Monte-Carlo Testing for AUV Planning Software

Zeyn A. Saigol, University of Birmingham
Mentor: Kanna Rajan
Summer 2008

Keywords: AI, Artificial Intelligence, Automated Planning, T-REX, AUV

ABSTRACT

Constraint-based planning systems, especially those used for real-world applications, have a very large space of possible states. This makes them very hard to test, as only one path through the state space can be tested at once, and it is tricky to manually select a small set of paths that will thoroughly test a system. This paper describes my implementation of a Monte-Carlo testing system for the T-REX planner, which was developed at MBARI for AUV control. The test harness performs repeated runs of the planner, forcing each run through a series of randomly sampled states, in an attempt to uncover bugs in the system that manual methods may fail to detect.

INTRODUCTION

Autonomous underwater vehicles (AUVs) are increasingly useful tools for marine scientists to acquire high-quality oceanographic data. The Autonomous Systems group at MBARI aims to enhance the capabilities of AUVs by using on-board AI software, which enables the vehicle to respond to conditions and observations at sea. They have successfully developed and deployed a constraint-based planning system called T-REX (McGann et al. 2008a). T-REX is built upon EUROPA, a proven planning system developed by NASA, and extends EUROPA by allowing the interleaving of planning and execution.

However, T-REX is a complex piece of software and given a fixed mission specification, it can output widely differing paths and behaviors for the vehicle depending on the environment and sensor inputs it actually receives. On occasion, this has led to bugs in the system manifesting only during a sea deployment, which means the mission is aborted early and returns little or no scientific data (see McGann et al. (2008b) for a discussion). Extensive shore-based testing is performed prior to every mission, but it is impossible to test every condition that can be encountered at sea, and hard to even get a systematic coverage of a range of conditions.

This was the motivation for my internship project at MBARI: to develop a Monte-Carlo testing system that would automatically execute a large number of simulated T-REX missions, where each mission would test the system's response to a different set of inputs and observations. To help me get up to speed with T-REX, I was also involved in developing some simple functionality to send messages to and from the AUV while it was at sea. This was part of the "mixed initiative" project, which has the aim of combining automated and human control of the vehicle to facilitate longer missions.

This document firstly contains a brief summary of my mixed initiative work, then a detailed design description of the Monte-Carlo test harness. Finally it contains a user guide for the test harness, and a list of potential future enhancements.

MIXED INITIATIVE – COMMUNICATION AT SEA

In T-REX, the domain is specified in NDDL, which is EUROPA's modeling language. This means that the actions the agent can take (at both a high-level and low-level), the data stored by the agent, the format for data exchange with the hardware, and the structure of missions are all written in NDDL. Therefore it was important that I gain a good understanding of NDDL and T-REX in general, and to help with this I worked on several small development tasks for the mixed initiative project at the start of my internship (and in fact these continued throughout the internship).

The aim of the mixed initiative project is to get the AUV to send messages via an Iridium satellite modem to the shore whenever it surfaces for a GPS check-in. I was responsible for the NDDL side of development, which involved creating two types of message: DMO messages, which include basic status information about the AUV for the use of the Division of Marine Operations (DMO), and Science messages, which contain a summary of the observations made by the AUV.

For DMO messages, I:

- Added variables to represent the battery voltage to the NDDL model
- Created a framework to populate data in DMO messages
- Ensured that a DMO message is sent whenever the vehicle performs a GPS check-in, except in the case that a DMO message was sent within the previous 10 minutes

For Science messages, I developed code to:

- Populate and send messages containing the cluster_id at regular intervals
- Populate and send a message with the cluster_id and probability whenever a water sample is taken

I also developed code to build Science messages incrementally, so that a single message would be sent containing data from 5 different times in the recent past. However, this proved unreliable in EUROPA, as it depended on placing data in tokens that will become active in the future, and EUROPA occasionally garbage-collected these tokens before the data could be used.

MONTE-CARLO SYSTEM DESIGN

OVERVIEW

The existing T-REX testing infrastructure is shown in Figure 1 below. Normally at sea, T-REX communicates with a VCS Adapter, which sends commands to, and receives observations from, the VCS – the computer that actually runs the AUV. For shore-based testing, there are two options: firstly the real-time simulator, which is a physics-based simulator of the vehicle, runs on the same platform as the real VCS, and exposes exactly the same interface to the VCS Adapter. The disadvantage of this is that it runs in real-time, and missions generally last over 6 hours, which severely limits the amount of testing that can be done on this simulator.

The second option for testing T-REX is the pseudo-simulator, which runs much faster but provides a more crude approximation of the vehicle dynamics and environment. This is written purely in NDDL, and it runs as a separate instance of the EUROPA planner. The Monte-Carlo test harness uses the pseudo-simulator to run tests; the pseudo-simulator is completely deterministic, but it has several parameters in the NDDL model that can be changed to reflect different sea conditions, or a different set of observations the AUV may receive. An example is the parameter ERROR_RATE_X, which determines how far off course the AUV will be when it finally gets a GPS fix (this error comes from compass error, plus errors in dead-reckoning when descending and ascending). The Monte-Carlo tester is able to inject new values for certain parameters during a T-REX run, and thus simulate different environments.

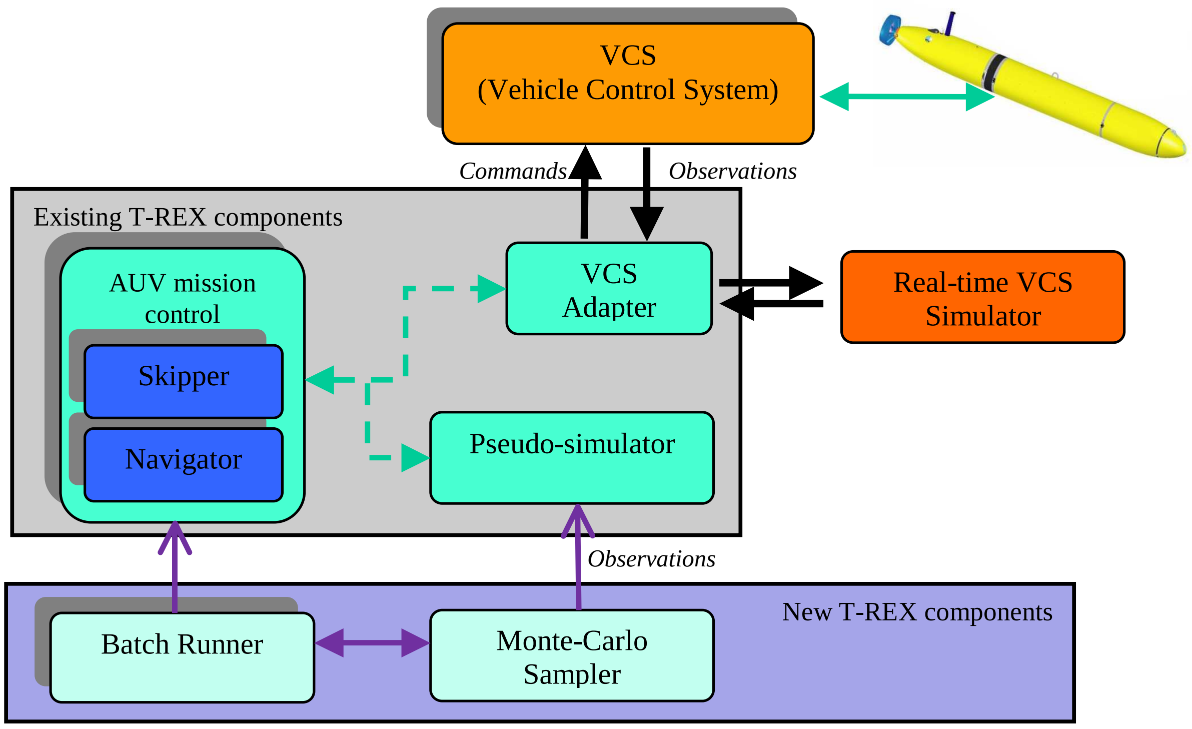

Figure 1. How the Monte-Carlo test harness integrates with the existing T-REX test infrastructure

REQUIREMENTS

The original requirements for the system were:

- Should run multiple missions simultaneously, to take advantage of the multi-core test server
- Each mission should have different values for model parameters
- The parameters should be variable during a mission, as well as between missions
- Each mission must be reproducible, i.e. must store all parameters/random seeds in a file
- Should not affect the behavior of single missions run using the `amc` executable and the pseudo-sim – these should produce exactly the same results as before
- Should determine if missions completed successfully or not by analyzing the output log file

BINDING VARIABLES

The first design decision to make was how to bind NDDL variables to a value during a run. Options were:

- Choose a fixed value for each variable for each run, and write this value into the EUROPA XML file (or into an NDDL file and then run the NDDL compiler)
- Use a custom EUROPA solver to pick values
- Write a custom constraint and apply it to the variable

We discounted the first option, as we wanted to be able to change the value of a variable during a mission. Using a Solver seems logical, but the problem is that after the Monte-Carlo system has chosen a value for a variable, if EUROPA finds that this breaks the plan it may disregard the value and backtrack, which is *not* the behavior we desire. This left us with the last option, using a custom constraint.

When applying the constraint to a variable, we had to require a default value, which will be used when not running the Monte-Carlo system to ensure that T-REX works exactly the same as before.

BATCHER AND SAMP

The system is split into two separate executables: `batcher` and `samp`. `batcher` decides on the parameters for each mission, and repeatedly runs `samp`. `samp` then starts up T-REX to run a single mission in such a way that parameters values can be injected into the pseudo-simulator. `samp` is closely based on the existing `AMC.cc` file.

Figure 1 shows a diagram of how the Monte-Carlo test harness fits in with the rest of T-REX.

LOGGING OUTPUT

By default, T-REX chooses a logging directory by finding the latest directory and incrementing a numeric identifier in the directory name; this has two problems from a batch run point of view:

1. May be a conflict if two child processes are started at the same time
2. Parent process doesn't know which logging directory corresponds to which child process

This second point was especially important as the parent process has to examine log output to determine if the mission failed or not. Therefore we decided to allow the parent process to tell the child which logging directory to use, which meant re-writing some of the T-REX logging code.

MULTIPLE PROCESSES

The `batcher` launches each mission as a child process, running `samp`. It can be configured to keep several child processes running at once, but the parent process is only single-threaded. This meant the main loop had to be designed quite carefully to check that the correct number of missions is currently running, and no extra missions are launched, despite the significant amount of time that processing a terminated mission may take. It also means the time for analyzing terminated missions should be kept to a minimum, as during this analysis time fewer than the requested number of processes may be running.

SAMPLERS AND SAMPLER GENERATORS

There are two options for varying a parameter:

1. Choose a new value for each mission, but keep the value fixed during the mission
2. Change the value at every timestep during the mission

These are shown in figure 2.

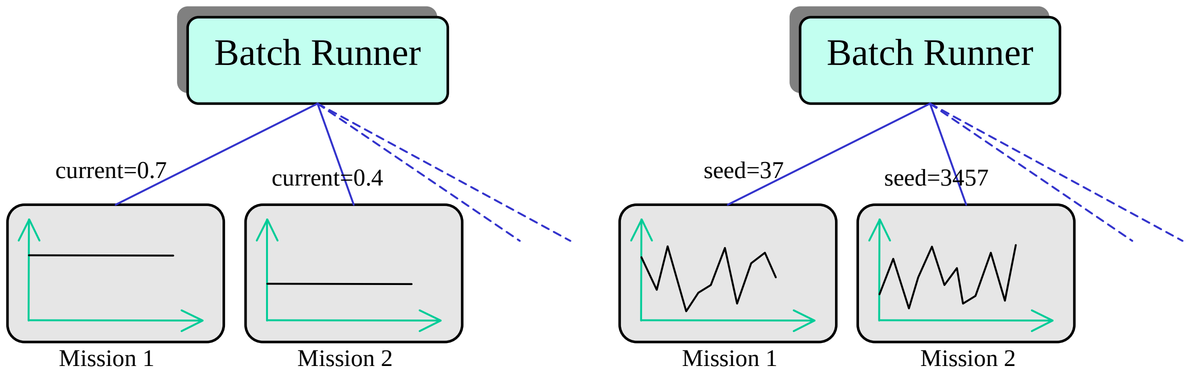

Figure 2. Inter-mission (left) versus intra-mission (right) parameter varying

Note that for option (2), changing the parameter during the mission, we still need a meta-parameter at the mission level (for example a different random seed) as otherwise every mission will be the same.

In the system, two interfaces are used to generate new values for parameters: `BatchSamplers` control how parameters are varied between missions, and `Samplers` provide parameter values within a mission. Different implementations of these interfaces allow parameters to be varied in different ways, for example different `Samplers` may return a fixed value, or sample from a Normal distribution. XML files drive configuration of the `BatchSamplers` and mission `Samplers`, and much of the EUROPA XML parsing and Factory class framework has been re-used.

DESIGN WALKTHROUGH

Figure 3 shows an overview of the main objects in the Monte-Carlo system. It is driven by a single configuration file, in which every parameter to be sampled has a separate section that sets up the `SamplerGenerator` to use for that parameter. The `BatchManager` class reads in the config file and creates a `SamplerGenerator` for each parameter, using EUROPA's `AbstractFactory.allocate` method, and stores them in a list. Note that the `SamplerGenerators` often rely on a nested `Sampler` to generate random numbers.

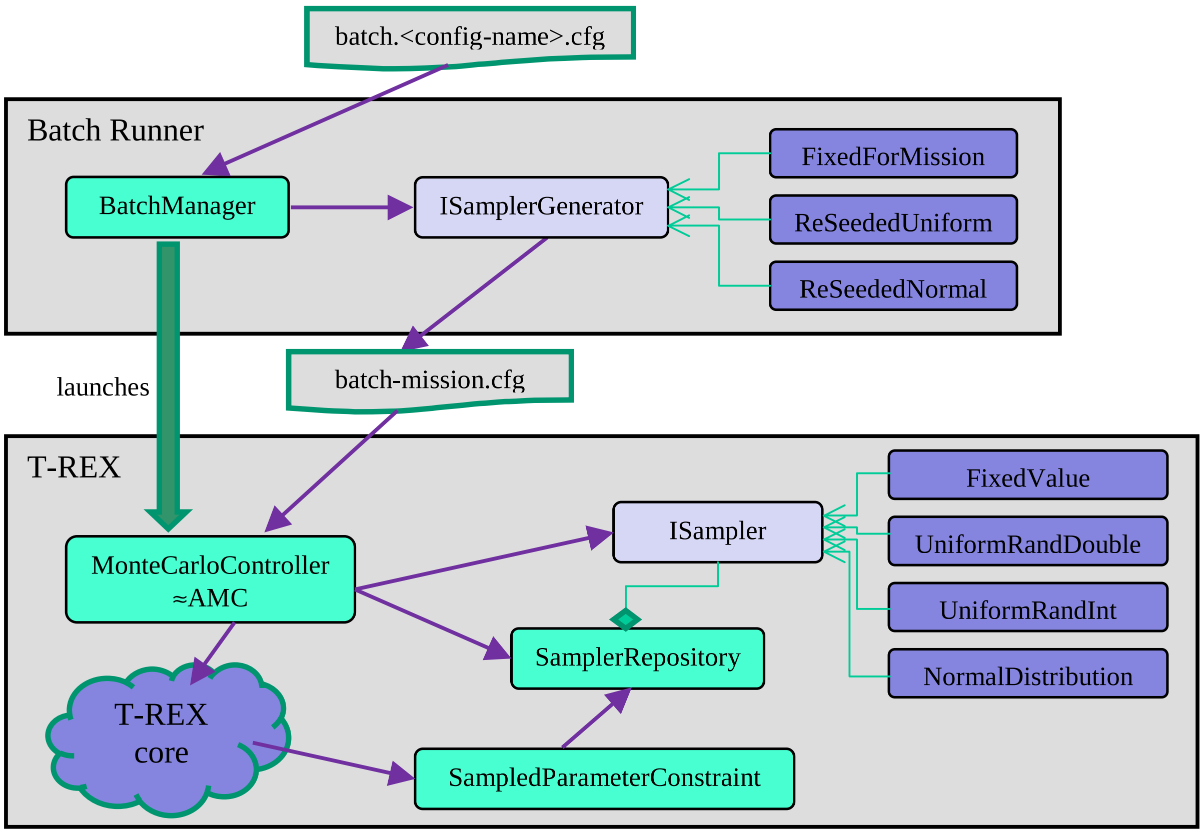

Figure 3: Key objects in the system design. Class names are not exact, and conceptual interfaces have an "I" prefix in this diagram.

The main loop in `BatchManager` runs missions by firstly creating a logging directory, and then writing a unique configuration file for that mission to the directory. It uses the `SamplerGenerators` to create the `Sampler` element for each parameter and writes that directly to the `batch-mission.cfg` file. Finally, it forks and executes `samp`.

`MonteCarloController` is the main class for the `samp` executable. It is quite straightforward – it simply reads in the `batch-mission.cfg` file, uses the `AbstractFactory` to create `Samplers` for each parameter, registers these with a `SamplerRepository` singleton, and finally runs T-REX as normal.

When T-REX runs, `SampledParameterConstraint` uses the name of the parameter to look up a `Sampler` from the singleton `SamplerRepository` class. If no `Sampler` is found, it simply sets the variable to the default value. If a `Sampler` is found, it calls the `getContinuousValue` method to find the value to set the parameter to.

ADDING NEW SAMPLERS AND SAMPLERGENERATORS

To create a new Sampler, the following steps should be followed:

1. Write the Sampler class. This should inherit from `TREX::Sampler` (or one of it's subclasses, such as `SamplerPeriodicUpdate` for `Samplers` that can be configured to only produce a new value every X ticks)
2. Call `REGISTER_COMPONENT_FACTORY(TREX::NewSampler, NewSamplerName);` when the application starts up (e.g. from `initCTDApplication`)
3. Update the `batch-sample.cfg` file with an example of how to configure the new `Sampler`

To create a new SamplerGenerator, exactly the same process should be followed with the sole difference that the new class should inherit from `TREX::SamplerGenerator`.

MONTE-CARLO SYSTEM USER GUIDE

RUNNING BATCHER

All `batcher` settings are controlled by a single config file, which should be fairly self-explanatory. `batcher` operates in two modes: it can either run a fixed number of missions, or run missions until a specified time (the time indicates a cut-off for starting missions rather than for missions to have all ended by). The outline of the config file is:

```
<Batch>
  <!-- Config for missions: copied directly to mission config files  -->
  <AgentConfig>science.4.cfg</AgentConfig>
  <!-- Steps Per Tick is optional -->
  <StepsPerTick>50</StepsPerTick>
```

```
    <!-- BatchRunner control settings -->
    <MaxConcurrentMissions>7</MaxConcurrentMissions>
    <!-- Format for times is HH:MM:SS (24-hour local time)
       EITHER LastNewMission OR TotalNumMissions must be specified  -->
    <!--  <LastNewMission>12:30:00</LastNewMission> -->
    <TotalNumMissions>14</TotalNumMissions>
    <EndTickTolerance>5</EndTickTolerance>

    <!-- Sampled variables -->
    <SampledParameter name="gpsHitRate" component="…">
      <BatchSampler component="…">
        …
      </BatchSampler>
    </SampledParameter>
  </Batch>
```

Batcher uses the last entry in TREX.log to determine if a mission has failed or not: if the last entry is within `EndTickTolerance` of the mission length (as specified in the file given by the `AgentConfig` argument), the mission is assumed to have been successful.

Batcher is run by providing just the config file name as an argument (note however that it must be run from the $TREX_HOME/ctd2007 directory):

```
      batcher_o_rt <path-to-config-file>
```

To launch an overnight run, the command should resemble:

```
      nohup nice ./batcher_o_rt batch.science.4.cfg &
```

The logging output of the batch runner is kept separate from normal T-REX runs, in the structure:

```
$TREX_LOG_DIR
      /batch
            /<year>.<day>.<batch-id>
                  /<mission-number>
```

For example, the first mission of a batch run might have the logging directory `~/autonomy/TREX/log/batch/2008.233.01/0001`. The batch runner records the success or failure of each mission in the debug log in the top-level directory, for example `~/autonomy/TREX/log/batch/2008.233.01/Debug.log`.

SAMPLED PARAMETERS

The following sampled parameters are available:

Name	**Description and default**
`gpsHitRate`	Number of GPS hits per second. Determines the length of GPS.Active (along with MIN_GPS_HITS), smaller number mean GPS.Active will take longer. Default value is the constant GPS_HIT_RATE.
`errorRateX / errorRateY`	Represents the navigation error that will be applied once a GPS fix is obtained, i.e. the agent's location will be instantaneously corrected by this amount following a GPS.Active. Units are meters/second. Defaults to the constants ERROR_RATE_X / ERROR_RATE_Y.

`cluster_id`	Sets the cluster_id reported as part of VehicleState.Holds, which represents the INL probability based on sensor readings. The default for this is a local variable, localClustId, whose value depends on the constant CLUSTER_STABLE: If CLUSTER_STABLE is set to true, then cluster_id will default to having the same value as on the previous timestep. If CLUSTER_STABLE is set to false, cluster_id will be cycled from 1 to the maximum defined id, changing every 10 ticks. If CLUSTER_STABLE is undefined, then cluster_id will also be undefined.

`dxNoise /`
`dyNoise`
	Represent a current that causes the vehicle's path to be offset from the expected path. Defaults to 0.

SAMPLERS

The following samplers are available:

Name	**Arguments/Example**
`FixedValue` – sets the parameter to a fixed value throughout the mission	`Value` – value of parameter

	`<Sampler component="FixedValue">`
	`  <Value>-0.235081</Value>`
	`</Sampler>`

`UniformRandInt` – chooses random integers from a uniform distribution	`Seed` – seed for the random number generator `Min` – min value `Max` – max value `UpdateFrequency` – (optional) how many ticks should elapse before a new value is generated. If not specified, returns a new value every time it's called.

	`<Sampler component="UniformRandInt">`
	`  <Seed>1047926106</Seed>`
	`  <Min>1</Min>`
	`  <Max>39</Max>`
	`  <UpdateFrequency>13</UpdateFrequency>`
	`</Sampler>`

`UniformRandDouble` – chooses random reals from a uniform distribution	`Seed` – seed for the random number generator `Min` – min value `Max` – max value `UpdateFrequency` – (optional) how many ticks should elapse before a new value is generated. If not specified, returns a new value every time it's called.

	`<Sampler component="UniformRandDouble">`
	`  <Seed>37</Seed>`
	`  <Min>-0.9</Min>`
	`  <Max>0.9</Max>`
	`</Sampler>`

`NormalDistribution` – chooses random real numbers from a Gaussian distribution	`Seed` – seed for the random number generator `Mean` – mean of the distribution `Sigma` – standard deviation of the distribution `UpdateFrequency` – (optional) how many ticks should elapse before a new value is generated. If not specified, returns a new value every time it's called.

	`<Sampler component="NormalDistribution">`
	`  <Seed>1167779653</Seed>`
	`  <Mean>0.550809</Mean>`
	`  <Sigma>0.3</Sigma>`
	`  <UpdateFrequency>10</UpdateFrequency>`
	`</Sampler>`

SAMPLER GENERATORS

The following sampler generators are available:

Name	**Arguments/Example**
`FixedDuringMission` – uses a Sampler to generate a value for the parameter for each mission	`BatchSampler` – Sampler to generate values, can be `UniformRandDouble`, `UniformRandInt` or `NormalDistribution`.

	`<SampledParameter name="gpsHitRate"`
	`component="FixedDuringMission">`
	`  <BatchSampler component="UniformRandDouble">`
	`    <Seed>9381</Seed>`
	`    <Min>0.1</Min>`
	`    <Max>2</Max>`
	`  </BatchSampler>`
	`</SampledParameter>`

`ReSeededUniformRandom` – generates a new seed for each mission.	`BatchSampler` – Sampler to generate a new seed for each mission, MUST be `UniformRandInt`. `Sampler` – Sampler used during the mission, can be `UniformRandDouble`, `UniformRandInt` or `NormalDistribution`. Has its `Seed` argument populated by the `BatchSampler`.

	`<SampledParameter name="cluster_id"`
	`component="ReSeededUniformRandom">`
	`  <BatchSampler component="UniformRandInt">`
	`    <Seed>9281</Seed>`
	`    <Min>0</Min>`
	`    <Max>2147483647</Max>`
	`  </BatchSampler>`
	`  <Sampler component="UniformRandInt">`
	`    <Min>1</Min>`
	`    <Max>39</Max>`
	`    <UpdateFrequency>40</UpdateFrequency>`
	`  </Sampler>`
	`</SampledParameter>`

`ReSeededNormalDistn` – generates a new seed and a new mean value for each mission.	`BatchSampler` – Sampler to generate a new seed for each mission, MUST be `UniformRandInt`. MeanSampler -- Sampler to generate a new mean for each mission, can be `UniformRandDouble` or `UniformRandInt`. `Sampler` – Sampler used during the mission, should be `NormalDistribution`. Has its `Seed` argument populated by the `BatchSampler`, and `Mean` argument populated by the `MeanSampler`.

```
<SampledParameter name="errorRateY"
component="ReSeededNormalDistn">
  <BatchSampler component="UniformRandInt">
    <Seed>7234</Seed>
    <Min>0</Min>
    <Max>2147483647</Max>
  </BatchSampler>
  <MeanSampler component="UniformRandDouble">
    <Seed>57251</Seed>
    <Min>-0.5</Min>
    <Max>0.5</Max>
  </MeanSampler>
  <Sampler component="NormalDistribution">
    <!-- <Seed>352</Seed>   Set by BatchSampler -->
    <!-- <Mean>-0.5</Mean>   Set by MeanSampler -->
    <Sigma>0.3</Sigma>
    <UpdateFrequency>10</UpdateFrequency>
  </Sampler>
</SampledParameter>
```

RUNNING SAMP DIRECTLY

The `samp` executable can be run directly if needed, to re-run a failed mission from a batch run, or to try out a single mission using sampled parameters. To do this:

1. Create a logging directory anywhere on your file system
2. Copy the `batch-mission.cfg` config file into the logging directory (note the file *must* have this name)
3. Run `samp_o_rt <path-to-log-dir>`

ADDING A NEW SAMPLED PARAMETER TO THE MODEL

The pseudo-simulator (essentially `simulator.nddl`) contains several other variables that it would be nice to set using the Monte-Carlo system. To enable a variable to have values injected by a `Sampler`, you have to apply the `sampledParameter` constraint to it, specifying a default value:

```
      sampledParameter(gpsHitRate, GPS_HIT_RATE);
```

The default value is used whenever the variable is not being sampled – either if running the pseudo-sim via `amc`, or if the variable is not configured as a `SampledParameter` in the batch configuration file. The default can be a constant global variable (as above), a literal value (e.g. 0.0), or a local variable. The variable to be sampled can be either a local variable, or a variable of a predicate.

No changes to C++ code are needed to enable sampling for a new parameter – but usually a new variable will be needed in the NDDL code, as the existing variable will probably become the default value for the sampled variable (or a new variable will be needed to act as the default value).

CONCLUSIONS AND FUTURE WORK

I have implemented a practical and useful system for more fully testing T-REX prior to sea missions. The Monte-Carlo system has been integrated into the nightly build process, and will hopefully make it easier to find any bugs in the future.

SHORT-TERM ENHANCEMENTS

- Enhance mission failure analysis by finding the number of occurrences of specific keywords in the T-REX log, such as “Recall”, “Reject”, “No plan found”, “Relax failed”. A maximum count could be specified for each of these messages before a mission is considered as failed
- Integrate a graphical method of viewing how parameters have been varied over a mission, for example in MATLAB
- Would be nice if `batcher` could respond better to being killed, by explicitly killing all child processes
- Instead of just relying on the base name of an NDDL variable, the fully qualified name (i.e. `class-name.predicate.var-name`) should be used to find the correct `Sampler` to use
- A `Sampler` to simulate a random walk

FUTURE DIRECTIONS

- Create a GUI to display the history of parameter values while a mission is being run, and to allow a tester to alter the evolution of that parameter
- Alter parameters based on previous failures
- Group together failures that may have a common cause (i.e. be due to the same bug)
- Trace failures back to a particular part of the model

ACKNOWLEDGEMENTS

I would like to acknowledge:
Frederic Py and Conor McGann
My mentor, Kanna Rajan
George Matsumoto and Linda Kuhnz
All the 2008 interns
Tom O’Reilly, Hans Thomas, Doug Conlin and Duane Thompson
The crew of the R/V Zephyr
MBARI and the Packard Foundation
My supervisors at Birmingham, Jeremy Wyatt and Richard Dearden

References:

McGann, C., F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen (2008a).
A deliberative architecture for AUV control. *IEEE International Conference on Robotics and Automation, 2008 (ICRA '08)* :1049-1054

McGann, C., F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen (2008b).
Preliminary Results for Model-Based Adaptive Control of an Autonomous Underwater Vehicle. *11th International Symposium on Experimental Robotics 2008 (ISER '08)*