Setting Up & Using the ACS

Table of Contents

Walkthrough of the Simple Hello World Task

In this section, we will go line-by-line through one of the simplest tasks to simulate: *Hello World*. This simple task will provide you with the basics for getting up and running building simulations in Clarion. It should not, however, be considered to be the definitive method for developing simulating environments (or constructing agents for that matter). We encourage you to look at all of the samples and consult the various guides and documentation when building your own simulations. Once you get comfortable developing simulations, we encourage you to explore the capabilities of the Clarion Library and use your own creativity when constructing your models.

To fully convey the point that simulations need not follow any rigid format, the `SimpleHelloWorld` simulation is written entirely within a single method (i.e., the "`Main`" method). The basic layout for this task is as follows:

- It uses **ONLY** the ACS with a reinforcement trainable backpropagation network in the bottom level and RER (Rule Extraction and Refinement) turned on in order to extract rules based on the reinforcement.
- The goal is for the agent to learn the following:

  > *If someone says "hello" of "goodbye" to me, then I should respond with either "hello" or "goodbye" respectively.*

- At the end of the task, the above statement should be represented as rules in the top level of the ACS (having been learned via RER).
- Other than knowing the inputs and outputs, the agent should have no *a-priori* knowledge of the task dynamics.

So let's look now at how we would go about setting up this simple task. The first thing we need to do is add the "Simple Hello World" sample to our project. This is generally accomplished by simply right-clicking on your project (i.e., the one we setup in the "*Getting Started Guide*") and choosing the "*Add Files*" option (under the "*Add*" menu item). The file name for this task is "*HelloWorld-Simple.cs*" and it can be found in the "*Simple*" folder in the "*Samples*" section of the Clarion Library package. Once you have added the file, it should open in the main window of your development environment. If it does not, you should be able to open it by double-clicking on it from the Solution Explorer.

The `using` Clause

In the previous ("*Getting Started*") tutorial, we discussed how to add the Clarion Library as a resource (or reference) for a simulation project. However, if you are familiar with Java (or maybe Python), then you will already be aware that a resource library needs to also be signaled within a class file by specifying an inclusion keyword at the top of a code file (for example, with the `import` statement in Java). C# uses the keyword `using`. The code below lists the basic resources we use for our example:

```
using Clarion;
using Clarion.Framework;
```

The `Clarion` and `Clarion.Framework` namespaces are the primary locations for the majority of the classes you will need when setting up and using a Clarion-based agent in your simulation. More advanced examples might use additional libraries.

Declaring the `SimpleHelloWorld` Class

Now that we have signaled the necessary namespaces, we need to specify the initial declaration for the class that will perform the task:

```
public class SimpleHelloWorld
```

This class contains all of the code for my simulating environment, including the initialization of the world, my agent, and the mechanisms that allow the agent to interact with the world. Since this is a simple task, we will actually perform all of these steps within a single (`Main`) method.

The `Main` Method

If you are familiar with Java, C, or C++, you know that a method called `Main` is where any program begins its execution. In C#, this routine is always spelled with a capital letter, and must be declared as `static void` (or `static int`, if you want to

return an exit value to the operating system). As is usual for Java, C, or C++, command-line arguments are handled with the string array, `args`, passed to the `Main` method. While this argument is optional, including it is usually the default behavior in most development environments. In addition, you may find it convenient to use in some of your simulations.

```
static void Main(string[] args)
{
        ...
}
```

We should also note that it is usually a good idea to limit the amount of work that is performed in the `Main` method. In practice, this method should initialize its enclosing class, writing out a couple of progress messages, and call a few methods to initiate the task. The major work of the simulation should be performed in other methods such that your `Main` method is easy to follow and very high-level. This being said, the Simple Hello World example does not follow this convention. Remember, for this particular simulation, we simply wish to demonstrate the simplicity in which simulations can be created. The Clarion Library will still perform a task correctly even if you use poor programming practices.

The first lines you will see in the `Main` method are these:

```
//Initialize the task
Console.WriteLine("Initializing the Simple Hello World Task");

int CorrectCounter = 0;
int NumberTrials = 10000;
int progress = 0;

World.LoggingLevel = TraceLevel.Warning;

TextWriter orig = Console.Out;
StreamWriter sw = File.CreateText("HelloWorldSimple.txt");
```

This code first sets up some task-specific variables to use for tracking the results of the task. It also configures C# to output the results to a text file (instead of to the console window). You do not need to implement any of this within your simulations. We have merely implemented it this way here so that the console will look nice and clean when you actually run the simulation.

Notice also that we specify a value for `LoggingLevel`. The Clarion Library lets you trace the inner workings of your agents at varying levels of detail. By default, if logging is turned on, the logging information gets output to "*Clarion Library.log*". However, if you wish to change this location (or if you want additional details on how logging works within the Clarion Library) see the "*Useful Features*" guide[1].

[1] Located in the "*Features & Plugins*" section of the "*Tutorials*" folder

Initializing the World

Some amount of initialization of the world itself is needed before we can run our task. The next lines of code in our demonstration are these:

```
DimensionValuePair hi = World.NewDimensionValuePair("Salutation", "Hello");
DimensionValuePair bye = World.NewDimensionValuePair("Salutation", "Goodbye");
```

When building a simulation, the first thing we need to do is describe what the simulating environment looks like. For example, you will usually want to tell the World about is the existence of any dimension-value pairs we need for our task. For our Simple Hello World task, there is one dimension named "Salutation", and it is given a value of either "Hello" or "Goodbye." These DV pairs effectively represent the simulating environment for this task.

Note that we can refer to these objects as often as we need to as we continue setting up the simulation by either maintaining links (i.e., pointers) to them within our simulation, or by calling the "Get" methods that are provided in the World singleton object. Later, you will see that we can use our descriptive objects to create input nodes to the agent's implicit decision network. However, for now, simply note that these dimension-value pairs can now be considered to exist within the world.

We now need to define the external actions that the agent will be able to perform (in this case "Hello" and "Goodbye") when presented with salutation inputs. The NewExternalActionChunk method accomplishes this:

```
ExternalActionChunk sayHi = World.NewExternalActionChunk("Hello");
ExternalActionChunk sayBye = World.NewExternalActionChunk("Goodbye");
```

Here, our action chunks are about the simplest possible: indicating either responding "Hello" or "Goodbye", which is all we need for this task. Note, however, that in practice, action chunks can be made as complex as required (since they are in fact chunks[2]).

With these essential initializations out of the way, the simulating environment has essentially been set up. There are, of course, more complex aspects that can potentially be set up here, but we will leave those alone for now and address them in the other, more advanced, guides. At this point, it is time to initialize our agent (whom we will call John):

```
//Initialize the Agent
Agent John = World.NewAgent("John");
```

John is initialized within the world just like any other descriptive object, because technically John is just another object in the world. This is accomplished by calling the NewAgent method. Note that we also give the agent its name at this point, which is the string parameter "John" passed to NewAgent. Also note that this name (or

[2] Chunks are a well-established representational concept that contain arbitrarily large collections of dimension-value pairs

"label" when we are creating goal/action/declarative chunks) is optional and is only needed if you later want to "retrieve", or `Get`, the agent (or chunk) back from the `World`.

To this end, keep in mind that since the names/labels are optional, they can be assigned to multiple agents/chunks. In other words, it is completely possible to have more than one agent in the world named `John`. The system will not break if you want to give your (non dimension-value pair) world objects the same name/label. However, if you choose to do this, it will limit your ability to retrieve the objects from the `World` using the `Get...(by_name_or_label)` methods. In particular, when multiple world objects (of the same object type) have the same name/label, the `Get` methods cannot guarantee that the object that is retrieved will be the object you intended.

With that in mind, if you plan to build a simulating environment with multiple agents, it is very important to make sure you segregate your agent initialization from your descriptive object declaration. Otherwise, you could inadvertently create multiple "copies" of the same object and potentially cause major problems with the operation of your task (if you also use the `World.Get...` methods).

Moving back to our example, we have now created an "empty" agent. We say it is empty, because it does not know anything about those DV pairs and action chunks that were set up earlier, nor does it contain any functional mechanisms with which to interact with the world. Therefore, we need to finish setting up our agent, `John`.

Agent Initialization

In this task, the only functional object we need to initialize ahead of time is an Implicit Decision Network (IDN) in the bottom level of the ACS. To set up this network, we will initialize a backpropagation network. More specifically we use a simplified Q-learning backpropagation network. This line demonstrates how to accomplish this initialization:

```
SimplifiedQBPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
      (John, SimplifiedQBPNetwork.Factory);
```

The network type that we use for the implicit decision network in this simulation is a `SimplifiedQBPNetwork`, one of several artificial neural network components located in the `Clarion.Framework` namespace of the Clarion Library. We generate this network (and associate it with agent `John`) through the `AgentInitializer` (located, as with the `World` class, within the root `Clarion` namespace).

Note again that the `AgentInitializer` is the primary means by which we attach the myriad of internal functional objects (e.g., implicit components, drives, meta-cognitive modules, etc.) to a Clarion agent. Along with the `World` class, the agent initializer represents one of the primary foundational objects for interacting with a Clarion agent within a simulated task environment.

As we mentioned earlier, for the simple Hello World task, all we need is an implicit decision network (or IDN) in the bottom-level of the ACS, so we call the

`InitializeImplicitDecisionNetwork` method to generate the neural network. To initialize the IDN, we pass two items to the initializer method in the `AgentInitializer`:

- The agent (`John`, in this case) to which the network is to be attached
- A factory that will be used to generate the network we want

Note that this method will place the network within `John` in an "initializing" state (more on this in a moment). Since a complex agent could literally have dozens or even hundreds of special-purpose networks, we have implemented a standard factory pattern, which allows for you to specify any type of internal object you desire (including customized ones of your own creation[3]). Here, we want to use a `SimplifiedQBPNetwork` so we pass in the factory that the system will use to generate this network.

For all of the internal functional objects in the Clarion Library (as specified by the Clarion theory), a factory has already been provided for you and can be accessed by statically invoking the `Factory` property. In our current example, specifying `SimplifiedQBPNetwork.Factory` will provide the factory for creating the `SimplifiedQBPNetwork` in the bottom level of the ACS. Note here that we can also pass along any number of parameters to the factory. In this particular example, we do not need to do so, however, you will see other examples as you go along with these tutorials where such parameters are required. For the "built-in" functional objects, lists for both the required and optional parameters can be found in the documentation.

The `InitializeImplicitDecisionNetwork` returns an implicit component (of the type that is specified by the factory). For our example, we are initializing a `SimplifiedQBPNetwork`, and have assigned it to the `net` variable. This `net` can be considered as being part of our agent, `John`. In other words, the network has not only been generated by calling the initialize method, but it has also been attached to the agent that was specified when the method was called. This means that the network belongs to the agent and cannot be used by any other agent.

In this simple example, we are creating only one neural network component (as an IDN in the bottom-level of the ACS). In more elaborate simulations, you will, no doubt, call several other methods in `AgentInitializer` many times in order to generate all of the components you will need for constructing your agents. For details on initializing functional objects (within the other "agent internals containers") consult the documentation for the various "`Initialize`" methods in the `AgentInitializer`. Additionally, further information on initializing certain other components can also be found in the later, more advanced, guides.

At this point, however, we need to finish initializing our IDN, `net`, using the DV pairs and action chunks that we defined earlier. This will give our agent, `John`, the ability

[3] See the "*Advanced Customization*" tutorial (located in the "*Customizations*" section of the "*Tutorials*" folder) for details on how to implement a custom component

to choose actions based upon the sensory information it receives from the world. The four lines below accomplish this:

```
net.Input.Add(hi);
net.Input.Add(bye);

net.Output.Add(sayHi);
net.Output.Add(sayBye);
```

First, we begin by specifying two input nodes for the two DV pairs that represent the salutations. To accomplish this, we call the `Input.Add` method. Similarly, we will also add two output nodes to represent the two response actions by calling `Output.Add`. Note that the number of nodes in the hidden layer is computed automatically, which is fine for this example. In more advanced simulations, however, you can take control of this number, but that is outside the scope of this guide (see the documentation for more details).

As part of the initialization phase for any simulation, your job will be to create and set up all of the functional objects you wish for your agent(s) to use. During initialization these objects are open and available for you to configure using your initialization code. Once initialized, the objects will exist in a special, temporary, initialization area within the agent until you are finished setting them up. When you are finished, you will need to signal to the agent that it can begin using the functional object. When the agent is notified of this fact, it will remove the object from the temporary location and wire it into the appropriate container (e.g., the bottom level of the ACS in our current example). At that point, in most cases, you will **NOT** be able to make additional changes to that functional object (except to "tune" its parameters). In the Clarion Library, this concept is called "committing." All functional objects in the Clarion Library **MUST** be "committed" to an agent after they have been initialized. Committing a functional object makes it "immutable" (i.e., "locked" or read only). You should not attempt to make changes after you make this call.

At this point we have finished initializing `net`; however, we now need to tell `John` that it can begin using the network. To signal to our agent that we have completed the initialization of `net` we make the `Commit` call:

```
John.Commit(net);
```

Tweaking Parameters

The final initialization that is usually performed is parameter tweaking. The Clarion technical specification (which can be found on Ron Sun's website) contains a list of default settings for the various mechanisms described by the Clarion theory. However, you will probably find that, in many cases, these settings do not provide you with the optimal results for your task. In such cases, you will likely want to tune some of the parameters that are introduced in this tutorial.

We will discuss how to tweak parameters in the next guide (i.e. the "*Intermediate ACS Setup*" tutorial located in the "*Intermediate Tutorials*" section of the "*Tutorials*" folder). However, for the Simple Hello World task, the following parameter settings are used to optimize the agent's performance:

```
net.Parameters.LEARNING_RATE = 1;
John.ACS.Parameters.PERFORM_RER_REFINEMENT = false;
```

Running a Simulation (& Continued Walkthrough)

In addition to initializing the world and agents, it is also the job of a simulating environment to act as the intermediary between the agent and the world during the actual running of a task.

In general, the simulating environment needs to handle the following communications:

1. Specifying to an agent the sensory information it perceives on a given perception-action cycle
2. Capturing and recording the action that is chosen by an agent
3. Updating the state of the world (if necessary) based on an agent's actions
4. Providing feedback to an agent as to the "goodness" or "badness" of its actions
5. Tracking the performance of the agent (for the sake of reporting results at the end of the task)

Now that the world and agent have been set up, it is time to give `John` a means for interacting with the world. This is where the bulk of our simulation code actually comes into play.

```
//Run the task
Console.WriteLine("Running the Simple Hello World Task");
Console.SetOut(sw);

Random rand = new Random();
SensoryInformation si;

ExternalActionChunk chosen;
```

The above lines mainly set up some variables that will be useful as part of running the simulation: a random number generator (`rand`) for choosing the configuration of the sensory information, a sensory information pointer to hold onto the sensory information for the current perception-action cycle, and a variable named `chosen` to capture the action chosen by the agent.

The next line begins the major body of the task:

```
for (int i = 0; i < NumberTrials; i++)
{
      ...
}
```

The `for` loop (above) is in charge of the "flow" for the task. Inside this loop, the sensory information is set up for each perception-action cycle (i.e., each trial), the chosen action is captured, feedback is given, and the agent's performance is recorded.

Initializing the Sensory Information

The first thing we must do on any given trial is get a sensory information object. The code below handles this request:

```
si = World.NewSensoryInformation(John);
```

We obtain sensory information objects from the world by calling the `NewSensoryInformation` method and specifying the agent for whom the sensory information is intended. Note that, while it isn't particularly pertinent for the present task, sensory information objects ***cannot*** be shared between agents. This is necessary because the sensory information object not only tracks the state of the world (as seen by that agent), but it also tracks the "internal meta information" of an agent (i.e., the state of its goals, working memory, drives, etc.). If you would like to "share" a certain configuration of the world between two agents, an overload for the `NewSensoryInformation` method exists that will copy the configuration of a sensory information object into a new sensory information object (for the specified agent).

These next lines in the simulation are used to decide on the configuration for a sensory information object:

```
//Randomly choose an input to perceive.
if (rand.NextDouble() < .5)
{
    //Say "Hello"
    si.Add(hi, hi.MAXIMUM_ACTIVATION);
    si.Add(bye, bye.MINIMUM_ACTIVATION);
}
else
{
    //Say "Goodbye"
    si.Add(hi, hi.MINIMUM_ACTIVATION);
    si.Add(bye, bye.MAXIMUM_ACTIVATION);
}
```

For our simple task, we are basically just "flipping a coin" to decide whether John perceives someone saying "hi" or "bye." More complex simulating environments will likely have more complicated methods for determining the appropriate sensory information. However, for this task, a random choice is sufficient.

We set up the sensory information object by adding descriptive objects to it (along with an activation level for each object). This is done using the `Add` method.[4] Also, note that the agent's "internal meta information" is pre-loaded into the sensory information object, so if your simulation needs to set the activation for a part of the agent's "internal state", you would do so by "setting" the activation rather than "adding" it. The following is an example of how you could do this:

```
si[...] = 1;
```

Perceiving and Acting

The next lines initiate the agent's perception of the sensory information and retrieve the agent's action (when one is chosen):

```
//Perceive the sensory information
John.Perceive(si);

//Choose an action
chosen = John.GetChosenExternalAction(si);
```

The `Perceive` method initiates the process of decision-making and `GetChosenExternalAction` returns the action that is chosen by `John` (given the current sensory information). Note that both of these methods can only be called **ONCE** for any given sensory information object (or time stamp). This means that an agent **cannot** perceive a sensory information object more than once, so you **must** generate a **new** sensory information object each time you want your agent to perceive something. It also means that your agent will not repeat itself, so make sure that you either apply its action immediately or store its choice somewhere in the simulating environment.

The above example represents the simplest method for initiating a perception-action cycle. There are other, more complicated, ways to interact with an agent (e.g., using asynchronous operations[5]). These other methods are outside of the scope of this guide, so we will leave further discussion on that topic to later, more advanced, guides.

Processing Outcomes and Delivering Feedback

After the chosen action has been captured, the following `if` statement determines the "consequences" of that action, records the outcome, and reward or punishes John accordingly:

---

[4] The `+/-` operators can also be used to add or remove items from the sensory information (as well as the inputs/outputs of implicit components). Note, however, that the activation when using these operations will always be set to the minimum activation.

[5] Details on how to set up asynchronous operation can be found in the "*Useful Features*" tutorial (located in the "*Features & Plugins*" section of the "*Tutorials*" folder)

```
//Deliver appropriate feedback to the agent
if (chosen == sayHi)
{
    //The agent said "Hello".
    if (si[hi] == hi.MAXIMUM_ACTIVATION)
    {
        //The agent was right.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was correct");
        //Record the agent's success.
        CorrectCounter++;
        //Give positive feedback.
        John.ReceiveFeedback(si, 1.0);
    }
    else
    {
        //The agent was wrong.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was incorrect");
        //Give negative feedback.
        John.ReceiveFeedback(si, 0.0);
    }
}
else
{
    //The agent said "Goodbye".
    if (si[bye] == bye.MAXIMUM_ACTIVATION)
    {
        //The agent was right.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was correct");
        //Record the agent's success.
        CorrectCounter++;
        //Give positive feedback.
        John.ReceiveFeedback(si, 1.0);
    }
    else
    {
        //The agent was wrong.
        Trace.WriteLineIf(World.LoggingSwitch.TraceInfo, "Jon was incorrect");
        //Give negative feedback.
        John.ReceiveFeedback(si, 0.0);
    }
}
```

You should notice two primary things about this segment of code. First, notice that we are able to compare our actions and sensory information objects using the standard "==" comparator. All of the descriptive objects in the Clarion Library have had their operations overloaded so that they act more like value types.[6] Second, to give `John` feedback, all we need to do is call the `ReceiveFeedback` method. Calling this method automatically initiates a round of learning inside John. Even if we were

[6] This is also the case for sensory information objects, and the input/output layers of implicit components.

using a Q-learning network instead of a simplified Q-learning network, we would still only need to call `ReceiveFeedback` and the system will take care of the rest.[7]

Note also that the feedback on this simulation is between 0 and 1. Feel free to use whatever scales you want (although see the "*Intermediate ACS Setup*" for additional considerations regarding this). However, unless it is absolutely necessary, you should try to stick to keeping your activations and feedback between 0 and 1. Using this convention, a feedback of 1 would be tantamount to the highest level of positive feedback possible, and 0 would equate to the highest level of negative feedback (with .5 being more or less neutral feedback).

The remaining lines of code in the simulation basically just update the progress of the simulation and report the results after all of the trials have completed. Therefore, we will not go over these lines. However, once we have finished processing all of our results, there is still one more thing that has to be done before out simulation can be exited.

Killing an Agent

The last step we need to take in order to terminate our simulation is to terminate our agent. For our current example, we kill `John` by calling the following line of code:

```
John.Die();
```

This command initiates the termination of all of `John`'s internal processes (in other words, he "dies"). Note that even though the agent's processes are terminated, its internal configuration is still maintained. This means you can still access/view or "save" (i.e., serialize) the agent's internal configuration even after the agent dies.[8] If we didn't remember to kill John, then the application would not be able to close because `John`'s internal mechanisms would still be running. However, once `John` is dead, the simulation will be able to exit by returning from the `Main` method.

At this point, you should have everything you need to start writing your first (simple) simulation. If you get stuck at any point, consult this guide again or take a look at the API resource document (in the "*Documentation*" folder). When you are ready to move on to the more complicated aspects of the ACS, the next tutorial ("*Intermediate ACS Setup*"), can be found in the "*Intermediate Tutorials*" section of the "*Tutorials*" folder.

In addition, you may also want to check out the "*Useful Features*" tutorial.[9] It contains information about some of the "built-in" enhancements of the Clarion

---

[7] In the case that an implicit component expects new input (as is the case for Q-learning), the system will actually wait until the perceive method is called on the next sensory information object before performing learning.

[8] Details on the latter can be found in the "*Using Plugins*" guide (located in the "*Features & Plugins*" section).

[9] Located in the "*Features & Plugins*" section of the "*Tutorials*" folder.

Library. These enhancements have been designed to aid you in the development of your simulating environments.

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.