**Role & Strategy Tree Overview**

Role & Strategy trees are two distinct concept within the RTT codebase. A Role tree is reponsible for controlling the behavior of one specific robot. It does this based on the various types of leafs it contains, as well as the contents of it’s public blackboard and it’s per-node private blackboards. At the leafs of a role tree you can find various skills and conditions; the Skills control the robot assigned to the tree, and the conditions succeed or fail depending on the state of the world. A Strategy tree is responsible for directing the entire team, be it 6 robots or two robots. At the leafs of a strategy tree one finds Plays and Conditions. A play is responsible for direction 0 or more robots. A condition works as described earlier.

There are several ways of passing information to a tree, which all work through the interface of a blackboard. These are the public blackboard within a tree, scoped variables within the public blackboard, and the private blackboard of each leaf. In practice these three manners of passing information around are only used by role trees; it’s however also possible to do this in strategy trees, but we haven’t found a lot of cases where it is explicitly needed.

**Blackboard**

A blackboard is simply a categorized map (or a fixed multi-type map) from strings to a bunch of types. The types we use most frequently are:

* Doubles
* Ints
* Strings
* Bools

For each type you can add and remove entries, i.e. you can add a double named “goal” with the value 3.5. The reasons we use blackboards are the following:

* It’s a common thing to do in the behavior tree literature
* The Behavior Tree library we adopted shipped with a blackboard class.

The blackboard class included with the behaviortree library also has space for floats.We decided to only use doubles for greater precision, and to avoid confusion.

**Public blackboard**

Each tree has a public blackboard. This is a blackboard that is accessible to all the leafs, i.e. each leaf can query this blackboard for information. It stores both global data (data meant for all the leafs) and scoped data (data meant for specific leafs).

**Global data**

At the time of writing we only have two pieces of information that always reside in the public blackboard. These are

* ROBOT\_ID: The ID of the robot this tree belongs to
* KEEPER\_ID: The ID of the current keeper. Can possibly be identical to ROBOT\_ID

These can be used liberally by skills and conditions to control the robot and query the world. These two variables are only present in the case of Role trees; at the time of writing strategy trees have no predefined “global” variables.

The global data can be accessed by querying the blackboard variable within a Skill or Condition.

**Scoped data**

We needed a way to pass data down the behavior tree. For example, let’s say you have a behavior tree with a GoToPos node called GoToPos\_A and a Kick node called Kick\_A. We pass data to the nodes by putting it in the public blackboard, which usually works fine. But what if GoToPos and Kick both share a piece of information that’s named identically, e.g. speed? If we do this:

// Set the speed for the kicker

blackboard->SetInt(“speed”, 5);

// Set the speed for the driving

blackboard->SetInt(“speed”, 60);

The program will definitly break, since you overwrite the speed value with the second statement. To fix this we introduced scoped variables. This means that if you want to pass a specific piece of information to a specific node, you prefix the name of the node to the name of the variable, separated by an underscore. In our example, the blackboard statements would become:

// Set the speed for the kicker

blackboard->SetInt(“Kick\_A\_speed”, 5);

// Set the speed for the driving

blackboard->SetInt(“GoToPos\_A\_speed”, 60);

This way each node can check if a piece of data meant for that node only is present or not. The class ScopedBB in roboteam\_tactics automates this process by allowing the user to enter the node name only once, and prefixing the name to all subsequent entered variables. Example:

ScopedBB(bb, “GoToPos\_A”)

.setInt(“speed”, 60)

.setDouble(“xGoal”, 3)

.setDouble(“yGoal”, 2)

.setBool(“passOn”, true)

;

Scoped data can be accessed by manually prefixing the node name to the parameter and then looking up the parameter in the public blackboard, or by using the family of functions (GetBool, GetDouble, etc.) when within a Skill or Condition class.

**Private Blackboard**

When talking about our behavior trees we also have the concept of “private blackboards”. When talking about a private blackboard people usually mean a blackboard that belongs to a specific node. In our earlier example of the behavior tree with a GoToPos node and a Kick node there are two private blackboards present: the one from the GoToPos node and one from a Kick node.

Private blackboards are meant to be just that: private. It’s contents are only meant to be accessed by the node (i.e. the respective GoToPos or Kick node) that owns it. They can be extracted and edited by accessor functions, but this is usually only done by the tree generation module. Sometimes private blackboards are also used to pass information to nodes that are class members of the owning node. For example, at some point GetBall had a class member instance of GoToPos, and GetBall used it’s private blackboard to pass information to GoToPos. This is done in some cases, but not in all, so do not rely on it.

We use private blackboards to fill in node-specific parameters via the behavior3 editor. When you select a node in the behavior3 editor on the right you can fill in values that should be passed directly to the node. If you enter a value there, they get put in a node’s private blackboard at compile time. In a way this feature allows you to construct behavior trees without writing code that creates the needed parameters and puts them in a scoped entry. It can also be used to document a behavior tree, i.e. if an AimAt node has a parameter that says “aimAt: ourgoal”, then it’s clear that the AimAt node aims at the goal, which illustrates the context in which the node is executed.

The private blackboard can be accessed through the private\_bb while within a skill or condition class.

**Synchronisation**

Currently we provide only one mechanism to synchronize between role trees. We do this with the ParamCheck condition and the ParamSet skill. These nodes check and set ros parameters respectively. They can be used to “signal” from one robot to another that a certain task has been completed, or that a robot is ready for the ball.

The upside is that this is a fast, simple method that’s easy to understand and implement. The downside is that it does not allow multiple names, i.e. one parameter can only be used for one purpose. If two plays run side-by-side and use the same parameter, they will inevitably break.

We have been thinking about a more extensive/detailed way of communication/synchronization, namely via a “Global Blackboard”. This would be an imaginary blackboard that would set values per robot which avoids name clashes. It would also reduce the number of calls each robot does to the ros master, therefore being more efficient. See Software/Onderzoek/Getball lock brainstorm.gdoc on the roboteam twente drive for more info.

**Mem-Principle**

First things first: you can turn off mem-principle checking in the roboteam\_tactics’ CMakeLists.txt: just set ENABLE\_MEM\_PRINCIPLE\_CHECKING to FALSE.

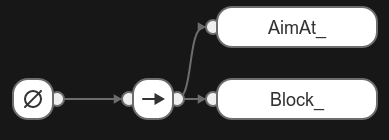
When you have a sequence (not a mem-sequence!) which has two skills/actions (nodes that can return running), what happens when the AimAt returns success and Block returns running (see picture below)? The node will be paused, and the following frame updated again. Now this is where the problem happens: instead of running the action which returned running the previous frame, Block, it runs AimAt (the one that returned success last frame)! Normally this is not a problem, but once the actions start allocating resources (e.g. Plays do this when they allocate players or the keeper via RobotDealer!) this might bite you.

Therefore, to adhere to the mem principle, every tree must ensure that:

Every (non-mem) sequence or priority has either:

- only conditions

- or only conditions, with exception of the last child, which is allowed to return running.



Following these rules ensures that if there’s a sequence or priority that has a mem-node (a node such as block or aimat) as child, it will be guaranteed to be executed next frame, provided that every condition preceding it succeeds. If not every condition succeeds the non-mem node will be terminated, and the sequence or priority node will be exited cleanly.