Overview of the MCL ODL
Matt Schmill
7/6/2009

Ontology Description Language (ODL)

The MCL ODL allows users to describe the Indication, Failure, and Response ontologies of MCL in a more readable, flexible manner than was possible by building them directly with C++ code. This document describes the format of an ODT file, as well as how one is used. Supplied ODL files can be found in the MCL distribution under config/netdefs. See the file config/netdefs/basic.ont for an annoted example of a small, simple definition file that can be used by MCL.

At the top level, the ODL can process two structure types: *ontology* and *linkage*. Ontology blocks are used to define nodes, grouped into ontologies, and linkage blocks are used to define linkage between the ontology nodes.

The "ontology" block

First, we consider ontology blocks. They take the form:

ontology <name> (node_def1 ... node_defN)

For MCL's purposes, there should be at least three ontology blocks per ODL file: "indications", "failures", and "responses". **Those ontologies must be defined.** Other ontologies are technically allowed, although they are not guaranteed to work with all types of glue (they should work fine with SMILE glue). Inside the ontology block, nodes are defined using the *node* construction:

node <type> (name=nodename, arg1=argval1, ... , argN=argvalN)

where legal arguments arg1 ... argN are defined by the node type specified. For example,

node iCore(name=stuck, doc="a spatial sensor has stopped changing as specified.")

Note that the name parameter, *stuck*, does not need to be enclosed in quotation marks, but the *doc* parameter does. String literals containing spaces or commas should be enclosed in quotation marks. Other literals need not be.

As of this writing, there are 8 node type specifiers that correspond to the 8 implemented node types in MCL:

- *hostProp(name,propcode,doc)*
  A host property node.
  name is the node name
  propcode is an integer or symbol referring to the internal MCL property constant(see mcl_symbols.h or "Overview of the MCL API")
  doc is a documentation string

- *genInd(name,doc)*
  A general-purpose indication node.
- *concInd (name,doc)*
  A concrete indication node (one that can be clamped by MCL on an expectation violation).
- *iCore (name,doc)*
  An indication core node (one that is derived from other indications and links out to the failure ontology).
- *HII(name,doc)*
  A Host-Initiated Indication (one that is clamped by MCL as a response to a call from the host).
- *failure(name,doc)*
  general purpose failure node
- *genResponse(name,doc)*
  general purpose response node
- *interactive(name,rcode,yes,no,doc)*
  an MCL response that will receive yes/no feedback from the host.
  Rcode is an integer or symbol referring to the internal MCL response constant (see "Overview of the MCL API").
  Yes is the name of the node to clamp to 1 if the host responds positively to the interrogatory.
  No is the name of the node to clamp to 1 if the host responds negatively to the interrogatory.
- *concResponse(name,rcode,doc)*
  concrete (implementable) response node

The "linkage" block

Next, we consider linkage blocks. They take the form:

linkage <name> (link_def1 ... link_defN)

The *name* parameter is for descriptive purposes only and will be ignored by the parser. There may be as many linkage blocks as you like, and they may be interleaved with ontology blocks or not, but they must appear at the top level of the ODL file. sThe only other restriction is that the links are instantiated as they are read – so it is important that links are only defined after the nodes they connect have been created. You will note in basic.ont, that all the linkage is created in a single block at the end of the file. It may be more logical/readable to define intraontological linkage in linkage blocks directly following the ontology blocks they are defined over.

A link is defined similarly to a node:

link <type>(src=srcnodename, dst=dstnodename)

As of this writing, there are 7 link types. Each type takes a src and dst parameter, which names an existing node.

- *link abstraction(src,dst)*
  dst is an abstraction of *src* (indication or failure ontology intraontological link)
- *link IFC(src,dst)*

a link from an indication node to an indication core node

- *link specification(src,dst)*
  dst is a specification of *src* (response ontology intraontological link)
- *link diagnostic(src,dst)*
  the indication node *src* suggests the failure node *dst*
- *link inhibitory(src,dst)*
  the indication node *src* suggests the response node *dst* is not useful
- *link support(src,dst)*
  the indication node *src* supports the probability that response node *dst* is useful
- *link prescriptive(src,dst)*
  the failure node *src* prescribes the response node *dst*

Additional ODL

Any line beginning with the pound sign '#' is considered a comment and is ignored by the ontology reader.

```
# this is a comment explaining part of the ODL code
```

Ontology specifications may also span multiple files. This is useful when a common core is to be used for several different host systems. The more specific *fringe* nodes may be included in a separate file uniquely tied to the agent that used them. For example, we might create the file *rover.odt* as follows:

```
include basic
ontology failures (
        failure(name= bogusMap, doc= "class of failures where the map is bad")
)
linkage all (
        abstraction(src= bogusMap, dst= predictiveModelError)
)
```

Note that the argument to include may either be an ontology basename, or a fully specified pathname (it *must* start with a slash '/' in this case).

Verifying the ontology structure

It is sometimes useful to verify that an ODL file is generating an internal representation that matches a desired result. This is most easily done by producing a graphical output of the generated ontologies for viewing by the ontology designer. This can be done using tools provided in the MCL distribution. The user must make sure that **dot** is installed on your host machine for these tools to work.

First, build the dot generation app by making the target dot. From the shell prompt, in the mcl_2_0 directory.

```
make dot
```

Then, from the mcl_2_0 directory, run the supplied shell script dot2ps, providing the root of the ontology filename as an argument. Thus, if you wanted to generate graphs for basic.ont, you would run the following shell command:

bin/dot2ps basic

This command will use the ontology reader to read the ontologies in basic.ont, produce dot files in docs/ontologies, and then use the dot application to generate ps files in the docs/ontologies directory. You can use these to verify the correct structure is being generated. You can also use the output to console to help diagnose errors in the ODL file.