Basic Customization

Table of Contents

Customized Methods (Using Delegates)

For several of the algorithms specified by the Clarion theory (e.g., eligibility checking, rule refinement, match calculating, etc.), the library only implements the default method, even though there may be other ways to perform those operations. This being said, you may run into instances where you will need to designate a different operation to replace the system's default behavior for a certain algorithm. To address this need, the Clarion Library leverages C#'s delegate feature and defines a series of "delegate signatures" that can be used to define your own custom algorithms. Whenever a custom method is specified during initialization, the system will use this method in lieu of its default behavior.

So let's begin our tutorial on setting up and using delegates by looking at one of the algorithms that you are most likely to want to customize: checking the eligibility of a component. For customizing this method, the Clarion Library defines the `EligibilityChecker` delegate. The signature for this delegate is:

```
public delegate bool EligibilityChecker
       (ActivationCollection currentInput = null, ClarionComponent target = null);
```

The default eligibility checking algorithm for an `ImplicitComponent` is simply to return the value of the component's "`ELIGIBILITY`" parameter. While this provides a simple way for you to manually prevent or allow a component to be used by the system, it does not, otherwise, provide any additional logic for determining the eligibility. For example, suppose you wanted to define some conditions for when a component should be used and you want the system to be able to integrate this "condition eligibility check". To accomplish this, you need to implement a method that will perform the eligibility checking operation and then inform the system of the result of this check.

Implementing a custom eligibility checking delegate is accomplished by creating a method within your code that uses the same inputs and returns a value of the same type as is specified by the "delegate signature" (from above). The following pseudo-code demonstrates how such a method might look:

```
public bool Custom_EligibilityCheck (ActivationCollection currentInput = null,
       ClarionComponent target = null)
{
    ... // Do operations to determine if the target component is eligible
    return true or false;
}
```

After the method is set up, if we wanted a particular component to make use of it when checking its eligibility, we would need to specify the method as a parameter (in the form of an `EligibilityChecker` delegate) during the initialization of that component. The system will use our custom delegate method to check the eligibility of any components with which it was initialized.

Specifying Delegates as Parameters during Initialization

For the most part, we specify delegate methods during the initialization (using the `AgentInitializer`) of an internal (functional) object. For example, suppose we created a method called `Custom_EligibilityCheck` (in our own code) that matches the signature for the `EligibilityChecker` delegate. The following code demonstrates how our custom method could be specified as part of the initialization of a `BPNetwork` in the bottom level of the ACS of the agent, John:

```
BPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
     (John, BPNetwork.Factory, (EligibilityChecker)Custom_EligibilityCheck);
```

Note that we have explicitly casted our custom method to the correct delegate specification (i.e., `EligibilityChecker`) during the initialization call. We do this because it is required in order to pass a `delegate` using the `dynamic` type designation. This being said, there are other ways to specify the type of our delegate method. Specifically, we can "wrap" our delegate method in a property and use that property instead of explicitly casting our custom method. From the previous example, initializing the `BPNetwork` can alternatively be done as follows:

```
// During the initialization method:
BPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, BPNetwork.Factory, CustomEligibilityCheckerDelegate);
... // At some other point in your code:
public EligibilityChecker CustomEligibilityCheckerDelegate
{
    get
    { return Custom_EligibilityCheck; }
}
```

You can use whichever method you'd prefer, however, we recommend using the property method as it is generally cleaner and easier to follow.

Specifying custom delegate methods during the initialization of an internal object is usually optional. To find out which delegates an internal (functional) object can use, consult the API resource document (in the "*Documentation*" folder) for the **factory** class that is used to generate that internal object.

At this point, we should point out that while custom delegates are mainly optional, there some internal (functional) objects that do actually **require** you to implement some custom delegates in order to be initialized.[1] In the following section, we will look at two such internal objects: IRL rules, and fixed rules.

Creating Custom Rules

Depending on the specifics of the task you are simulating, you may discover that you need to implement a rule whose dynamics are more complex than what can be captured using a simple `RefineableActionRule`. To handle this case, the Clarion theory defines two types of rules: IRL rules, and fixed rules.[2]

In the Clarion Library, we have implemented these rule types using two classes: the `IRLRule` class, and the `FixedRule` class. These classes provide the large majority of the mechanisms that are required for the rules. All you need to do is specify a single custom delegate method in order to initialize either an IRL rule or a fixed rule. More specifically, you need to define the method that is used for calculating the support for the rule (via the `SupportCalculator` delegate). The system uses this support measure to determine if a rule is eligible for action recommendation at a given time step (based on a "partial match threshold"[3]). The signature for the `SupportCalculator` delegate is as follows:

```
public delegate double SupportCalculator
    (ActivationCollection currentInput, Rule target = null);
```

To help clarify this concept further, let's look at a few examples where we may want to use each of these rule types.

Using the `SupportCalculator` Delegate to Set Up an IRL Rule

In this section we will cover an example of where an `IRLRule` would be necessary. One of the most common instances when this rule type is necessary is when a certain factor of a rule's condition is, itself, conditioned upon another factor of that condition. For example, let's assume that we have the dimension-value pairs: {`dim1`, `a`}, {`dim1`, `b`}, {`dim2`, `c`}, and {`dim2`, `d`} and the action: {`do_something`}. Now, suppose we want to create the following rule:

[1] Again, consult the API resource document of the factory class for this information

[2] See Sun (2003) for more details

[3] Captured by the `PARTIAL_MATCH_THRESHOLD` parameter

If {dim1, a} AND {dim2, c}*, but NOT* {dim2, d} *then recommend the* {do_something} *action, otherwise don't recommend it*

To capture the condition of this rule, we will need to write a custom method that can be initialized using the SupportCalculator delegate signature. Using pseudo-code, we could express this method as follows:

```
public double CalculateSupport_IRL(ActivationCollection currentInput, Rule r)
{
    return the maximum activation between {dim1, a} and {dim2, c} if both
           {dim1, a} and {dim2, c} are specified as being part of the condition
           while {dim2, d} is both specified as NOT being part of the condition
           and is NOT activated in currentInput. Otherwise, 0
}
```

Note that the IRLRule class derives from the RefineableActionRule<> class, so it has all of the same refinement capabilities as that rule type. Therefore, we can use the Clarion Library's built-in generalization and specialization processes to automatically refine our IRLRule without needing any additional customizations be set up to facilitate it. For instance, suppose that the system decided that the following refined rule is better able to capture the dynamics of our current task example:

If {dim1, a} OR {dim1, b} AND {dim2, c}*, but NOT* {dim2, d} *then recommend the* {do_something} *action, otherwise don't recommend it*

We want to make sure, when constructing custom delegate methods, that we capture the most essential factors for your rule while still maintaining enough flexibility to accommodate any refinements that may be made to the rule. For our current example, the main factors are the co-activation of **any** dimension-value pair in `dim1` **and** the `{dim2, c}` dimension-value pair, but **NOT** the activation of the `{dim2, d}` dimension-value pair. Therefore, our support calculator method should capture these factors. Below is an example of how we might express this in C#:

```
public double CalculateSupport_IRL(ActivationCollection currentInput, Rule r)
{
    var d1 = from d in currentInput
                where d.WORLD_OBJECT.AsDimensionValuePair.Dimension == "dim1" &&
                r.GeneralizedCondition[d.WORLD_OBJECT] == true
                select d;

    return (r.GeneralizedCondition["dim2","c"] == true &&
                currentInput["dim2", "c"] > 0 &&
                r.GeneralizedCondition["dim2", "d"] == false &&
                currentInput ["dim2", "d"] == 0)? d1.Max(e => e.ACTIVATION) : 0;
}
```

To fully understand the above code, you need to be aware of the C# language features that it is leveraging. The first line uses LINQ to get all of the dimension-value pairs in `dim1` that are specified as being part of the condition. The second line (i.e., the `return` line) uses a combination of lambda expressions and the conditional operator to return the maximum activation (which captures the **OR** operation) of the dimension-value pairs that were found in the first line if the dimension-value pair `{dim2, c}` is activated and `{dim2, d}` is not. Alternatively, if the condition is not true (i.e., either `{dim2, c}` is not activated or `{dim2, d}` is), then the second line will return zero.

Now that we have setup our custom method, let's look at how we would go about initializing the `IRLRule`.

Initializing the IRL Rule

To setup the `IRLRule` we need to do three things. First, we need to initialize it:

```
// During the initialization method:
IRLRule rule1 = AgentInitializer.InitializeActionRule
    (John, IRLRule.Factory, some_action, SupportDelegate);

DimensionValuePair dv1 = World.NewDimensionValuePair(“dim1”, “a”);
DimensionValuePair dv2 = World.NewDimensionValuePair(“dim1”, “b”);
DimensionValuePair dv3 = World.NewDimensionValuePair(“dim2”, “c”);
DimensionValuePair dv4 = World.NewDimensionValuePair(“dim2”, “d”);

... // At some other point in your code:
public SupportCalculator SupportDelegate
{
    get
    { return CalculateSupport_IRL; }
}
```

Second, we need to setup the initial condition for the rule:

```
// Elided rule initialization (see above)

rule1.GeneralizedCondition.Add(dv1, true);
rule1.GeneralizedCondition.Add(dv2, false);
rule1.GeneralizedCondition.Add(dv3, true);
rule1.GeneralizedCondition.Add(dv4, false);
```

Finally, we need to commit the rule:

```
John.Commit(rule);
```

That is everything you need to do to setup an `IRLRule`. So let's turn our attention now to an example of how to setup a `FixedRule`.

Using the `SupportCalculator` Delegate to Set Up a Fixed Rule

The process for setting up a `FixedRule` is very similar to setting up an `IRLRule`. A good example of where we might want to use a `FixedRule` is when part of the algorithm for determining the support of a rule requires that we perform some sort of mathematical translation on parts of the `SensoryInformation`. For instance, let's suppose we want to setup the following rule, which determines whether a "carry-over" operation is needed when performing addition:

If {`operator`, +} and {`digit1`, x} + {`digit2`, y} > 9, then recommend the {`carry-over`} action, otherwise don't recommend it.

We capture the condition of this rule within our custom delegate method. The following cod demonstrates how we would do this in pseudo-code:

```
public double CalculateSupport_FR(ActivationCollection currentInput, Rule r)
{
    return 1, if {operator, +} is activated and {digit1, x} + {digit2, y} > 9
           Otherwise, 0
}
```

While it is a little longer, expressing the pseudo-code in C# would look something like this:

```
public double CalculateSuppot_FR(SensoryInformation currentInput, Rule r)
{
    if (currentInput["operator", "+"] > 0)
    {
        var d1 = (from d in currentInput
                  where d.WORLD_OBJECT.AsDimensionValuePair.Dimension == "digit1"
                  select d).OrderByDescending(e => e.ACTIVATION).First();

        var d2 = (from d in currentInput
                  where d.WORLD_OBJECT.AsDimensionValuePair.Dimension == "digit2"
                  select d).OrderByDescending(e => e.ACTIVATION).First();

        if (((int)d1.WORLD_OBJECT.AsDimensionValuePair.Value.AsIComparable) +
            ((int)d2.WORLD_OBJECT.AsDimensionValuePair.Value.AsIComparable) > 9)
            return 1;
        else return 0;
    }
    else
        return 0;
}
```

The LINQ queries (i.e., the first 2 lines inside of the first `if` statement) are used to find the maximally activated digits (represented as dimension-value pairs). Note that we assume, in this example, that only one digit will be activated for each dimension, so getting the `First` value from the dimension (after it has been sorted in descending order by activation) should give us digits x & y (from the pseudo-code). The first `if` statement checks to see if the + operator is activated in the `SensoryInformation`. The second `if` statement checks to see if the sum of the two digits will require that the `carry-over` action be performed and will return 1 (i.e., the action should be recommended) if it does, or 0 (i.e., the action shouldn't be recommended) if it does not.

A Note on the Generically Typed DimensionValuePair<DType,VType> Class

You may have noticed the following from the example we are using to demonstrate how to set up a FixedRule:

```
Value.AsIComparable
```

Beginning with version 6.1.0.7 of the Clarion Library, dimension-value pairs actually come in two flavors:

- The standard DimensionValuePair class
- A generically typed DimensionValuePair<DType,VType> subclass

Although this is the case, you likely have not realized it until this point, since the vast majority of your interaction with the World class automatically gives you the generically type DimensionValuePair<DType,VType>. This generically typed DimensionValuePair<DType,VType> is very useful as it significantly simplifies your interaction with dimension-value pairs by reducing the amount of explicit casting that is necessary. For example, suppose we did the following:

```
World.NewDimensionValuePair("Digit", 1)
```

If we were to subsequently call the Dimension or Value properties of the DimensionValuePair<DType,VType> object that is returned by that method, the dimension or value that we will get back will already be cast as the appropriate type (i.e., as a string or an int respectively). This is the case whenever working with generically typed dimension-value pairs.

While this is certainly a very useful addition, the primary reason that we added the generically type DimensionValuePair<DType,VType> class, was to enable values of different types to inhabit the same dimension. This has been accomplished by implementing a special "wrapper" class (called V) within the standard DimensionValuePair class. This wrapper class has specially overloaded IComparable methods that allow differently typed values to be compared using their ToString() representations. While there are **MANY** benefits to implementing this, the downside is that calling the Value property of the standard DimensionValuePair (e.g., when using the AsDimensionValuePair property of the IWorldObject interface) will actually return the V class "wrapper" instance as opposed to the underlying IComparable value itself (which is what is returned by the Value property of DimensionValuePair<DType,VType>).

We have gotten around this issue by defining a property, called AsIComparable, within the V class that exposes the underlying IComparable value. Note, however, that you will likely also have to explicitly cast this value back to its appropriate type to make use of it.[4] The following example (taken from the previous FixedRule

[4] We acknowledge that this solution is "suboptimal", however, unless (or until) Microsoft decides to allow custom implicit casting to interfaces or (better yet) allows custom down casting in C#, this is the best solution we could come up with.

example) demonstrates how we might go about exposing (and explicitly casting) the underlying `IComparable` value when working with the standard `DimensionValuePair` class:

```
if (((int)d2.WORLD_OBJECT.AsDimensionValuePair.Value.AsIComparable) +
     ((int)d2.WORLD_OBJECT.AsDimensionValuePair.Value.AsIComparable) > 9)
        return 1;
else return 0;
```

Now that we have addressed this point, let’s look at how we might go about initializing a `FixedRule`.

Initializing the Fixed Rule

Setting-up the `FixedRule` is essentially the same process as it was for the `IRLRule`:

```
// During the initialization method:
FixedRule rule = AgentInitializer.InitializeActionRule
    (John, FixedRule.Factory, carry_action, SupportDelegate);
John.Commit(rule);

... // At some other point in your code:
public SupportCalculator SupportDelegate
{
    get
    { return CalculateSupport_FR; }
}
```

Note that, since fixed rules are not refineable, we technically do not need to specify a condition for them. We call these types of fixed rules “condition-less.” In addition, by default, fixed rules are also not deletable (e.g., via density considerations[5]). In general, you will want to use a fixed rule to capture the “one-shot-learning” paradigm. That is, fixed rules are usually obtained in some sort of explicit fashion (e.g., via instruction).

This concludes the basic customization tutorial. You should now have everything you need to know in order to do basic customizations using delegates within the Clarion Library. Feel free to explore the documentation to discover all of the places throughout the system where custom delegate methods can be used. Using delegate methods is a great way to customize your simulations without needing to “reinvent the wheel”, so to speak.

However, if you find that you have reached the limit of what custom delegates can provide or you feel like taking on a challenge, then you should know that you can implement your own customized internal (functional) objects (e.g., implicit components, drives, rules, etc.). Details on how to do this can be found in the “*Advanced Customization*” tutorial. However, be forewarned that implementing a custom internal (function) object is **NOT** a simple process. Therefore, you should have a thorough understanding of the Clarion theory as well as significant

[5] Although this can be enabled by toggling the `DELETABLE_BY_DENSITY` parameter

experience working with the Clarion Library before endeavoring to take on this sort of customization.

Generic Equations

It is not uncommon that, during the development and tuning of your task, you may find it both quicker and more preferable to temporarily "shortcut" some of the more laborious aspects of initialization (e.g., pre-training implicit components such as neural networks) for the bottom level of an agent's subsystems. Frequently, these implicit components are simply expected to report the results of an already known equation. For this sort of event, the Clarion Library provides `GenericEquation`[6], which can easily be setup and "plugged into" the bottom level anywhere within your agent. Specifically, this component makes use of the `Equation` `delegate` in order to allow you to easily create your own, custom equation. The signature for this delegate is:

```
public delegate void Equation
    (ActivationCollection input, ActivationCollection output);
```

As was the case for the other basic customizations (discussed previously), to implement a custom equation, all you need to do is define a method within your own code that conforms to the above signature and the Clarion Library will then be able to use that method to set the activations for the "nodes" on the output layer (given the specified input).

For example, let's suppose that we want to create a `GenericEquation` that can be used to solve a simple linear equation (i.e., $= X$). We can accomplish this using the following method:

```
public void LinearEquation
    (ActivationCollection input, ActivationCollection output)
{
    output["Variable", "Y"] = input["Variable", "X"];
}
```

You should note that the `GenericEquation` class conforms to the library's standard convention of transforming/bounding activations between 0 and 1. However, you can specify your own range for your equation by simply changing `Parameters.MIN_ACTIVATION` and `Parameters.MAX_ACTIVATION` (found by using the `Parameters` property in the `GenericEquation` class). By setting these parameters, both the input and output that is passed to your `delegate` method will be automatically transformed/bounded to between your specified range.

After we have created our `delegate` method, all we need to do in order to use it is provide it as a parameter during the initialization of a `GenericEquation`. For example, suppose we wanted to use the simple linear equation in the bottom level of the ACS for our agent, John. This could be accomplished as follows:

[6] Located in the *Clarion.Framework.Extensions* namespace

```
GenericEquation eq = AgentInitializer.InitializeImplicitDecisionNetwork
    (John, GenericEquation.Factory, (Equation)LinearEquation);
```

As a reminder, in order to complete the initialization of our `GenericEquation`, we also need to define the inputs and outputs, as well as “commit” the component to the agent. Taken together, the following code demonstrates how we might setup and initialize the simple linear equation (with a range between ±10) in our agent, John:

```
public void InitializeAgent()
{
    DimensionValuePair x = World.NewDimensionValuePair("Variables", "X");
    DimensionValuePair y = World.NewDimensionValuePair("Variables", "Y");
    Agent John = World.NewAgent("John");
    ... //Elided additional agent initialization
    GenericEquation eq = AgentInitializer.InitializeImplicitDecisionNetwork
        (John, GenericEquation.Factory, (Equation)LinearEquation);

    eq.Input.Add(x);
    eq.Output.Add(y);

    eq.Parameters.MIN_ACTIVATION = -10;
    eq.Parameters.MAX_ACTIVATION = 10;

    John.Commit(eq);
}
... //Elided code for running the task
public void LinearEquation
    (ActivationCollection input, ActivationCollection output)
{
    output["Variable", "Y"] = input["Variable", "X"];
}
```

Using the `GenericEquation` can save valuable time while in the debugging and tweaking phases of developing your task. However, keep in mind that ultimately you will want to replace these “shortcuts” with the actual pre-trained bottom level constructs that are defined within the Clarion theory (e.g., a backpropagation neural network, or `BPNetwork`). Note that, in order to help simplify these sorts of pre-training processes, the Clarion Library provides a very useful tool, called the `ImplicitComponentInitializer`[7]. Details on how to use the `ImplicitComponentInitializer` can be found in the *Useful Features* tutorial (located in the *Features & Plugins* section of the *Tutorials* folder).

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.

[7] Located in the main *Clarion* namespace