Cozmo Tutorial

James Boggs

February 2019

1 Introduction

The aim of this tutorial is to teach people somewhat familiar with the Soar cognitive architecture how to write a Soar agent to control a Cozmo robot. It assumes that Soar is already installed on your computer and that you have the Soar-Cozmo Interface code ready to be run. Files associated with each part of the tutorial are included in this directory, and should be able to be sourced presently. When reading a lesson, I strongly suggest opening the associated `.soar` file or tutorial folder and keeping it handy so that you can reference it while reading the tutorial. Ideally, by the end of the tutorial, you should understand what the inputs made available to Soar by Cozmo are and what they do, as well aome of the commands Soar can output to Cozmo and what these do. Although the tutorial will not cover every available Cozmo action currently supported, I hope it will cover enough to give you the context and background needed to understand the rest through inspection of the documentation and the interface code. Before getting into examples, we will first go over the higher-level mechanics of how the interface works.

2 How It Works

First, it is important to note that the interface code is purely *reactive* to the Soar kernel, which is considered primary. That means the Soar kernel and its cycle is what drives the interface, rather than any internal loop in the interface itself. All of the functionality of the interface is within various *callback* functions, which are triggered by the Soar kernel, execute their code, and then return control to the kernel. Two callback functions in particular are used to provide the two primary functions of the interface: providing information about the Cozmo and its perceptions on the input link and watching the output link for actions the agent wants Cozmo to perform.

2.1 Input

The input is achieved by a callback triggered when the Soar cycle is about to enter the input phase. When the cycle reaches that point, the interface polls

the Cozmo robot for all the information which needs to be presented to Soar such as its pose, the status of its lift, and any objects or faces it can see. We will cover the details of how this information is presented later. Once it has the appropriate information from Cozmo, the interface updates the agent's working memory elements through the Soar Markup Language, and passes control back to the kernel, which proceeds into the input phase. By updating the working memory elements before the input phase, we ensure that the agent has the latest information available.

2.2 Output

Control of the Cozmo robot is achieved by a callback which listens for changes to the output link, then scans it for new working memory elements with names of actions Cozmo can take. For example, if the output link is initially empty and Soar adds a new identifier augmentation `^move-lift M1` which has an float augmentation `^height 0.5`, the new output link would look like

```
(I3 ^move-lift M1)
  (M1 ^height 0.5)
```

and the output callback would be triggered. It would be given the new output link identifier augmentation `^move-lift M1`, and then see if the attribute name "move-lift" is an action it recognizes. Since it is, it will look at the identifier `M1` and try and find the `^height` attribute. The value of the `^height` attribute is used in a function call to the Cozmo SDK which will move the lift to the specified level, in this case 50% of its maximum height. Once the Cozmo finishes the action, a `^status complete` augmentation is added `M1`, so the final output link looks like

```
(I3 ^move-lift M1)
  (M1 ^height 0.5 ^status complete)
```

Control is then handed back to the Soar kernel, which continues to run its cycle. If more than one new valid action is added to the output link Cozmo will execute all of them before handing back control. The order of execution should be treated as random.

3 Lessons

To ground that rather abstract description of the interface, and to provide examples of the specific inputs and outputs the interface provides and handles, we will go through a series of lessons which incrementally build on each other, ultimately producing a Soar agent which will look around its environment for light cubes and a face, and then bring the "most valuable" light cube to the face.

3.1 Lesson 1: Reset Cozmo's Head and Lift

First, we'll explore how to move Cozmo's head and lift to specific positions using the `move-head` and `move-lift` commands. The `move-head` action sets the angle on Cozmo's head relative to a horizontal plane through its axis of rotation. Thus, if the angle specified is -0.25, Cozmo's head will move so that it forms a -0.25 radian angle with the plane, which ends up having Cozmo look towards the ground. An angle of 0.25 will similarly have Cozmo look upwards. The `move-lift` action moves the lift to a position specified in the command by a real number between 0 and 1. The number indicates the percentage of the lifts maximum height it should move to, so a value of 0 is the lowest possible height for the lift, a value of 1 is the highest, and 0.5 is the lift's midpoint.

We will be using the `move-head` and `move-lift` actions to reset Cozmo's head and lift positions to default ones. Specifically, we want Cozmo to move its head to be parallel with the ground and to lower its lift as far as it can. Together, these actions will help Cozmo see better, since Cozmo often starts with its tilted down, and the lift can occasionally block Cozmo's camera. In order to make sure Cozmo always resets when the Soar agent starts, we will make a slight addition to the usual initialization production for a Soar agent. The initialization proposal rule is a cookie-cutter proposal which just proposes the `initialize-cozmo` operator. The application rule checks for the presence of this operator, and then has two parts. The first is fairly standard: it adds a `^name` string augmentation to the top state and sets its value to "cozmo". The second is what resets the positions of Cozmo's lift and head:

```
(<out>  ^move-head.angle 0.0
        ^move-lift.height 0.0)
```

This part of the right hand side (RHS) adds two working memory elements to the output link, `^move-head` and `move-lift`, and gives them each an augmentation, `^angle` for `^move-head` and `^height` for `move-lift`, which are set to 0.0. Recall that the interface listens for new additions to the output link. This means that when this rule fires, the interface will pause the kernel to look for valid actions, which both new WMEs are. The interface will be looking for an `^angle` attribute on the `^move-head` identifier and a `^height` attribute on the `^move-lift` identifier. Since both are present and supply valid values, the interface will execute the specified actions in the Cozmo robot.

3.2 Lesson 2: Saving the Origin Pose

The first input we will be looking at will be Cozmo's pose information, which is always placed by the interface on the agent's input link and will look similar to:

```
(I2 ^pose P1)
    (P1 ^rot 2.733525 ^x 31.371500 ^y 1.045640 ^z 0.000000)
```

The pose identifier will have four attributes with floating point values, ^rot, ^x, ^y, and ^z, indicating Cozmo's rotation on the z (vertical) axis in radians and its position on the x-y plane from the origin in millimeters. Although these values are just estimations based on Cozmo's internal sensors, they are nevertheless useful in keeping track of where Cozmo is. Right now, we are going to make sure that Cozmo also keeps track of where it started by saving Cozmo's initial pose when it starts up. This will involve modifying both the proposal and the application rule we touched on in the last lesson.

First, we need to modify the proposal rule so that the initialize-cozmo operator has the initial pose information. In the left hand side (LHS), add a new condition ^io.input-link.pose <p>, like so:

```
(state <s>  ^superstate nil
           -^name
            ^io.input-link.pose <p>)
```

Then add a few more conditions based on augmentations of <p> to get the actual values of the pose:

```
(<p> ^rot <r-val>
     ^x <x-val>
     ^y <y-val>
     ^z <z-val>)
```

This will look at the pose information on the agent's input link and store the values in the variables <r-val>, <x-val>, <y-val>, and <z-val>, so that we can attach them to the operator.

On the RHS, add a new attribute ^origin to the operator <op> which has a new identifier as its value, so it looks like:

```
(<op>   ^name initialize-cozmo
        ^origin <ogn>)
```

and then expand <ogn>, adding ^rot, ^x, ^y, and ^z augmentations, giving each as their value the matching variable from the LHS:

```
(<ogn>  ^rot <rot>
        ^x <x-val>
        ^y <y-val>
        ^z <z-val>)
```

Now the operator has the pose information we want to store, which means the application rule an pull directly from the operator. Now we need to modify the application rule so that it actually stores the origin pose on the top state. First, modify its LHS a bit by looking for an ^origin <ogn> augmentation on the operator and, like above, get the relevant augmentations from <ogn> so that the agent stores the rotation and location information in the variables <r-val>, <x-val>, <y-val>, and <z-val>. On the RHS, create an ^origin <origin> augmentation on the top state <s>, like so:

```
(<s>    ^name cozmo
        ^origin <origin>)
```

Note that the name of the new identifier (i.e., <origin>) must be different than the variable used to match the operator's augmentation (i.e., <ogn>). Then add the necessary augmentations like before by adding a new effect:

```
(<origin>   ^rot <rot>
            ^x <x-val>
            ^y <y-val>
            ^z <z-val>)
```

Note that <origin> is an augmentation to the top state, rather than the input link, because it is not an input but a memory. Additionally, because we check for an operator in the LHS, the <origin> WME is *o-supported*, meaning it will persist even after the rule which added it (apply*initialize-cozmo) is retracted. Thus, we have a permanent record of where Cozmo started.

3.3 Lesson 3: Remembering Objects and Faces

Now that Cozmo moves its face and lift so that it can see, and remembers where it started, we need to begin giving it some capabilities for observing the world around it. In order to keep the tutorial files small, from here on out we'll be working on a single Soar project, contained in the cozmo/ folder. If you want to work through the this tutorial, I suggest similarly creating a folder to house multiple soar files and to copy and paste the elaborations/ sub-folder and the cozmo_source.soar file to it. These provide some basic functionality like elaborating the top state and giving you a single soar file to run. Specifically, you can run the tutorial agent by using cozmo_source.soar as your agent file. For this lesson, look at the file cozmo/remember.soar.

Before Cozmo can bring you the most valuable light cube it sees, it must not only find light cubes and your face but be able to remember where those things are. First, we will be giving Cozmo the ability to remember details like the location, name, and id of objects and faces it sees, even after they leave its pereption. The core of this process is watching for new objects or faces, copying the various augmentations attached to these, and then storing them on the top state as o-supported WMEs for later access. Additionally, we want Cozmo to be able to update its old knowledge if anything about the object (e.g., it's locaton) changes, so we need Cozmo to be comparing objects in its perception with those it remembers and be able to update the information in its memory if needed.

3.3.1 Remembering New Cubes

For this task, the only kinds of objects we need to remember are light cubes, which is helpful because light cubes have some unique pieces of information which we want to capture, and we can do so without needing to separately check for plain objects. The first thing we need to do is have the Soar agent

propose an operator remembering a cube which it hasn't seen before, and attach whatever information is pertinent to that operator. We will call this rule `cozmo*propose*remember-new-cube`, and it should look for an `^object` <obj> identifier augmentation on the input-link with certain augmentations of its own, and should also make sure that the object seen isn't one we already remember. The entire LHS of the proposal should look like:

```
(state <s> ^name cozmo
           ^io.input-link.object <obj>)
   (<obj> ^type cube
          ^cube-id <cid>
          ^object-id <oid>
          ^pose <p>)
   (<p> ^rot <r-val>
        ^x <x-val>
        ^y <y-val>
        ^z <z-val>)
  -(<s> ^cube.cube-id <cid>)
```

The `^type cube` augmentation we look for on the <obj> identifier ensures that the object in question is indeed a light cube. This isn't strictly necessary, since only light cubes are given a `^cube-id` attribute, but I want to introduce the presence of that augment as a simple test for cubeness. The `^cube-id` <cid> condition captures the cube's id in the <cid> variable; each light cube has a unique ID which is tied to the image on the cube itself. This means that we are guaranteed that any object on the input link with an augmentation matching `^cube-id` <cid> is guaranteed to be this particular cube. <oid> stores the cube's object id, which is a temporary internal id given to the object by Cozmo. An object's object id will remain the same as long as Cozmo senses the object, and no two objects will have the same object id simultaneously. We need to keep track of the cube's object id because it is needed for certain actions such as picking up the cube.

Finally, we have the pose data, stored in <p>. <p> itself is an identifier with four attributes: `^rot`, `^x`, `^y`, and `^z`, which have as their values the rotation of the cube on the z-axis (in rads) and the distance (in mm) from the origin along the x-, y-, and z- axes respectively. This pattern will be see very often when working with Cozmo, so it's worth exploring a bit more. Anything which Cozmo can estimate the location of (which primarily means objects, faces, and Cozmo itself) has pose information which is included by the Soar-Cozmo interface on the input link. The pose information is based on an internal grid Cozmo maintains, which has as its origin an apparently arbitrary location chosen on start-up. Pose information is always presented in a `^pose` attribute on the thing in question, which connects to an identifier WME. This identifier then has the four pieces of information outlined above. Later on, we will create elaborations which add the angle from Cozmo's heading to the object's location and the distance from Cozmo to the object, but this information is not provided by the interface. In

the above Soar code, we use <r-val> and the rest to capture this information, and use these names, rather than the names <rot>, <x>, and so on to avoid confusion.

The last condition checks that there isn't already a ^cube attribute on the top state with the cube id of the cube we've matched on the input link. Now that we have this information, particularly <cid>, we need check the top state to make sure we don't already remember a block with the same cube id as the one we have found. If we do, we shouldn't be remembering it for the first time (since that would just add a new WME), but updating what we have. Supposing that we find a match, we now need to create an operator and propose it. The operator should include all the information we pulled from the observed cube: the cube id, the object id, and the pose id. As such, the RHS should look like:

```
(<s> ^operator <op> + =)
(<op> ^name remember-new-cube
      ^cube-id <cid>
      ^object-id <oid>
      ^pose <pose>)
(<pose> ^rot <r-val>
    ^x <x-val>
    ^y <y-val>
    ^z <z-val>)
```

As can be seen, this just creates an operator which has the attributes, including ^pose, which we extracted from the perceived cube. Once this rule fires, we need to apply the operator, which will store the cube's information in working memory. Unsurprisingly, the application rule simply looks for the operator and collects the information it carries then creates a ^cube attribute on the top state which holds an identifier carrying all of this information. The entire application rule is

```
sp {apply*remember-new-cube
   "If a cube with this id not seen yet, add to wm"
   (state <s> ^operator <op>)
   (<op> ^name remember-new-cube
         ^cube-id <cid>
         ^object-id <oid>
         ^pose <p>)
   (<p> ^rot <r-val>
        ^x <x-val>
        ^y <y-val>
        ^z <z-val>)
-->
   (<s> ^cube <c>)
   (<b> ^cube-id <cid>
        ^object-id <oid>
        ^pose <pose>)
```

```
   (<pose> ^rot <r-val>
        ^x <x-val>
        ^y <y-val>
        ^z <z-val>)
}
```

3.3.2 Updating Known Cubes

Just memorizing all the information about a cube the first time we see it isn't enough, since cubes may move around. Cozmo also needs to be checking whether it sees a cube it already memorized in a different place than expected, and merely update the existing `^cube` attribute if it does. Since we still need all the information about the cube, the LHS will look very similar to the LHS of the initial remembering rule, but instead of looking for the *absence* of a remembered cube with the same cube id, we want to look for its *presence*. Thus, we want to copy and then slightly change `cozmo*propose*remember-new-cube` into a new rule `cozmo*propose*update-cube-knowledge`. Just like before, we want to search the input link for an object with the relevant attributes. However, we also want to match a `^cube` attribute on the top state which has a cube id equal to the cube id of the object on the input link to ensure we actually know about this cube. Thus, the new rule looks like

```
sp {cozmo*propose*update-cube-knowledge
   "Propose an operator to update cube information"
   (state <s> ^name cozmo
              ^io.input-link.object <obj>
              ^cube <c>)
   (<obj> ^type cube
          ^cube-id <cid>
          ^object-id <oid>
          ^pose <p>)
   (<p> ^rot <r-val>
        ^x <x-val>
        ^y <y-val>
        ^z <z-val>)
   (<c> ^cube-id <cid>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name update-cube-knowledge
         ^cube-id <cid>
         ^object-id <oid>
         ^pose <pose>)
   (<pose> ^rot <r-val>
        ^x <x-val>
        ^y <y-val>
        ^z <z-val>)
```

```
}
```

The application rule for this operator has an similar LHS to the application rule for `remember-new-cube`, since the `update-cube-knowledge` operator has the same information as the `remember-new-cube` operator. However, now we also need to find and capture in a variable (<c>) the `^cube` WME on the top state which matches the cube's cube id, so that we can alter it. This means making two changes. First, add a search for a `^cube` augmentation to the top state:

```
(state <s> ^operator <op>
           ^cube <c>)
```

Note that the variable <c> captures the relevant identifier on the top state. This allows us to remove it correctly later on, so we can replace it with more up-to-date information. We also need to check to make sure that the cube <c> has the proper cube id:

```
(<c> ^cube-id <cid>)
```

Since this condition needs the variable <cid> to exist, it must come after the condition which captures the cube id stored in the operator.

The RHS is also quite similar to the RHS of the remembering production, because we are essentially doing the same thing, but just removing the old augmentation first. We need to delete and replace the WME because Soar does not allow direct manipulation of WMEs. Thus, we can essentially copy the RHS of the `apply*remember-new-cube` production and add an additional effect before the other effects:

```
-(<s> ^cube <c>)
```

3.3.3 Prioritization of Remembering Cubes

As of now, Cozmo can both remember and update its memory of cubes and their locations. However, since the operators are prioritized equally Cozmo will choose between them at random if both `remember-new-cube` and `update-cube-knowledge` are proposed. Although this will usually be fine in the long run, we really want Cozmo to prefer remembering new cubes over updating its memory of old ones. In fact, there are very few proposals we might want to prioritize equally or more than remembering a newly-seen cube, since that is both fast and often quite important. To ensure remembering a cube always has priority, we will introduce a very simple operator comparison. The comparison will check whether two operators are both `remember-new-cube` operators and, if not, prioritize the remembering operator over the other operator. This means that `remember-new-cube` operators will *always* be chosen first. The comparison production will look fairly standard:

```
sp {cozmo*compare*remember-cube
   "Make sure remembering new cubes has highest priority"
   (state <s> ^name cube-stack
              ^operator <op1> +
                        <op2> +)
   (<op1> ^name remember-new-cube)
   (<op2> ^name {<> remember-new-cube })
-->
   (<s> ^operator <op1> > <op2>)
}
```

The critical condition is the final one, which only matches if <op2> isn't a remember-new-cube operator.