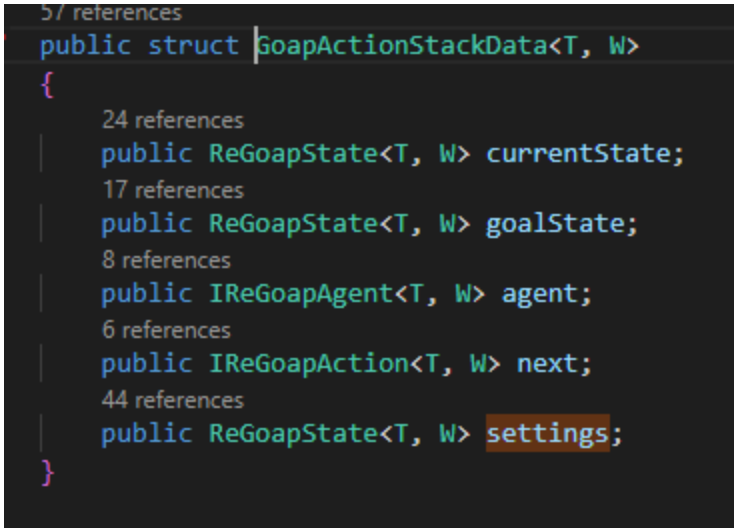
Actions & stackData

1. First thing to do when defining actions is implement how the GetSettings() returns a stackData.
2. Based on the stackData provided by the GetSettings(), calculate how the preconditions given the resulting stackData, will be defined to be added to the goal state of another node.
3. From the resulting stackData from GetSetting(), define what key from the stackData the action needs for the action's effects to be applied. Define what conditions and values are applied in the effects.
4. In the Run method, override it and define it. Run also relies on the settings defined by GetSettings().
5. For every action have a private List<ReGoapState<string, object>> settingsList.

Also:

1. IsInterruptable() by default is spelt wrong and returns true. This is called inside TrWarnActionFailure(), and if the action is not interruptible, then call action.AskForInterruption() which sets interruptWhenPossible in the ReGoapAction class is true.
2. TrWarnActionFailure() inside Agent doesn't get called anywhere in the architecture or in the example.
3. Inside WarnPossibleGoal() which is called by the Update function inside ReGoapGoalAdvanced. If there is a currentActionState and if the action isn't interruptible in which Action.IsInterruptable() returns false., then we set interruptOnNextTransition to true, and Action.AskForInterruption(), so that the action's interruptWhenPossible to true.
4. During the next PushAction(), interruptOnNextTransition if true , we will calculate a new goal and subsequent plan.

Actions can be very versatile in the ReGoap architecture in which a single action could produce multiple child nodes. Action States could have the same actions, but different settings. This is possible with the GoapActionStackData<T, W> class commonly associated with the variable stackData.



When defining actions, in order to further define the method CheckProceduralCondition in which the planner uses in the Expand() function in the Node class, the settings of that stack data must be modified.
Inside the Expand() function in Node,

Inside Init(planner, newGoal, parent, action, settings):
This method focuses on creating a node with the given action and its settings. Based on these settings the associated preconditions are retrieved and effects, as well as the cost. With this the retrieved state from the parent is added with these retrieved effects and goal's conditions are fulfilled with these effects and preconditions are added too.

GetPreconditions(stackData) and GetEffects(stackData) both take a stackData of type GoapActionStackData<T, W> which holds the following information:

- currentState = parent.GetState().Clone(); // state of the parent action node at that point
- goalState = newGoal; // the unfulfilled conditions of the parent node
- Next = action; // the next action in the plan which is the parent node's action
- agent = planner.GetCurrentAgent(); // the agent that the planner is working with
- Settings = settings; // clone of the parent's action's settings

GetCost(stackData) also depends on this stackData at this particular point.

Inside the Expand():
Focuses on whether an action can be expanded into a node based on its multiple settings which could result with different preconditions, effects, and, or procedural conditions.
The Expand() function returns a list of INode<ReGoapState<T, W>>
There is a different GoapActionStackData<T, W> which holds the following information:

- currentState = state;
- goalState = Goal;
- agent = agent;

- settings = null;

This is the stackData of the current node (this node) that is being expanded upon.

With this, every possible action will run PreCalculations(stackData) to reference the same agent as the node being expanded. The settingsList variable acquires the resulting value of type List<ReGoapState<T, W>> with the call GetSettings(stackData) which can also initialize local ReGoapState<T, W> settings in that action class which will be used by Run().

For each settings in settingsList

The stackData's settings will equal the setting in the list, and the calls GetPreconditions(stackData) and GetEffects(stackData) are made again, so that the checks of whether the effects fulfill the expanded node's goal state, no conflicting effects or preconditions and CheckProceduralCondition(stackData) returns true.

This is very important, why there is even a list of ReGoapStates is because there can be multiple preconditions and effects to check based on the different settings, and different procedural conditions that need to be checked too for different settings.

With these different settings returned by GetSettings from an Action, two or more different nodes could be made. So a single action could result with multiple child nodes when expanding a parent! This can make a single action very versatile, and have different preconditions and effects based on different settings.

Goals

In the Agent class there is a public bool called BlackListGoalOnFailure which if ticked, any action that failed to properly run on failure, the call WarnActionFailure() will blacklist the current goal if the public bool field is ticked. So when calculating a new goal, the current goal will be considered to be blacklisted even if the goal were to properly finish, so the same goal won't be consecutive.

Blacklisted goals can be considered again for planning when updating possible goals depending if they surpassed the calculation delay threshold which is by default 0.5f. The Agent will only pass non-blacklisted goals, excluding the current goal which if BlackListOnFailure is ticked, then the planner will consider the current goal blacklisted too.

Inside the planner, it gets the set of possible goals and emits the current goal if blacklisted. Next it'll filter whether the goal is possible with the call IsGoalPossible(). Next the filtered out possible goals are sorted with the goals of lowest priority at the front (last to come out) and the highest priority at the front (first to come out).

Goal has a Run method which the agent will run whenever calling to OnDonePlanning(). This could be a final check to verify whether the goal is still viable. By default this is empty and the example doesn't use it.

The most important method to consider: IsGoalPossible().