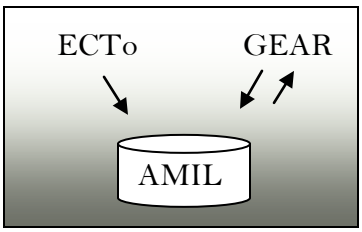
Demonstration of Ontological Reasoning

As a demonstration of the META language toolset, we want to apply a customized archetypal reasoner (GEAR) to help with the down-select from a set of ECTo design alternatives.

The pre-condition is that the reasoner needs to know what part of the conceptual model to filter on. The user of the ECTo model facilitates this by tagging those elements with a "DESIGN_SET" property. This then gets saved into the current graph database as additional triples to the individual elements. The "DESIGN_SET" becomes a property of its parent.

The reasoner next accesses the graph database to get all the relevant information needed to make a decision. It does this by making an HTTP request to the `ArrowWebServices` entry called `/arrow/arrowGraphData`. The response to this call is a JSON text string that contains the representation of the internal triple-store data (with the active content executed).



The JSON is parsed and then stored in the GEAR reasoner's local knowledgebase as an equivalent set of triples (the active content has been executed so no loss of information occurs at this point). The predicate logic for doing this is in the following code snippet:

```
process_triples(_, []).
process_triples(Subject, [(Pred=Obj)|Rest]) :-
   assertz(triple(Subject,Pred,Obj)),
   process_triples(Subject, Rest).
clean_triples :- retractall(triple(_,_,_)).

scan_amil([]).
scan_amil([json([A,(B=json(List))]) | Rest]) :-
   process_triples(B, [A|List]),
   scan_amil(Rest).
scan_amil([First | Rest]) :-  %% If we dont understand discard
   write(user_error, First), nl(user_error),
   scan_amil(Rest).
scan_amil(_) :- write(user_error, 'Not in AMIL format\n').

stream_json(Stream) :-
   clean_triples,               write(user_error, '% starting to stream\n'),
   json_read(Stream,Text),      write(user_error, '% finished stream\n'),
   json_to_prolog(Text,Prolog), write(user_error, '% converting to
prolog\n'),
   scan_amil(Prolog).
```

Figure 1: JSON parser and store rules

This essentially scans the JSON stream, pulling out each of the AMIL data items and saving these as (*subject*, *predicate*, *object*) triples. Anything not recognized in this format is discarded (such as AMIL create, preconditions, and postconditions).

At this point, all the information is available locally to do sophisticated reasoning against. The following snippet of logic queries the local knowledgebase for individual elements of a "DESIGN_SET", next reasons about specific derived properties, and then applies a set of criteria to those properties. This is essentially a logic+control strategy, with a search through the description logic filtered through control predicates.

For the demo we keep it simple, with the understanding that this ontological rule can easily be changed. For this case, we decide to filter for the element with the largest value of mass density.

```
triple_num(Subj, Pred, Obj) :-  % convert atom to number
   triple(Subj, Pred, O),
   atom_number(O, Obj).

get_design_set(S, mass, Value) :-     % triple store lookup
   triple(S, parent, 'DESIGN_SET'),
   triple_num(S, mass, Value).

get_design_set(S, volume, Value) :-   % lookup with computation
   triple(S, parent, 'DESIGN_SET'),
   triple_num(S, height, V1),
   triple_num(S, width, V2),
   triple_num(S, length, V3),
   Value is V1*V2*V3.

get_design_set(S, density, Value) :-   % higher-level rule
   get_design_set(S, mass, V1),
   get_design_set(S, volume, V2),
   Value is V1/V2.

maximize_value([], (Subj,Amt), (Subj,Amt)).
maximize_value([(Subj,Amt)|Rest], (Subj,Initial), Value) :-
   Amt > Initial,
   !, maximize_value(Rest, (Subj,Amt), Value).
maximize_value([(_,_)|Rest], (Subj,Initial), Value) :-
   !, maximize_value(Rest, (Subj,Initial), Value).

find_maximum_density(Subj, Density) :-
   findall((Subj, Value), get_design_set(Subj, density, Value), L),
   maximize_value(L, (_,0.0), (Subj,Density)).
```

Figure 2: GEAR reasoner which finds the maximum density from elements in a set.

The logic predicates labeled `get_design_set` polymorphically match to the ontological constraints of mass, volume, and density. The mass predicate looks up the mass property directly. The volume predicate requires matches for the three dimensions of height, width, and length and then computes the volume after successfully binding to values for each property. The even higher-level density predicate combines the mass and volume predicates to calculate an element density.

This demonstrates the powerful potential for reuse of rules via ontological classification.

A reusable rule called `mazimize_value` searches a set of paired tuples for the maximum value after the meta-call `findall` provides a list of potential candidates.

The top-level rule `find_maximum_density(Subj, Density)` is potentially invoked as a web service or as a command line invocation.

For example, from the server logic below the HTTP query
http://localhost:5000/load?amil=http://localhost:8080/ArrowWebServices/arrow/arrowGraphExport

will load from the local AMIL server and http://localhost:5000/query will execute the `find_maximum_density` query.

```
server(Port) :-
   http_server(http_dispatch, [port(Port)]).

:- http_handler(root(home), index_page, []).
:- http_handler(root(load), load_data, []).
:- http_handler(root(query), max_density, []).

index_page(_) :-
   reply_html_page(title('Home'),
      [ h2(a(href('load?file=JSON_graph.txt'), 'Load data from a file')),
        h2(a(href('load?amil=http://wcsn262:8001/backup/JSON_graph.txt'),
                 'Load data from an arbitrary web-served file')),
        h2(a(href('load?amil=http://localhost:8080/ArrowWebServices/arrow/arrowGraphExport'),
                 'Load data from a local AMIL server')),
        h2(a(href(query), 'Query data example')) ]).

load_data(Request) :-   %% File name version
   http_parameters(Request, [file(Name, [ optional(true) ])]),
   nonvar(Name),
   load_json(Name),
   write(user_error, Request), nl(user_error),
   reply_html_page(title('File loader'), [p(Name), p(' load completed.')]).

load_data(Request) :-   %% URL version
   http_parameters(Request, [amil(URL, [ optional(true) ])]),
   nonvar(URL),
   http_open(URL, In, []),
   stream_json(In),
   close(In),
   reply_html_page(title('AMIL loader'), [p(URL), p(' load completed.')]).

max_density(_) :-
   find_maximum_density(Subj, Density),
   reply_html_page(title('Max Density'),
                   [ h1('Max Density Query'),
                     table([tr([th('Engine'), th('Density')]),
                            tr([td(Subj), td(Density)])]) ]).

:- server(5000). %% load server
```

Figure 3: GEAR web server with handlers pointing to rules.

The reasoner code is high-level enough that knowledge engineers can quickly adapt design rules and analysis archetypes as customized GEAR rulebases and store these in CML.

1. INSTRUCTIONS

These are the instructions to run the GEAR demonstration on Windows

- Change directory to `trunk\galileo\gear`
- Execute `'run_lp.bat'` and leave it running as a server (exit if needed by entering `halt.`)
- From a web browser connect to http://localhost:5000/home
    1. Any new load will clear out the previous set of data
    2. The query should return something interesting if the "DESIGN_SET" node is in AMIL, see below, otherwise no match on return