

World Models

Draft Version 0.2 - 19 Nov 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 what it can see and hear, 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.

1 Introduction

In order for a robot to have reasonably competent conversations about the world around it, three things need to happen: the robot has to be able to see and hear; the robot needs to have an internal model of the world around it; and, finally the robot needs to have some basic language capability. The internal model and it's relationship to the senses and to language is the primary topic of this review.

Internal model

The core idea of the internal model is shown in figure 1. More abstractly, it is a set of software structures that describe the the world that the robot sees. The description must be sufficiently abstract that the robot can perform basic reasoning about the world. Consider, for example, spatial reasoning: if Alice is standing behind Bob, and Carol is in front of Bob, then Alice is farther away than Carol. All of this needs to be represented such that basic rational reasoning about distances can take place. Almost any example of reasoning will do: 'if cats are animals, and animals are things, then cats are things'.

The internal model must also be abstracted in such a way that it is not too hard to map natural language onto the model. Words in a sentence or question need to be matched up to specific objects in the model, so that the different parts of the model provide a "meaning" or a "grounding" for the words. There needs to be a correspondence between nouns and objects, and also between prepositions ("in", "near", "next to", "of", "with", ...) and the relationships between the objects in the internal model. That is, the internal model must also have a natural "prepositional" or "relational" structure. This extends to all word classes in natural language: verbs must correspond to actions that might be performed on objects in the internal model. Adjectives must correspond

Figure 1: Self and World Models



The internal model is a highly simplified representation of the world and of the self. It provides the software with a specific data structure that can be reasoned about and manipulated. Here, the robot Sophia thinks simplified thoughts, much as this illustration simplifies the idea of a representation. (Robot credit: [Hanson Robotics Sophia](#). House image credit: [Small House Bliss](#), [The Cypress Laneway House](#).)

to attributes that an internal object might possess. And so on. The internal model needs to naturally correspond to natural language, so that basic reasoning and basic language are possible.

The internal model must also be abstracted in such a way that it can be kept up to date as new sensory information comes in. This means that raw video must be converted into a representational form, with objects (people, their position, movement, and facial expressions) picked out from the raw video. Similarly, raw audio must be converted into representations of the form “heard a loud bang” or “there is a quiet rushing sound”, etc.

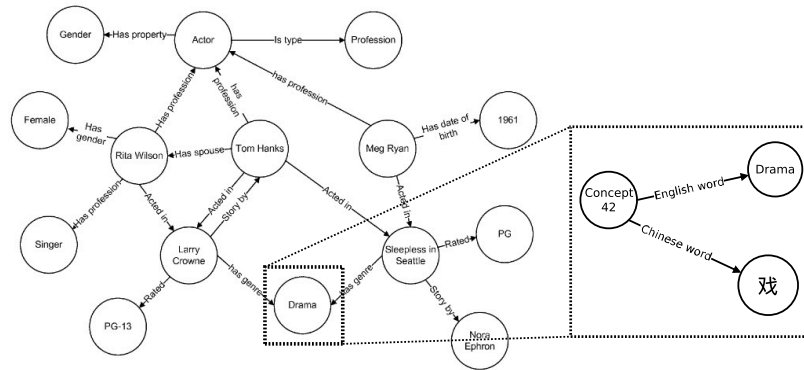
Symbolic Representation

It is assumed that everything described here can be accomplished with a symbolic representation. That is, there is an explicit network of vertexes and edges, joined to form a graph, that encodes the knowledge, explicitly and overtly. Thus, it is assumed that the core architecture is *not* a neural net, nor that the basic process is that of deep learning, where data is represented implicitly, not explicitly.

In the end, this assumption may not matter very much: a symbolic representation can still have floating point numerical weights and probabilities attached, and result in a network that is so complex that it challenges easy comprehension: the distinction between neural and symbolic can be blurry. None-the-less, by picking a symbolic representation, the goal is to have a system whose internal state can be examined, reviewed, debugged and manipulated. In the end, some of that manipulation might employ deep-learning techniques; however, that is not the starting point.

Thus, the core symbolic representation should be thought of as a large, complex graph, such as that shown in figure 2. By having an explicit symbolic representation,

Figure 2: Concept Graph



An example of a symbolic representation of conceptual data. This kind of representation is commonly employed in symbolic knowledge representation systems, such as in John Sowa's **Concept Graphs**, Dick Hudson's **Word Grammar** or in Igor Mel'čuk's **Meaning-Text Theory**. Note that each circle and arrow may itself be a graph with additional structure: for example, the circle labeled "Drama" may be represented by the abstract concept 42, which can be expressed as English or Chinese words, each of these possibly having multiple synonyms, grammatical properties and relationships to other words and concepts. Thus, properly speaking, the graph is a hypergraph. The point is that labelled bubbles, such as "Drama", or arrow relationships, such as "Acted in", are themselves undefined and have no meaning, unless they are themselves expressed (grounded) by their relationships to yet other objects, some of which may be concrete, sensory entities or physical (motor) movements. Typical graphs are more complex, and the relationships may be assigned probabilistic strength and confidence weights. (Image credit: Unknown)

one has the benefit, and requirement, of having an explicit software layer that can access, query and update the internal model, both from the language side, and the sensory side. Such conceptual graphs can be manipulated in a style similar to the widely-known “model-view-controller” paradigm. Software “controllers” can change and edit the model, while a structured query system can present “views” of the model to the language subsystem.

Outline

This document proceeds by fleshing out the above definitions in greater detail. Some of the early statements may seem overly simplistic or pedantic; this seems needed, so that the later stages are clear.

The first step is to review the overall concept of an “internal model”, and the “control language” used to interface with it. This is followed by a section reviewing the current prototype. After this, difficulties with natural language are discussed. There are multiple software issues encountered during development; these are discussed, and possible solutions are proposed.

The work described here has been chartered by Hanson Robotics, and is specifically intended to interface with the robots manufactured by Hanson, such as Sophia and others.

1.1 Robot architecture overview

XXX TODO: This section to be written in a later version. A conceptual diagram of the system is shown in figure 3.

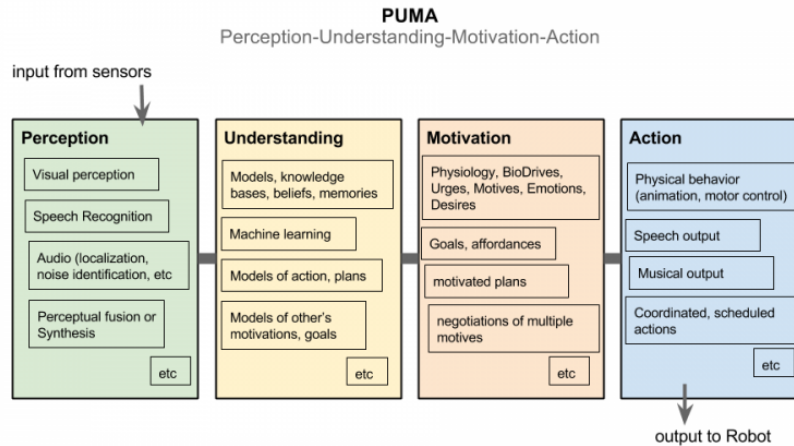
2 Internal Models

The basic premise 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. Two depictions of models are given in figures 1 and 4.

2.0.1 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.

Figure 3: The PUMA Architecture



This diagram shows the Hanson Robotics PUMA architecture: it offers a hint of what an embodied artificial intelligence architecture might look like. The architecture described in this document is a horizontal slice through the above diagram; it is focused primarily on how natural language interacts with models of the self, and of the world, to drive behavior and conversation. (Diagram credit: [Hanson Robotics](#)).

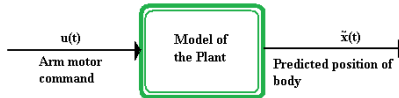
2.0.2 Self Model

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 person standing in front of me.” 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. A simplified model of the concept of a model is shown in figure 1.

2.0.3 World Model

Although the above argues that both the self-model and the world-model are special cases of the internal model, this is merits discussion. The world model describes not only inanimate objects, such as stacks of boxes seen in the corner of a room, but includes models of people and moving things. 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

Figure 4: A Control Theory Model



A typical depiction of a model, as described in control theory. This is considerably narrower than the intended meaning for this document, but serves to illustrate the general idea. Here, a forward model of an arm movement is shown. The motor command, $u(t)$, of the arm movement is input to a model of the arm and its motors and the predicted position of the arm, $\tilde{x}(t)$, is output. This shows how a model can be used to predict hypothesized future movements. Inverse models can be used to convert desired positions into motor commands, or, more generally, desired outcomes can be converted into specific actions to take, to achieve those outcomes. In this document, both uses, and many more, are anticipated. In control theory, “plant” refers to the device under control. (Image credit: Wikipedia, By Katie Amenabar, Brian Baum - Theories of Motor Control Class UMD Spring 2008, CC BY-SA 3.0.)

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.

2.0.4 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 Haskell 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.

2.0.5 Data

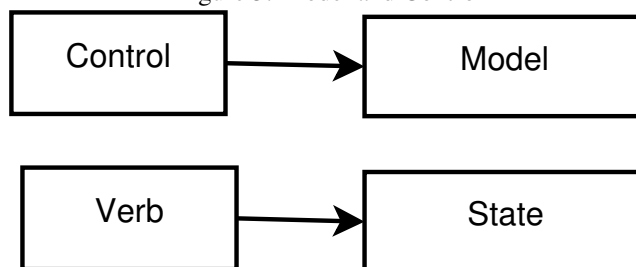
When the system is turned off, the data encoded in the conceptual graphs of figure 2 must be stored in some way. This is accomplished by tying the system to a database. The most natural encoding is that of an **Entity-Attribute-Value** (EAV) model: that is, a graph database. Although there are numerous design issues related to graph and EAV stores, this is outside of the domain of this document. The OpenCog AtomSpace is able to, under the covers, connect to various different databases. It is sufficient to imagine that graph components are stored as JSON-like fragments.

2.0.6 Model and Control

It is not enough to create an internal model of the world, and represent it as state: a control API or control language 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 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 dynamics 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 5). In object-oriented programming, these “actions” are usually called “methods” or “messages”. In what follows, these will be called “verbs”. There is a reason for this (non-standard) term. Due to the nature of how data is represented in the AtomSpace, it is the case that some given action can often 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”. It is not as-if 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

Figure 5: Model and Control



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

does not provide a good way of thinking about what goes on in the atomspace. More generally, the OO paradigm does not naturally lend itself to the idea of editing graphs.

Another 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 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”.

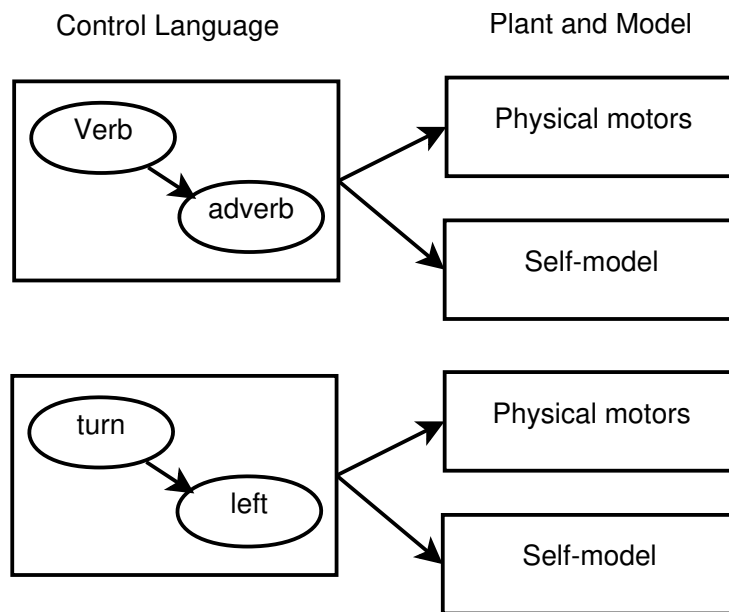
Verbs are themselves a form of data. In the same way that model state must be represented graphically (in terms of OpenCog Atoms, or “atomese”), so too must the verbs. That is, the “control API” is not some C++ code (or python or scheme or Haskell...) but rather, it is also a collection of Atoms (a graph). 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. 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 is how such composition and performance-scripting can be achieved.

2.0.7 Control language

The idea of scripting actions to control behaviors leads naturally to 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.

Figure 6: 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 need for a control language seems to be unavoidable: it is important to be able to specify motor speeds, arm heights, 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”. 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 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, JavaScript members are 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 the atomese query system, 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. Postgres is able to do the former (with the **JSONB column type**); however, SQL is itself not JSON, and there is no natural way to convert the one into the other. Atomese also has other language features that are lacking in JavaScript. It is ProLog-like, and has a **DataLog**-like subset. It is **ML-like** (CaML, Haskell) in that it has a **type system**.

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”.

2.0.8 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, and an “inverse model” as the interface to the physical body. Both of these are distinct from the internal model, which models the current state. This is, in turn, distinct from a model of 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 endpoints will be 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.

2.0.9 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.

2.0.10 Dependency Grammar

When one has a syntactic analyzer, then translation, from a natural language (e.g. English) to the control language can be performed. This can be done 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. But what syntactic analyzer, and what theory of natural language should be used?

It appears that the most natural theory of morphology and grammar, in relation to conceptual graphs, such as those of figure 2, is one or another form of **dependency grammar** (DG). This is in contrast to **phrase structure grammar** (PSG), which, although very popular in the United States, seems less suitable for practical use in concept and action extraction. Particularly inspirational forms of dependency grammars include Igor Mel’čuk’s **Meaning-Text Theory** (MTT) and Dick Hudson’s **Word Grammar** (WG). A benefit of MTT is that it offers a clear, direct articulation of the

relationship between syntax and semantics, and how these can be transformed into one another. Another key concept in MTT is the idea of a **Lexical Function** (LF): that is, a function that tells you which word one should use to express a certain idea or relation. A benefit of WG is that it offers the concept of “landmarks”, which control word-order relationships in English.

De facto, the prototype uses the **Link Grammar** (LG) parser. This is done for several reasons. It is a strong, stable, mature parser of English that works well, and is easy to integrate, today. It also has a particularly elegant mathematical grounding, closely related to that of categorial grammars and category theory. This grounding in category theory helps make clear exactly how natural language morpho-syntax can be manipulated into graphical representations using abstract theories and algorithms.

The model and control interfaces of the prototype connect directly to Link Grammar. This is called out here, because the prototype explicitly and intentionally does *not* use **RelEx** or **Relex2Logic** (R2L), for reasons explained later. RelEx is a dependency parser layered on top of LG, generating **standard CoNLL-style DG markup**. R2L is a layer on top of RelEx, used to extract logical relationships from factual English-language statements. Both are OpenCog subsystems. The R2L subsystem is used in OpenCog to perform natural reasoning on English statements, using the PLN subsystem.

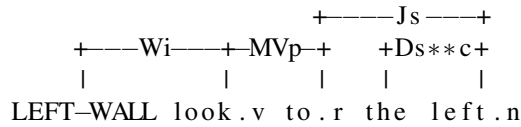
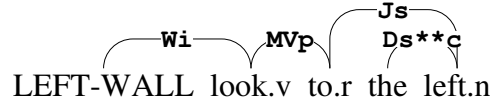
2.0.11 Link Grammar

The syntax of English-language sentences 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 7. 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 serve to 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 7, it can be seen that the word “look” has a link W_i- to the left, and a link MV_p to the right: these two form a disjunct $W_i- \ \& \ MV_p+$, 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 $W_i- \ \& \ MV_p+$ 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 (Webster’s, OED) organize word-meanings by

Figure 7: 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. **LEFT-WALL** is simply the start of the sentence; it is a pseudo-word, simply the first word of the sentence. 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).

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 plays a minor role, at this stage, but is important in the long-run.

2.0.12 Graph rewriting

The translation process proceeds by taking the syntactic-parse graph, such as 7, and converting it into an equivalent control-language graph, such as in figure 6. 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 $p \rightarrow q$, or, with variables, $p(x) \rightarrow q(x)$.

To provide a worked example: to rewrite figure 7 into figure 6, one creates a rule “IF word x has disjunct W_{i-} & 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.

2.0.13 Planning and rule selection

The next section describes the OpenCog rule selection and rule chaining subsystems. Before doing so, it seems important to motivate these, and to provide several concrete examples to keep in mind.

Classic examples of rule engines, and why they are needed, are provided by AIML and ChatScript. These systems allow authors to script a large number of rules that, when matched against a specific input pattern, provide a specific (verbal) response. Because there are a large number of rules (often tens of thousands), it is impractical to “try them all”; instead, these must be pruned down to a manageable set, usually to only one, or a small set, and then applied.

AIML/Chatscript rules usually go only “one level deep”, or (with AIML SRAI) they “forward-chain”. This essentially means that they are deterministic and reactionary: they can only react to a given input. This should be contrasted to goal-seeking behavior, which typically requires “backward chaining”: a desired end-result is specified, and the rule system must discover the rules that can achieve those results.

An example of goal-oriented behavior is this: the robot seeks to make the audience laugh (the goal). Which, of the myriad performance rules it has available, are suitable for achieving this goal?

A very different rule application scenario arises when only partial inputs are available to an ordinary forward-chaining system. For example, someone said something to the robot, but it was not fully understood – possibly due to noise, bad speech-to-text conversion, mumbling. It may be possible that the full text is available, but it is not clear what the speaker meant. In this case, one needs a selection of fallback actions: perhaps the statement cannot be responded to directly or in full (“raise your arm” except that the robot’s arm is already raised: a reasonable action is to raise it even higher), or only various sub-parts are understood (“mumble mumble turn left mumble”: a reasonable action for the robot is to turn left). In these cases, one could try to write a rule that covers every eventuality, every situation: but of course, this is legendarily impossible. Yet, a response is required.

In this case, it is important to be able to pick out a set of fragmentary rules, all of which might be partially appropriate for the situation, and then assemble a consistent, non-conflicting subset of them, for performance. If several subsets of consistent rules are available, the one with the highest likelihood or confidence can then be selected for performance. This scenario is neither forward nor backward chaining; it is rather the assembly of a fallback plan to (gracefully?) handle a new, unexpected scenario.

However, in the default mode of robot operation, a simple, direct stimulus-response reaction chain is sufficient for most situations. Almost all of the prototype runs in the forward-chained mode. Simple, direct rule application is sufficient for many or most cases. It’s just not enough to handle all cases.

2.0.14 Rule engine

A collection of rules require a rule engine to manage and apply them. OpenCog has several forms of these, that can be used in various different situations. These are an ad-hoc system, OpenPsi, and the chainers.

Rules can be applied, one-by-one, on an *ad hoc* basis, by calling `cog-execute!`, for example. Typically one has already selected some rule; the rule is typically some pattern or query, and if the query is satisfied, then the specified action is performed.

The OpenPsi subsystem maintains classes of rules, enabling goal-driven behavior. Although the rules that OpenPsi manages are only applied one by one (one at a time), the sorting of rules into distinct classes allows distinct goals, with distinct priorities to be assigned to each. Thus, to reach a certain goal, it is possible to select sets of rules that are most likely to achieve that goal. As the importance of various goals change dynamically over time, different rules-sets can come into play; that is, the goals modulate the selection of the rule-sets. In more poetic, psychologically-colored terms, the OpenPsi subsystem allows the internal model to have a set of desires and “hopes” and “emotions”, which can be used to control the rule-sets being deployed at a given moment.

The OpenPsi subsystem was explicitly developed to offer more powerful and more flexible rule management than several other popular control systems: **behavior trees**, **state machines** and **hidden Markov models**. Behavior trees are popular in game AI systems; they allow game developers to craft explicit behaviors for game avatars. In the end, though, behavior trees prove difficult to work with, because of the lack of goal-directed behavior (which OpenPsi can provide). State machines are popular among Hollywood animators, as they allow the animator to create responsive animations in a modular fashion, that can be composed together to obtain new, surprising and effective behaviors. They do not provide a way of specifying goal-directed behavior, nor the modulation of behavior based on the current model state and desires. Hidden Markov Models can be thought of as probabilistic state machines: they add a statistical component to behavior. They are particularly useful and popular for automatically discerning patterns in data. Thus, they are useful for learning. For the expression and selection of rules, though, they offer no particular advantage.

The forward and backward chainers can be used to discover sequences of rules that connect two endpoint-graphs. That is, given a particular starting graph, and a particular ending graph, the chainers can explore different sequences of rules that might result in a valid path between these two end-points. The chainers use traditional search algorithms, essentially a variant of an A-star search algorithm variant. Currently, the forward-backwards chainers are employed primarily for probabilistic reasoning (PLN), and not in the robot control subsystem.

More efficient rule chaining can be done by means of parsing; that is, collections of compatible rules can be discovered and assembled by applying parser theory (as opposed to blind A-star search).

A proposal for a generic parser, suitable for rule selection and chaining, is given in Appendix A. It is hoped that this approach can provide a more performant, more directed and principled manner of making plans: that is, of discovering routes through sets of rules that accomplish certain goals. Properly implemented, it can not only

replace blind search and rule chaining, but can also enable reasoning about plans. More colloquially, it allows thinking about the future, and planning for it. In practical terms, it allows faster responses to current situations.

2.0.15 Action performance

Once a natural language utterance has been converted into the internal language form, a decision must be made as to what the robot should do next, if anything, as a result of hearing that utterance. This decision, of what to do next, is modulated by OpenPsi. If multiple actions are selected, they must be coordinated, in terms of timing, as well as resolving any conflicts between them. This is meant to be done by the incompletely-developed “Action Orchestrator”.

2.0.16 The language pipeline

Figure 8 illustrates the complete language pipeline, assembling together figures 6 and 7 for a typical imperative sentence. This illustrates only the “input” side; so, questions, for example, will also generate a verbal output; the output side is not shown, although it can roughly be imagined to be the “reverse” of this figure.

2.0.17 Semantics

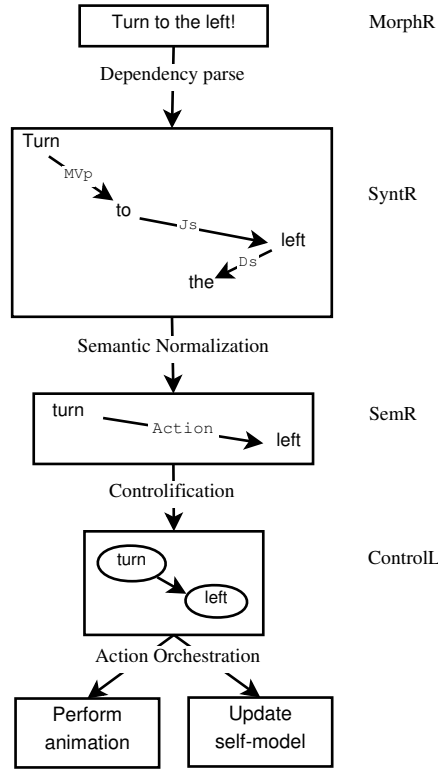
The figure 8 distinguishes a semantic representation layer SemR, and a control-language layer ContrL. The SemR layer provides a “normalized” description of the meaning of some natural language expression, independent of the precise surface form of the sentence. It provides a representation that is easier to work with, moving away from less important syntactic details. Also, unlike the control language, it can avoid being closely tied to the state representation.

The label SemR comes from “**Meaning-Text Theory**” (MTT), where it has a very specific and well-defined meaning. There, it is a graphical network, showing all of the relationships between concepts and ideas in some natural language text; it is supposed to capture *all* of the semantic content of that text. MTT elaborates, in considerable detail, exactly how to move from text (the MorphR format) to meaning (the SemR format). However, there are several things that MTT completely ignores: action, and reasoning.

In order to perform actions, the semantic network must be re-interpreted as a “control language”, and that is exactly what is being done here. At this point in the design, these two ideas are being intentionally conflated: it is not exactly clear how to untangle the different requirements and needs. Note, however, the prototype *does* currently distinguish the internal, “semantically normalized” from the control language.

Another deficiency of MTT is that it ignores logical reasoning. That is, given a specific form of SemR, one would like to perform deductions and inferences. In the case of OpenCog, this is to be done by PLN. However, the specific semantic representation that seems to fit the needs of PLN the best, is NOT the specific semantic representation that seems to fit the needs of performing embodied actions. This will be hotly debated

Figure 8: Language Pipeline



The complete language pipeline, for imperative English sentences. This combines figures 6 and 7 into one. The input sentence, “Turn to the left!”, is parsed by a dependency parser, generating a graph that shows the syntactic relations between words. Because syntax is correlated with semantics, it becomes relatively straight-forward to transform the syntactic graph into a normalized, semantic form. The semantic form is close to the control language, but not the same. The control language is used to directly to perform actions, such as moving motors or updating the self-model. The four steps shown here, dependency parsing, semantic normalization, and action orchestration, can each be taken to be rule-driven re-writes of graphs. That is, they are each achieved by applying graph-rewriting to each graph, in turn.

The labels on the right, MorphR, SyntR and SemR are the standard names that are given to each representation by Mel’čuk’s “**Meaning-Text Theory**” (MTT). In MTT, SemR is short for “semantic representation”, while SyntR stands for the dependency parse, and MorphR refer to a time-linear sequence of morphemes. Although MTT develops the relationships between these three acronyms in considerable detail and precision, it almost completely ignores the relationship between SemR and behavior, the actual performance of actions. Thus, the label ControlL is new and specific to this document. One very powerful concept in MTT, the “**Lexical Function**”, can be and should be generalized to the idea of performing actions.

in the months (years) to come; the exact nature of the best design remains a research project.

Thus, within the system, there are really three distinct forms of semantic representation: the one that PLN uses, and is generated by the Relex2Logic component; the one that the embodiment subsystem uses, currently somewhat ad hoc in design, and referred to as a “control language” in this document; and finally, external, academic conceptions of what a semantic representation should be, ranging from Mel’čuk’s MTT, to John Sowa’s “[Conceptual graphs](#)”, to many other proposals for [semantic networks](#).

2.0.18 Summary

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.

3 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.

3.0.1 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.

3.0.2 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 internal “semantic representations”. 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 7. 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”.

3.0.3 Action Performance

The connection between the semantic representation and the control language is made in the file `semantics.scm`. There are not one, but two implementations here, exactly as suggested by figure 6. One implementation converts the semantic representation into the control language that is capable of “making motors turn” viz, launching Blender animations. The other directly updates the self-model. The implementation is, once again, in the form of a set of rules, that is, as `BindLinks`.

To simplify this dual implementation (to share common code), a generic rule template is constructed in lines 55-92, and then three specific rules are built. Lines 94-111 generate the “control language” that was carefully reviewed in the section above, titled “control language”. The control-language snippet is attached to an anchor-point called (`AnchorNode “*-action-”`), which the Action orchestrator can pick up, and perform, if desired (at this time, it is always performed, without question).

The self-model is updated directly, with two rules, at lines 113-133. When executed, these two directly set a `StateLink`, recording the the robots current facial expression and look-at direction. These later two rules are, strictly speaking, a bug: the internal state should only be updated *if* the action orchestrator actually decided to perform the action.

3.0.4 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 hoc* 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 its superiority as a rule-engine management system will become apparent.

3.0.5 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.

3.0.6 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 "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. These 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-bag hard-coded predicates.

4 Design Issues and Enhancements

During the prototype development, a number of design issues were encountered. These are discussed here. All of these should be addressed in some way, either through architecture changes or implementation improvements. A number of enhancements also suggest themselves: these ask for additional development and pursuit.

4.1 Language Processing Issues

A fair number of issues crop up in the middle layers of the language input system. These are reviewed here.

4.1.1 What is the correct semantic representation?

Although the label SemR was appropriated from MTT, that is not what is actually implemented: its entirely ad hoc. During early stages of prototyping, it was assumed that the surface representation would be provided by RelEx and the deep interface would be generated by R2L; that is, that the output of R2L would provide the normalized “semantic” format. This did not work out as hoped, for two different reasons.

First, semantically similar sentences were converted into very different R2L outputs. Examples include “Look to the left!” and “Look leftwards!”. Attempts to mitigate this became quickly stuck in the morass of R2L rules. It seemed like an awful lot of effort for very little return, so that approach was abandoned for the prototype.

Next, it was hoped that RelEx would provide the surface format, but that also did not work out. RelEx failed in two distinct ways: it sometimes generated highly variable outputs for semantically similar sentences *e.g.* “Look up!” and “Look down!” gave very different outputs. Other times, there were unexpected similarities, when outputs should have been semantically different. On top of this, pattern matching to RelEx was cumbersome: the patterns quickly became complex, requiring many different checks to many sub-parts. The large complex patterns proved fragile and hard to maintain. In the end, using Link Grammar as the surface representation proved to be a LOT easier; the patterns became much smaller and more regular.

4.1.2 RelEx issues

The prototype made clear that the RelEx output is actually more complex than the LG output, and it is more difficult to use, in practice. It does not align well with semantics. The reason that it is more complex is that it tries to re-package the LG data into “orthogonal” linguistic classifications: person, number, tense, aspect, mood, voice. This is a laudable goal, but for several problems:

- Many semantically equivalent sentences have completely different values for number, tense, mood.
- This unpacking into into distinct classifications behaves like a data decompressor: it makes the atomese graphs larger, without actually increasing the information content.
- RelEx has poor coverage, and makes a fair amount of errors in what it does. This makes it’s output unreliable and undependable.

The end-goal is to have a system that can recognize semantically-similar sentences. There are ways to do this: the old and oft-quoted work from Dekang Lin, Poon&Domingos show how that can be achieved. It should be possible to harness the “Patten Miner” to perform this work. However, there’s now a strong hint that the RelEx output is inappropriate, a poor choice for this.

Important Note: Although the above paragraphs use the word “RelEx”, the same comments and complaints can be levelled against the [Stanford Parser](#), and against Google’s [SyntaxNet \(McParseFace\)](#). Both suffer from the same issues listed above:

that is because they both assume the same theory, and generate the same kind of output. They might be more accurate than RelEx, thus alleviating one of the problems, but even this is not entirely clear.

The point here is that traditional dependency parsing, whether by RelEx, Stanford or SyntaxNet, seems to be appealing, especially if you are a linguist. However, the resulting graphs contain too much fluff to be useful for efficient computing. They unpack a lot of information, making it easy for a linguist to examine their output, and verify it's correctness. But the result of this unpacking is a large graph, and that graph becomes unweildy and inefficient for automated processing. It is large, but it does not seem to contain more data than the more compact LG output. Its large and awkward to work with.

Side comment: now that LG indicates the head-verbs of sentences and clauses, the primary *raison-d'être* for RelEx, the major reason for it's complexity, goes away. It does not seem to provide a lot of value-add any more.

4.1.3 Synonymous phrases

The infrastructure needs to support synonymous phrases, not just synonymous words. A proper definition of the problem, together with prototyping is required. The current architecture is inadequate.

4.2 Process Flow Issues

Several issues arise in the design of rules, and in the control of their application. Specifically, there has been little or no design effort invested in managing processing pipeline.

4.2.1 AnchorNodes vs. URE

The way that information is passed around, from subsystem to subsystem, and the way that it interacts with the pattern matcher can be at times awkward. Some general improvements are needed in this area. They should not be hard, but they are significant, architecturally.

The `AnchorNode` was invented to be an attachment point, where different subsystems could leave collections of atoms that other subsystems could pick up. This works OK, in some situations, but has issues. The RelEx subsystem drops off the reesults of a parse at an `AnchorNode`. This is OK, because RelEx is a process external to the `AtomSpace`, and new, incoming sentences are infrequent. However, the `AnchorNode` mechanism provides no hint of order or priority: if two things are attached, there's no hint of which came first, which is more important. Another issue is that its not thread-local, thread-atomic: two different racing threads could pick up the same work unit; currently there is no mechanism to avoid this.

A more serious problem is how the current prototype uses (abuses) the `AnchorNode`. The current prototype defines a number of rules (see the review above) but does not use the URE to apply them. Instead, it uses an *ad hoc* mechanism: different processing steps look for input attached to assorted anchors, and "return" thier results by attaching them to yet other anchor points. The informal goal of this *ad hoc* process is to be able to

designate sequences of processing steps: that certain rules should run only after other rules that ran earlier, and that certain rules should limit their scope of applicability only to certain situations.

This is just fine, you might think, except that each rule is more complex: it has to specify both the input anchor and the output anchor. In addition, one has to write some *ad hoc* scheme code watching each anchor point, and triggering if something arrives there. After running, it has to clear the anchor. It ends up being clumsy, cut-n-paste blobs of utility code.

The current URE, specifically, the forward chainer part of the URE, does not provide any interfaces for specifying this kind of chaining of rule-classes. It probably should.

If the URE is altered to provide this, then how should it be implemented?

At any rate, even without adding chaining support to the URE, the current prototype should be altered to remove most uses of the AnchorNodes, and replaced by the URE, instead.

4.2.2 URE bottlenecks

The URE currently limits the scope of its search by creating a new, stand-alone AtomSpace, copying all of the focus set into it, copying all of the rules into it, and only then applying the rules. This has the desired effect of limiting rule application to only the desired focus set. However, it surely has poor performance, as a result. [Issue #1004](#) tracks this.

4.2.3 Rule selection

The current prototype applies all rules to all inputs. This is fine, as long as there are only a few rules, but this will not scale, in general. A better solution is to first limit the number of possible rules to apply, based on the input. There already is infrastructure to do this, it's just not used in the prototype.

The existing infrastructure is a combination of `DualLink` and the `cog-recognize` function. It's already deployed in the AIML subsystem, where it is used to pick out a set of appropriate AIML rules. It is partly integrated into OpenPsi, but this could be improved.

4.3 Enhancements

The prototype ignores various established OpenCog subsystems and technologies. It also ignores several architectural directions that have been discussed, but have not been acted on. These are briefly reviewed here.

4.3.1 OpenPsi

OpenPsi as a flexible rule-selection system. It is not used in the prototype. How should it be used (aside from the rule-selection issue mentioned above?)

4.3.2 Fuzzy matching

The current system does not tolerate failures in the speech-to-text and the text-to-parse stages. Use of fuzzy matching during rule application should be able to make up for this fragility. Thus, for example, if the input is “*garbled garbled mumbmle turn left mumble*”, then the robot should still be able to latch onto the “*turn left*” portion of the input, and respond to that. Since there may be multiple fuzzy matches, some sort of confidence ranking would be required, as well.

4.3.3 Question-answering

The robot needs to be able to answer questions about self and the world. This has been prototyped, but very incompletely. From the architectural perspective, question-answering resembles 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).

However, the current prototype does not fit well with this idea. For example, 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`. These need rework.

Question answering requires a second pipeline, beyond that shown in figure 8. Of course, there must first be a conversion from English to the internal query language. The response must then be converted back into 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.

4.3.4 Enlarged world model

The current model of the self, and the world, is tiny. It consists entirely of emotional state, and the visible faces in the environment. It needs to get bigger.

4.3.5 More sensory input

The robot is currently almost completely starved for sensory input: it can see faces, and it can access the text that came from speech-to-text. The speech-to-text does not provide any audio cues: was it a whisper? A shout?

Raw sensory input needs to be converted into sensory perceptions: atomese representations of processed sensory input is needed. For example, if the audio volume suddenly got loud, and the visual field is suddenly moving, then maybe ... everyone is clapping? booing? getting up to leave? How should the robot react to these events? The reaction for an interrupted discussion must be different from that occurring at the end of a speech.

4.3.6 Action orchestration

One must be able to carry out multiple things at once. Ben has written about this. How it should fit in remains unclear, although it is obviously needed.

4.3.7 Acting memory and performance

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 what she was told, and what she did, and later on, be able to play that back as a performance. That is, there must be a mode where the robot behaves as an actor under directorial control: the actor can be instructed by the director, given stage notes, and respond to these.

Examples of such directives might be: *“do that performance again, except this time, do xyz more slowly”*.

Initial prototypes of this level of function should be “straight-forward”: one can record the control-language directives. They need to be marked up with timing information. Verbal editing might be harder, as portions of a directive, such as *“do xyz more slowly”* require associating the word “xyz” with some portion of the performance. This brings up a number of reference resolution issues, discussed previously.

4.3.8 General memory

A second, distinct type of memory is being able to remember past states of the world. A basic prototype of this should be relatively straight-forward. This memory would need to be made accessible to the question-answering subsystem, so that questions about the immediate past can be answered.

One minor associated infrastructure challenge is that a management layer for the Postgres DB interfaces is needed. The basic idea is that memories should not be lost during power-off or reboot. Progress on this infrastructure is tracked in [issue #734](#).

Memories must be segregated into classes: there are some things that need to be remembered, others that should not be.

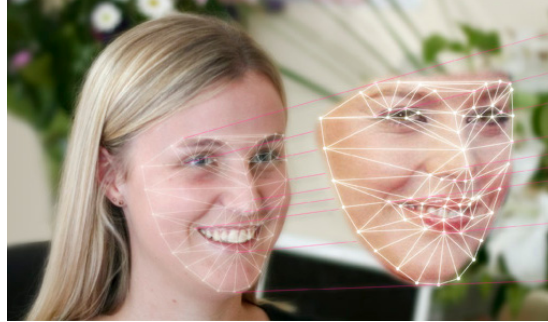
4.3.9 Learning

A deep, fundamental design issue is that, right now, essentially all of the rules are hand-authored, and not learned. Ultimately, the ability to learn is an imperative. This is a large topic that will not be covered here, with one exception. The next section describes a design proposal for expressive imitation learning.

5 Associating words and meanings to visual cues

Interaction with humans requires the ability to read facial expressions and gestural cues. This ability needs to be integrated with language abilities. This section proposes one possible way in which this could be done.

Figure 9: Facial landmark discovery



Example of facial landmark discovery. Software extracts boundaries of a face from an image, identifying significant landmarks, such as the eyes, nose and lips. These are referred to as “landmarks”. The relational structure of the landmarks is also presented: that is, neighboring points are identified by means of a triangular mesh. Knowing this relational structure is important, as it helps disambiguate points that (for instance) indicate the nose from points that indicate the mouth. This is a demo image taken directly from the [ci2cv website](#); it is meant only to be illustrative of the general idea, rather than promoting this technology over any other.

5.1 Imitation learning of facial expressions

Here’s a proposal for imitation learning of facial expressions. The goal is to be able to perceive different facial expressions, via analysis of incoming video, to remember what was seen, to mimic what was seen, and to associate words and language with the perceptions and memories.

Many of the details here could be done in a different way, so this is a sketch.

5.1.1 Landmark discovery

An imitation learning system needs to start with some way of perceiving human faces. One such system, for example, is [ci2cv](#), although there are others. This system is able to recognize facial landmarks, such as eyes, nose, lips and boundary of the head, and stitch these together into a mesh showing the relative relationship between these points. This is illustrated in figure 9. The goal of the landmark discovery system is to “dimensionally reduce” high-dimension data, i.e. the raw video pixels, to a much lower amount of data (dozens or hundreds of 2D or 3D floating-point numbers at 10 to 30 frames per second: that is, 500 floats per second to 15K floats per second) in such a way that the reduced data is still meaningful, and has some sort of reasonable fidelity to the input image.

There are a large number of reasons why this is a difficult task, including poor lighting, fast motion, and distant or blurry faces.

5.1.2 Expression recognition by classification and clustering

With a such dimensional reduction system in place, the next step would be to perform training and analysis of “typical” facial expressions. That is, working with a large collection of visual data, consisting of many different types of facial expressions on many different faces, the goal is to extract “typical” feature vectors. This does NOT have to be a labeled dataset; it would be better if it was not labeled, as the goal is to extract commonalities in the dataset, rather than to extract labels. Labeling comes at a later stage.

To extract unlabeled but classified and clustered data, it might be possible to use one of several relatively simple data analysis tools, such as principal component analysis (PCA), possibly other kernel methods, or possibly non-linear tools. Because unsupervised training is being proposed here, its not clear how to apply deep-learning neural net techniques.

The goal of this stage is to have a relatively small set of “face types” that are recognized by the system, perhaps a few dozen, but less than one hundred. Because the training is unsupervised, its not clear what common features are being learned: perhaps they are facial expressions (happy, sad), perhaps they facial types (round, square-jawed), perhaps they are racial (Asian, European, African), perhaps they are mouth-centered (open, closed-mouth, speaking vs listening), and perhaps they include face orientations (direct full-face, three-quarters face, profile). Perhaps the recognized types are data characteristics (well-lit, distinct, poorly-lit, blurry, distant, small). The point here is to somehow reduce the data rate of thousands of floats per second, to maybe 1 byte (256 different face recognitions) per second, and to have some sort of reasonable fidelity to the input image, including stability (low noise).

This appears to be a research task. Exactly how hard or easy it is to accomplish this is not clear.

5.1.3 Imitation learning

If the above can be accomplished, the next step would consist of imitation learning. That is, the robot would be set up to look at itself, using the same (trained) analysis pipeline. One would then embark on a search of the rather large space of possible motor settings, with the goal of associating certain particular motor settings and movements to the different types or classes being identified. If a motor setting is found that corresponds to known characteristic, then that motor setting is memorized, and associated with that type. An exploration of the size and shape of the motor parameter space needs to be performed: Similar motor settings should trigger the same recognition, but what, exactly, are the boundaries of what can be recognized?

The goal of this stage is to discover how to make the specific types of facial gestures that were learned. That is, the goal is to get a software API whose messages are of the format: “make facial expression type 7, or anything close to that”, as well as “now make a similar, but different expression of type 7”.

Such automated exploration and definition of boundaries should be relatively straightforward. What is not clear is whether the results will be significant in human terms: it may turn out that “expression type 7” encompasses everything from smiling to frown-

ing in profile, while raising one eyebrow: it might turn out that “type 7” is a grab-bag of nonsense expressions. This makes it clear that a form of feedback to the classification stage will ultimately be needed, later if not sooner.

Some, possibly many or most expression classifications might not be reproducible on the robot. For example, “type 7” might correspond to “poorly-lit and square-jawed European male” which is not something that any motor setting on a well-lit female face would be able to achieve.

Again, this appears to be a research task.

5.1.4 Language association

The goal of this stage is to associate specific words with specific facial expression classes. This would be done through natural language dialog. An example dialog might be:

Human: Can you see me?

Robot: Yes.

Human: I am smiling.

Robot: OK.

Human: Can you imitate smiling?

Robot: (Performs animation)

Human: That’s good (or) No, that’s wrong.

If a positive response can be elicited, then that particular word or phrase (smiling, turning away, hanging head down, etc.) can be associated with some confidence to two distinct things: the perceived feature vector, and also the motor settings. One learned, these referents can be used in other language dialog and discussion.

5.1.5 Feedback and continuous learning

It is possible, or even likely, that there will be serious data flaws at every step: landmark extraction may fail spectacularly in all but the best of lighting conditions. Feature classification may group together features in a way that is not natural to humans. Imitation learning may fail to find a motor match to most expressions. Language association may appear to work, but later fail, because the word “smile” was associated with a feature set that includes turning away and opening the mouth, while shutting the eyes. Thus, mechanisms for un-learning word associations are required, or of transferring them to new classifications, as these are learned.

Because all of these are hypothetical issues, proposing distinct solutions for these seems premature, at this point, with the exception of language association. That is because the creation of associations between perceptions and words is a broader task than merely that of perceiving facial expressions. This is discussed in greater detail in section XXX (where?)

6 Ideas

This section provides some brief sketches of interesting behaviors that should be achievable within the current prototype framework, or are suitable for the next step of prototyping. Some of these are very easy, but simply have not been explored or done.

- A person walks into room, who she recognizes. Depending on the current emotional state, she should perform non-verbal greetings, acknowledgements (play one of 3-4 different animations, such as brief look-at glance, chin push, nod) and verbal greetings (“*hello, what’s up?*”). The current interplay between state and psi rules is a mess, these need to be split up properly. There are additional comments in [issue #80](#), which see.
- Someone asks: “*so what are we talking about?*” This requires the robot to have some rudimentary memory of the conversation, and have the ability to recapitulate it. A simple prototype mock-up would be to extract keywords/key-topics from sentence, and remember them. These can be queried with the fuzzy matcher, and used to recapitulate the conversation.
- Someone says “*Sophia, bark like a dog!*” A barking dog animation is needed.
- Someone asks: “*Why did you smile?*” Output: have her explain the most recent OpenPsi decision-making. This requires recording that decision-making into a set of atoms, queryable using the above-described question-answering subsystem.
- Someone says: “*I’m so sorry about that.*” Output: have her perform a small, cute pout (a frown, but without the eyebrow animation).
- Someone says: “*Look at me.*” The robot should determine the number of visible faces. If only one, then it could say “*I see only one face, and I am looking at it now*”. If there are two faces, then either change the gaze, or use the audio localization to determine the audio source, and respond appropriately, by changing gaze, or keeping it, and appropriate verbal remarks.
- Someone says: “*What are you doing?*”. Describe a the recent past, indicating the presence of memory. The verbal reply should show that the robot is sentient of it’s surroundings.
- When last person leaves, she should say goodbye. If no goodbyes were exchanged, then remark loudly, “*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?*”
- If no one is visible, but there is an ongoing chat session, she should complain about the visibility of the chat partners. If there is no ongoing chat session, but there loud noises (and she cannot see any faces), she should say something like “*Hey, what’s going on? Who’s there? Show yourself!*”

7 Conclusion

To be written.

Appendix A - Reasoning

The relationship between rules, rule application and reasoning continue to be a point of on-going confusion in the community. This appendix attempts to provide an abstract, but mathematically precise, view of what it means “to perform reasoning”. The desire here is to move past the current conception of reasoning as a kind-of forward/backward chaining, and bring it into a more modern era of reasoning as parsing. This change of viewpoint is required in order to be able to formulate faster, more robust reasoning algorithms, and thus to be able to perform far more complex inferences and deductions.

Jigsaw puzzle pieces and dominoes

One particularly direct way in which reasoning-as-parsing can be understood, is as the assembly of a jigsaw puzzle. Here, each rule is represented by a jigsaw puzzle piece, and rule application can be imagined as a chaining together of jigsaw puzzle pieces. Thus, for example, if a particular rule requires, say, an input of a particular kind, and generates an output of a particular kind, then each of these can be envisioned as a distinct connector on a jigsaw puzzle piece. A rule requiring a particular input can *only* be mated to something that provides that actual shape. The next rule to run, must, of course, also be of a kind that can properly mate.

Unlike an ordinary jigsaw, there can be many different puzzle pieces having the same exact shape. This allows the reasoner a lot of latitude, a lot of discretion, as to which pieces might be chosen and used next, during this chaining operation. Perhaps the game of dominoes offers a more accurate model: for one’s next move, one can lay down any one of a number of different dominoes, in any one of a number of different locations, and one has many choices as to what to play next. However, those choices are constrained: one cannot just lay down any domino anywhere; one must follow certain mating rules. Unfortunately, the game of dominoes is not widely known, and so the model of many identically-shaped jigsaw puzzle pieces will have to do.

Combinatoric explosion

The large number of choices as to which jigsaw piece to lay down next leads to the infamous problem of “combinatoric explosion” during chaining. At every step of the chain, there seem to be ever more choices possible, and the sheer number of these choices appears to overwhelm any hope of straight-ahead chaining.

When met with this problem, one commonly proposed solution is some form of “inference control”, some way of whacking down the number of possible choices. A different approach, that of adding more constraints to the system, is almost never considered. I believe that is because the actual mechanics of reasoning is not generally understood, and thus obscures algorithmic possibilities. However, before embarking

on an exploration of constraints and inference control, some additional general concepts must be laid down.

Side-note: there are extremely simple rule systems with horrific combinatoric explosion properties. See the Azimuth blog on racks, quandles, shelves, and Laver tables for more.

Chaining

The use of the word “chaining” is very misleading: it very strongly suggests a linear sequence of events, like a chain. Nothing could be further from the truth, and this one word alone is the source of huge conceptual stumbling blocks. Real-world puzzle-pieces are two-dimensional, and, just like the case of dominoes, there is no particular constraint that pieces must be laid out in linear order. Attachments can be made in any number of different directions.

At first, one might imagine that the combinatoric explosion can be even worse. In fact, this can serve to constrain the problem. There are three distinct ways in which this is done: one way is statically, by means of systemic constraints; another way is dynamically, by means of consistency; a third way is to enforce completeness.

Systemic constraints

In real-world dominoes and puzzle assembly, there are two constraints: a finite-sized table, whose edges cannot be crossed, and the constraint that pieces must not be laid on top of one-another. In the assembly of abstract rule systems, there are other constraints that can come into play. One is a no-crossing constraint, often used in theories of grammar (e.g. Link Grammar). This is essentially the same as saying “two pieces must not be laid on top of one-another”. Another is a relaxation of this constraint to use landmark transitivity (e.g. in Word Grammar).

Consistency

Another way to constrain a puzzle assembly problem is by requiring consistency. This can be achieved with an *a priori* boundary condition: one might have some edge, or boundary, and require that the final assembly *must* meet that boundary, without violating it. Imagine, for example, a string or chain of puzzle pieces, given *a priori*. As new pieces are added, they *must* fit into the given chain.

An example of a consistency constraint is the link-grammar parse of a sentence: A list of words, the sentence, is given *a priori*, and one must assemble puzzle pieces in such a way that these words, and only these words, are connected to.

A looser example might be the constraints of evidence in a crime: the evidence is given *a priori*, and the detective must show that the suspects behavior is consistent with each of the points of evidence. Here, the puzzle-pieces correspond to the grouping (suspect, time, place, capability, opportunity); each element of the grouping is like a connector in the puzzle-piece, and it must connect appropriately with other pieces, or with the evidence. In criminology, this consistency constraint tends to very sharply narrow the list of suspects: combinatoric explosion is mitigated by constraint.

Completeness

Yet another way to constrain a puzzle assembly is to require completeness: there must not be any connector tabs left unconnected! If a connector is left unconnected, one is not allowed to ignore it, or cut it off, or pretend its not there: instead, one declares the arrangement incomplete, and discards it.

The completeness constraint is well-known in proof theory; it corresponds to the idea that both contraction (**idempotency of entailment**) and weakening (**monotonicity of entailment**) are forbidden.

The completeness constraint is employed both in link-grammar parsing, and in criminology (all criminal evidence must be explained; it cannot be arbitrarily ignored).

Unfinished stuff

DRAFT VERSION 0.2

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

—
practical examples, for PLN.

—
so how is assembly to be performed?

—
difference between counting (performing the assembly) and linkage parsing (evaluating the truth values, i.e. applying the actual the PLN formulas) which is done only after a valid (complete, consistent) assembly has been obtained.

—
A universe is like a partially assembled puzzle.

—
Relationship between

—
Andre Joyal lemma

—
The word “choice” in the above is not an accident ... ditto for contraction and weakening in the “completeness” section – linear logic .