Advanced ACS Setup

Table of Contents

A Brief Note

The primary focus of this tutorial is to demonstrate how to interact with the NACS using the ACS (and by creative extension the MCS). However, there is another, simpler, method for using the NACS (i.e., as a stand-alone mechanism). That method is described in the "*Setting Up and Using the NACS*" tutorial (found in the *Advanced* folder of the *Tutorials* section).

Keep in mind that you should read that tutorial first before attempting to integrate the ACS and NACS. At the very least, it will show you how to initialize the top and bottom levels of the NACS. Furthermore, you may find that the stand-alone method is very useful for testing whether the NACS is operating correctly before moving onto the somewhat more complicated matter of integrating the NACS with the other subsystems.

Interacting with the NACS

As part of performing more advanced tasks, you may wish you agent to reason over its declarative knowledge about the world before (and in order to aid in) action decision making. To this end, the ACS (or MCS) and NACS have been designed to interact with one another. In the following sections we will look at an example of how the NACS and ACS can be setup to work in tandem on a simple reasoning-type task. In particular, we will be looking at the sample "*Reasoner – Full.cs*", which can be found in the *Advanced* section of the *samples* folder. This example extends "*Reasoner – Simple.cs*" to integrate ACS and NACS functioning. If you have not

already done so, you will want to refer to the *Advanced* tutorial "*Setting Up & Using the NACS"* for an overview of this simple reasoning task as well as instructions on how to initiate the NACS.

The full reasoner task is similar to the simple reasoner, except that the specifics of the task itself are actually a bit simpler. In particular, the ACS is set up using only the top level whereas the NACS is set up with only the bottom level. The task itself entails the perceiving of one of 5 "noisy" patterns. This pattern triggers a call by the ACS to the NACS. The NACS then uses an auto-associative network on the bottom level to complete the pattern and returns a fully activated chunk back to the ACS (which represents the full pattern). That chunk is then stored in working memory. Finally, the agent is asked whether the initial input was the partial pattern for a particular target (i.e., the 2nd pattern). The expected output (from the agent) for this simulation is: *No*, *Yes*, *No*, *No*, *No*.

Making Reasoning Requests

Initiating the NACS reasoning using the ACS is as simple as setting a goal in the goal structure or a chunk in working memory. In other words, to have the ACS initiate reasoning, all you need to do is use a `ReasoningRequestActionChunk`. The following code demonstrates how to initialize this action chunk:

```
ReasoningRequestActionChunk think =
       World.NewReasoningRequestActionChunk ("DoReasoning");
think.Add (NonActionCenteredSubsystem.RecognizedReasoningActions.NEW, 1);
```

Note that the second line is similar to how we specify set/reset commands when using `GoalStructureUpdateActionChunk` or `WorkingMemoryUpdateActionChunk`.[1] Specifically, we use the `RecognizedReasoningActions` to specify the type of reasoning request that is being made by the ACS. In our current example, we are specifying that the "think" action will prompt a new round of reasoning and perform 1 iteration. In total, the `ReasoningRequestActionChunk` recognizes the following actions:

- `NEW` – Specifies that a new round of reasoning is being requested
- `CONTINUE` – Similar to `NEW`, but specifies that reasoning should be continued from a previous round
- `PEEK` – Specifies that the chunks that have thus far been concluded should be returned, but that reasoning should continue
- `INTERRUPT` – Similar to `PEEK`, but specifies that reasoning should be halted immediately

Note that if you do not specify the number of iterations, the NACS will reason continuously. The action commands can also be chained. For example, suppose you

[1] See the *Setting Up & Using the Goal Structure* or *Intermediate ACS Setup* tutorials for details on how to implement these action chunk types

want to reason for 20 iterations, but look at the conclusions halfway through. The following shows how this could be accomplished:

```
think.Add (NonActionCenteredSubsystem.RecognizedReasoningActions.NEW, 10);
think.Add (NonActionCenteredSubsystem.RecognizedReasoningActions.CONTINUE, 10);
```

On top of this, by leveraging the "parameter change" capabilities inherent to all action chunks, you can also specify various settings that you wish to use for the round of reasoning that is being requested. For example, suppose you only wanted the top level of the NACS to be used, the following demonstrates how this could be set up:

```
think.Add (John.NACS, "USE_BOTTOM_LEVEL", false);
think.Add (John.NACS, "USE_TOP_LEVEL", true);
```

Of course keep in mind that this will only work if the "think" action is initialized specifically for and is used solely by John. Also note that, as is the case with other internally-oriented actions, if a `ReasoningRequestActionChunk` is chosen by the ACS, the agent will deliver the "`DO_NOTHING`" action to the simulating environment as its chosen action for the particular sensory information that caused the `ReasoningRequestActionChunk` to be selected.

Although setting up a `ReasoningRequestActionChunk` is fairly straightforward, handling the coordination of the subsystems requires a little more thought. For example, the determination of which level(s) should recommend the reasoning request action and what the ACS should do while waiting for the conclusions from the NACS (or if it should wait) must be carefully devised and implement. Of course the particulars of how you implement this is based on the requirements of your task. However, the primary reason why this consideration is so important is because the NACS is designed to run in parallel with the ACS. This is done so that an agent can continue to interact with the world while reasoning is being performed. However, in many tasks (such as this), you may want the ACS to wait until the NACS has completed reasoning. For the full reasoner sample, a fairly simple process is used, with rules on the top level of the ACS and "states" utilized to coordinate the two subsystems. In other words, we will use the state concept to "modulate" when certain actions are recommended by the ACS. The following code demonstrates how this can be set up:

```
World.NewDimensionValuePair ("state", 1);
World.NewDimensionValuePair ("state", 2);
World.NewDimensionValuePair ("state", 3);

RefineableActionRule doReasoning =
       AgentInitializer.InitializeActionRule (reasoner,
       RefineableActionRule.Factory, think);

doReasoning.GeneralizedCondition.Add (World.GetDimensionValuePair ("state", 1));
reasoner.Commit (doReasoning);

RefineableActionRule doNothing =
```

```
        AgentInitializer.InitializeActionRule (reasoner,
        RefineableActionRule.Factory, ExternalActionChunk.DO_NOTHING);

doNothing.GeneralizedCondition.Add (World.GetDimensionValuePair ("state", 2));
reasoner.Commit (doNothing);
```

The idea here is that only certain rules will be eligible in certain states. For example, the first rule from above will only be eligible to fire when the `("state", 1)` dimension-value pair is activated and second rule is only eligible when the `("state", 1)` dimension-value pair is activated. Within the simulating world, the states transition as follows:

1. This state occurs in conjunction with the first presentation of the noisy pattern
2. This state occurs after the first time the agent issues the "DO_NOTHING" action (which is also correlated with the ACS making the reasoning request)
3. This state occurs when the working memory is updated inside of the agent

The code below demonstrates how this "state transition" process might be accomplished:

```
int state_counter = 1;
while (chosen == null || chosen == ExternalActionChunk.DO_NOTHING)
{
        SensoryInformation si = World.NewSensoryInformation (reasoner);
        si.Add (World.GetDimensionValuePair ("state", state_counter), 1);

        int count = 0;
        foreach (DimensionValuePair dv in dvs)
        {
                if (((double)count / (double)dc.Count < (1 - noise)))
                {
                        if (dc.Contains (dv))
                        {
                                si.Add (dv, 1);
                                ++count;
                        } else
                                si.Add (dv, 0);
                } else
                        si.Add (dv, 0);
        }

        reasoner.Perceive (si);
        chosen = reasoner.GetChosenExternalAction (si);

        if(reasoner.GetInternals(
                Agent.InternalWorldObjectContainers.WORKING_MEMORY).Count() > 0)
                state_counter = 3;
        else
                state_counter = 2;
}
```

The `while` loop will continue the perception → action process for the current input until the agent responds either "yes" or "no" to whether that noisy pattern was the 2nd one. The rules and actions that enable this response can be set up as follows:

```
ExternalActionChunk yes_act = World.NewExternalActionChunk ("Yes");
ExternalActionChunk no_act = World.NewExternalActionChunk ("No");


RefineableActionRule yes =
       AgentInitializer.InitializeActionRule (reasoner,
       RefineableActionRule.Factory, yes_act);

yes.GeneralizedCondition.Add (World.GetDeclarativeChunk (1), true);
reasoner.Commit (yes);

RefineableActionRule no =
       AgentInitializer.InitializeActionRule (reasoner,
       RefineableActionRule.Factory, World.GetActionChunk ("No"));

no.GeneralizedCondition.Add (World.GetDeclarativeChunk (0), true, "altdim");
no.GeneralizedCondition.Add (World.GetDeclarativeChunk (2), true, "altdim");
no.GeneralizedCondition.Add (World.GetDeclarativeChunk (3), true, "altdim");
no.GeneralizedCondition.Add (World.GetDeclarativeChunk (4), true, "altdim");
reasoner.Commit (no);
```

Specifying Alternative Dimensions

You might have noticed in the above code that we have specified something called `"altdim"` when adding conditional inputs to our "no" rule. This is the case because chunks themselves, by default, are considered self-contained and therefore are technically represented at the dimension-value pair level within their own dimension. That is, the library will treat the relation between chunks, when added to the condition of a rule, using the AND operation. However, at times it may be preferable to have the chunks organized together (i.e., within the same dimension) so that they can be compared using an OR operation. To enable this possibility, the Add method of a rule's condition has an option "alternative dimension" parameter. Any world object (be it a chunk, dimension-value pair, etc.) that is added to the condition with an alternative dimension, will be "placed" in that dimension (for that rule). This does not change the objects dimension at the "descriptive" (i.e., world) level. However, it does allow for objects that are naturally correlated (by default) to be compared as if they were.

Filtering Input/Conclusions

The interaction between the ACS and NACS occurs seamlessly. That is, when reasoning request actions are chosen in the ACS, the NACS will automatically begin the reasoning process. Subsequently, when the NACS is finished reasoning, it's conclusions will automatically be sent back to the ACS and the ACS will populate the working memory all on its own. However, this is done based on certain default assumptions. For instance, the ACS will send the entire sensory information object to the NACS to use as its input into reasoning whenever a reasoning request action

is chosen. Additionally, the ACS will always load all of the chunks that are concluded by the NACS into working memory (provided that the number of chunks do not exceed the working memory capacity). However, you may find instances where you will want to filter the inputs into the NACS and the conclusions from the NACS. To enable this, the `InputFilterer` and `KnowledgeFilterer` delegates have been defined:

```
delegate ActivationCollection InputFilterer(ActivationCollection input);

delegate IEnumerable<ChunkTuple> KnowledgeFilterer(IEnumerable<ChunkTuple> input);
```

These delegates allow you to specify your own custom filtering operations. If specified (using the `ReasoningInputFilterMethod` or `ReasoningOutputFilterMethod` properties, found in `Agent.ACS`), the ACS will call these delegates prior to initiating reasoning or loading the conclusions into working memory.

Retrieving Chunks from the GKS

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Interacting with Episodic Memory

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Retrieving Episodes

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

"Offline" Learning

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

Generative Actions

This feature is currently under development and, therefore, is not available in the current release of the Clarion Library.

In future releases, this section will contain information about how to use this feature (when it becomes available).

An Example: Using Generative Actions to Change Local Parameters

Remember, as always, if you have any questions, want to submit a bug, or make a feature request, please 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.