"answer.observable."` is received (the `*` has the semantics of .+ in traditional regular expressions).

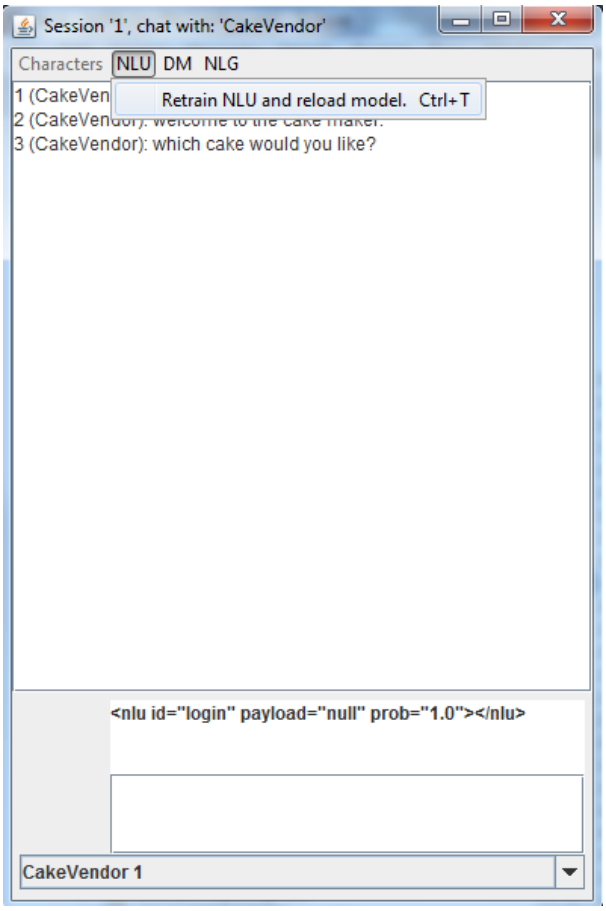
the update parameter defines what will be executed when the specified event is received. The `imply` update will increment the variable `break_timer` by the value of the variable `timerInterval` if the variable `questionnaire_flag` has value 2. Basically this allow to define an event listener that executes the update when a particular event is received and a particular information state condition is satisfied.

the second update using the `assign` keyword is a simple assignment (the variable `smalltalk_pause_lock_auto` is assigned the value returned by the function `isQuestion` applied to the value of the variable `systemEvent`.

The listing order is important as the listeners are evaluated in the order in which they were defined.

Step 3: Train the natural language understanding module

After defining the content and the dialogue policy we are ready for training the NLU. We need to start the FLoReS module. After the interface pops up:



select the NLU menu and under it the training voice. The first time the interface is opened the training happens automatically but if you update the content as described in step 1, you need to manually select this menu to update the NLU models.

An example

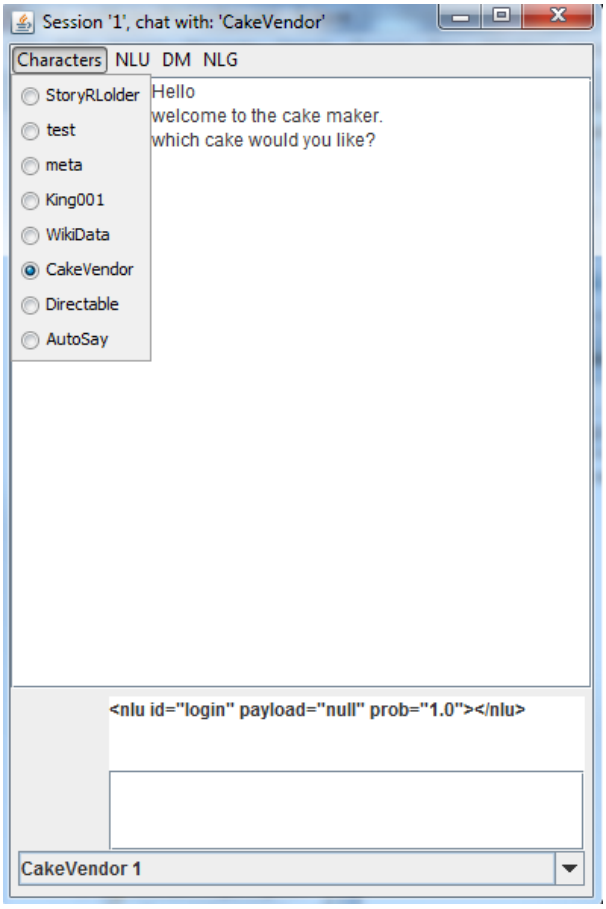
CakeVendor contains all is required to define a CakeVendor character that is an extension of the character created in this other tutorial for NPCEditor.

Running a character

The FLoReS module comes with a chat interface that allows to easily test a dialogue policy without requiring any other module to run.

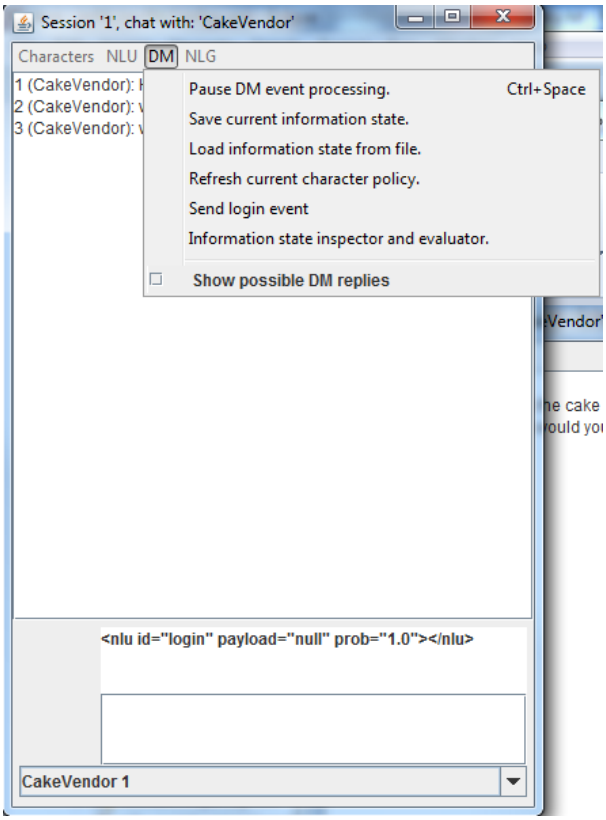
The chat interface allows the user to select a character, train the NLU, see what the system says, send text to the system, momentarily block event processing in the dialogue manager, test all nlg speech acts and inspect the information state.

If you design multiple characters defined under resources/characters/, the one for which the policy was loaded correctly will be available in the `Characters` menu. select one to chat with it. the following is a screen capture of a typical characters menu:



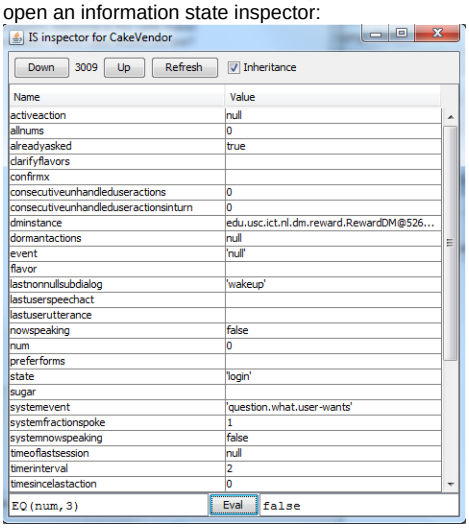
The NLU menu allows you to retrain the NLU after you have made changes to the user utterances file.

The DM menu:



allows to control the dialogue manager:

- pause the event processing so that, for example, you can check the logs or the information state before the next timer event comes in
- save the current information state so it can be loaded at a later time to initialize another character
- reload the entire character content (to get fresh changes made to the files on disk)
- send a login event to simulate a login from a remote user (useful for policies designed to start when a login is received)
- open an information state inspector:



  in which one can see the value of all assignments and assertions in the information state and evaluate all expressions
- enable a mode that displays for every user input the possible system replies that were available sorted by their expected reward

The interface also allows to send text to the character, just type in the bottom part of the interface and press ENTER and select the particular character to which to send the text in case multiple are running (advanced configuration that uses the meta protocol).

Configuration

The system is completely configured using a set of xml files. the configuration is separated into 4 main modules:

1. the messaging bus
2. the NLU module
3. the DM module
4. the NLG module

The messaging bus is common across all characters. The NLU, DM and NLG configuration instead is (can be) specific to each individual character. A default configuration must be provided but that can be overriden by specific configuration provided in the characters/CharacterName folder.

Here follows a list of all the current configuration fields for each of the types above:

Messaging bus configuration

- AllowNluTraining: enables retraining of the NLU through the chat interface.
- Character: default character attempted to start at startup.
- ChatLog: the prefix (including path) of the chat log files.
- ContentRoot: root directory where all characters are found. Typically it is resources/characters.
- DisplayNluOutputChat: if true, the system displays the NLU output in the chat interface.
- LoggingEventsInChatLog: if false, it disables saving the chat log file.
- Protocols: list of external messaging protocols to enable. Each must be the name of a class extending edu.usc.ict.nl.bus.protocols.Protocol.
- UseVrExpressOnly: use this is you want the system to listen to other's vrexpress messages and treat them as input text utterances.
- UseVrSpeakOnly: similar to the UseVrExpressOnly. This enables listening to the vrSpoke messages instead of the vrExpress ones.
- VhComponentId: the string identifier used to respond to the VHToolkit messanger API.
- VhOtherSpeaker: a string identifying the name of other vh speakers to which we want to listen (for multi agent configurations). Can be "*" to indicate listen to all.
- VhServer: ip/name of the activemq server to which to connect.
- VhSpeaker: name of the sender of vh messages, in general this should be automatically set using the current character, but it can be overwritten by this setting.
- VhTopic: topic for the vh messages.
- ZoomFactorChat: a float used to configure the font size in the chat window.
- FileRoot: deprecated
- InternalDmClass4VhMsgWrapper: deprecated.
- IsLoadBalancing: not used in this context.
- RunningMode: deprecated.
- ValidatePolicies: deprecated.

NLU configuration

- AcceptanceThreshold: if configured (i.e. not null or negative) the NLU will return its 1-best result only if the confidence score associated with it is above this threshold.
- ChartNluInSingleMode: used by the class edu.usc.ict.nl.nlu.chart.MXChartClassifierNLU to disable the multiple speech acts extraction described in the paper: http://www.aclweb.org/anthology/P11-2017
- ChartNluMaxLength: used by the class edu.usc.ict.nl.nlu.chart.MXChartClassifierNLU to automatically disable extracting multiple speech acts for longer utterances, see http://www.aclweb.org/anthology/P11-2017
- EmptyTextEventName: if the user enters no text, then this event is returned by the NLU.
- ForcedNLUContentRoot: this is the path to the NLU models in case you don't want to use the default location in a character folder.
- FstInputSymbols: configuration used by the attempts to use Finite state transducers to do NLU for SPS. Read the code for more info.
- FstOutputSymbols: see FstInputSymbols
- RunningFstCommand:the actual command to be executed for the FST experiment. See src/NLUConfigs.xml for an example.
- TrainingFstCommand: see RunningFstCommand
- HierNluReturnsNonLeaves: boolean, default true. used in hierarchical NLU models, if true the model will return also non leaves (i.e. result from a NLU model that has children NLU models) when the result has a higher probability than the results of its children
- HierarchicalNluSeparator: separator used in the labels to recognize hierarchical structure. For example, "." is the hier separator for java packages.
- InternalNluClass4Chart: internal NLU class used by the chart classifier (multiple speech acts in a single line of text)
- InternalNluClass4Hier: internal NLU class used in hierarchical NLU models.
- InternalNluListForMultiNlu: list of NLU beans to run simultaneously in the multi NLU setup (e.g. SPS).
- LowConfidenceEvent: event sent out if the 1-best NLU result is below the AcceptanceThreshold.
- MaximumNumberOfLabels: maximum number of labels to be found in the training set. Used to generate an error or warning if it's known that the particular classifier used has this limitation (on the number of labels).
- MergerForMultiNlu: bean to use to reach a single output from a multi NLU setup. For example, if classifier 1 returns result r1 then run classifier 2 and return result r2 as the global result, otherwise return r1.
- NluClass: the basic NLU class used (could be a hierarchical or multi or chart or simple classifier). check out src/NLUConfigs.xml for some examples. Check out the resources/characters/*/NLUConfig.xml for other examples.
- NluDir: the name of the directory under which the nlu stores its model: ContentRoot/characterName/NluDir
- NluExeEnv: setup needed only when running specific external nlu exe that requires custom environment variables (check the source code, never used).
- NluExeRoot: see NluExeEnv.
- NluFeaturesBuilderClass: the class used to build the features from the training data for the NLU class.
- NluHardLinks: file that contains direct links between surface text and speech acts. if a text matches that string 1-to-1 then the NLU is not invoked and the associated label is returned.
- NluModelFile: the name of the file that stores the NLU model.
- NluTrainingFile: the name of the file that stores training data used by the NLU classifier to train its model. Usually the data is generated from the user utterances found in xlsx files in the ContentRoot/characterName/content directory. Then that data goes to the features builder class and then it gets dumped in the training data format of the specific NLU class used in this file.
- NluVhGenerating: if true the NLU generates the vrNLU vh message
- NluVhListening: if true and the NL bus has VHProtocol enabled, then the system will listen to vrSpeech messages.
- PreprocessingRunningConfig: the spring bean name of the preprocessing config to be use at runtime.
- PreprocessingTrainingConfig: the spring bean name of the preprocessing config to use to generate the NLU training data. You need a different one, for example, if you want to run different preprocessing steps to prepare the training data as opposed to
- PrintNluErrors: not used
- Regularization: regularization parameter.
- SpsMapperModelFile: sps specific, should be moved out to a sps specific class.
- SpsMapperUsesNluOutput: see SpsMapperModelFile.
- TrainingDataReader: class used to read the NLU training data format.
- UseSystemFormsToTrainNLU: the system will extract training data from the forms definition file is there (forms define the multiple choice questions).
- UserUtterances: defines the name of the file that contains the user utterances to be used for training.
- nBest: defines hoe many results should be returned by the NLU.

DM configuration

- ApproximatedForwardSearch: if enabled the system runs a simplified search. faster but less accurate.
- CaseSensitive: if true, variables are case sensitive, otherwise everything is lowercased internally.
- DmClass: the DM class to be used (e.g. RewardDM)
- DmVhGenerating: if true the DM generates the vrGenerate message.
- DmVhListening: if true listens to vrNLU.
- ForcedIgnoreEventName: name of the event generated when the dm ignores a user event.
- InitialPolicyFileName: the name of the policy file.
- LoginEventName: the name of the event generated at login.
- LoopEventName:the name of the event generated if the DM recognizes that it's stuck in a loop.
- MaxIterations: maximum number of iterations for a single event. it's a safeguard for bugs in the policy and event handling.
- MaxSearchLevels: used to stop the search, defines the maximum depth of the visited search space.
- PreferUserInitiatedActions: if true, it prefers a user initiated action from the possibilities found by the search.
- SkipUnhandledWhileSystemSpeaking: if true, doesn't generate the unhandled event while the system is speaking.
- SpecialVariablesFileName: the name of the file used to dump the list of special variables at startup.
- SpokenFractionForSaid: percentage of a line that needs to be said (before interruption) for a line to be considered said.
- StaticURLs
- SystemEventsHaveDuration: if true, the system will track the NLG to wait for a line to be finished before moving on.
- TimerEvent: the name of the timer event.
- TimerInterval: the length of time in seconds between timer events. if negative, timer events are disabled.
- TrivialSystemSpeechActs
- UnhandledEventName: the name of the event generated when the system doesn't have an executable operator that can handle the current user event.
- UserAlwaysInterrupts: if true, the user always interrupt the system.
- ValueTrackers: trackers used to update specific variables with high precision. See src/DMConfigs.xml for examples.
- VisualizerClass: used to visualize the DM state, mostly deprecated.
- VisualizerConfig: see VisualizerClass

- WaitForUserReplyTimeout: number of seconds for which we allow a user to take to reply.

NLG configuration

- AllowEmptyNLGOutput: if true the NLG can return empty text, otherwise it'll generate an error if empty text is returned.
- AlwaysPreferForms: global flag used to prefer forms (multiple choice) for a given speech act if forms are defined for it/
- DefaultDuration: default duration of a speech act. Usually VH messages are used to compute this, or the audio file, or the length of the text. If all custom methods fail, then this default is used.
- DisplayFormAnswerInNlg: in case multiple choice is used, the nlg will return the full selected answer.
- IsAsciiNLG: if true, it filters non ASCII characters from the nlg text.
- IsNormalizeBlanksNLG: if true, normalizes blanks in the nlg text (removes duplicates, clean end of line).
- IsStrictNLG: if true, it returns errors for each speech act used in the DM policy from which the NLG cannot return text.
- LfNlgLexiconFile: not used
- NlgClass: the NLG class used to generate text from speech acts.
- NlgVhGenerating: not used, typically specific NLGs generate vh messages.
- NlgVhListening: is true, listens to vrGenerate.
- Nvbs: file that contains the nvb info.
- Picker: class that specify how to pick one text realization from multiple possibilities for a given speech act.
- SystemForms: name of the file that contains the definition of the multiple choice system lines.
- SystemResources: name of the file that contains resources (e.g. links)
- SystemUtterances: name of the file that contains the system lines (mapping between speech acts and surface form).