MODEL-BASED AUTONOMY ABOARD MBARI AUVS: PROJECT HANDOVER DOCUMENT

Shuonan Dong[*], MIT
Mentor: Kanna Rajan
Summer 2006

Keywords: AUV, autonomy, model-based programming, fault diagnosis

ABSTRACT

We desire more autonomy aboard MBARI AUVs, especially in the area of onboard fault monitoring and diagnosis. Thus we would like to introduce model-based autonomy aboard AUVs to perform the fault diagnosis task. This paper discusses the concept of model-based autonomy and introduces a model-based executive called Titan.

INTRODUCTION TO AUVS

MBARI's autonomous underwater vehicles (AUVs) are designed for unmanned and untethered scientific ocean sampling. The goal is to reduce personnel costs from sampling operations. The vehicles are modular and reconfigurable to host different scientific payloads amidst the basic components such as the propulsion, navigation, power, control, and emergency location systems.[1] An example of an AUV is shown in Figure 1.

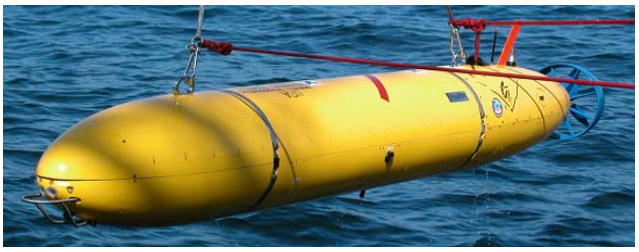

Figure 1. The Conductivity Temperature & Depth AUV has a modular design.

[*] Contact: <dongs@mit.edu>

AUV HARDWARE

In the current configuration of the Conductivity Temperature & Depth (CTD) AUV, the CTD instruments are located in the nose of the vehicle. The three tubular devices toward the left side of Figure 2 are the CTD instruments, which includes the temperature, conductivity, and oxygen sensors. The depth (i.e. pressure) sensor is located in the aft of the vehicle. The mid-body of the AUV contains mainly the lithium-polymer batteries (as shown in Figure 3), and other optional components such as the gulper or Ultra Short Base Line (USBL). The three large battery cells are connected in parallel to a power supply in the main vehicle computer.

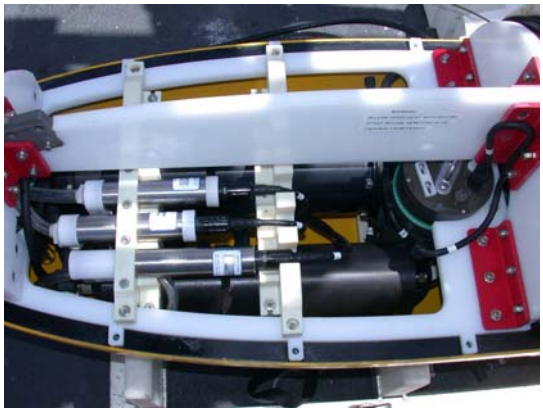

Figure 2. The CTD instruments are located in the nose of the AUV.

Figure 3. There are three Li-Po batteries in mid-body of the AUV.

The main vehicle computer (MVC) is located in the aft portion of the vehicle, as shown in Figure 4. It is a pressurized bubble containing most of the onboard electronics. Figure 5 shows a diagram of the components connected to the MVC.

Figure 4. The main vehicle computer is located in the pressurized bubble toward the aft of the vehicle.

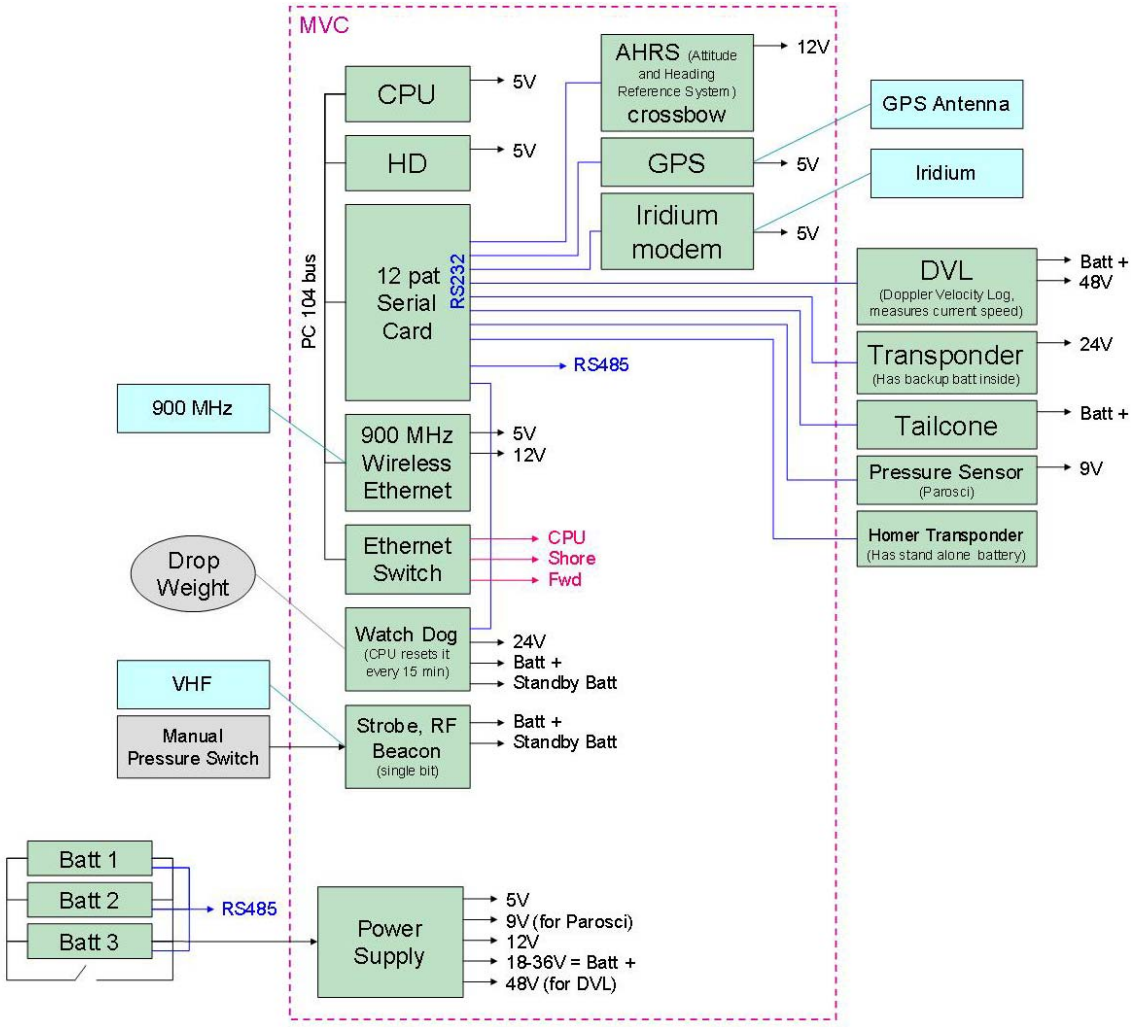

Figure 5. The main vehicle computer contains most of the onboard electronics.

The scientific instruments are connected to the MVC via Ethernet. There are several different scientific instruments onboard, including the Hydroscat2 which measures turbidity and chlorophyll fluorescence, ISUS which measures the amount of nutrients in the water through UV absorption spectra, Bathyphotometer which measures bioluminescence, LISST-100 which measures particle sizes, and two OCR-507s which measure radiance and irradiance, respectively. Connected to the CTD boards are the CTD instruments, including SBE 3 and 4, which measure temperature and conductivity, SBE 43 oxygen sensor, and SBE 5T pump. A multifunction board ensures flow through the CTD. These components are connected as shown in Figure 6.

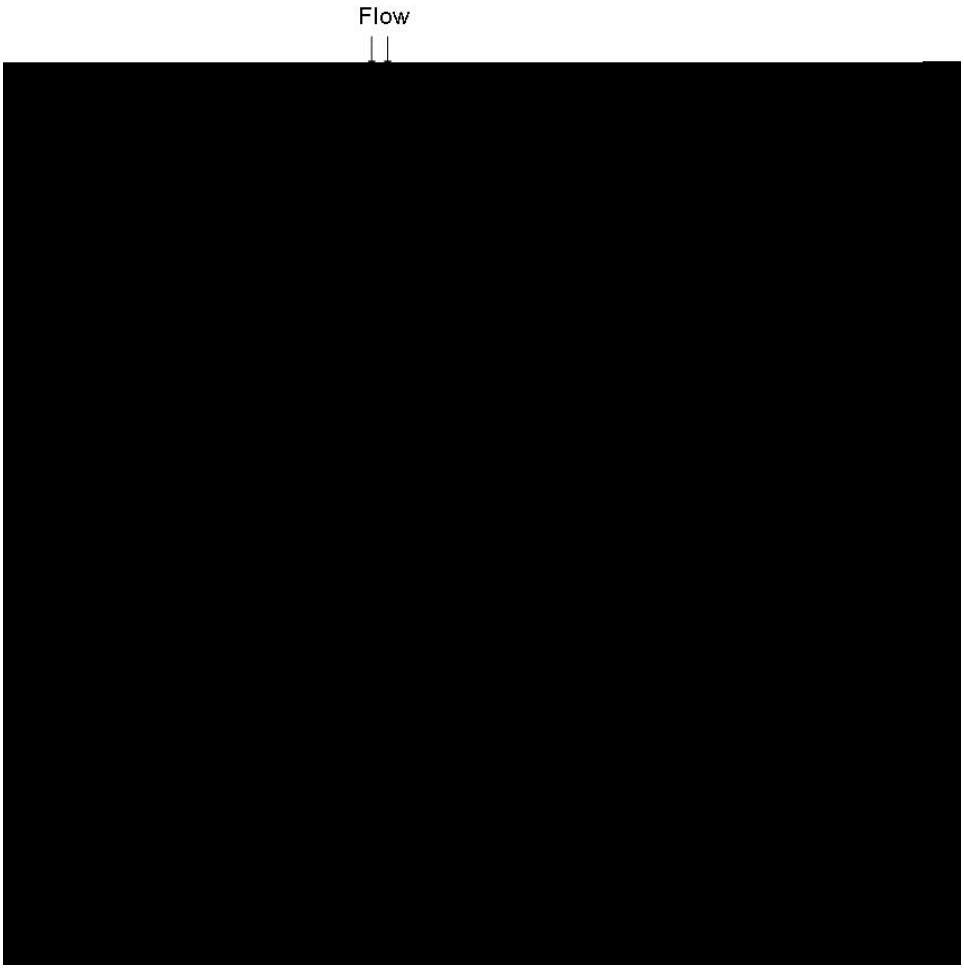

Figure 6. The scientific instruments are connected via Ethernet.

User manuals and specification sheets for most of the scientific instruments are located at \\tornado\ProjectLibrary\368000_DMO_AUV\manuals\Payload_Instruments.

AUV SOFTWARE

The MBARI AUV software uses the subsumption architecture. Each action that the vehicle can take is called a "behavior." Under the subsumption architecture[2], one behavior takes over (i.e. "subsumes") another in a forward moving linear fashion. A mission operator must decide what behaviors to trigger and for how long. Each behavior has a set of parameters that are defined. For example, the "ascend" behavior might have parameters called "duration," "horizontal," pitch," "speed," and "endDepth." The mission operator creates a mission file (*.cfg format) which contains the behaviors of interest. The mission file is executed by the onboard computer from the bottom upward (i.e. the

AUVs might be moving among hydrothermal vents. Dynamic planning would greatly enhance the vehicle's capability to collect the most relevant data near the vents.

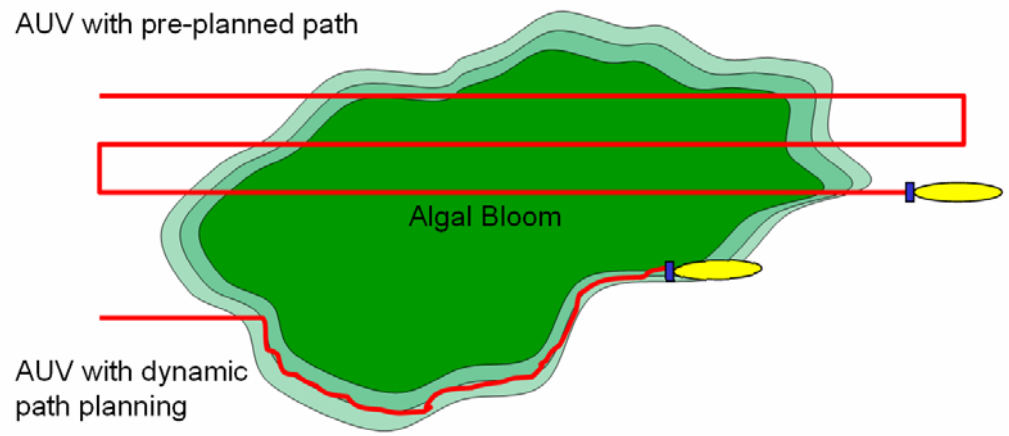

Figure 8. Dynamic path planning could be useful to characterize the boundary condition for an algal bloom.

Finally, in the future, we would like to employ multiple autonomous assets to work together to perform larger ocean observing tasks. This would require distributed command and control technology for multi-vehicle systems, which is another area of research in artificial intelligence.

MODEL-BASED AUTONOMY[3]

The ideal high level software architecture of an AUV is shown in Figure 9. We would like a deliberative planner to generate detailed AUV mission plans from a goal level task plan which the mission operator can create. Currently, AUV mission operators must generate the detailed mission plans themselves. The plan is fed into the executive, which is responsible for carrying out the plan to the controlled system. Usually a part of the executive, the fault monitor compares the commands given to the controlled system by the executive to the output of the controlled system after it interacts with the environment. If a fault is detected, the fault monitor will use a set of algorithms to diagnose probabilistically which component(s) triggered the fault. This information is returned to the executive, which will decide what kind of recovery procedure is necessary.

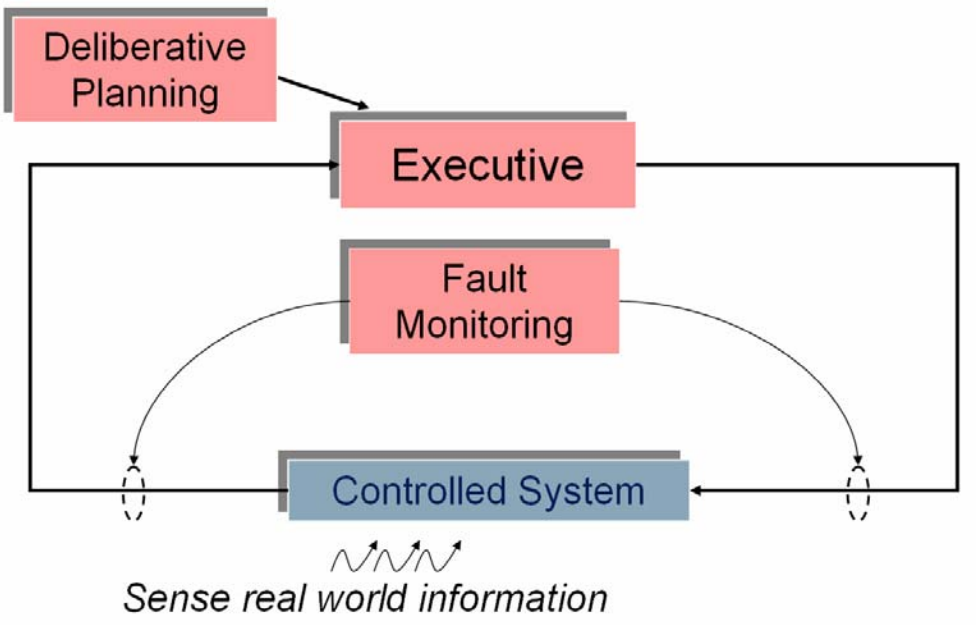

Figure 9. The high level software architecture of a system would ideally incorporate fault monitoring.

In order to reason about embedded systems, we can follow system state evolutions. If we know the current state of a system and the possible state transitions, we can probabilistically predict or estimate the next state (also called mode) of a system. Fault monitoring is a specific case of mode estimation. In order to estimate the states of a system, we must describe the system in a formal language. Large systems can be described by subsystems and their interactions. The set of states, state transitions, and subsystem interactions can be captured into a model to fully describe the behavior of a system. Model-based autonomy uses such a model written in some model-based programming language to perform estimation, reconfiguration, or execution for the system.

The model-based executive of interest for this project is called Titan, developed by the Model-based Embedded & Robotic Systems group (led by Professor Brian Williams) in the Computer Science and Artificial Intelligence Lab at MIT. As a side note, Titan is a derivative of Livingstone, the model-based executive developed at NASA Ames and successfully deployed on the Deep Space One spacecraft in 1999. The architecture of the Titan model-based executive is shown in Figure 10.

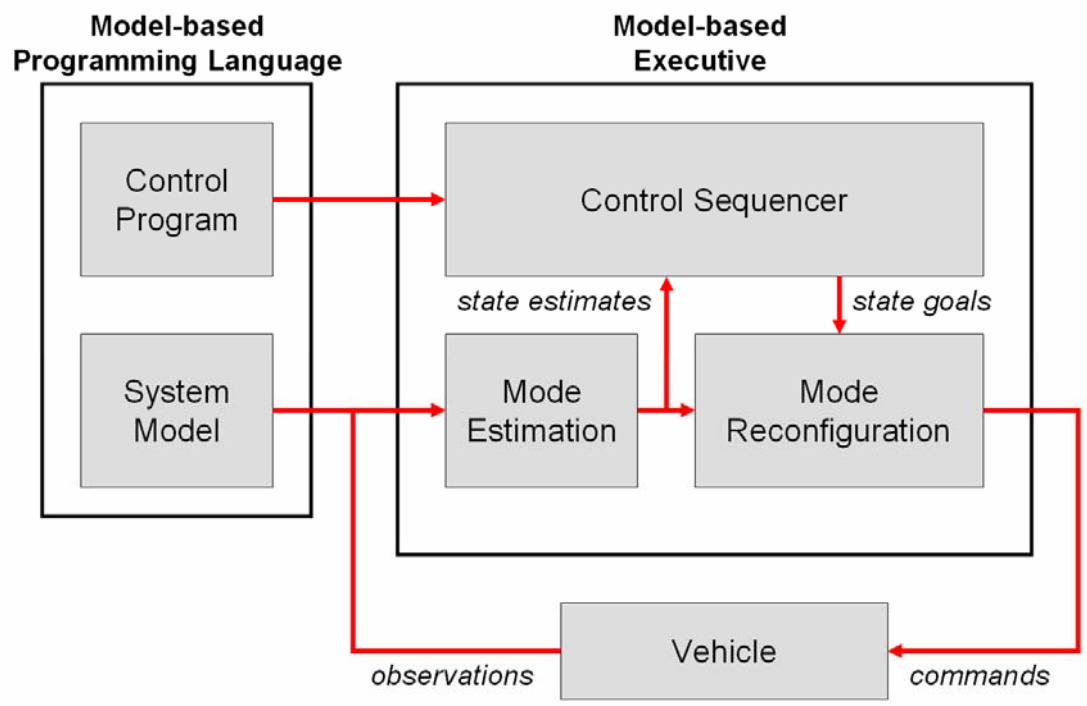

Figure 10. The Titan model-based executive reads in control programs and system models written in a model-based programming language.

Since the MBARI AUV already has an onboard executive which carries out the mission operators' plans, the part of Titan that we are interested in includes only the system model and the mode estimator.

LOGISTICAL CONNECTIONS

Now I will combine all the aforementioned system architectures and provide a more concrete idea of how everything works together. See Figure 11 for the general flow of procedures. An AUV has onboard instruments that output continuously sampled data which gets stored in netcdf log files. Example mission log files can be found at \\tornado\AUVCTD\missionnetcdfs. Netcdf files can be read in from MATLAB with the netcdf toolbox. Obtain the toolbox from Michael Godin. Typing `help netcdf` in the MATLAB command prompt can provide a list of commands. A typical session for reading a netcdf file could look like:

```
addNetcdfPath;
nc = netcdf('measurements.nc'); % nc is the netcdf file object
nc_names = ncnames(var(nc));
```

```
      % nc_names is cell array of variable names in the object
nc_temp = nc{'temperature'};    % nc_temp is a netcdf variable object
temperature = nc_temp(:);       % get the actual data for use in MATLAB
nc_temp_dims = ncnames(dim(nc_temp));
      % cell array of names of dimensions of the variable
nc_time = nc{nc_temp_dims{1}};  % assuming first dimension is time
time = nc_time(:)*86400+datenum(1970,1,1);
      % gets time and converts to MATLAB time
...
close(nc);  % closes the netcdf object.
```

The data from a log file shows a string of events, and when matched with the string of commands during the mission, can simulate live command and output sequences during an actual mission. The logged data are continuously sampled, so we must discretize them, then convert to a Titan readable file, before feeding to Titan as input.

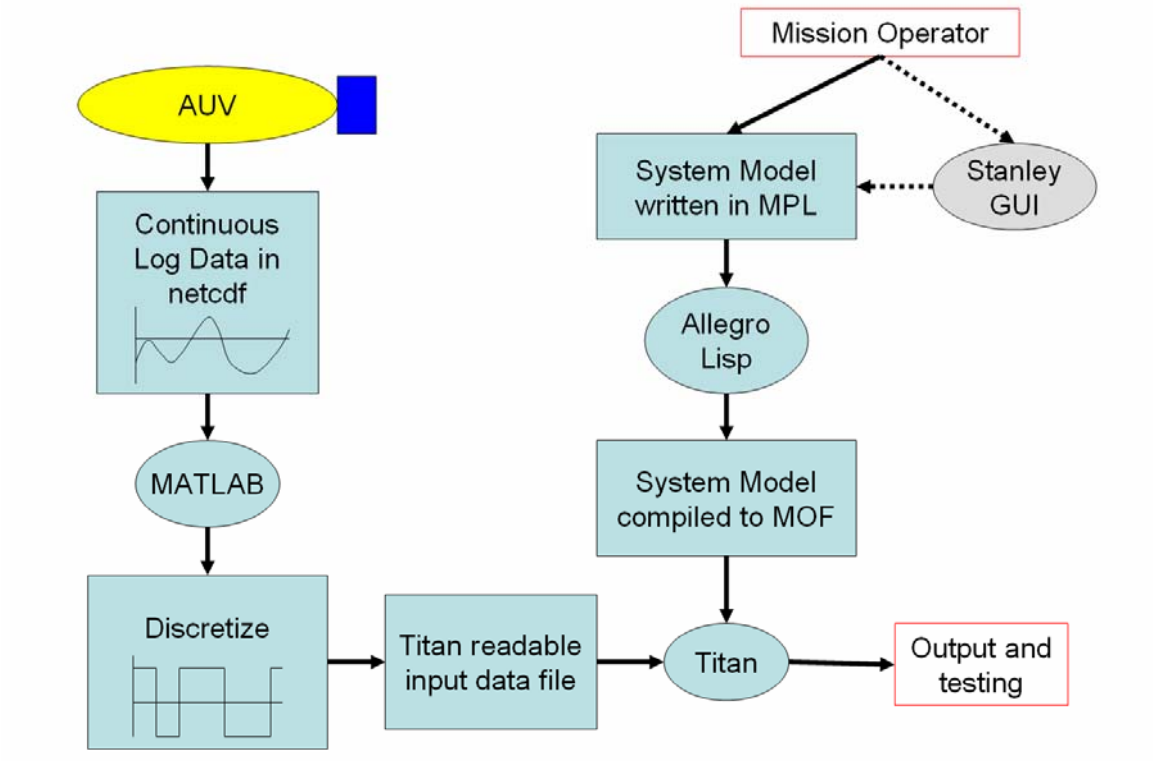

Figure 11. There are many steps in setting up the software to work together.

The mission operator must create a system model, either by directly writing it in the model-based programming language[4,5] (MPL) or (more ideally,) through a GUI called Stanley, which is an open source software provided by NASA Ames as a part of the Livingstone 2 package. The Livingstone 2 package can be found at

http://opensource.arc.nasa.gov/project.jsp?id=6. Keep in mind, however, that Livingstone 2 with Stanley was created a few years ago, and requires many supporting software programs to run, many of which are near obsolete versions. Nonetheless, if you can get Stanley to work, it will greatly increase the usability and efficiency of creating system models. After the system model is generated in MPL, it needs to be compiled to a MOF format via Franz Allegro Common Lisp 6.0 in order to be read by Titan. It is possible for mission operators to write their own MOF files, but this is a highly tedious and time consuming task.

RUNNING TITAN

Titan can be run on a Windows machine via Cygwin. Under the ModeEstimation directory, run the Titan engine by typing `./me_driver-CygDebug.exe ./?.mof` where `?` is the filename of the system model to be used. Enter just `./me_driver-CygDebug.exe` to see the options for adding an input file, etc. Once the Titan engine is running, enter `h` for a list of options. An example session in Titan may look like the following, where the user input is highlighted in red.

```
dongs@csail-759f66398
~/Titan/src/Mers/Estimation/Discrete/ModeEstimation
$ ./me_driver-CygDebug.exe ./switchOrGate.mof
Loading model './switchOrGate.mof'...
Success!

Titan ME Shell: Enter 'h' for help.

1> h
  var val ...            : assert (var = val) using integer UIDs or
string names

  a [option] var val ... : assert (var = val) for specified types and
trajectories
  e [option]             : estimate the next state
  k #                    : sets the maximum number of trajectories
  r                      : resets commands and observations
  v [option]             : displays variables
  h [option]             : displays this help
  d #                    : sets the debug level
  #                      : toggle echo input
  q                      : quit

2> v
State Variables:
SWITCH2.MODE[0] { >+BROKEN[0]:1, ON[1]:0, OFF[2]:0 }
```

```
OR_GATE.MODE[3] { >+BROKEN[0]:1, NOMINAL[1]:0 }
SWITCH1.MODE[7] { >+BROKEN[0]:1, ON[1]:0, OFF[2]:0 }

Command Variables:
SWITCH2.COMMAND_IN[2] { TURN_ON[0], TURN_OFF[1], >NO-COMMAND[2] }
SWITCH1.COMMAND_IN[9] { TURN_ON[0], TURN_OFF[1], >NO-COMMAND[2] }

Observation Variables:
OR_GATE.OUTPUT[4] { HIGH[0], LOW[1] }
3> 0 1 3 1 7 1
4> a 4 1
5> v
State Variables:
SWITCH2.MODE[0] { +BROKEN[0]:0.00497513, >+ON[1]:0.995025, OFF[2]:0 }
OR_GATE.MODE[3] { +BROKEN[0]:0.000999001, >+NOMINAL[1]:0.999001 }
SWITCH1.MODE[7] { +BROKEN[0]:0.00497513, >+ON[1]:0.995025, OFF[2]:0 }

Command Variables:
SWITCH2.COMMAND_IN[2] { TURN_ON[0], TURN_OFF[1], >NO-COMMAND[2] }
SWITCH1.COMMAND_IN[9] { TURN_ON[0], TURN_OFF[1], >NO-COMMAND[2] }

Observation Variables:
OR_GATE.OUTPUT[4] { HIGH[0], >LOW[1] }
6> e
Calculating next trajectories...


Leading estimates for the next trajectories:
{0.9662:
        (SWITCH2.MODE[0] = ON[1])
        (OR_GATE.MODE[3] = BROKEN[0])
        (SWITCH1.MODE[7] = ON[1])}
{0.02415:
        (SWITCH2.MODE[0] = BROKEN[0])
        (OR_GATE.MODE[3] = NOMINAL[1])
        (SWITCH1.MODE[7] = BROKEN[0])}
{0.004831:
        (SWITCH2.MODE[0] = BROKEN[0])
        (OR_GATE.MODE[3] = BROKEN[0])
        (SWITCH1.MODE[7] = ON[1])}
{0.004831:
        (SWITCH2.MODE[0] = ON[1])
        (OR_GATE.MODE[3] = BROKEN[0])
        (SWITCH1.MODE[7] = BROKEN[0])}
7> q
```

In the previous example, user input #3 (`0 1 3 1 7 1`) means we are setting the `SWITCH2.MODE[0]` to the `ON[1]` condition, the `OR_GATE.MODE[3]` to the `NOMINAL[1]` condition, and the `SWITCH1.MODE[7]` to the `ON[1]` condition. User input #4 is similar, but the `a` is required for non-state variables. To progress the states of the system forward, enter `e -p`. The program is defaulted to generate results for 4 trajectories. To change the number of trajectories (i.e. the number of diagnoses), enter `k #`. This is useful if the suspected diagnosis did not appear because other diagnoses had higher probabilities.

PEOPLE RESOURCES

The following people can provide further assistance in this project.

Kanna Rajan <kanna.rajan@mbari.org>: PI, gives general project direction, has worked on Deep Space 1 and Mars Exploration Rover projects in the past, has done higher level modeling for space systems, worked on Livingston2, has many NASA Ames and DS1 connections.

Hans Thomas <hthomas@mbari.org>: AUV lead, knows AUV subsystems very well, knows AUV software code very well. Hans is usually very busy and hard to get a hold of, and is more often out of his office helping people or in the DMO AUV lab. Hans also has NASA Ames experience and connections.

Duane Thompson <drthom@mbari.org>: AUV hardware knowledge source, although shares an office with Hans, is almost never there. Duane is most often found in the DMO AUV lab or on a boat at the dock.

Tom O'Reilly <oreilly@mbari.org>: Has worked on AUV software in the past, and can provide some documentation on AUV software.

Brian Williams <williams@mit.edu>: MIT professor who leads the Model-based Embedded and Robotic Systems group[6] (MERS) in the Computer Science and Artificial Intelligence Lab (CSAIL). Our group's e-mail is mers-mit@mit.edu.

Seung Chung <chung@mit.edu>: MIT Ph.D. student in MERS who has worked extensively with Titan.

Adam Sweet <sweet@email.arc.nasa.gov>: NASA Ames employee, has worked on creating system models for spacecraft, using Livingstone 2.

REFERENCES

---

[1] http://www.mbari.org/auv/. Accessed 8/22/06.

[2] Rodney A. Brooks. *How to Build Complete Creatures Rather than Isolated Cognitive Simulators*.

[3] Brian Williams et al. *Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers.*

[4] See `RMPL Documentation.pdf`

[5] http://mers.csail.mit.edu/papers/isairas01_rmpl.pdf

[6] http://mers.csail.mit.edu/mers-publications.htm