Intermediate MS & MCS Setup

Table of Contents

Setting Up and Using Drives and Meta-Cognitive Modules

Drives use factors from both the internal and external state information (located within the `SensoryInformation` object) to transform them into a "drive strength" (i.e., the amount of activation for a drive). However, without mechanisms to process these drive strengths and make decisions based upon them, the drives alone will have little effect on the overall operation of an agent. Therefore, we rely on meta-cognitive modules to make decisions based upon these drives strengths (as well as other factors) and to initiate a variety of internally-directed meta-cognitive actions.

It is because of the tight coupling between drives and meta-cognitive modules that we have chosen to present these concepts together. We begin by demonstrating how to set up and initialize a drive in the bottom level of the motivational subsystem. Afterwards, we present an example of a meta-cognitive module that will combine the drive strengths from various drives and then use that information to update the goal structure using a `GoalStructureUpdateActionChunk`. But first let's begin by describing how to initialize a drive.

Initializing a Drive

To start, we should note that a drive object is considered to be a "special form" of a functional object within the CLARION Library. We say this because the drive object doesn't directly extend from the base `ClarionComponent` class but is essentially just a wrapper around an `ImplicitComponent` that provide a few additional drive-specific features.

The CLARION Library comes equipped with all of the primary drives specified in the technical specification document. Each of these drives is defined by their own class and use the following naming convention:

```
PrimaryDriveNameDrive
```

For example, the class names for the food drive and dominance and power drive are "FoodDrive" and "DominancePowerDrive"[1] respectively. These drives are initialized using the `InitializeDrive` method in the `AgentInitializer` class. For example, the following code will initialize a `FoodDrive` in our agent, `John`:

```
FoodDrive food = AgentInitializer.InitializeDrive(John, FoodDrive.Factory, .5);
```

Drives are designated as being one of the following groups within the bottom level of the MS:

- Approach drives (i.e., BAS drives)
- Avoidance drives (i.e., BIS drives)
- "Both" drives (i.e., both approach and avoidance oriented)
- "Unspecified" drive types (i.e., they either don't belong to a behavioral system, or their behavioral system has not been specified)[2]

In general, you shouldn't need to access this group specification in order to use a drive. All of the "built-in" drives in the CLARION Library (based on the CLARION theory) specify their appropriate group during initialization. However, if you want to change the group affiliation of a built-in drive, you can specify it as an optional parameter during initialization. Below is an example of what this might look like if we were to change the group specification for the `FoodDrive` in our agent, `John`:

```
FoodDrive foodDrive = AgentInitializer.InitializeDrive
    (John, FoodDrive.Factory, .5,
     MotivationalSubsystem.DriveGroupSpecifications.BOTH);
```

Additional, you may also want to create your own "deficit change processor" to process how the deficits change over time.[3] This is accomplished using "custom delegates", however implementing something like this is a more advanced concept outside of the scope of this guide. [4]

The Drive Equation

Recall that we mentioned earlier that a drive is just a "wrapper" for an `ImplicitComponent`, so the next thing we need to do is initialize an `ImplicitComponent` inside of the `FoodDrive`. Ideally, you will want to use

---

[1] For those drives that contain the "&" symbol in their name, the "&" conjunction has been left out of the class name for those drives.

[2] The specification of the drive's behavior system can be found in the documentation for the drive's class as well in an instance of a class (via the `BehaviorSystem` property).

[3] By default, drive deficits change by a multiplicative factor of the `DEFICIT_CHANGE_RATE` (located in the local parameters class instance of the drive).

[4] Details on how to implement a "custom delegate" can be found in the "*Basic Customization*" tutorial in the "*Customizations*" section.

something like a pre-trained BPNetwork in your drive. However, since pre-training an implicit component can get a bit complicated, for our current example we will demonstrate a quicker and easier component, the DriveEquation, instead.[5]

The DriveEquation is an extension of the CLARION theory.[6] In general, you should usually only use it when you are testing the configuration of your agents. Once an agent is properly configured, you should replace it with a more "distributed" type of implicit component (such as a BPNetwork). The following code demonstrates how to set up a DriveEquation within the FoodDrive of our agent, John:

```
DriveEquation foodEq =
    AgentInitializer.InitializeDriveComponent(foodDrive, DriveEquation.Factory);
```

We initialize the equation by calling the InitializeDriveComponent method located in the AgentInitializer. Note that the DriveEquation class uses the following equation for calculating the drive strength of a drive:

$$DS_{d,t} = Gain_{universal} \times Gain_{system} \times Gain_{drive} \times Stimulus_{d,t} \times Deficit_d + Baseline_d$$

The details regarding this equation can be found in addendum #1 of the CLARION Technical Specification (located on Ron Sun's website). What is of particular importance here is the series of variables defined by the equation.

We refer to these variables as "typical drive inputs." They can be found within variables located in either the "parameters class" for the drives (e.g., UNIVERSAL_GAIN, DRIVE_GAIN, or BASE_LINE) or in the "parameters class" for the motivational subsystem (e.g., SYSTEM_GAIN).

Generating dimension-value pairs to represent these "typical inputs" can be readily generated by calling the GenerateTypicalInputs method, which is statically available in the Drive class. Furthermore, if the ImplicitComponent within a drive contains any of these "typical inputs", the system will automatically fill them in with the values appropriate for those inputs.

When an instance of DriveEquation is initialized using the AgentInitializer, the following will already be configured for you in the DriveEquation instance that gets returned:

1. The inputs to the DriveEquation (populated with the dimension-value pairs representing the "typical inputs" for the equation
    - Note that the dimension ID will be set to the Type of the drive in which the equation is being initialized and the value IDs will be of the enumerated type Drive.MetaInfoReservations.
    - Note also that these inputs are generated by statically calling the

[5] The CLARION Library comes equipped with a feature (the ImplicitComponentInitializer) that can aid with the initialization and pre-training of implicit components. The details on how to use it can be found in the "*Useful Features*" tutorial located in the "*Features & Plugins*" section.
[6] Located in the *Clarion.Framework.Extensions* namespace.

Drive.GenerateTypicalInputs method.

2. The output from the DriveEquation (specified as a dimension-value pair representing the "drive strength" for that drive.
    - Note that the dimension ID will be set to the Type of the drive in which the equation is being initialized and the value ID will be the DRIVE_STRENGTH specification from the enumerated type Drive.MetaInfoReservations.
    - Note also that this output is generated by statically calling the Drive.GenerateTypicalOutput method

This pre-loading behavior of the input layer (as was described in #1, above) is unique to DriveEquation. However, the pre-loading of the output layer is not. The CLARION Library requires that, for those implicit components that are being used within a drive, the output layer of those components **MUST** contain the "DRIVE_STRENGTH" dimension-value pair (as was described in #2, above) and can **ONLY** contain that dimension-value pair. If you attempt to put a different dimension-value pair in the output layer, the ImplicitComponent will fail when you try to commit it to the drive. As a result of this requirement, the InitializeDriveComponent method will automatically populate the output layer of the ImplicitComponent it generates with the appropriate "DRIVE_STRENGTH" dimension-value pair for the drive in which the ImplicitComponent is initialized.

Finally, as is always the case, once you have finished initializing the drive's ImplicitComponent, you must commit that component to the drive. In addition, you will also then need to commit the drive itself to the agent. The following code demonstrates how this is done for the FoodDrive of our agent, John:

```
foodDrive.Commit(foodEq);
John.Commit(foodDrive);
```

Stimulating a Drive

If you use only the "typical inputs," then you will only need to specify the activation of the STIMULUS input. The following code demonstrates how you might specify the STIMULUS for the FoodDrive in a sensory information object (generated during the running of a task) for our agent, John:

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

si[typeof(FoodDrive), FoodDrive.MetaInfoReservations.STIMULUS] = 1;

... //Elided initialization of other aspects of the sensory information

John.Perceive(si);

... //Elided code for running the rest of the task
```

Note that you are not required to use the "typical inputs" for your drives. However, if you don't use them, you will then have to specify the activations for the inputs of your drives during the running of the task every time you create a new sensory information object. Also, as you may have noticed from the above code, we do not need to "add" the `STIMULUS` variable to the sensory information object. This is because the `NewSensoryInformation` method automatically populates the sensory information object with all of the agent's "meta information" (which includes the drive inputs and outputs, among other things) before the object is returned to the simulating environment. Instead of "adding" the internal meta information, we can simply access it from the sensory information object (as was demonstrated in the above code).

You now have everything you need in order to set up drives in the bottom level of the MS. However, in order to use these drives, you also need to implement one or more meta-cognitive module(s) that will act based on these drives. So now let's turn to discussing how to initialize meta-cognitive modules, following which we will demonstrate how to integrate the drives with them.

Initializing a Meta-Cognitive Module

Operationally, a meta-cognitive module acts essentially like a "mini-ACS," except that the actions of a meta-cognitive module is directed towards manipulating the internal aspects of an agent (such as the goals in the goal structure, certain parameters within other subsystems, etc.). A meta-cognitive module can be comprised of any combination of `ImplicitComponent` instances in the bottom-level and `RefineableActionRule` instances in the top level. However, unlike the ACS, meta-cognitive modules are more limited in their capabilities. For example, a meta-cognitive module does not use `FixedRule` instances in the top level and the action recommendations from the top and bottom levels are always combined.

In general, you should mainly set up a meta-cognitive module using the bottom level. This makes sense conceptually, since meta-cognitive processes tend to be sub-conscious. This being said, rule extraction and refinement is enabled by default within the modules. Note, however, that no mechanism is provided for delivering the specialized feedback that would be needed in order to take advantage of RER within these modules. In fact, in order to leverage the RER capabilities in the MCS, we would need to develop a meta-cognitive module that could interpret both internal and external factors and then deliver the reinforcement signal to the other modules in the MCS.[7]

The process for setting-up a meta-cognitive module is similar to initializing a drive. For this tutorial, we will look at a commonly used module: the `GoalSelectionModule`. Below is an example of how you would initialize this module within the agent, `John`:

[7] This capability, while within the scope of the CLARION theory, is currently only conceptual. However, future research into this concept may eventually lead to the implementation of such a module.

```
GoalSelectionModule gsm =
    AgentInitializer.InitializeMetaCognitiveModule
    (John, GoalSelectionModule.Factory);
```

After we have initialized the module, we can start populating it with implicit components and rules. You can initialize these components by calling either the `InitializeMetaCognitiveDecisionNetwork` or the `InitializeMetaCognitiveActionRule` methods located within the `AgentInitializer`. Below is an example of how we could initialize a `GoalSelectionEquation`[8] within the bottom level of the `GoalSelectionModule`:

```
GoalSelectionEquation gse =
    AgentInitializer.InitializeMetaCognitiveDecisionNetwork
    (gsm, GoalSelectionEquation.Factory);
```

The input layer for the implicit components (and conditions of any rules for that matter) of a meta-cognitive module can consist of any type of (descriptive) `IWorldObject` (just like in the ACS). However, they can also make use of several other types of inputs that you wouldn't normally use in the ACS. Specifically, meta-cognitive modules will often specify "`DRIVE_STRENGTH`" dimension-value pairs (from the previous section) as part of the input layer of their implicit components.

The Goal Selection Equation

Remember that the bottom level of the MS is in charge of determining drive strengths based on the combination of stimulus from the sensory information as well as certain "individual differences" considerations (i.e., gains, deficit, etc.). For example, the `GoalSelectionEquation` combines the drive strengths (and any other descriptive world object) to make goal recommendations for the goal structure based on the following equation:

$$GS_{g,t} = \sum_{d} ds_{d,t} \times Relevance_{d,g} + \sum_{i} dv_{i,t} \times Relevance_{dv,g}$$

Let's break down this equation to better understand how to set up the `GoalSelectionEquation` within your code. The first half of the equation relates specifically to the drive strengths. This part of the equation sums together the drive strengths for all of the drives. In addition, a weighting factor is applied to each drive strength. This weighting factor specifies the "relevance" that each drive has to the goal whose "goal strength" is being calculated. The second half of the equation is essentially the same as the first half, except that it applies the process to the other descriptive world objects (e.g., dimension-value pairs, chunks, etc.) that are "relevant" to the goal. The goal strength of each goal, which is the output of this equation, indicates the "value" for setting a goal within the goal structure.

---

[8] Like the `DriveEquation`, the `GoalSettingEquation` is an extension component and can be found in the *Clarion.Framework.Extensions* namespace.

Correlating Drives and Meta-Cognitive Modules

The input layer of the GoalSelectionEquation can contain any number of "drive strength dimension-value pairs" or other relevant descriptive world objects. The output layer can **ONLY** contain goal structure update action chunks. Recall that in the previous tutorial we demonstrated how a GoalStructureUpdateActionChunk could be used to set (or remove) goals within the GoalStructure.[9] The GoalStructureUpdateActionChunk is what enables the GoalSelectionModule to update the GoalStructure.

Once the GoalSelectionEquation is set up within the GoalSelectionModule, it gets used to calculate the goal strengths, which the GoalSelectionModule then uses to select a GoalStructureUpdateActionChunk. The goal associated with that action chunk is set (or removed) in the GoalStructure by initiating a goal structure update event within the system. That event will prompt the MS, which will perform the update based on what is specified by the action.

At this point, let's step through an example to demonstrate the process of setting up the GoalSelectionModule. This example correlates the FoodDrive to a GoalStructureUpdateActionChunk that "resets" the goal structure and then "sets" a goal, **g**, in the goal structure of our agent, John. The first line of our example is as follows:

```
gse.Input.Add(foodDrive.GetDriveStrength());
```

This line adds the "drive strength dimension-value pair" of the FoodDrive to the input layer of the GoalSelectionEquation that we initialized earlier. The next three lines initialize the GoalStructureUpdateActionChunk that "sets" goal **g** in the goal structure and add it to the output layer of the GoalSelectionEquation:

```
GoalStructureUpdateActionChunk gAct = World.NewGoalStructureUpdateActionChunk();

gAct.Add(GoalStructure.RecognizedActions.SET_RESET, g);

gse.Output.Add(gAct);
```

After the input and output layers have been set up we need to specify the relevance that each input has to each output. This is done using the following convention for each of the input → output relevance pairings:

```
SomeGoalSelectionEquation.SetRelevance(SomeGoalStructureUpdateActionChunk,
    SomeDrive or SomeWorldObject, SomeRelevanceValue);
```

To correlate the FoodDrive to the GoalStructureUpdateActionChunk for our example the code will look something like this:

```
gse.SetRelevance(gAct, foodDrive, 1);
```

[9] See the "*Setting Up & Using the Goal Structure*" tutorial located in the "*Basics Tutorials*" section of the "*Tutorials*" folder.

Finally, as was the case with initializing a drive, once we are finished setting up a component for our meta-cognitive module, we need to commit it to the module. Additionally, after the module has been completely initialized, we have to commit it to the agent. The code below shows how this would be done in our current example:

```
gsm.Commit(gse);

John.Commit(gsm);
```

Meta-Cognitive Module Integration

Once all of the components and modules have been committed, the system will automatically integrate them into the internal processes of the agent. No other interventions are required to make the modules interact correctly with the other parts of the system. In general, when a meta-cognitive module has been set up and committed to an agent, it will operate "behind-the-scenes." However, if you would like to view the inner workings or outcomes from either the motivational subsystem or the meta-cognitive modules, several features are available in the CLARION Library to accomplish this.

For instance, the results from either updating the drive strengths in the MS or choosing meta-cognitive actions in the MCS will be viewable as part of the `SensoryInformation` that was perceived by the agent **AFTER** the agent is finished choosing an external action based upon it. Below is an example (from the "*Full Hello World*" simulation sample[10]) of what this might look like:

```
Activations:
    (Dimension = AffiliationBelongingnessDrive, Value = DRIVE_GAIN), Activation = 1,
    (Dimension = AffiliationBelongingnessDrive, Value = SYSTEM_GAIN), Activation = 1,
    (Dimension = AffiliationBelongingnessDrive, Value = UNIVERSAL_GAIN), Activation = 1,
    (Dimension = AffiliationBelongingnessDrive, Value = STIMULUS), Activation = 0,
    (Dimension = AffiliationBelongingnessDrive, Value = DEFICIT), Activation = 0.5015015005,
    (Dimension = AffiliationBelongingnessDrive, Value = BASELINE), Activation = 0,
    (Dimension = AffiliationBelongingnessDrive, Value = DRIVE_STRENGTH), Activation = 0,
    (Dimension = AutonomyDrive, Value = DRIVE_GAIN), Activation = 1,
    (Dimension = AutonomyDrive, Value = SYSTEM_GAIN), Activation = 1,
    (Dimension = AutonomyDrive, Value = UNIVERSAL_GAIN), Activation = 1,
    (Dimension = AutonomyDrive, Value = STIMULUS), Activation = 1,
    (Dimension = AutonomyDrive, Value = DEFICIT), Activation = 0.4985014995,
    (Dimension = AutonomyDrive, Value = BASELINE), Activation = 0,
    (Dimension = AutonomyDrive, Value = DRIVE_STRENGTH), Activation = 0.4985014995,
    (Dimension = GoalChunk, Value = Bid Farewell), Activation = 1,
    (Dimension = Salutation, Value = Hello), Activation = 0,
    (Dimension = Salutation, Value = Goodbye), Activation = 1
```

You can also turn on logging, which, depending on the level, will provide you with additional details concerning the internal operations of the system (including the motivational subsystem and meta-cognitive modules). The details concerning how

[10] Located in the "*Intermediate*" section of the "*Samples*" folder

to use the logging feature can be found in the "*Using Features*" tutorial (located in the "*Features & Plugins*" section of the "*Tutorials*" folder).

This concludes the tutorial for setting up drive and meta-cognitive modules. At this point, you should have the necessary foundation for using these aspects of the CLARION theory. Note, however, that the example we demonstrated herein was fairly easy, however, setting up the MS and MCS can actually become quite complex. If you find that you need additional information on setting up a particular meta-cognitive module, please consult the API resource document (located in the "*Documentation*" folder). It will provide you with additional details for how to use any of the "pre-packaged" modules that come with the CLARION Library.

Furthermore, we suggest consulting the "*Advanced Customization*" tutorial (located in the "*Customizations*" section of the "*Tutorials*" folder). This tutorial provides details on how to create a custom drive (as well as other types of custom components). Additionally, while it is not included as part of the CLARION Library package, a guide on how to create a custom meta-cognitive module is available upon request. However, be forewarned that implementing a custom meta-cognitive module is a **VERY** advanced (i.e., developer level) undertaking. Therefore, before endeavoring to undertake any developer level (or even advanced level) customizations, you will need to have a thorough and complete understanding of the CLARION theory as well as extensive experience working with the CLARION Library.

Remember, as always, if you run into any problems, have additional questions, want to report a bug, or wish to request the tutorial on implementing a custom meta-cognitive module, you can contact us at clarion.support@gmail.com.