World Models

Draft Version 0.02 - 14 Sept 2016

Abstract

A description of how OpenCog currently implements self-awareness and world-awareness, together with a sketch of how this could be improved and expanded. The short-term goal is to create a robot (embodied chatbot) that can hear and see, and carry on conversations about perceived objects, as well as to carry out conversations about the self. These conversations can be verbal, and can also have physical-body performance components: body and face expressive movements, such as smiling or waving a hand.

Introduction

The paper proceeds in several steps. The first step is to review the overall concept of an "internal model", and how one can, in principle, interface with it. This is followed by a section reviewing the current prototype. After this, difficulties with language are discussed.

Internal Models

The basic premise that will be elaborated here is that interacting with the world requires the creation of an internal model of the world: in systems theory, this is sometimes called "the good regulator theorem." An internal model provides the system software with a natural API, a natural representation, that various different software components can agree on, and use, and manipulate, and reason with.

AtomSpace

To achieve a unification of speech, behavior and perception, one must have a software infrastructure that allows these to be represented in a unified way: that is, the data and algorithms must reside in some unified location. From here on out, it is assumed that this unified location is the OpenCog AtomSpace. This is stated explicitly, because, in the course of discussion, various other technology platforms have been nominated. Although one could debate the merits of alternative technologies, this will not be done here.

Self Model

The self-model, and together with it, the controversial term “self-awareness”, is here defined to simply be an internal model of the robot itself: both of low-level physical variables, such as motor angles, as well as higher-order concepts such as “I smiled just a few seconds ago”, or “I just said this-and-such”. A basic assumption taken in the following is that the engineering and design of the self-model is not any different than the engineering and design of the world-model: the data types and access methods are the same for both. Thus, ideas like “I know that my arm is raised” are represented in much the same way as “I know that there is a box in the corner of the room.” Thus, in what follows, the expression “internal model” or “model” will refer to both the self-model and the world model, there being no particular difference.

World Model

Although the above argues that both the self-model and the world-model are special cases of the internal model, this is worth more discussion. The world model describes not only inanimate objects, such as boxes seen in the corner of the room, but includes models of other people. In the current software base, this is quite shallow: it is just a list of the human faces currently visible to the video camera, and their 3D coordinates. The model could include a lot more information: names to be associated with the faces, memories of past conversations with those faces, a personality profile associated with each face. This can be expanded to a general model of “other”, including stereotypes of profession, gender, cultural and geographical background, and so on.

For most of this document, the distinction between models of self, other, and the rest of the world does not matter: rather, the discussion centers on how to interface such models to perceptions and to actions. An vitally important side topic is how to automatically learn and update the model: this is discussed briefly, later, but is not a primary topic.

State

The model is meant to implemented as “program state”. In the context of OpenCog, this means that state is represented as set of Atoms in the AtomSpace: the AtomSpace is, by definition, a container designed specifically for the purpose of holding and storing Atoms. Much of this state is to be represented with the StateLink, and much of the rest with EvaluationLinks and PredicateNodes. The precise details follow what is currently the standard best-practices in OpenCog. That is, there is no particular proposal here to change how things are already handled and coded in OpenCog, although a goal here is to clarify numerous issues.

It is critically important that state be represented as Atoms, as, otherwise, there is no other practical way of providing access to that state by all of the various subsystems that need to examine and manipulate that state. This is an absolutely key insight that often seems to be lost: if the state data is placed in some C++ or Python or Scheme or Haskel class, it is essentially “invisible” to the very system that needs to work with it. This applies to any kind of state: it could be chat state (words and sentences) or

visual state (pixels, 3D coordinate locations): if it is not represented as Atoms, then the myriad learning and reasoning algorithms cannot effectively act on this state. This is an absolutely key point, and is one reason why non-AtomSpace infrastructures are not being considered: they lack the representational uniformity and infrastructure needed for implementing learning and reasoning.

However, the AtomSpace does have certain peculiar performance characteristics and limitations that make it not suitable for all data: for example, one would never want to put raw video or audio into it. Yet, one does need access to such data, and so specific subsystems can be created to efficiently handle special-purpose data. A primary example of this is the SpaceTime subsystem, which represents the 3D locations of objects in an OctTree format, as well as offering a time component. Although the SpaceTime subsystem can store data in a compact internal format, it is not, however, exempt from having to work with Atoms: data must be accessible as Atoms, and suitable query API's must be provided. In this example: it is possible to query for nearby time-like events, or to answer questions about whether one object is nearer or farther, or maybe bigger or smaller, than another.

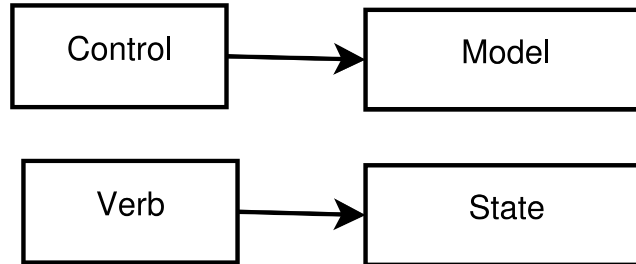
Model and Control

It is not sufficient to create an internal model of the world, and represent it as state: a control API or control language to manipulate that state must also be provided. The control is the active snippet of code that performs the actions needed to update the internal model. It can be thought of as the "control" aspect of the "model-view-controller" (MVC) paradigm from GUI programming. There are both engineering and philosophical reasons for having a control API. The engineering reasons include things like code-reuse, error-checking, encapsulation and ease-of-use. The philosophical reason is that a control API provides a shim between the world of static data, and the world of action and movement. That is, as events occur in time, and as the world is in flux, so must also be the internal model.

It is useful to think of the control API as a collections of "actions" or "verbs" that can be applied to "objects" (see 1). In object-oriented programming, these "actions" are usually called "methods" or "messages". In what follows, these will often be called "verbs", or possibly "meta-verbs" (XXX TODO: we need a good name for this). There is an important reason for this choice of terminology. First, due to the nature of how data is represented in the AtomSpace, it is the case that some given action can be applied to a large swath of the data. That is, most actions are NOT tightly coupled to the data they are manipulating, but are quite general. This means that the object-oriented paradigm does not work well with our concept of "internal model": its not like there are many different kinds of objects, and they all need to have methods. More accurately, there are only a few kinds, and many (most?) actions are in principle (*de facto*?) capable of manipulating many (most?) kinds of state. The OO paradigm does not provide a good way of thinking about what goes on in the atomspace.

Another handy reason for why these "actions" can be called "verbs" is that they are really "potential actions": nothing happens until they are performed. However, they can still be talked about, and reasoned about, and even learned: that is, the actions themselves can also be represented with Atoms, thus allowing the reasoning subsystem

Figure 1: Model and Control



The control API alters the model. It does so by applying "verbs" or "actions" to the model state.

---

to make inferences such as "if I do X, then Y will happen" e.g. "if I stick out my tongue, people will laugh, or maybe they will get offended".

In the same way that it was argued that model state must be represented in terms of OpenCog Atoms, or "atomese", so too must be the verbs. That is, the "control API" is not some C++ code (or python or scheme or Haskel...) but rather, it is also a collection of Atoms. Again, the reason for this is to allow the system to automatically generate new verbs, by means of learning, as well as to reason about the results of actions. Another key idea is that this allows actions to be combined and composed, in sequential or parallel order, with different timing. That is, the actions are primitives that can be composed into performances, that play out over time. Representing these as atomese how such composition and performance-scripting can be achieved.

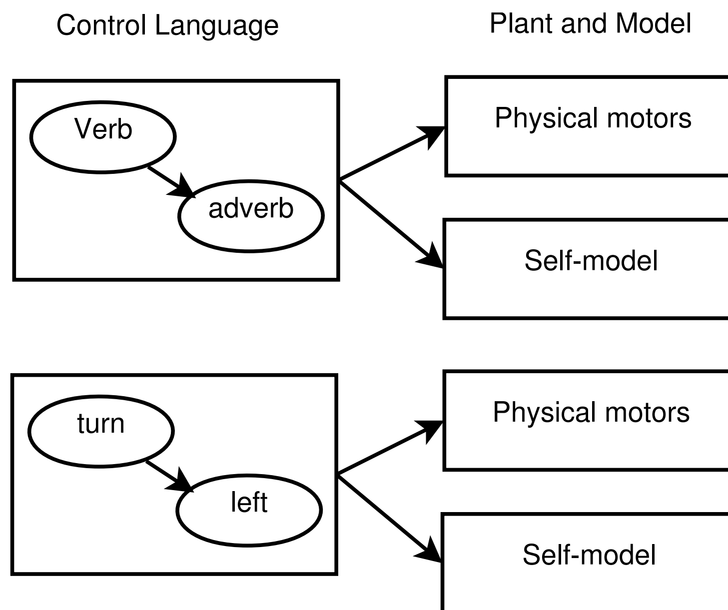
Control language

Following the idea of needing to script actions to control behaviors leads one naturally to the need for concepts such as modifiers, which come in various forms, including "adjectives" and "adverbs". So for example, if "arm" corresponds to the atomese describing a robot arm, and "raise" is an action that can be applied to "arm" (that is, the atomese for performing that action), then it is plausible to want to say "raise the left arm quickly". Here, "quickly" is the atomese needed for modulating the rate at which the motors controlling the arm are run. Likewise, "left" is the atomese specifier indicating which collection of motors are to be controlled.

Thus, the concept of a "control language" arises naturally within the system. The control language is NOT English! Although it does have a grammar, the grammar is NOT that of English.

The need for a control language seems to be unavoidable: it is important to be able to specify motor speeds, etc. Now, if one had an object-oriented system, then one would have a "motor" object (or an "arm" object composed of "motor" objects), which had a "slot" (or "method" or "message") called "speed", and to control the arm, one would send the "fast" message to the "speed" slot (or "speed" method) on the "arm" instance. There's nothing particularly wrong with this view, except for the following points. If

Figure 2: Control Language



The control language can have a non-trivial grammar associated with it; for example, one can "turn to the left", but one cannot "turn to the eye-blink" – they eye-blink animation not being valid with the turn directive. The language can control different systems: the physical body or "plant", the internal model (or self-model, in this case), as well as models representing hypothetical future behavior, or even models that represent memories of the past. This is possible because the plant and models all use the same representation scheme, and thus, the language can act on each equally well.

---

the OO language was C++, then the classes and methods must be known at compile time: new classes, methods and messages cannot be dynamically created, at run-time. If the OO language was JavaScript, then many of these issues go away: JavaScript does allow new methods to be added, at run-time, to pre-existing objects. Indeed, in many ways, OpenCog Atomese resembles JavaScript. In particular, the JavaScript members are very similar to OpenCog Atoms, in that, at run-time, new members and methods can be added to objects. One could also say that OpenCog Atomese is a lot like JSON, in that one can specify arbitrary state structures in JSON.

There are also some important ways in which atomese differs from JavaScript or JSON: atomese allows introspection, i.e. it allows for some atoms to control and operate on other atoms, which is not possible in JSON. Atomese also provides a query language, which JSON does not provide (To understand what atomese does, one might imagine writing a SparQL or SQL wrapper to query the contents of giant blobs of JSON, or possibly dumping large blobs of JSON into Apache Solr or Lucene or Cassandra). Atomese also has other language features (it is Prolog-like, it is ML-like) that are lacking in JavaScript.

Anyway, the goal here is not to debate the design of atomese, but rather to indicate that motor-control directives behave more like a language, than like an API: thus, it is more correct to think of the system as offering a "control language" rather than a "control API".

Internal model vs. physical body

The control language needs to control two distinct things: it needs to control both the internal model, and also the physical body! That is, a directive such as "raise the left arm" can be used to update the internal model, and it can also be used to control the motors on the actual physical body ("the plant", in control-theory terminology). There may also be more than one internal model: in control theory, it is not uncommon to have a "forward model", which is used to estimate what might happen if an action was performed, or an "inverse model" as an interface to the physical body. Both of these are distinct from the internal model, which models the current state, as opposed to some hypothetical future state.

Other models are possible: this includes memories of past events, where some remembered actions might be re-enacted: in this case, the remembered actions can be replayed on a remembered model, to reconstruct what happened (for example, to answer questions about those events). Another possibility is that of predicting the future, where a sequence of actions are played out on a model of the hypothetical future, to see what might happen. In such a case, there might be a model of the audience, as well as a model of the self: one is interested in predicting how the audience might react to a particular action.

Rather than inventing a new language for each of these different systems, it is convenient to be able to use the same language. The under-the-covers implementation is different: in one case, motors must be moved; in another case, the model must be updated. In either case, the verbs, nouns, adjectives and adverbs should be the same. (In control theory, this is termed the "efference copy"). This is possible as long as the different systems use the same underlying design scheme: as long as the underlying

hyper-graphs have the same structure, they can be manipulated the same way, never mind that one might represent the physical body, and another the self-model.

English language interfaces

Given the above description of the concept of "model" and "control language", one can now imagine that controlling the robot using the English language might not be too hard: just translate English to the internal control language, and one is done! Thus, for example, it is easy to imagine that simple English sentences, such as "look left!" and "pretend you're happy!", can be converted to the control language.

There are several ways to accomplish this. There is a simple, brute-force approach: create templates such as "Look ____" or "Pretend you're ____" and implement a fill-in-the-blanks algorithm. Simple string matching will suffice, and (for example) AIML excels at this kind of string-search and string-matching. This approach is sufficient to tell the robot to look in different directions, and to make different facial expressions.

A core premise of what follows is that brute-force string-matching or string-templating is NOT sufficient for more complex, more abstract conversations. For that, a syntactic analysis of the sentence is needed. Thus, although there is plenty of room and utility for string-matching in the natural-language subsystem, this will NOT be the primary focus of this document (nor is it used in the prototype). Complex conversational abilities are anticipated, and so are planned for, from the start.

The prototype does perform syntactic analysis using the Link Grammar parser. It could have used RelEx or Relex2Logic, but does not, for reasons explained later.

If one has a syntactic analyzer, then translation can be performed by extracting the syntax of a given English-language sentence, and re-writing it into an equivalent control-language structure. Along the way, specific English-language words and phrases are remapped to equivalent control-language atoms.

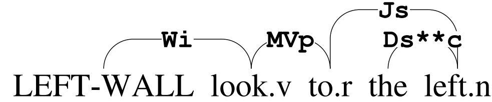
Link Grammar

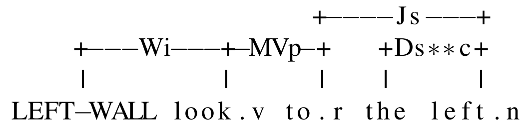
The syntax of the English-language sentence is extracted by parsing the input sentence with the Link Grammar parser (wikipedia). There are certainly many other natural-language parsers out there, including famously the Stanford parser, Google's Parsey-McParseface, and any number of phrase-structure parsers. Link Grammar is used because it fits particularly well with the theory of Atomese, and because it has particularly high accuracy and very broad coverage.

The Link Grammar parser discovers relationships between the words in a sentence, and marks up those relationships with links connecting pairs of words. These links have a type or kind, indicating the relationship between the words – typically, subject, object, adjectival-modifier, adverbial-modifier, prepositional-object and so on. This is illustrated in figure 3. The relationships, taken together, can be understood to form a directed graph, often a tree. Most links have an implicit directionality in them. The graph does sometimes have cycles: these constrain the parse choices.

The graph fully encodes the syntactic structure of the parsed sentence, as well as a fair amount of the semantic structure, encoded in the so-called "disjuncts". The disjuncts are the graph-duals to the linkages. Thus, in the figure 3, it can be seen that the

Figure 3: Example Link Grammar Parse





The above illustrates a parse of the sentence "Look to the left" (as post-script and ASCII-graphics). It consists of a collection of typed links or edges connecting pairs of words. The `Wi` link points at the main verb of the sentence, and indicates that it is an imperative. The `MVp` link attaches the verb to a verb-modifier, in this case, to the preposition "to". The `Js` link connects the preposition to it's object, in this case, the noun "left" (prepositions always have objects). The `Ds**c` link joins the noun to the determiner "the". Both `Js` and `Ds` indicate that the nouns is singular, and the `**c` indicates that it begins with a consonant (that is, phonetic analysis is also provided).

---

word "look" has a link `Wi` to the left, and a link `MVp` to the right: these two form a disjunct `Wi- & MVp+`, with the plus and minus signs indicating linkage to the right and left. Given only this disjunct, and nothing else - not even the word itself, one can already deduce that the word must be a verb, and that it must be an imperative, and that it will take a preposition, and thus, a prepositional object. Thus, the disjunct `Wi- & MVp+` acts as a fine-grained part-of-speech, and this carries semantic information. That is, the meaning of a word is correlated with the part-of-speech: this is obvious simply by observing that dictionaries organize word-meanings by parts-of-speech. The disjunct offers a particularly fine-grained distinction, and thus is more strongly correlated with meaning.

Note that there are "costs" or weights associated with different disjuncts. These can be interpreted as log-probabilities, and so the parse system, as a whole, has probabilistic or Markovian aspects associated with it. This play a minor role, at this stage.

Graph rewriting

The translation process proceeds by taking the syntactic-parse graph, such as 3, and converting it into an equivalent control-language graph, such as in figure 2. The conversion of one graph into another is accomplished via graph-rewriting. The OpenCog AtomSpace has a powerful and sophisticated graph-rewriting engine built into it, called the "pattern matcher"; it can easily convert graphs of one shape into another, even when the graphs contain variables in them (that is, use variables to represent subgraphs).

The graph rewriting is specified by writing down "rules" that indicate the shape of

the expected input graph, and the kind of output graph that should be generated, when a match is found. These rules are usually specified in the form of a BindLink, which can be thought of as an if-statement: "if graph p is recognized, then generate graph q". Alternately, they can be thought of as an implication $p \to q$, or, with variables, $p(x) \to q(x)$.

To provide a worked example: to rewrite figure 3 into figure 2, one creates a rule "IF word x has disjunct `Wi- & MVp+` and word x is 'turn' and word y has disjunct `Ds**c- & Js-`, THEN create control-language graph 'turn(y)"'. The prototype contains a rule of roughly this form.

Next steps

There are additional important concepts that are required to correctly implement a fully working system. Before discussing these, it is worth reviewing the current prototype, as it illustrates the above concepts in a specific, concrete manner. The issues encountered during prototyping also serve as an introduction to problems that must be solved in the full system.

Prototype Review

The prototype of the above-described system is located in github, in the nlp/chatbot-eva directory. It naturally splits into three pieces. These are:

- An implementation of the self-model and the control language.
- An English-to-control-language translation layer.
- A rule engine, to drive the system.

These are each reviewed, below. There are assorted design issues in each of these subsystems; these issues serve to anchor the next stage of the design discussion. Thus, it is important to understand how the current prototype works, as it makes clear both the how and the why of a more sophisticated design.

Control language prototype

The control language is implemented in knowledge.scm. All of the previous discussion is made concrete in this file, and a review of this file is strongly recommended. This is where the "rubber meets the road", where things actually happen.

Lines 80 thru 120 illustrate how spatial directions are grounded in specific x,y,z coordinates. Lines 127 thru 132 associate specific English-language words to these directions. Lines 135-139 group the control-language direction names into a single kind (in this case, into the class "schema-direction"). This will be used later, to make sure that the "look" and "turn" verbs can only take the direction-kind, as opposed to the facial-expression-kind. This forms the foundation of a crude grammar for the control language: it will not be legal to say "turn your head to face in the happy direction".

Lines 145-149 give the two kinds to looking-turning control verbs. Lines 166-170 define the control-language grammar: the only valid way to move the robot head is to specify either the "turn" or the "look" verb, followed by a direction-kind. (In the current Blender animation subsystem, "turn" rotates the entire head (turning the neck) while "look" only moves the eyes.)

Lines 173-213 duplicate the earlier portion of the file, and implement the control language for the internal model (here called the "self-model", because it is modeling the robot itself).

Lines 216-306 define the control-adverbs, in one-to-one correspondence to the Blender animation names for facial expressions. There is exactly *one* control-adverb for each animation: it is not desirable to have synonyms in this layer. Line 316 defines the one and only control-verb for facial expressions: this is the "perform a facial animation" verb.

Lines 321-336 associate fifteen different English-language words with this one control-verb. This is because, in English, synonyms are common and pervasive: it is quite natural to say "Look happy!" "Act happy!" "Be happy!", "Emote happiness!", "Portray happiness!" and mean the same thing. Thus, all of these different English-language words are mapped to the same control-verb.

Lines 338-511 associate more than one hundred(!) different English-language words with the fifteen-or-so different Blender animation names. For example, "perplexity", "puzzlement" and "confusion" are all valid synonyms for the "confused" animation.

Lines 520-531 group together the different Blender facial-expression animations into a single animation-kind.

Lines 534-538 define the control-grammar for performing a facial animation: it must necessarily consist of the single perform-facial-animation control-verb, and one of the fifteen Blender facial-animation adverbs. No other combination is possible: thus one cannot make the control-language statement "emote leftness".

These control-grammar rules not only define what it is possible to do with the robot, but they also disambiguate certain English-language expressions. Very specifically, one can say, in English, "Look left!" and "Look happy!". The English-language verb "look" is associated with both the control-language turn-verb (line 199) and also the control-language express-verb (line 335). Which of these two meanings for the English-language word "look" is intended becomes clear only after the English has been translated into control-language. The control grammar allows only one, or the other meaning, depending on how it is combined with the other control-words. In particular, this means that (in this prototype), the control-words are always and necessarily unique and unambiguous in their "meaning". The control words provide "grounding" for meaning.

Lines 540-560 duplicate the above, but are used for controlling the self-model, instead of controlling Blender.

Lines 570-680 (end of file) repeat the previous structures, but are used to control the Blender gesture-animations (blinking, nodding, shaking, yawning). The very same concepts apply.

Translation prototype

The translation consists of a set of hand-crafted rules that can recognize specific kinds of English-language sentences, and convert these into the internal-language forms. These are implemented in the file imperative-rules.scm. For example, the English sentence "look left" is recognized by the `look-rule-1` pattern, lines 176-189. The sentence "look to the left" is recognized by the `look-rule-2` pattern, lines 191-208. Both of these patterns specify several synonyms for the English verb. The pattern of the English sentences is recognized in lines 186-188 and lines 205-206. The lg-links `MVa`, `MVp`, `Js`, `Ju` come from the Link Grammar parse of the sentence; these are already illustrated in figure 3. There is a fair amount of monkey-business being done to make these rules relatively easy to write. One issue is that the atomese representation of the Link Grammar parses is fairly turgid (the RelEx Atomese format); complexities arise due to the need to represent multiple distinct sentences, as well as to distinguish the use of the same word in two different places in a sentence: these are "word instances". Thus, an imperative-sentence utility is provided in lines 130-172; if that utility is not used, then the two look-rules are more verbose, and are shown in lines 48-90 and 92-128. It is useful to compare these, to get the "big picture" of the rule format.

Lines 221-249 implement a utility for handling single-word imperative sentences, and lines 251-268 handle the various single-word imperatives.

Lines 272-286 implement a rule for sentences of the form "look happy", while lines 290-308 implement a rule for sentences of the form "show happiness". The difference here is that "happy" is an adjective, while "happiness" is a noun. The sentences, although short, are syntactically different: in English, one cannot correctly say "look happiness" or "show happy".

Translation, normalized form

The above section was a bit glib: currently, translation proceeds in two steps. First, the English form is converted into an intermediate, "normalized" form, and then the intermediate form is converted into the final control language. This second conversion is done in the file semantics.scm. The reason for such a dual-stage conversion is discussed in greater detail below. In brief, though: Link Grammar itself remains rather close to the surface syntax of the input text, whereas the actual semantic content can be abstracted away a little bit. The abstraction done here is a bit more abstract than what RelEx provides, and yet is different from Relex2Logic. It seemed to be the easiest, most natural step for the prototype.

So, given the intermediate, normalized form, conversion to the final control language is done with several more rules: a generic rule template is constructed in lines 55-92, and then three specific rules are built: one to communicate with the action orchestrator (which issues Blender animations) and two more to update the self-model: see lines 94-133. The overall structure of the code is similar to that reviewed earlier: it consists of a set of rules, each implemented in a BindLink.

Rule engine

The file imperative.scm implements an *ad hoc* mini-rule-engine. It very simply cycles through all of the rules described above, applying each in turn. It is invoked whenever the language subsystem detects that an imperative sentence has been uttered.

It is intended that this ad-ho arrangement of rules be replaced by the OpenPsi rule system. OpenPsi is described in greater detail further on in this document, at which point the reason for it's superiority as a rule-engine management system will become apparent.

Glue code

Assorted *ad hoc* scaffolding is required to integrate the above systems into the generic chat framework. it has no particular significance, other than that it is needed to make things work. See, for example, bot-api.scm for this scaffolding.

The chatbot-eva.scm file defines a loadable scheme module that encapsulates all of this code.

Self-model, World-model

The self-model is not in the OpenCog repo, but in the ros-behavior-scripting repo. The file self-model.scm simply redirects there. The reason for this is that the self-model is directly hooked up to the sensory and motor systems, and all of those are in the ros-behavior-scripting repo; only the language-processing parts are in the OpenCog repo. The self-model, together with the world-model, are central to the robot's current behavior subsystem. The combined self+world models encode only a small amount of internal state: the visibility of faces in the video feed, and knowledge of the robots current emotional state.

The file faces.scm contains several predicates, used to check if the the "room is empty" or not. At this time, "the room" consists only of that part of the world that is immediately visible to the video camera, and extends no further than that. It is intended that this model is to be replaced by a more significant one.

The file self-model.scm contains the self-model, as well as a significant portion of the room model. Notable portions include the following:

- Lines 57-60 indicating if the robot is asleep, awake or bored – the "some state". The actual state is stored in line 63. Lines 65-76 are predicates that return true or false, in answer to questions: "is the robot sleeping?" "is the robot bored?" In principle, one could say that these predicates are part of the control language for the soma state. In practice, they are not very language-like: they are indivisible, rather than having any grammatical structure to them.
- Lines 77-87 deal with the "current emotional state", but, more accurately, should be called "current facial expression."
- Lines 88-122 deal with eye-contact state.
- Lines 124-176 deal with the chatbot state: is it talking, or listening?

- Lines 177-200 deal with the chatbot affect: was the last thing that the chatbot heard friendly and positive, or is it negative?
- Lines 201-242 deal with whether anything has been heard. Currently, the robot can only hear speech, and not other arbitrary noises in the room.
- Lines 244-294 deal with the setting of timestamps on the internal state. One needs to know when, or, more importantly, how recently something happened. The are candidates for being moved into the TimeServer, which offers greater time-related capabilities.
- Lines 305-763 deal with the interaction with visible faces in the room. The names of the various predicates are more-or-less self-explanatory. So,
  - line 324: "Did someone recognizable arrive?"
  - line 388: "Did someone leave?"
  - line 421: "was room empty?"
  - line 506: "Select random glance target"
  - line 555: "Is interacting with someone?"

  ... and so on.

One major issue with the current design of these predicates is that they do not really follow the conception of the control system being a “control language”: each of these predicates is an indivisible whole, rather than a control-word that can be combined with other control-words to achieve a desired effect. That’s OK for now, but is a potential stumbling block for the future, as a true language would be more compact, and more flexible, than some arbitrary grab-back hard-coded predicates.

Design Issues

DRAFT VERSION 0.02

This is a draft. Everything above is in some mostly-finished state. Everything below is in outline format.

Translation issues

There are several technical issues that crop up, at this point. These are central to later development, and so are discussed in detail here.

issue: learning vs. hand-crafting, obtaining synonymous phrases, not just synonymous words.

issue: its LG not R2L

issue: dual-stage conversion. Normal form looks more like relex or MTT.

issue: using openpsi to discover and apply rules. This is like using openpsi in general, to pick through non-verbal stimulus.

issue: fuzzy matching, partial matching

issue: picking out sentences attached to an anchor, vs other processing pipeline designs.

todo – the verb synonyms should not be needed!? as they can be gotten from the synonym lists for the control-action language...

OpenPsi

OpenPsi as a flexible rule-selection system

Self-model issues

The various predicates: e.g. (DefinedPredicate "Is sleeping?") do not really fit the “control language” concept described above. Ditto for lines 305-547 of self-model.scm

Question-answering

Answering questions about self and the world. Also has been prototyped. Its like the “view” part of MVC – the internal state has to be “viewed” easily, in order to be queried. Thus, there are a set of “standardized state queries”, analogous to the control language. Running these queries returns yes/no answers (truth queries) or multi-valued data (e.g. look-at direction) or more complex structures (sequences of actions that had been performed in the past)

There are two translation layers that are needed here: first, to convert English to the internal query language, second, to convert the response back to English. If the response is of the right form, then SuReal can be used to perform the final conversion. Right now, the query language is not generating SuReal-compatible results.

World model

Right now, there is only a self-model. A world-model is needed, so we can talk about that. Well –there is a world model – currently, it consists of the visible faces in the environment. It needs to get bigger.

Action Orchestration

Carrying out multiple things at once; using the internal model to do this.

Memory

Multiple types of memory are needed. Most important (for demo purposes) is memory consisting of imperatives: when she is told to do this and say that, she needs to remember this, and later on, play that back as a performance.

This should be “straight-forward”: one can record the control-language directives. They need to be marked up with timing information.

Implementing acting-coaching is interesting, and in particular, implementing directives such as “do that performance again, except this time, make xyz go more slowly”.

A second type of memory would be remembering past states of the world.

One technical challenge is that we will need a management layer for the Postgres DB interfaces, so that memories are not lost during power-off. Those memories need to be segregated: there are somethings that need to be remembered, others that should not be.

Learning from sensory input

Creating atomese representations of sensory input processing pipelines, so that specific sensory inputs can be recognized e.g. if the audio volume suddenly got loud, and the visual field is suddenly moving, then maybe ... everyone is clapping? booing? getting up to leave? ...I'm no longer the center of attention?

Free will

Free will, as defined here, is the over-riding of default behavior (as computed by psi-rules) by means of a logically, rationally reasoned course of action. For example, the default of the psi rules might be "lolly-gag about", while a rational decision would be "go and do that important thing". Free will is then the act of picking between these two alternatives, of balancing them out (at a critical phase-transition point).

Items

- a person walks into room, who she recognizes. Depending on psi, she should do non-verbal greetings (play one of 3-4 different animations (look at, chin push, nod)) and verbal greetings ("hello, what's up, yo dawg"). split up the state and the psi rule stuff properly. See comments here: https://github.com/opencog/ros-behavior-scripting/pull/80
- extract keywords/key-topics from sentence, and remember them, then apply fuzzy matcher to see which of these come up.

Random ideas

- Why did you smile? Output: have her explain recent openpsi decision-making.
- I'm so sorry about that. Output: a small cute pout (blend of frown and ???)
- Look at me. (verify look-at location or report visibility)
- What are you doing?
- (When last person leaves, she should say goodbye. If did not say goodbye, then say "hey where did everybody go?")
- If no one is visible, and no one has been visible for many minutes, she should say "hey where did everybody go?"

- Behavior – she should not get sleepy, as long as someone is visible.
- Behavior – she should complain, if no one is visible, but there is a chat session going on.