Intermediate ACS Setup

Table of Contents

Optimizing Task Performance via "Tuning" Parameters

Frequently, you will find that a task runs reasonably well by simply setting up an agent using all of the default settings. However, there will likely be times where the defaults simply do not provide "optimal" performance and you may want to "tune" the agent's settings to get it to perform a task more effectively.

The Clarion theory specifies several parameters for various mechanisms (see technical specification document here for more details). These parameters have been implemented into the Clarion Library in two ways: as global (`static`) parameters, and as local (instance) parameters. The parameters have been stored within "`Parameters`" classes, which are located throughout the system based upon their most logical position (as specified by the Clarion theory). For example, the `RefineableActionRule` class implements a rule type that is refineable (and is usually extracted via RER). As part of being "refineable", these rules contain methods for generalization and specialization, each of which have some threshold parameters that can be "tuned" in order to optimize the frequency in which either process (i.e., specialization or generalization) occurs. The following code (from the "*Full Hello World*"[1] simulation sample) demonstrates how two of these thresholds might be changed during the initialization of a task:

```
RefineableActionRule.GlobalParameters.SPECIALIZATION_THRESHOLD_1 = -.6;
RefineableActionRule.GlobalParameters.GENERALIZATION_THRESHOLD_1 = -.2;
```

In addition to providing all of the default parameters specified by the Clarion theory, the Clarion Library also provides several extra parameters designed to aid you in running a task. For example, suppose we wanted to turn off the various forms of learning that take place in the ACS (i.e., rule refinement, bottom-up learning or rule

[1] Located in the "*Intermediate*" section of the "*Samples*" folder (filename: "*HelloWorld - Full.cs*").

extraction, top-down learning, and/or learning in the bottom level). The following lines of code could be used to accomplish this:

```
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;
ActionCenteredSubsystem.GlobalParameters.PERFORM_RULE_EXTRACTION = false;
ActionCenteredSubsystem.GlobalParameters.PERFORM_TOP_DOWN_LEARNING = false;
ActionCenteredSubsystem.GlobalParameters.PERFORM_BL_LEARNING = false;
```

All of these are examples of "global" parameter changes, so let's begin our discussion on how to make parameter changes by first considering the global (`static`) method for making parameter changes.

Making Global Parameter Changes

The first thing you need to know with regards to global parameter changes is how these parameters are accessed. As mentioned earlier, global parameters are stored statically. This means that they are accessible from a static call (i.e., using the class name) to the class with whom the parameters are associated. For all of the "built-in" classes defined by the Clarion Library, the global parameters can be found within the parameters class that is returned by the `GlobalParameters` property. In our earlier examples, the global parameters associated with specialization and generalization of action rules were accessed via:

```
RefineableActionRule.GlobalParameters
```

And the parameters for turning on and off learning in the ACS were accessed by:

```
ActionCenteredSubsystem.GlobalParameters
```

However, before you begin tuning all of your parameters using the global (`static`) method, it is essential to understand a few important points about how global parameters are implemented and what the consequences are with regards to how and when global parameter changes can be made.

First, global parameters changes are only applicable to an instance of a class **BEFORE** it is initialized. The (`static`) global parameters for any given class are only used during the initialization process in order to set the values of the local parameters of an instance. Once the instance has been initialized, it will thereafter only use the local parameters. In other words, making a global parameter change **AFTER** an instance of a class has been initialized will have **NO** effect on the corresponding local parameter for that instance. For example, let's look at the following lines of code:

```
Agent John = World.NewAgent("John");

ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;
```

Note that the ACS is initialized as part of the initialization of an agent, so `John`'s ACS will already be instantiated by the time the global parameter change is made (in the second line). As a result, if it was our intention to turn off refinement in `John`'s ACS,

our code (from above) would fail. Instead, the local PERFORM_RER_REFINEMENT parameter would still be set to true and John's ACS would still perform refinement.[2] The correct way of changing the parameter globally (so as to achieve our intended behavior) would be as follows:

```
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;

Agent John = World.NewAgent("John");
```

Moving on, the second thing you need to know is that, for all of the "built-in" classes of the Clarion Library, the global parameters have been set up such that they can be changed at any point within the inheritance hierarchy. For example, suppose you wanted to change the POSITIVE_MATCH_THRESHOLD parameter for **ALL** rules (regardless of their type). The following line of code would accomplish this:

```
Rule.GlobalParameters.POSITIVE_MATCH_THRESHOLD = .75;
```

This command will change the parameter for any class that derives from Rule (e.g., RefineableActionRule, IRLRule, AssociativeRule, etc.). However, suppose you just wanted to change the POSITIVE_MATCH_THRESHOLD parameter for IRL rules only. This would be accomplished in essential the same way as before, except you would call it on the IRLRule class instead:

```
IRLRule.GlobalParameters.POSITIVE_MATCH_THRESHOLD = .75;
```

Making parameter changes at different points in the inheritance hierarchy provides a convenient way for making wholesale (and possibly even targeted) parameter tweaks. However, you may find that you want to change the parameters for specific instances of a class, or that you want different instances of a class to have slightly different settings (e.g., depending on where it is located within an agent, or based upon the "group" of the agent in which it was initialized). For this reason, the Clarion Library also provides a method for changing parameters locally.

Making Local Parameter Changes

The other method for performing parameter changes within the Clarion Library is to make such changes on a per-instance basis. This is what we refer to as "local" parameter changing. Local parameter changes are made on instances of a class through its Parameters property. For example, if we wanted to change the PERFORM_RER_REFINEMENT parameter for just John's ACS (from the previous example), then we could accomplish this by doing the following:

```
John.ACS.Parameters.PERFORM_RER_REFINEMENT = false;
```

All of the "built-in" classes in the Clarion Library (which contain parameters) have a local Parameters property. In addition, unlike the global parameters, local

[2] Although the local parameter will be changed for any agents that are initialized **AFTER** the global parameter change is made

parameters can be tuned at any point (after the local instance has been initialized, of course). Note that the local parameters are not subject to the "immutable" (that is, read-only) restriction placed on an agent's internal (functional) objects, so they can be altered even after these objects have been "committed" to the agent. For example, suppose you wanted to change the `LEARNING_RATE` parameter for an instance of a `SimplifiedQBPNetwork` (called `net`) that was initialized within the bottom level of `John`'s ACS. The following code could be called at any point during the task:

```
//Retrieves the network from the bottom level of John's ACS
SimplifiedQBPNetwork net = (SimplifiedQBPNetwork)John.
   GetInternals(Agent.Internals.IMPLICIT_DECISION_NETWORKS).First();

net.Parameters.LEARNING_RATE = .5;
```

This parameter change will take effect the next time the learning rate is applied (which is presumably the next time John receives feedback). Note also that the first line of code (above) will retrieve the network from the bottom level of John's ACS, but only if it is either by itself in the bottom level of John's ACS or it is the first network in the collection that is returned by the `GetInternals` method. We won't get into the specifics of the `GetInternals` method at this point. Instead, you can consult the "*Useful Features*" tutorial[3] for more information about how to use this method.

Finally, we should also mention here that there is an additional way to change parameters in a more "automatic" fashion (i.e., by having either the ACS or a module within the MCS initiate the parameter change). However, using this method is beyond the scope of this tutorial, as it makes use of a concept called "Generative Actions" (which we will discuss in a later tutorial[4]).

At this point, we have covered the two primary techniques for changing parameters. These techniques should suffice any time it is necessary to tune a simulation.

Setting Up the Working Memory

In this section we will discuss how to set up and use the working memory. Broadly speaking, the working memory can be thought of as being a "container" within an agent that holds knowledge about the world (i.e., declarative chunks, previous action chunks, etc.). Technically speaking, it is located within the ACS. However, all interaction with the working memory (from the simulating environment) is performed directly via the `Agent` class. For instance, we can view the contents of working memory by calling the `GetInternals` method. The code below demonstrates how we might accomplish this for our agent, `John`:

```
IEnumerable<Chunk> wmContents =
    (IEnumerable<Chunk>)John.GetInternals
```

[3] Located in the "*Features & Plugins*" section of the "*Tutorials*" folder.
[4] See the "*Advanced ACS Setup*" tutorial in the "*Advanced Tutorials*" section of the "*Tutorials*" folder.

```
    (Agent.InternalWorldObjectContainers.WORKING_MEMORY);
```

Note that the working memory can hold any type of chunk. Additionally, whenever a chunk is "set" in the working memory, it becomes a part of the "internal sensory information" and will automatically be "activated" in the `SensoryInformation` the next time one is perceived.

World objects (i.e., chunks) are added to working memory either manually or by using a "working memory update action chunk." First, let's look at how chunks can be set manually.

Manually Setting a Chunk in Working Memory

The simplest way to "set" (or add) a chunk in working memory is to do it manually. We do this by calling the `SetWMChunk` method for the agent where the chunk is being set in working memory. The code below demonstrates how we can do this for our agent, John:

```
John.SetWMChunk(ch, 1);
```

To set a chunk in working memory we must specify two things when calling the above method: the chunk to be set and the "activation level" for that chunk. This will "set" (or add) the chunk in working memory. To "deactivate" (or remove) the chunk from working memory we call the `ResetWMChunk` method. The following code demonstrates how we can manually reset (i.e., deactivate or remove) the chunk in working memory:

```
John.ResetWMChunk(ch);
```

These two simple methods provide you with all of the power you need to be able to use chunks within working memory. However, manually setting working memory may not be enough for your simulating environment. Recall that the Clarion theory provides many more details regarding various additional methods for setting chunks in working memory. For example, we can use "working memory actions" in the ACS or in the MCS to perform operations on the working memory itself. In the following section, we will look at how chunks can be set using "working memory actions" in the ACS.

Using Action Chunks

To begin, while the Clarion theory refers to actions that affect the working memory as being "working memory actions", the implementation uses a clearer term for describing these sorts of actions. In the Clarion Library, actions that perform updates on the working memory are defined using the `WorkingMemoryUpdateActionChunk` class. The contents of these action chunks contain information about the sorts of updates that are to be performed. For example, suppose we want an action that "sets" the chunk `ch` in working memory. The following code sets up such an action:

```
WorkingMemoryUpdateActionChunk wmAct = World.NewWorkingMemoryUpdateActionChunk();

wmAct.Add(WorkingMemory.RecognizedActions.SET, ch);
```

Note that we specify, as the first parameter in our `Add` method, an enumerator called `RecognizedActions`. Several classes within the Clarion Library (namely those mechanism that can be manipulated using actions, e.g., the `NonActionCeneteredSubsystem`, the `GoalStructure`, `WorkingMemory`, etc.) define a `RecognizedActions` enumerator. This enumerator provides the list of commands that an action can perform on an instance of that class. The `WorkingMemory` recognizes four types of actions:

- `SET`. "Adds" the chunk to working memory
- `RESET`. "Removes" the chunk from working memory
- `RESET_ALL`. "Removes" **ALL** of the chunks from working memory
- `SET_RESET`. Combines the `RESET_ALL` and `SET` actions

If we want a component in the ACS to use this action, all we have to do is specify it in the output layer of the component. Below is an example of how we would set up this action in a network on the bottom level of the ACS.

```
... //Elided code performing additional initialization for the network

net.Output.Add(wmAct);
```

Now, whenever the ACS selects this `WorkingMemoryUpdateActionChunk`, the system will perform the commands specified by that action.

At this point, you now know how to make use of the working memory and how to update it via two different methods. This concludes the tutorial for the intermediate aspects of the ACS. In the final guides on the ACS, we will cover:

- "***Basic Customization***" – How to do some basic customizations in the Clarion Library. As this relates to the ACS, this guide covers how to use delegates to setup IRL and Fixed rules in the top level.[5]
- "***Advanced ACS Setup***" – Covers how to interface the ACS with the NACS.[6] Note that you should familiarize yourself with setting up the NACS first before looking at this guide.[7]

Remember, as always, feel free to post on our message boards (http://www.clarioncognitivearchitecture.com) or email us at clarion.support@gmail.com and we will do our best to respond back to you as quickly as possible.

---

[5] This tutorial can be found in the "*Customizations*" section of the "*Tutorials*" folder.
[6] This guide can be found in the "*Advanced Tutorials*" section of the "*Tutorials*" folder.
[7] The details for setting up the NACS can be found in the "*Setting up and Using the NACS*" tutorial, which is located alongside the "*Advanced ACS Setup*" guide