Behavior Tree in YARP

Current status and future work

CARVE project

This work originated from CARVE project.

The goal was to achieve formal verification of code execution by mean of behavior trees

The big picture

Groot GUI

Runtime display

BT Engine

Generates

Reads from

Receives signals for Monitoring via YARP ports

Glue Code

BlackBoard

Monitors

BT description

Coordinates modules via YARP ports

YARP execs

Behavior Tree cycle

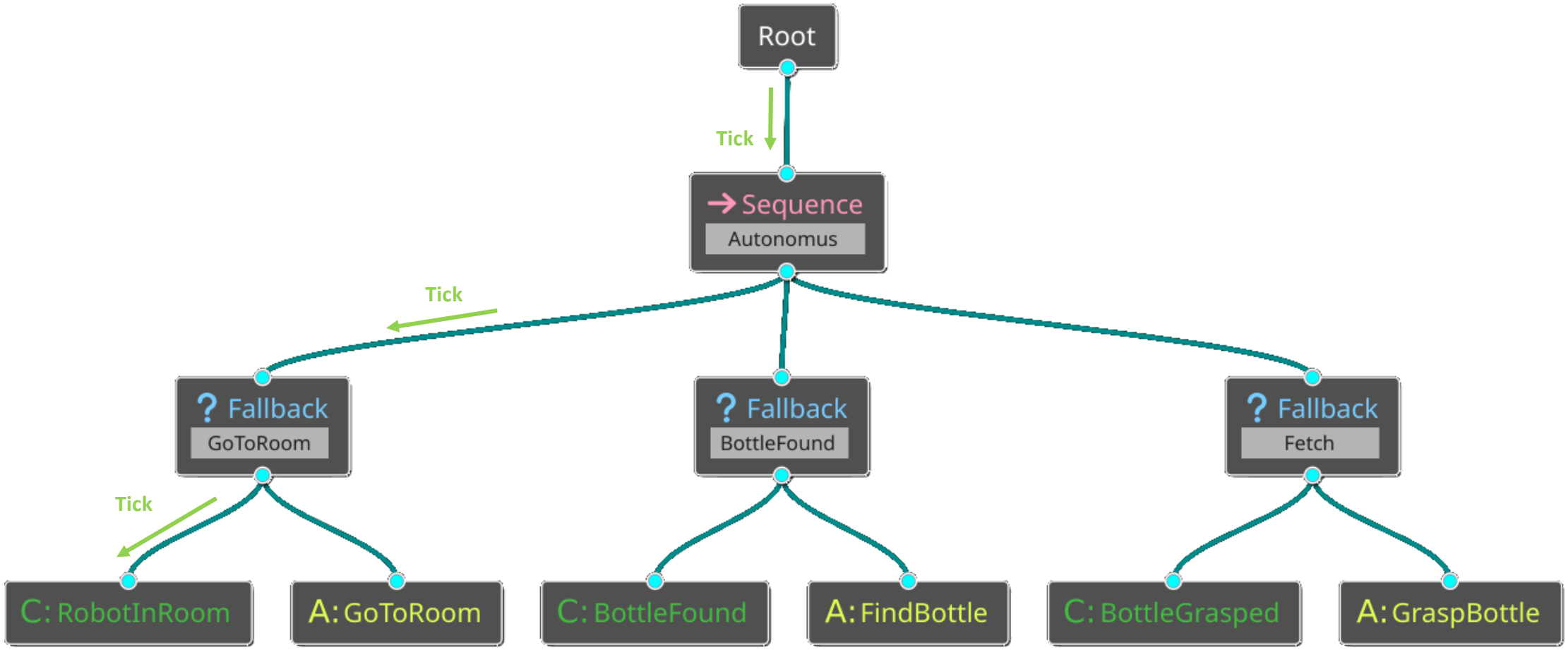
The engine periodically scans all the behavior tree and will return to the root node with a final result.

Each node can return a single value from the list

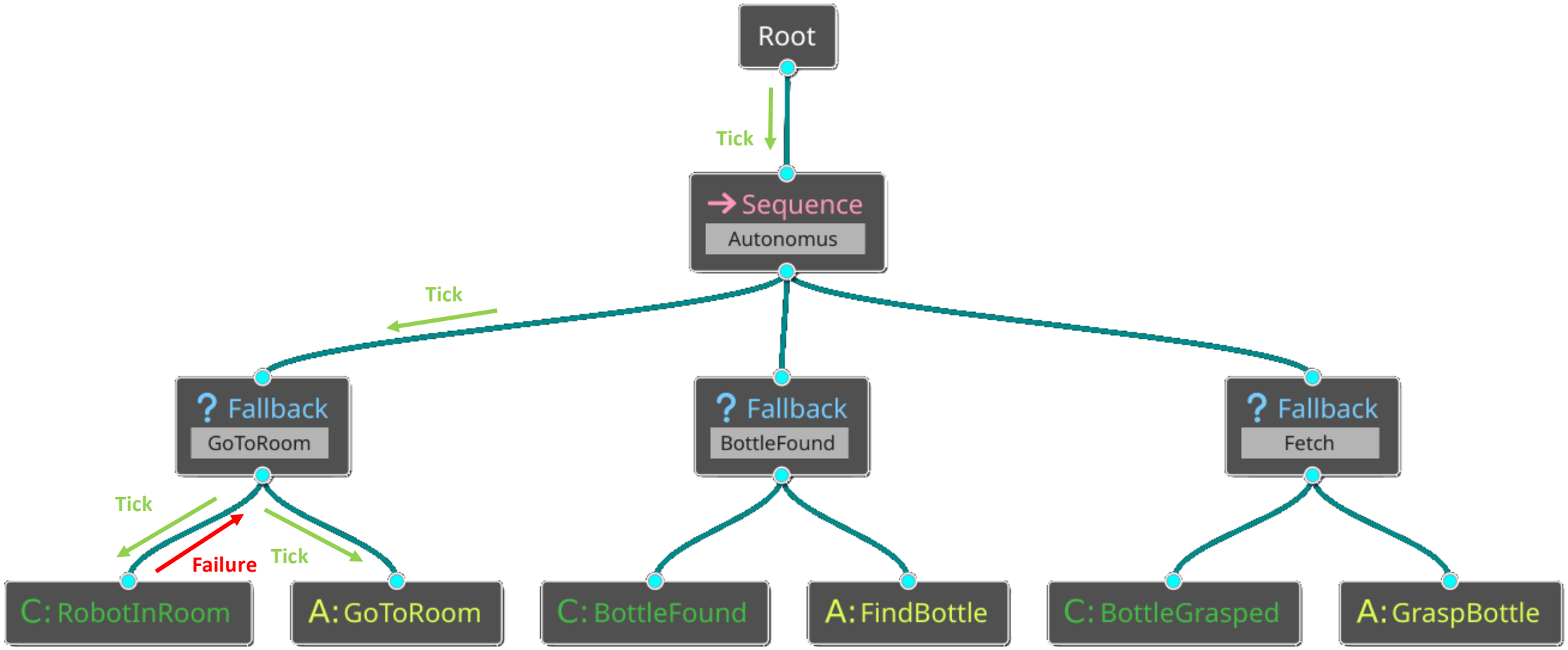
Running / Success / Failure

(N.d.a. Originally it was only Success / Failure)

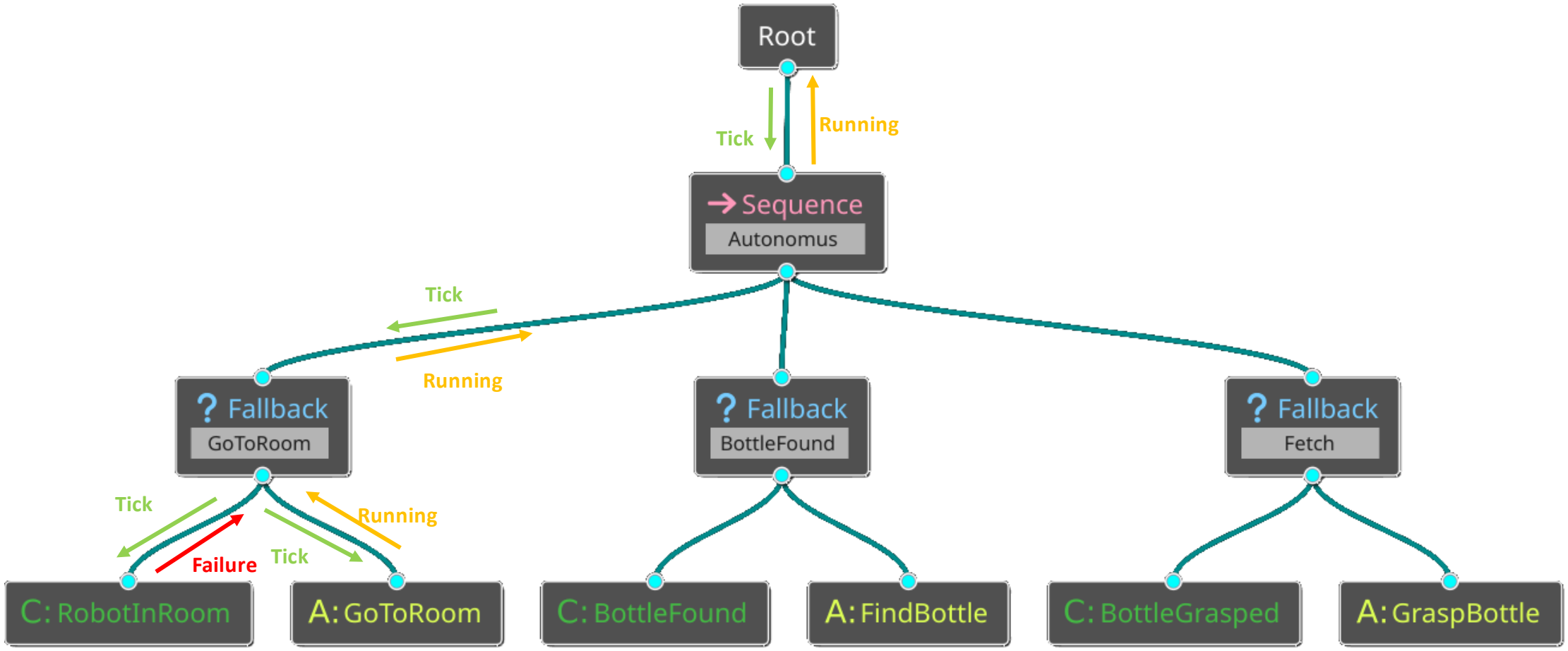
Behavor Tree in Robotics



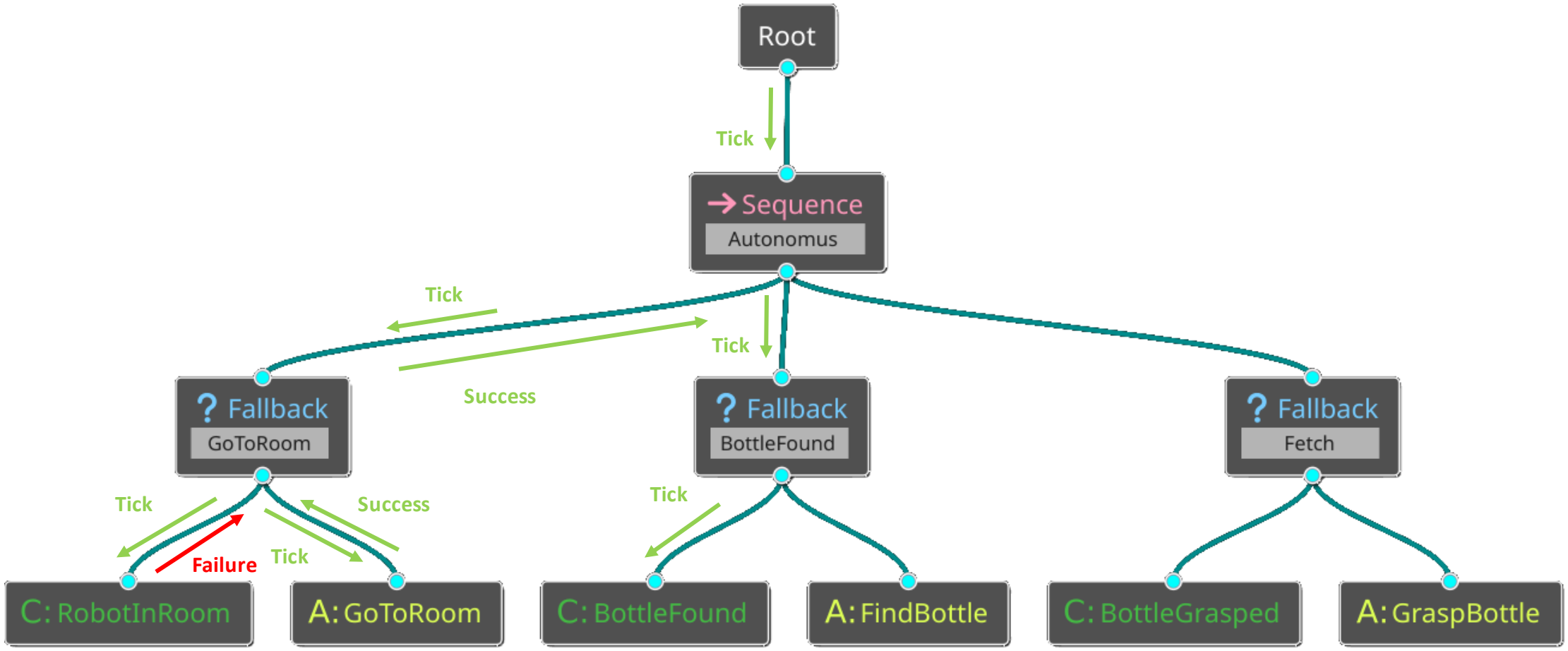
Behavor Tree in Robotics



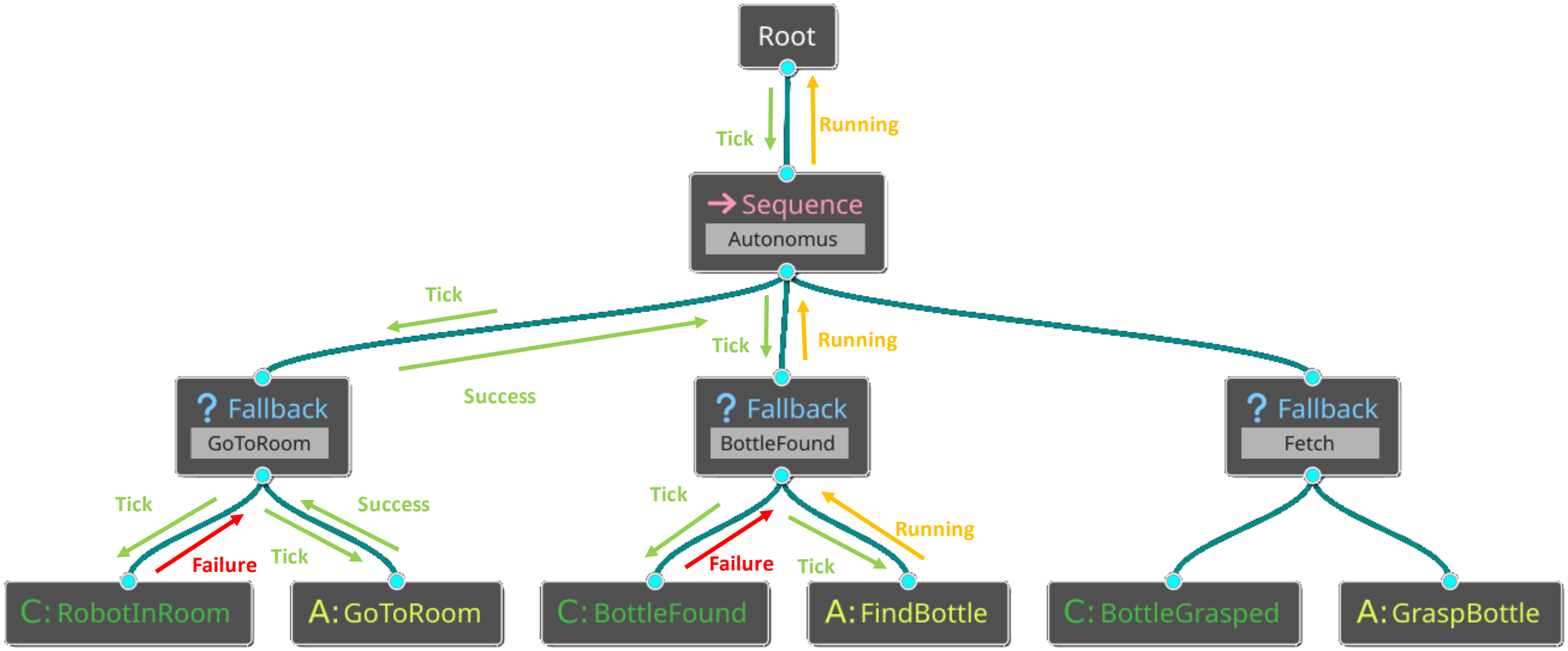
Behavor Tree in Robotics



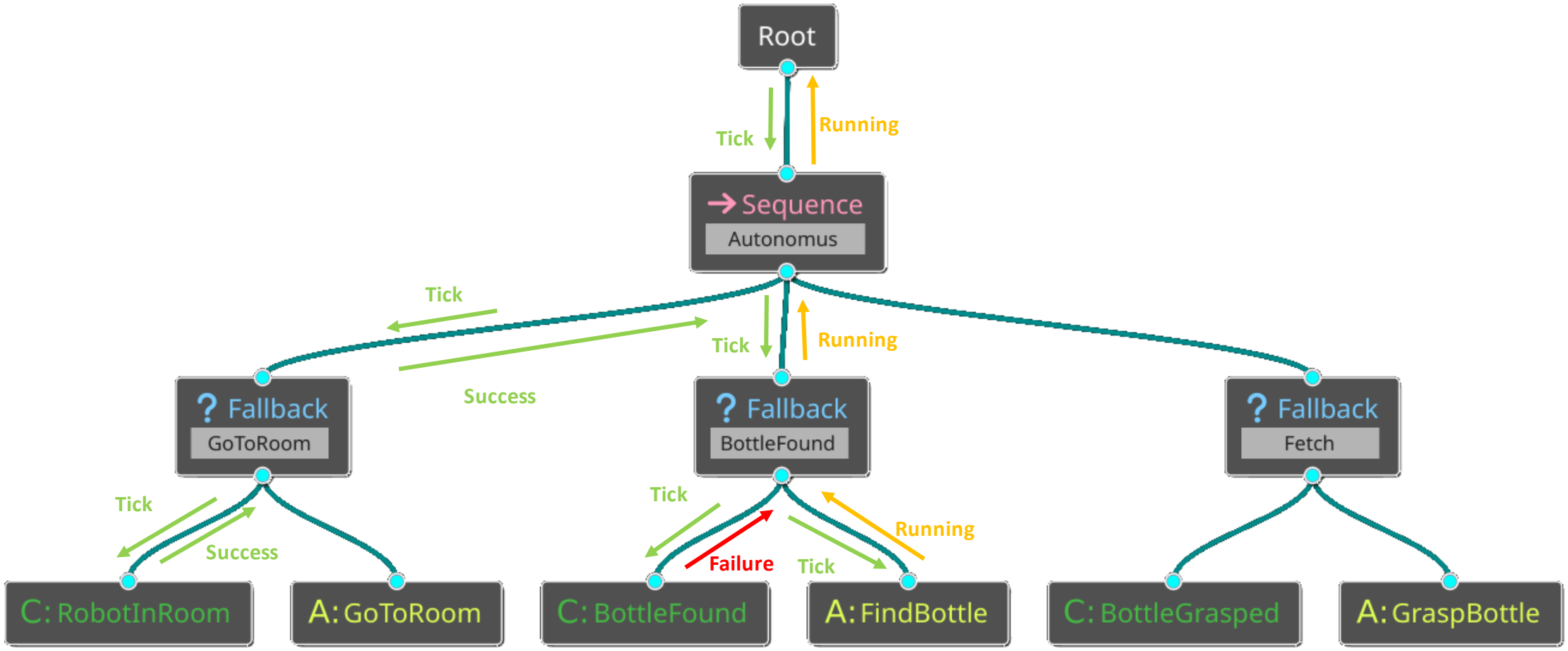
Behavor Tree in Robotics



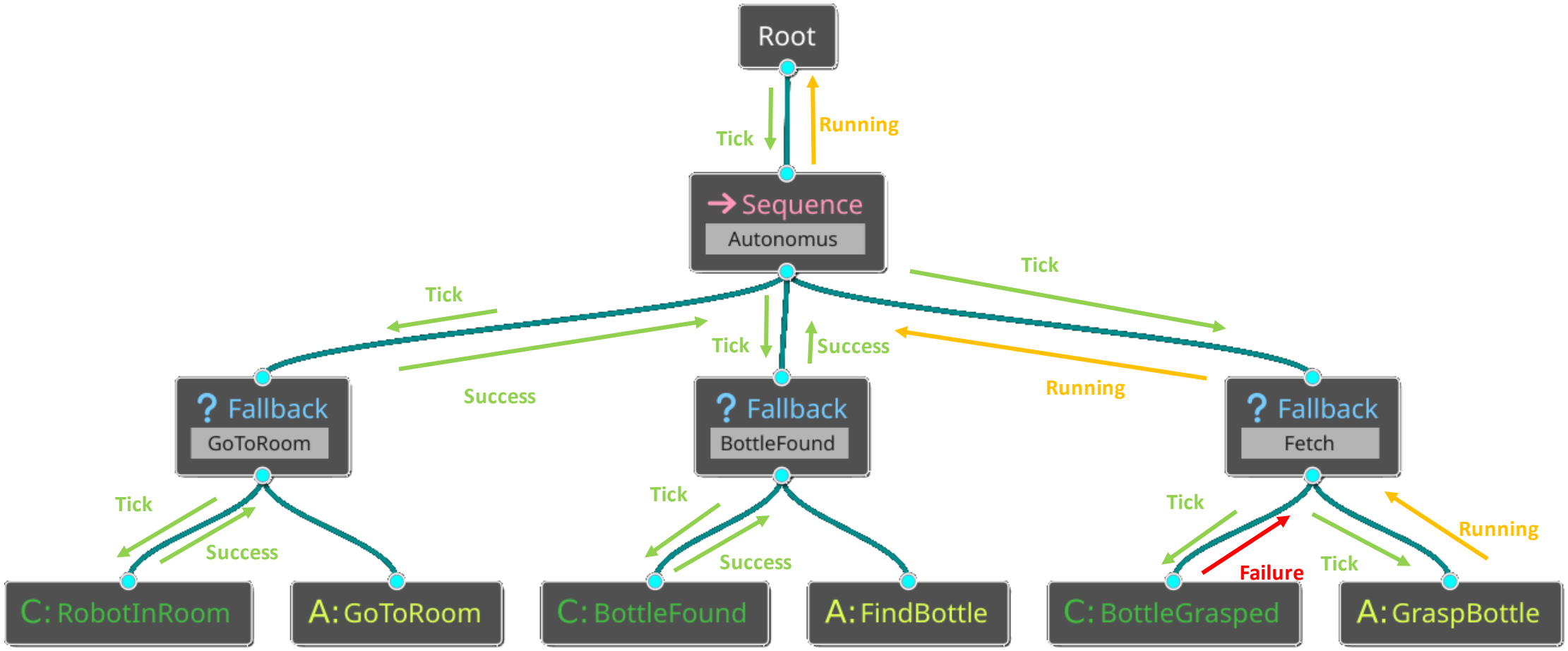
Behavor Tree in Robotics



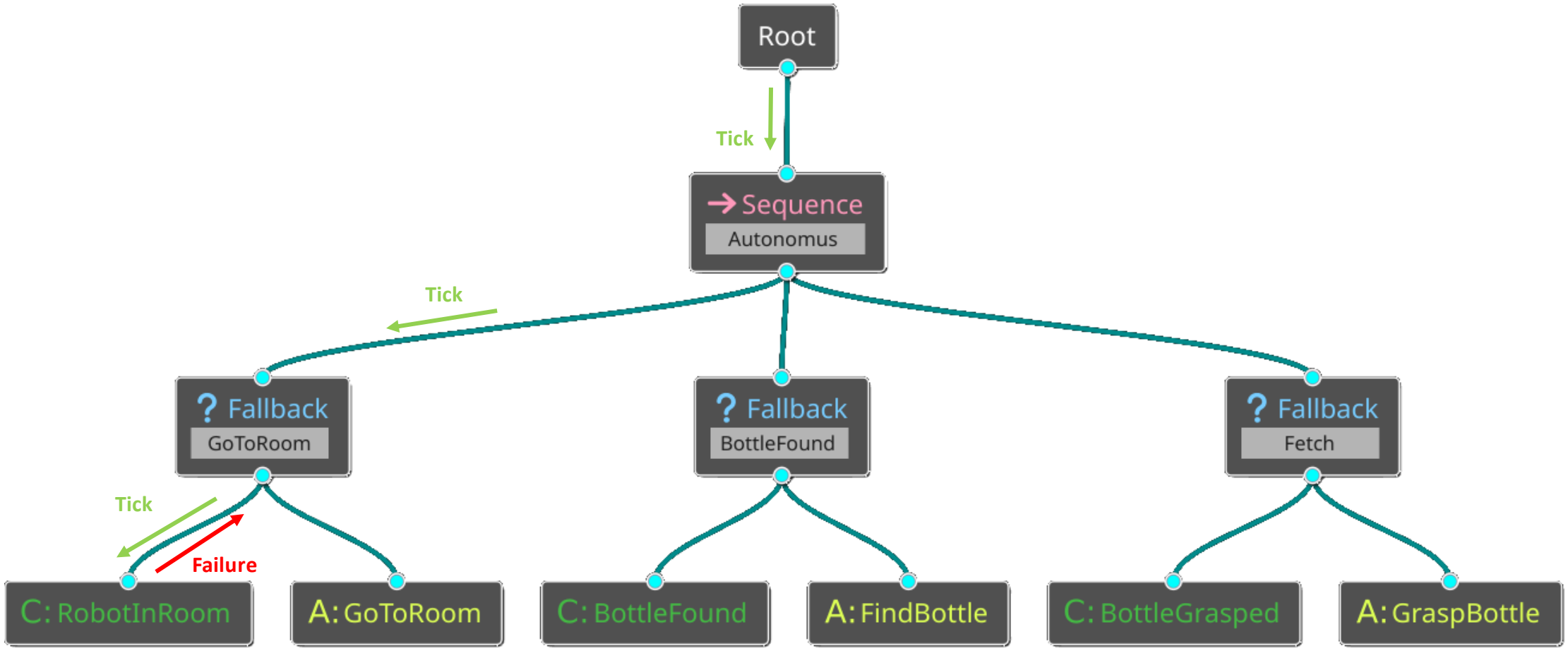
Behavor Tree in Robotics



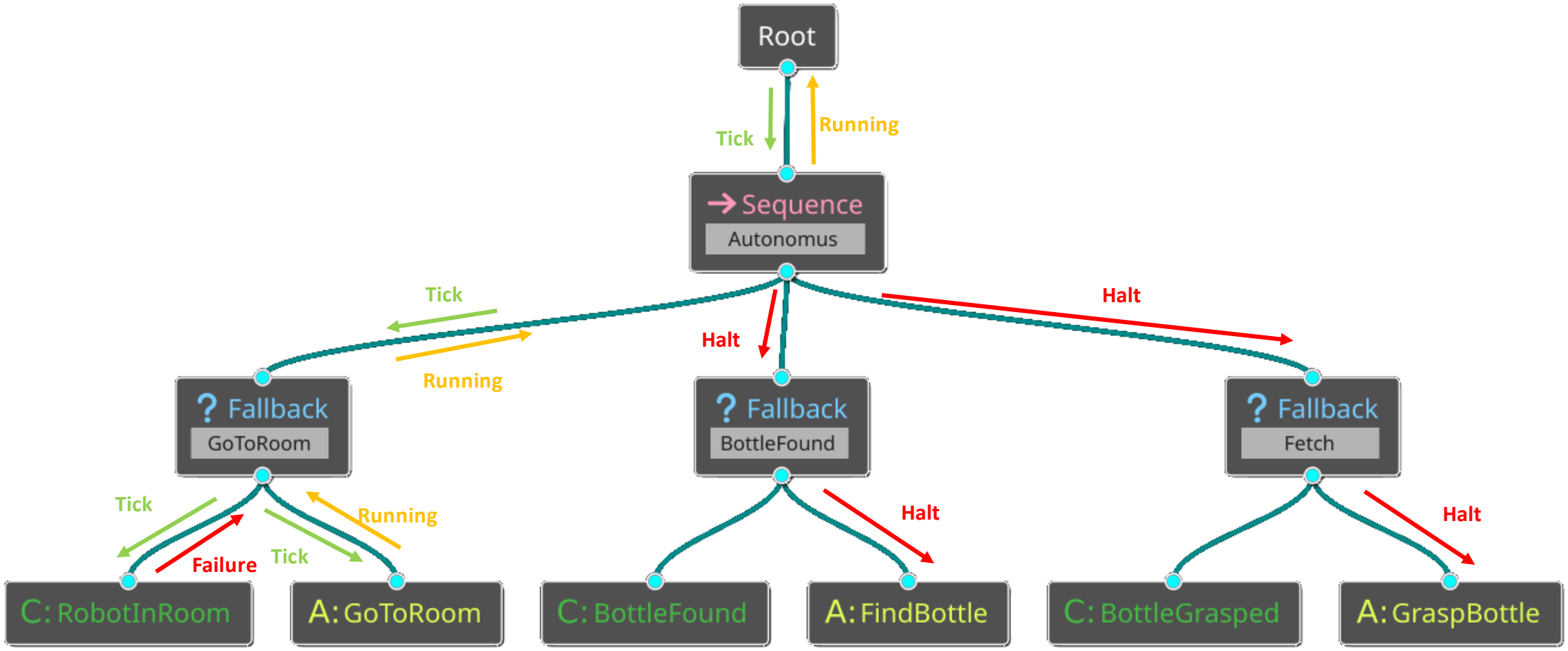
Behavor Tree in Robotics



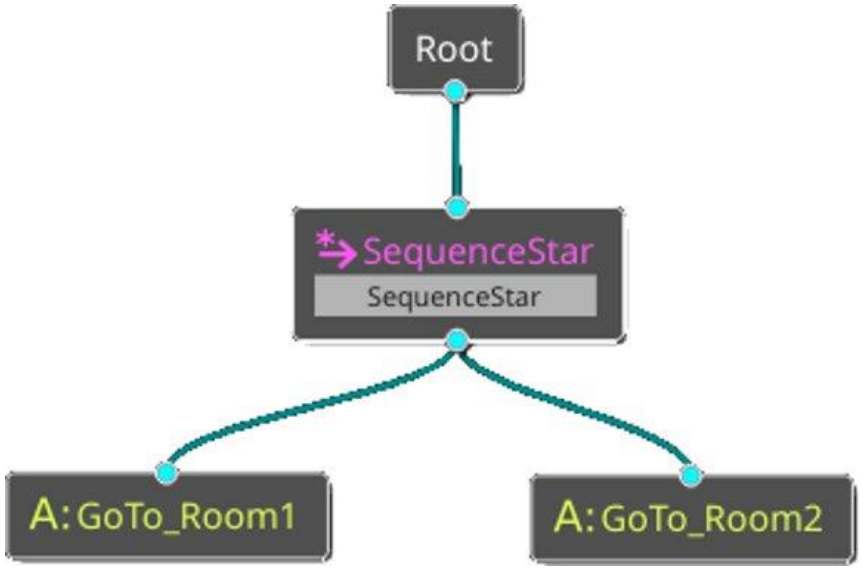
Behavor Tree in Robotics



Behavor Tree in Robotics



Star (*) Nodes



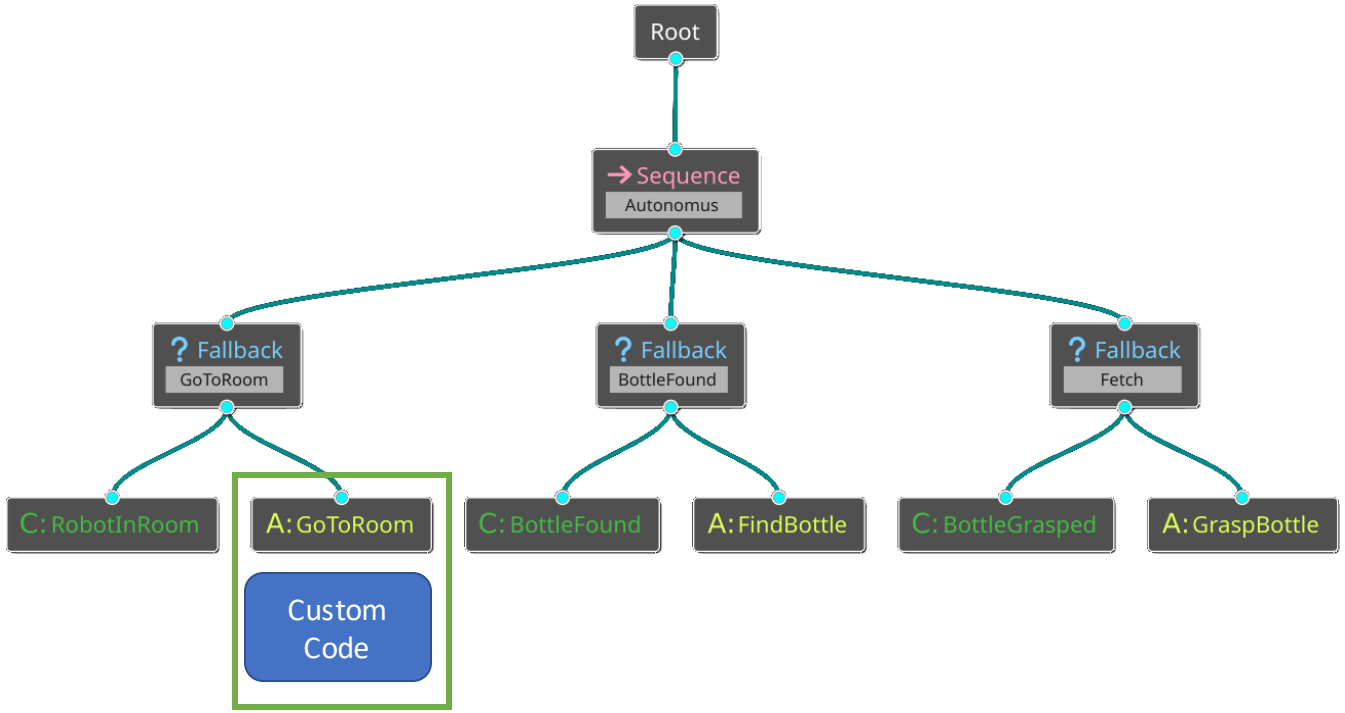
SequenceStar and FallbackStar are nodes with memory.

This means that once they get a result (Success/Failure, i.e. not Running) from the leaf node, they won't tick it anymore.

NOTE!!!!
In BehaviorTree_CPP library the naming convention is different:

- Sequence -> SequenceStar
- ReactiveSequence -> Sequence

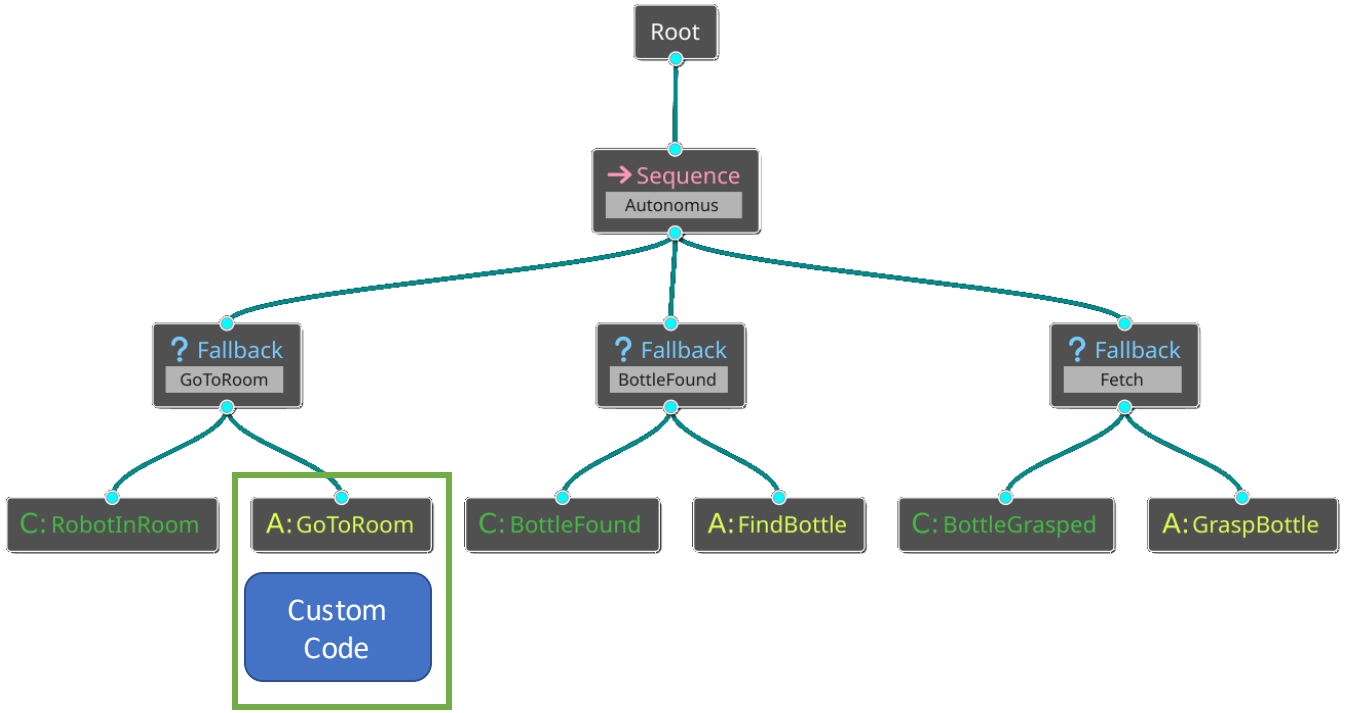
Implementing leaf nodes



All the conditions and actions require to run some code in order to compute the return value.

The routine can be directly embeeded inside the leaf node itself and loaded by the BT engine by subclassing the provided "ConditionNode" or "ActionNode" classes.

Implementing leaf nodes

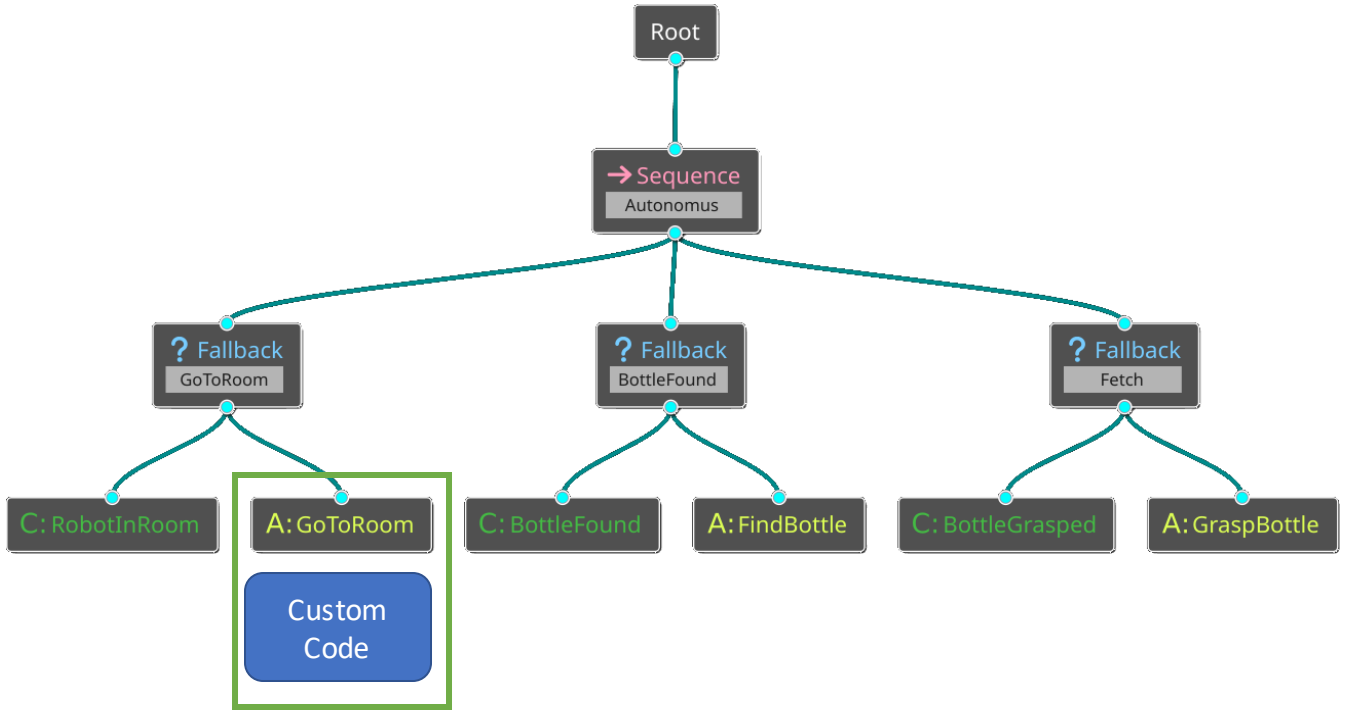


This was not good for CARVE for different reasons:

1) The purpose of the project was to better inspect the inside of the custom code and try to verify some property on the whole system: BT + custom code.

This implementation completely hides the code whitin the "ActionNode" class.

Implementing leaf nodes

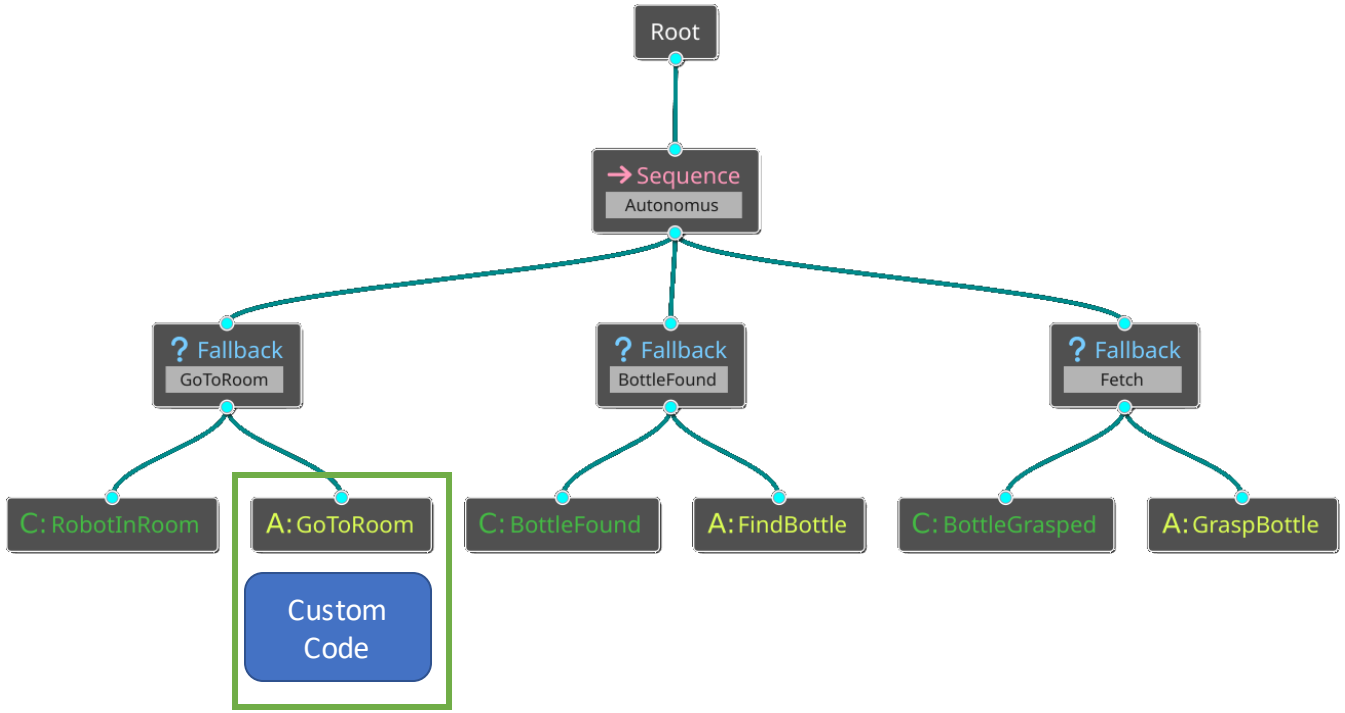


This was not good for CARVE for different reasons:

2) The engine needs to compile / link the code of the leaves, creating a sort of circular dependency:
The leaf require the Action definition in order to subclass, and the engine requires the class in odrer to link against it.

Each time a new Action is created or modified, the engine requires recompilation.

Implementing leaf nodes

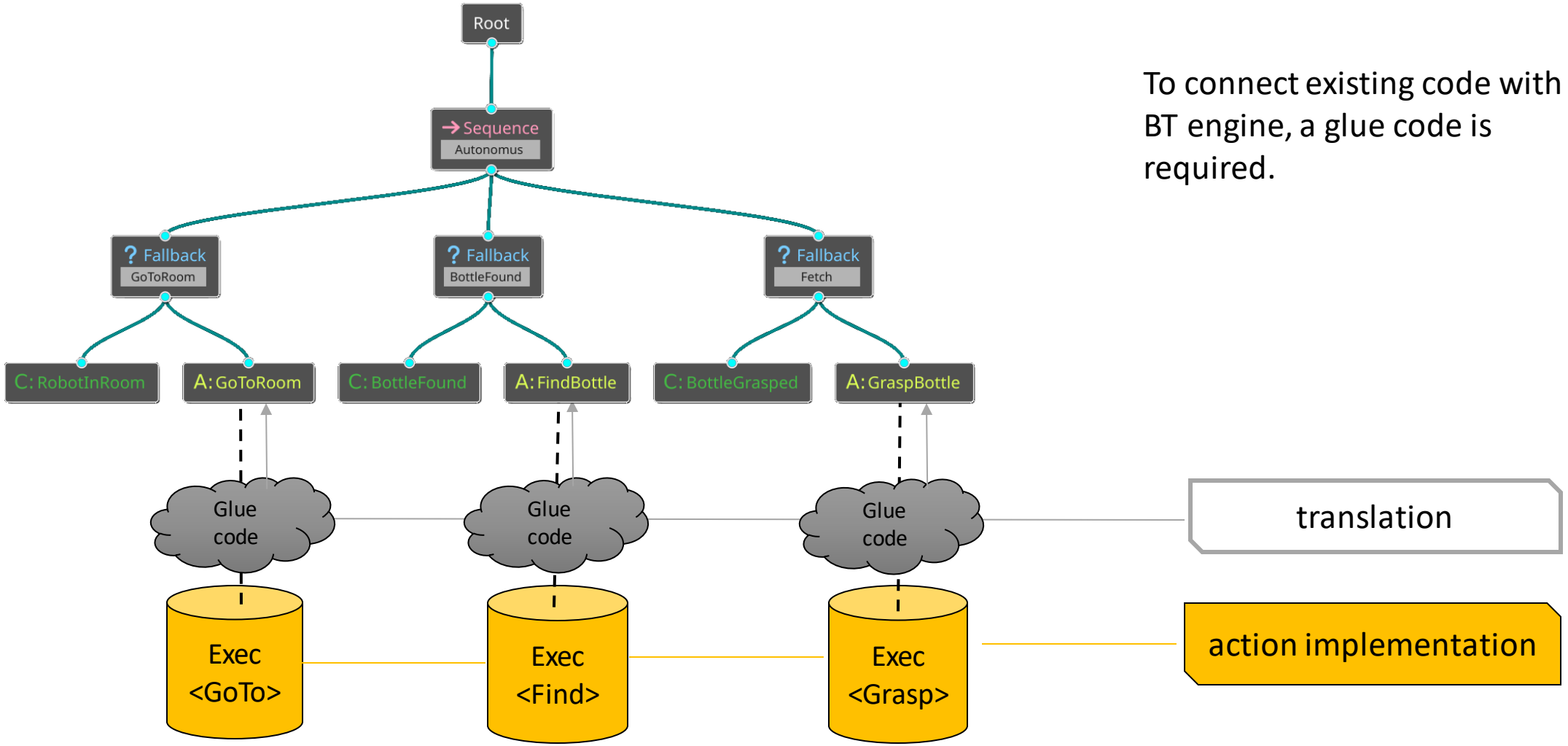


This was not good for CARVE for different reasons:

3) Not easy to integrate already existing code.

Navigation and manipulation stack in YARP are widespread across many executables (on different machines) and can't directly included in a single class.

Connecting BT with existing executables



To connect existing code with BT engine, a glue code is required.

BT nei confronti di una singola azione	Glue code	Singolo eseguibile nei confronti del chiamante
Solo 2 comandi possibili Tick / Halt	Cosa chiamo?	Una pletora di possibili chiamate a funzione
Tick e Halt NON hanno parametri	Dove recupero I parametri?	Ogni chiamata ha quasi sempre dei prarametri custom
Chiamare Tick periodicamente sulla stessa azione, anche durante l'esecuzione della stessa (running) , è safe e normale	Il primo tick ha un significato diverso dai successivi	Ad ogni chiamata, tendenzialmente, la routine riparte da capo. Una nuova chiamata durante l'esecuzione della stessa non è safe e può causare undefined behaviour
Un'azione è un'unità singola, atomica	Contiene una parte logica/deliberativa esso stesso	Ad un tick corrispondono più cose da fare, che possono cambiare con l'evolversi dell'azione.
Le azioni *terminano* sempre con Successo o Fallimento	Cosa ritorno?	Il valore di ritorno (se c'è) della singola chiamata RPC non rispecchia il Successo / Fallimento dell'azione (*1)
L'implementazione dell'azione è in grado di conoscere autonomamente il risultato dall'azione che esegue. Il risultato Success/Failure di una azione/condizione è, per il BT, un input esterno	Dove recupero lo status?	Per conoscere il successo / fallimento dell'azione (in senso BT) può essere necessario fare più chiamate e non una sola (*1)
Distingue tra Successo e Fallimento nella missione, ma non tra Fallimento ed Errore di esecuzione	Gestione errori e map logico dei valori di ritorno	Non distingue tra Successo o Fallimento ma tra Errore o Non-Errore (*2)

BT nei confronti di una singola azione	Glue code	Singolo eseguibile nei confronti del chiamante
Il Tick deve durare 'poco'	Adattare le tempistiche	La chiamata può prendersi tutto il tempo che gli serve prima di ritornare
Ha un clock periodico	Adattare le tempistiche	Dipende dall'implementazione, ma in generale non è sincrono col chiamante

Glue code's job

Remap Tick / Halt

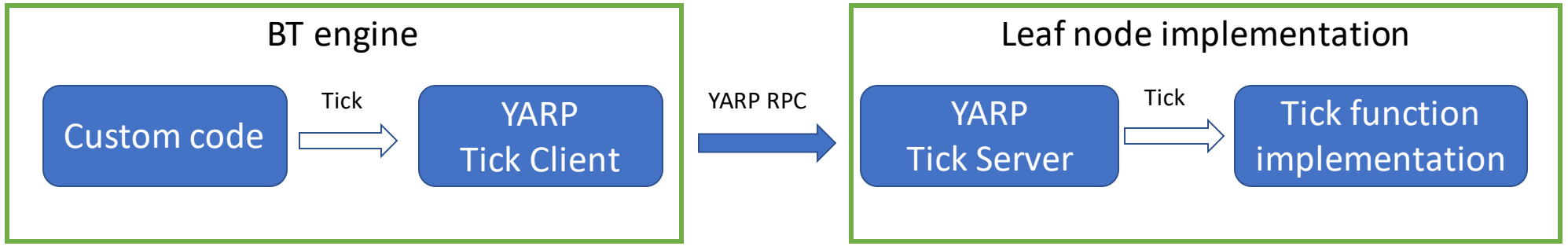
Remap Return values

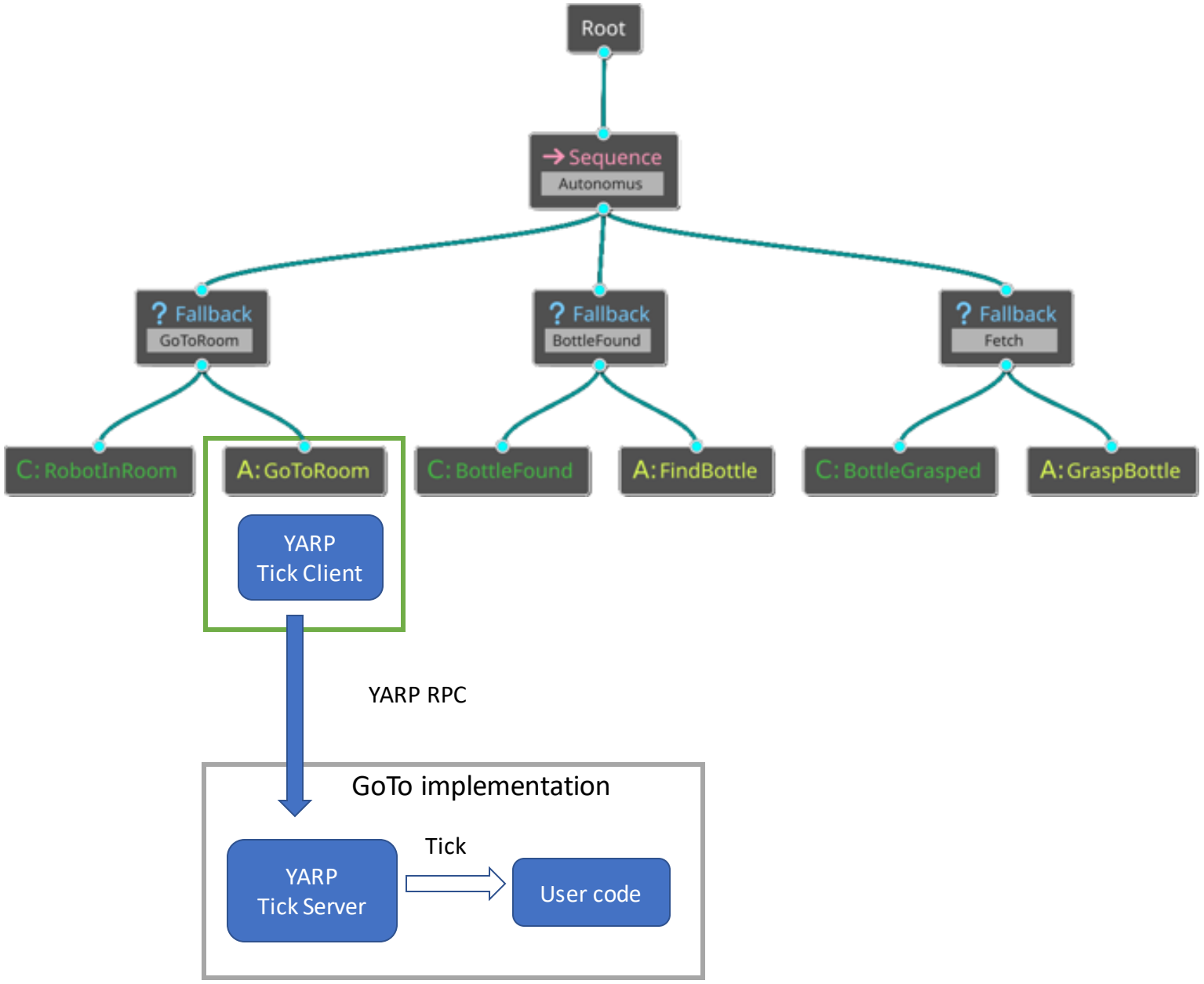
Handle parameters

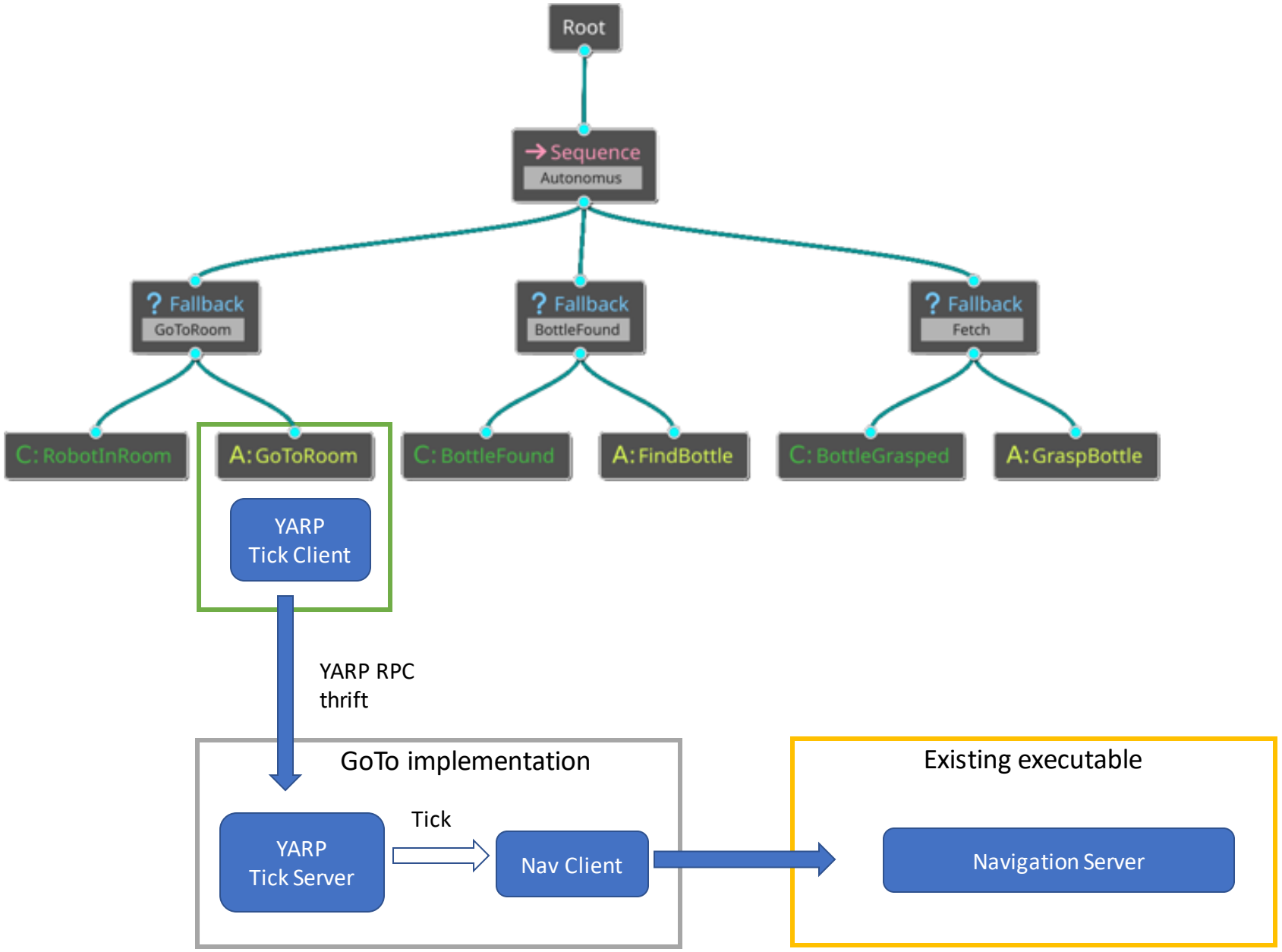
Asynchronous actions

Usual YARPish paradigm

The RPC client/server pattern has been used to propagate the Tick/Halt command from the engine to the action's glue code.




Root
Sequence
Autonomus
? Fallback
GoToRoom
? Fallback
BottleFound
? Fallback
Fetch
C: RobotInRoom
A: GoToRoom
C: BottleFound
A: FindBottle
C: BottleGrasped
A: GraspBottle
YARP
Tick Client
YARP RPC
GoTo implementation
YARP
Tick Server
Tick
User code


Root
Sequence
Autonomus
? Fallback
GoToRoom
? Fallback
BottleFound
? Fallback
Fetch
C: RobotInRoom
A: GoToRoom
C: BottleFound
A: FindBottle
C: BottleGrasped
A: GraspBottle
YARP
Tick Client
YARP RPC
thrift
GoTo implementation
YARP
Tick Server
Tick
Nav Client
Existing executable
Navigation Server

Thrift interface

```
ReturnStatus request_tick  (1:ActionID target, 2: Params params = {});

ReturnStatus request_halt  (1:ActionID target, 2: Params params = {});

ReturnStatus request_status (1:ActionID target);

bool request_initialize ();

bool request_terminate ();
```

Thrift interface

The ActionID is a small structure useful to uniquely identify the client requesting the action, so that the server can safely handle *multiple clients* at the same time.

```
struct ActionID  {
    1: string target;
    2: string resources;
    3: i32 action_ID;
}
```

target: this identifies the main target the action is referred to; e.g the location to reach.

resources: this identifies which resources the action will need to use; e.g. left or right arm.

action_ID: an ID number which is required to be unique for each Behaviour Tree for each node. In case the same action node is repeated more than once, each instance needs to have a different ID. (see later)

Synch/Asynchronous actions

Synchronous actions: the computation time of request_tick user function is negligible with respect to Behavior Tree period. In this case the Tick can return only Success/Failure (no Running).

Asynchronous actions: the computation time of request_tick user function takes a lot of time, e.g. navigation. In this case, the TickServer can spawn a thread for the Tick execution and return Running.

```
skill.configure_TickServer("/TickServer", "Test 1", true);
```

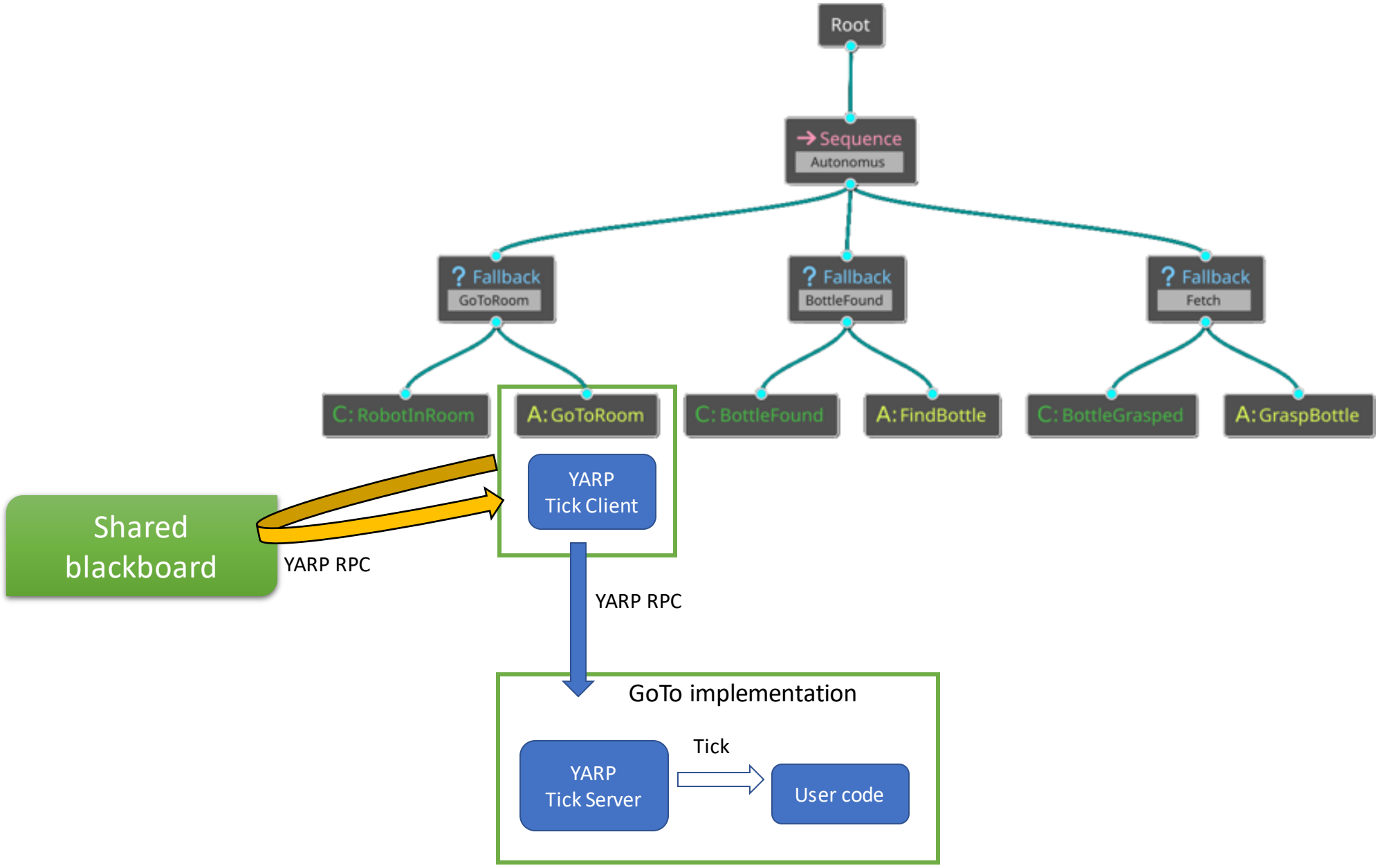
Note: if more actions are running in parallel, the server will handle it automatically, the user implementation does not require any addictional work.

Handle parameters

To handle the parameters required by the actual implementation to work correctly, a mapping has been introduced by mean of a shared blackboard.
Each node can have a single (optional) parameter called *'target'* which is used as a key to retrieve other data from a shared blackboard.

The parameters will then propagated to the TickServer to be processed.

At the startup the blackboard can read a .ini file to initialize the data.


Root
Sequence
Autonomus
Fallback
GoToRoom
Fallback
BottleFound
Fallback
Fetch
C: RobotInRoom
A: GoToRoom
C: BottleFound
A: FindBottle
C: BottleGrasped
A: GraspBottle
YARP
Tick Client
Shared
blackboard
YARP RPC
YARP RPC
GoTo implementation
Tick
YARP
Tick Server
User code

Targets

Many actions are called like:

GoTo_Kitchen **Grasp_Bottle** **Find_Bottle** ecc...

The part in red is the name of the action itself, while the part in blue indicates the main target it referres to.

For example, in case a BT has the leaves **GoTo_Kitchen** and **GoTo_Kitchen** they indicates due different actionsm but at implementation level they'll act onto the same executable, with a different target.

Targets

If we think to the target as a primary parameter, it can be used to retrieve other associated paramters.

Es:

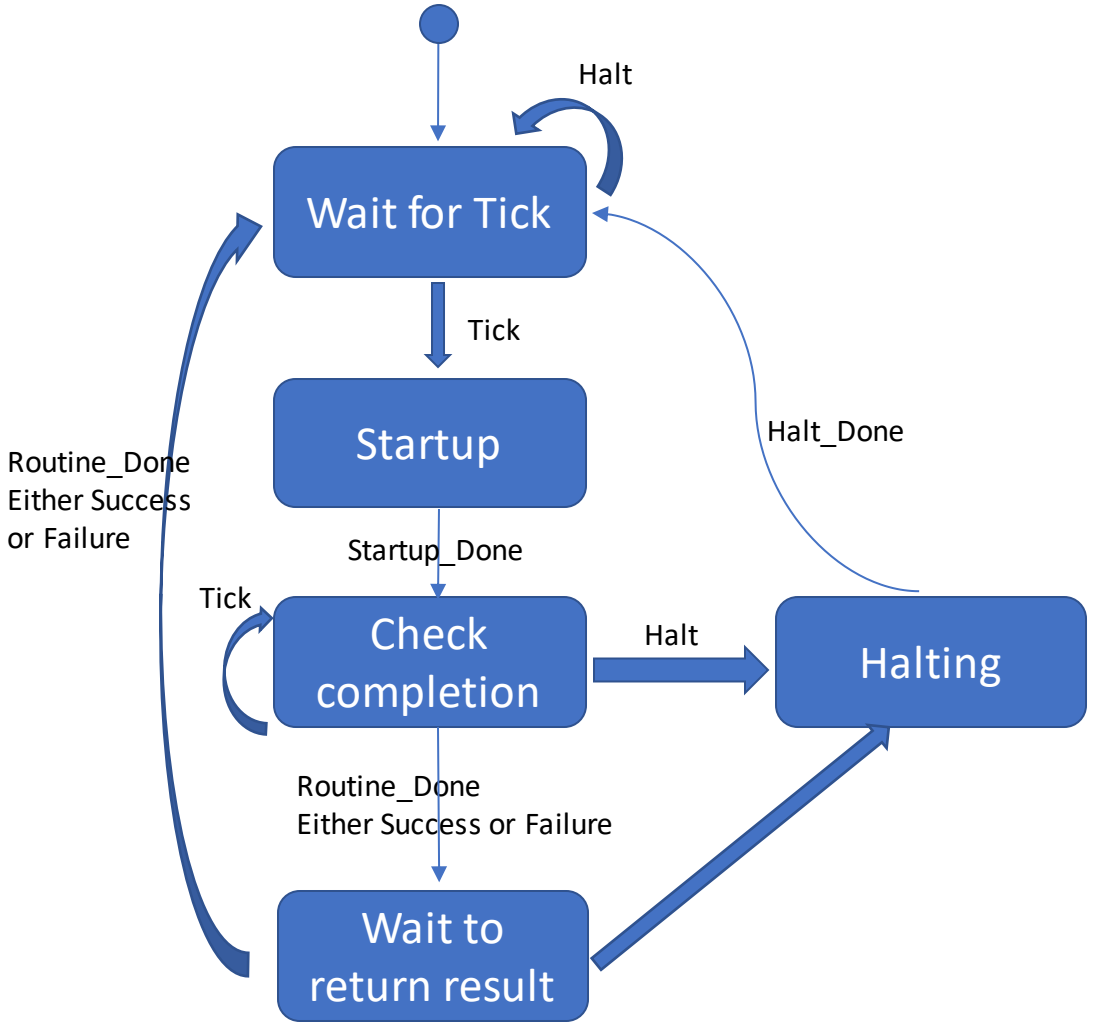
The navigation target Kitchen has a numeric value <x, y, theta>

The associated parameters are configurations of the navigation module, like

enable/disable obstacle avoidance, maximum speed and so on.

Those parameters may differ for different targets, so they depend on it

Glue Code / Tick Server (Skill)



The glue code will usually perform a few steps, mainly devided into two groups.

Startup:

- Set max speed;
- Set start navigation;

Check completion:

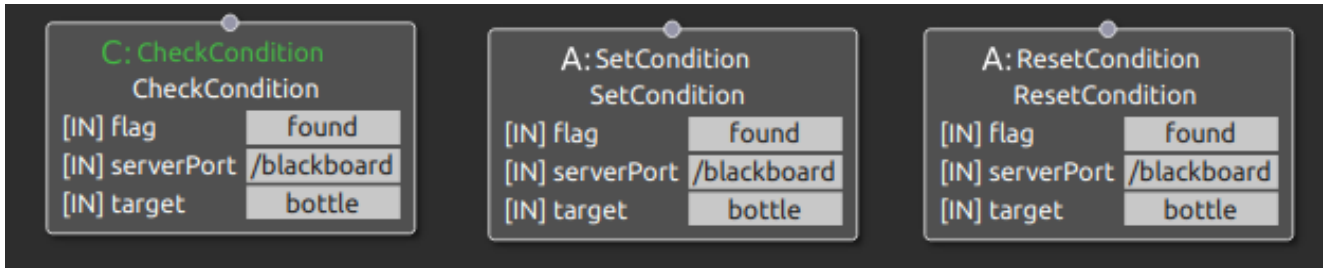
- Compare robot position with desired one

Blackboard interface

```
struct Data { }
(
  yarp.name = "yarp::os::Property"
  yarp.includefile="yarp/os/Property.h"
)

service BlackBoardWrapper
{
    Data getData(1: string target)
    bool setData(1: string target, 2: Data datum)
    void clearData(1: string target)
    void clearAll()
    void resetData()
    list<string> listTarget()
}
```

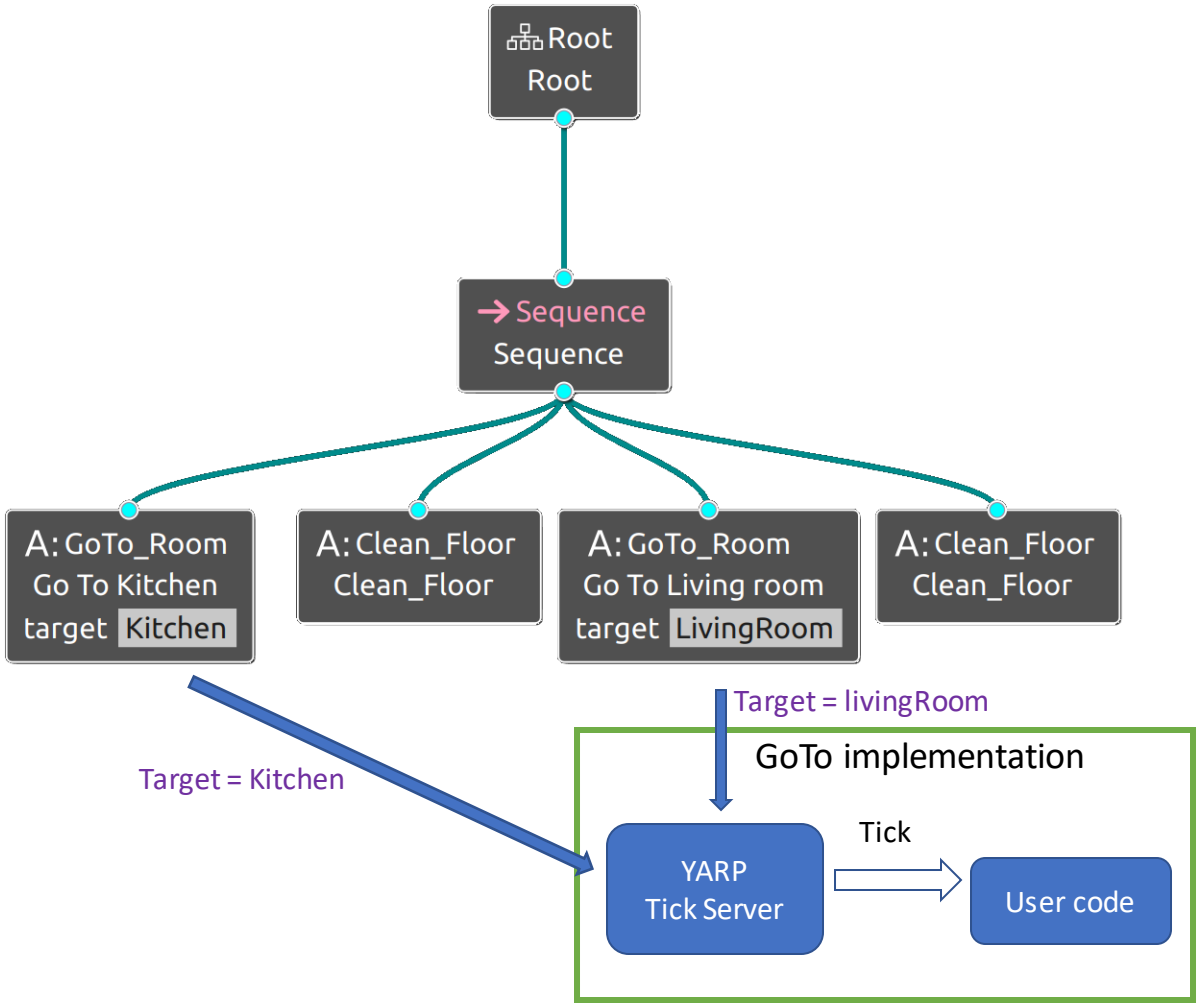
Blackboard Tick handler



The blackboard is also Tickable: it'll return success if a certain condition is True and Failure if it is false.

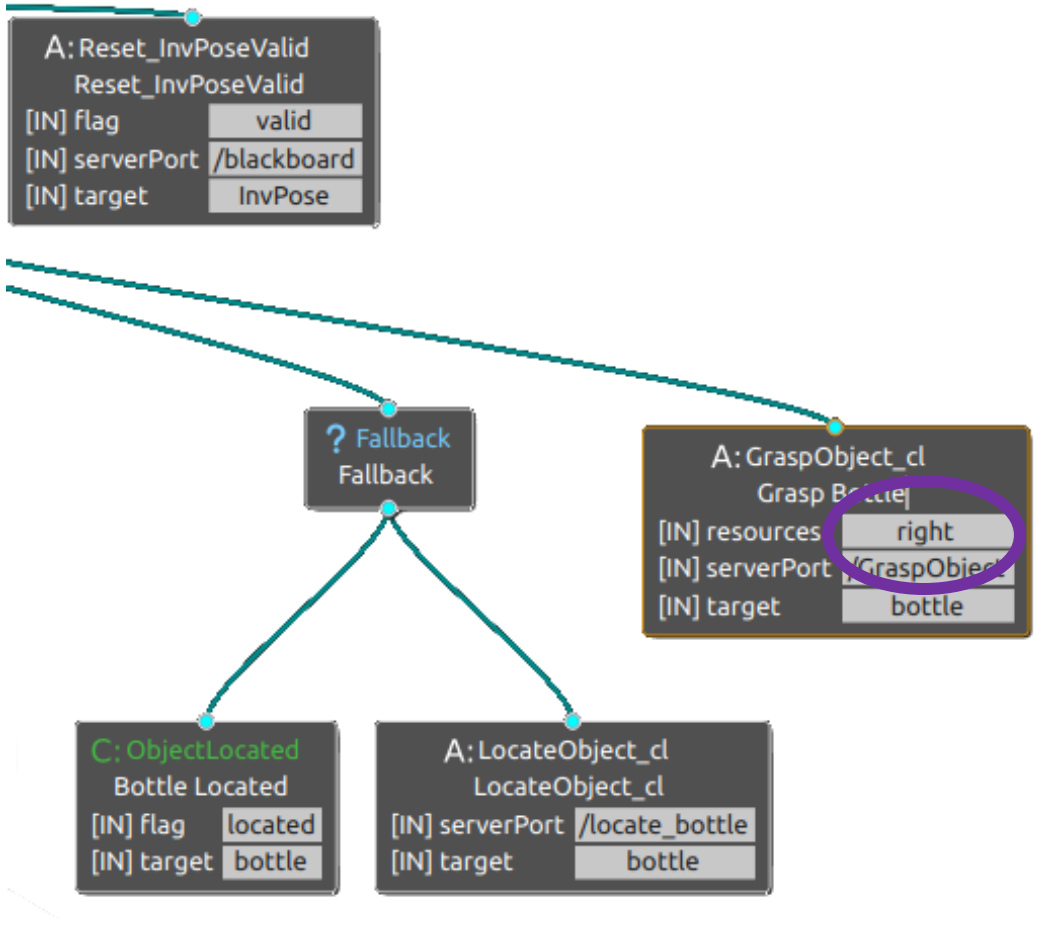
- CheckCondition checks if the parameter <found> for target <Bottle> is *true*
- SetCondition will set the parameter <found> for target <Bottle> to *true*
- ResetCondition will set the parameter <found> for target <Bottle> to *false*

Multiple Targets




```
struct ActionID {
    1: string target;
    2: string resources;
    3: i32 action_ID;
}
```

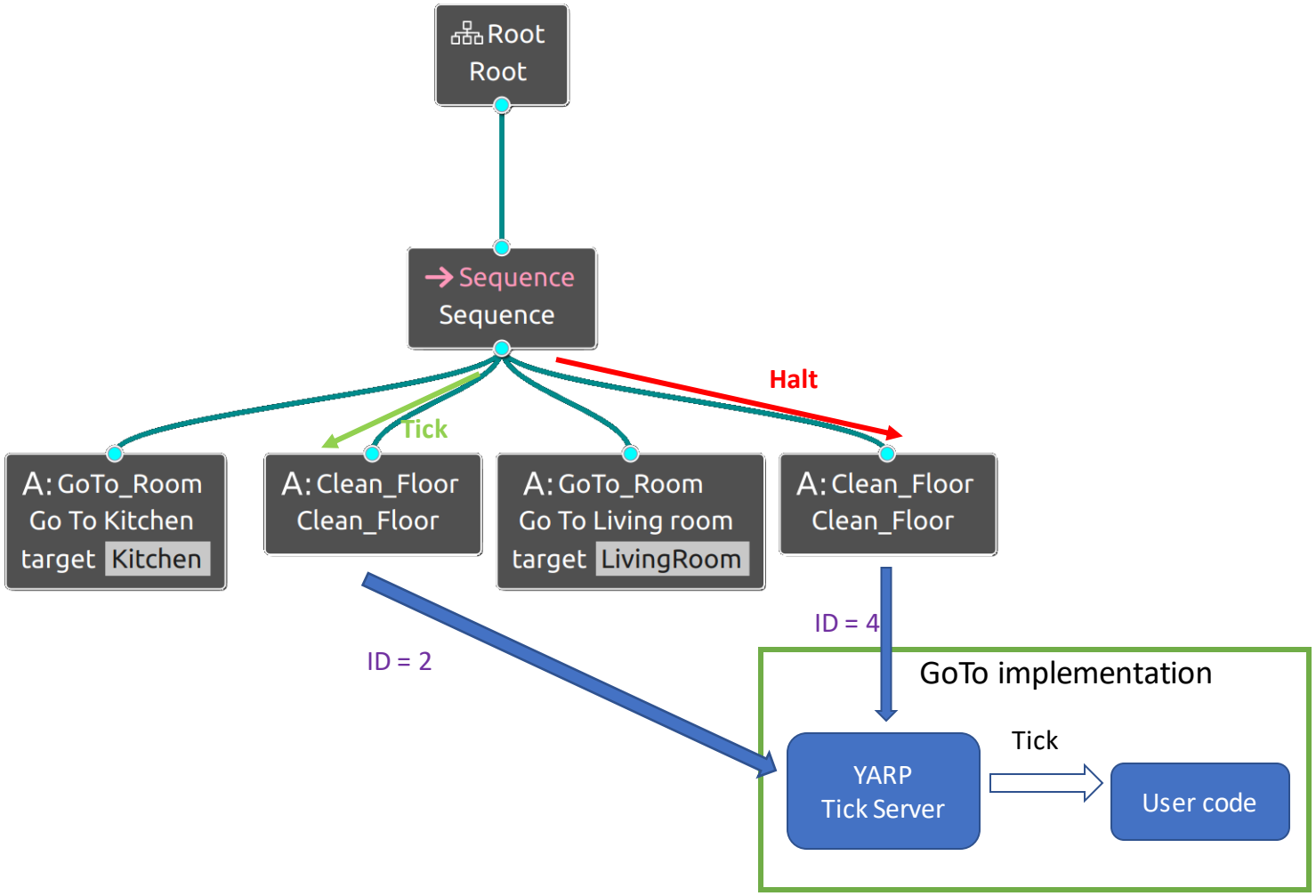
Resource usage description




```
struct ActionID {
    1: string target;
    2: string resources;
    3: i32 action_ID;
}
```

A special field has been reserved into leaves desription for resources.
Right now it is a string parameter, with no special handling, but they are read from BT description XML file and propagated to the TickServer via the Action_ID sctucture.
This example shows that the grasp action shall be performed with right arm. By changing right with left from the Groot GUI, we can easily change how the robot will perform the action.

Reusage of leaves



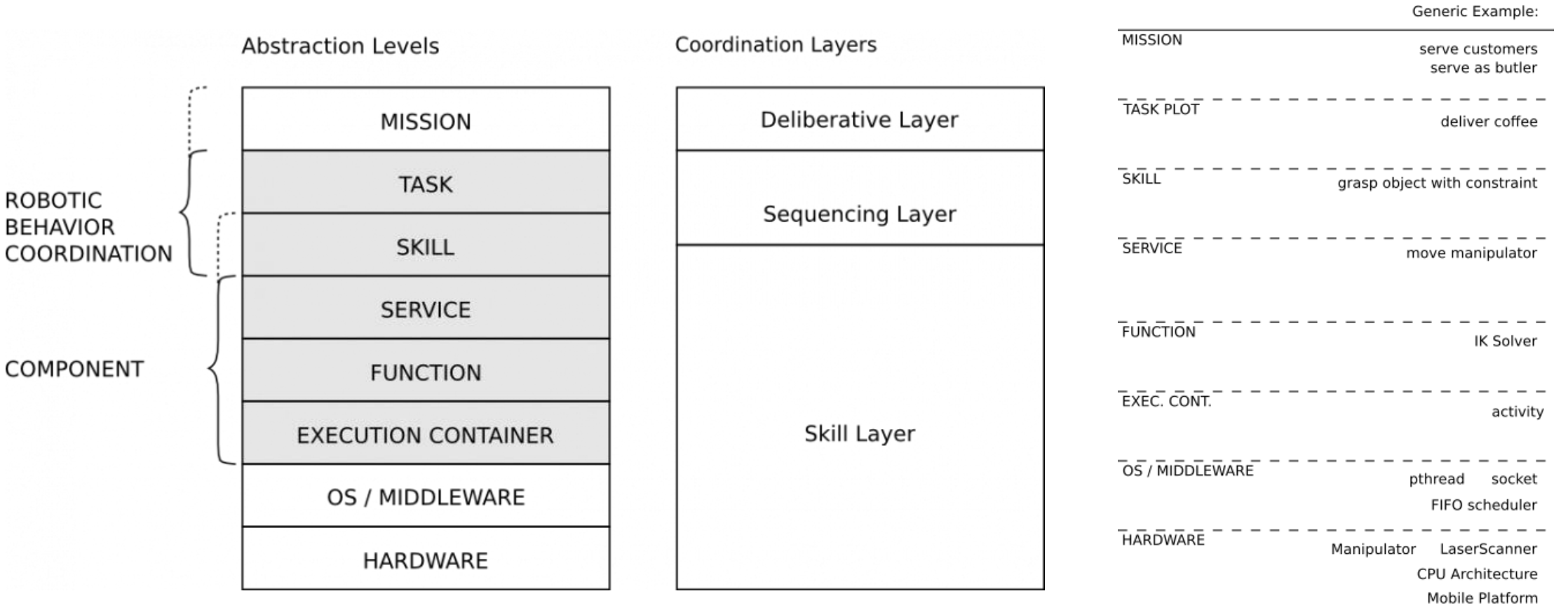

```
struct ActionID {
    1: string target;
    2: string resources;
    3: i32 action_ID;
}
```

Skills

To generalize the interaction between any **deliberative** level (regardless of type) and any **implementation** level (from different robots / producers) RobMoSys comes up with the concept of skill.

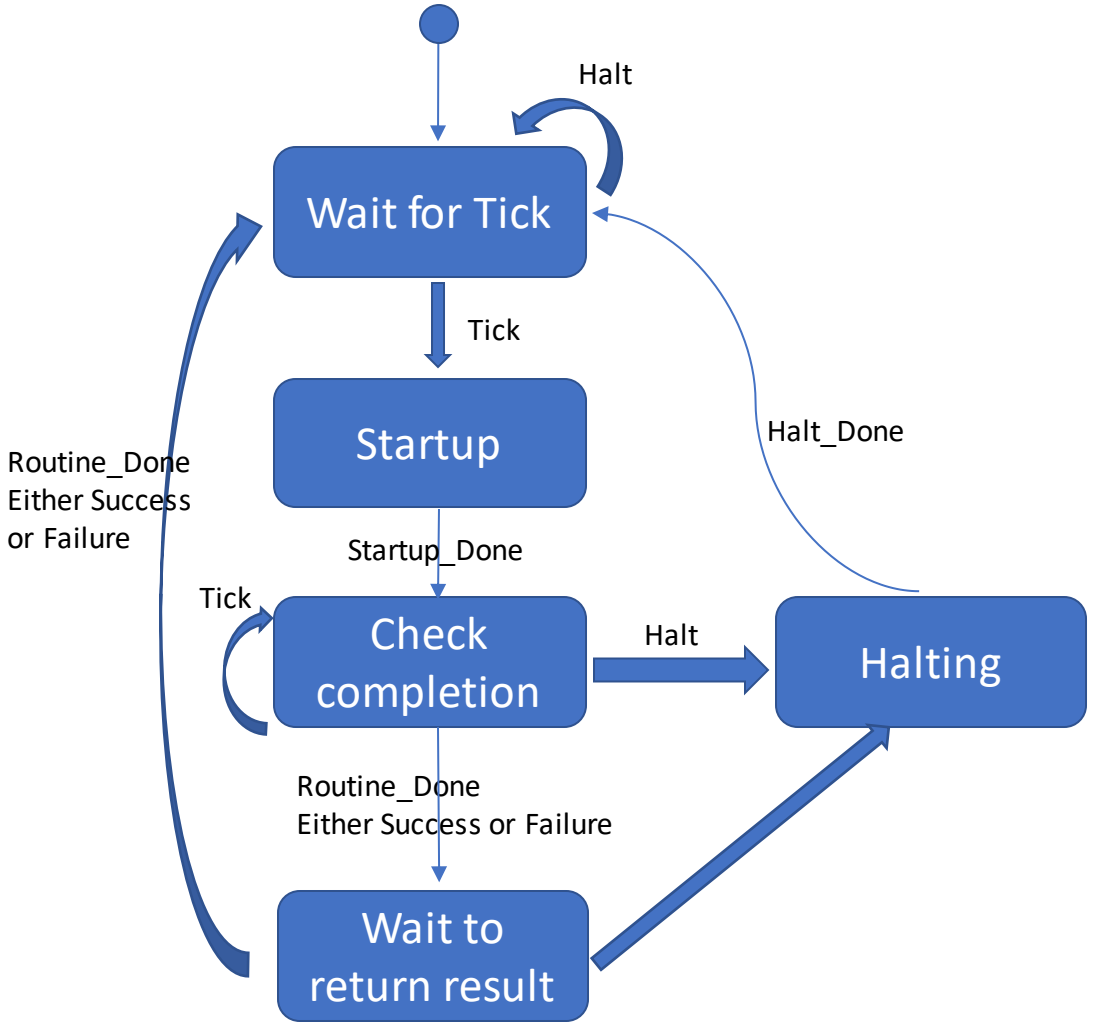
The skill fills the gap between the two levels and specify how to use a specific robot functionality.

The roles of a skill are ***configuration*** and ***coordination***.



Deliberative Level (Mission & Task)	Coordination Level (Skill)	Implementation Level (Service & Function)	Robot Level (Service & Function)
Behaviour Tree	BT Glue code	RF modules	robot
State Machine	SM Glue code	RF modules	robot
User Scripts / Code	User Scripts / Code (same)	RF modules	robot

'BT Skills'



The glue code will usually perform a few steps, mainly devided into two groups.

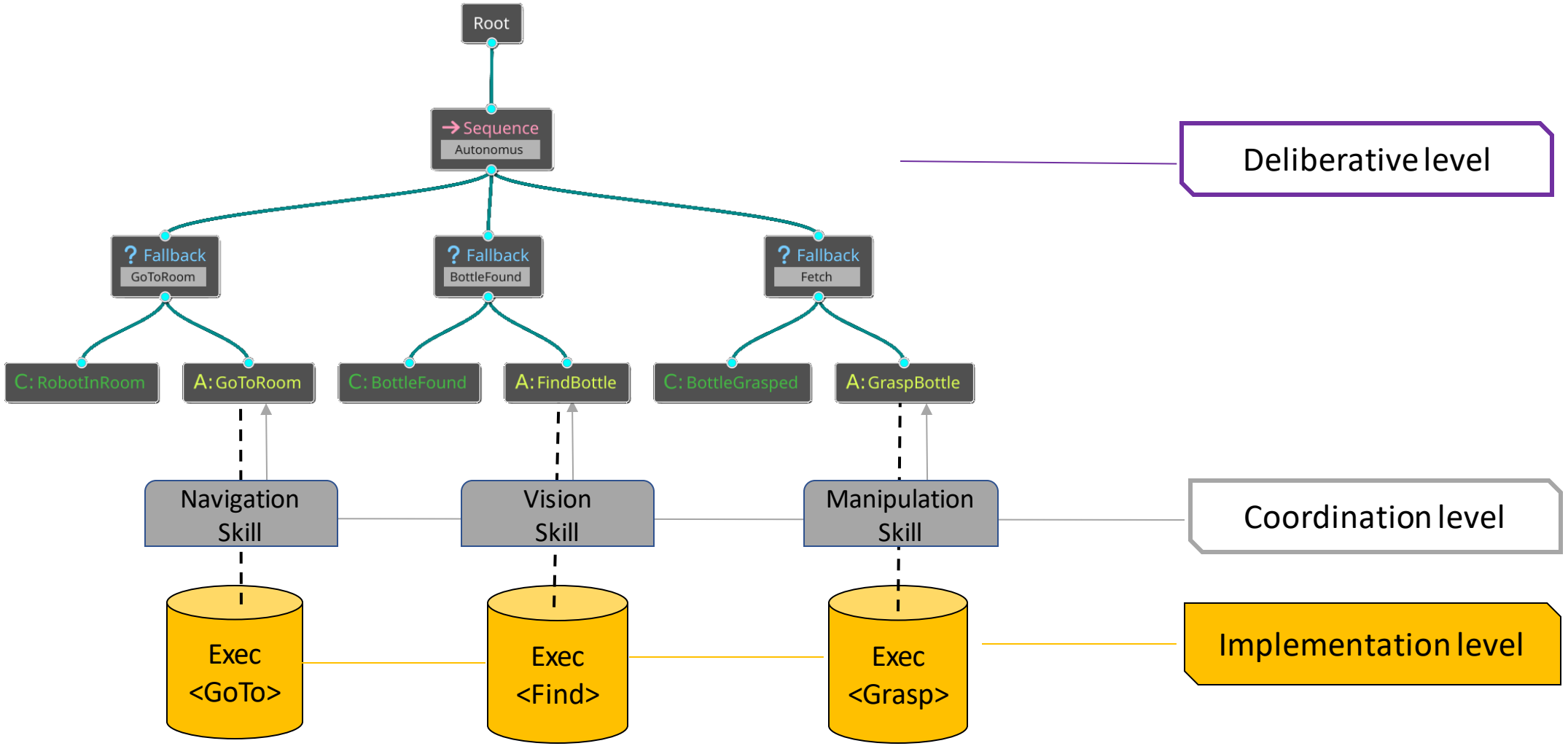
Startup:

- Set max speed -> RobotGoTo
- Set start navigation -> NavigServer

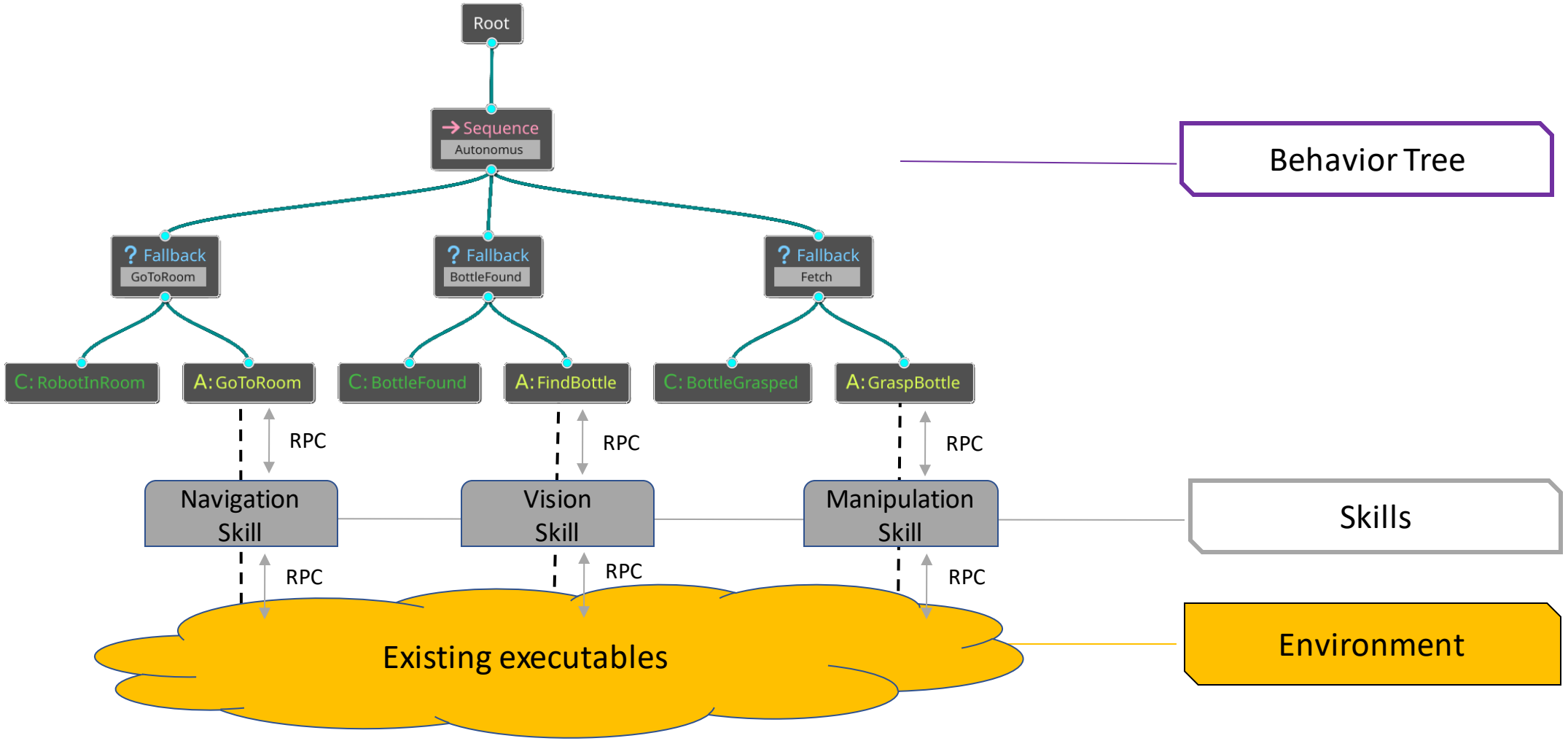
Check completion:

- Check robot's position -> localizServer

CARVE / RobMoSys naming convetion



CARVE setup


Root
Sequence
Autonomus
Fallback
GoToRoom
Fallback
BottleFound
Fallback
Fetch
C: RobotInRoom
A: GoToRoom
C: BottleFound
A: FindBottle
C: BottleGrasped
A: GraspBottle
RPC
RPC
RPC
Navigation Skill
Vision Skill
Manipulation Skill
RPC
RPC
RPC
Existing executables
Behavior Tree
Skills
Environment

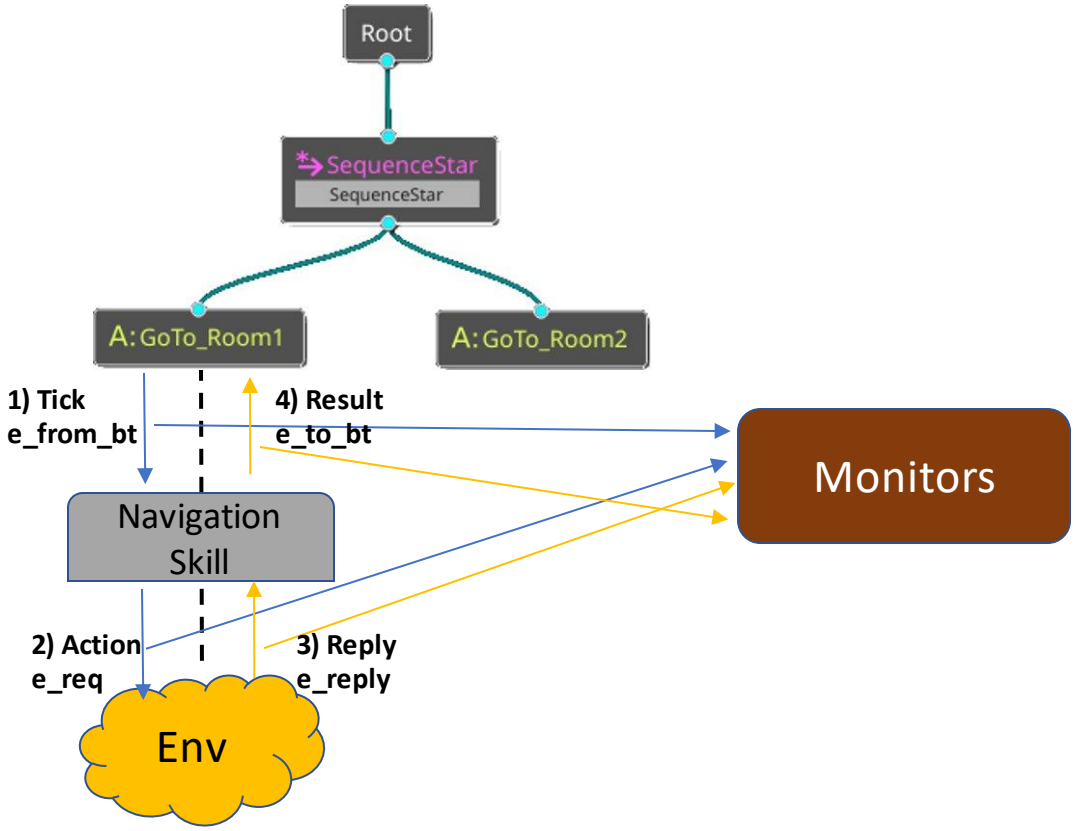
Monitor

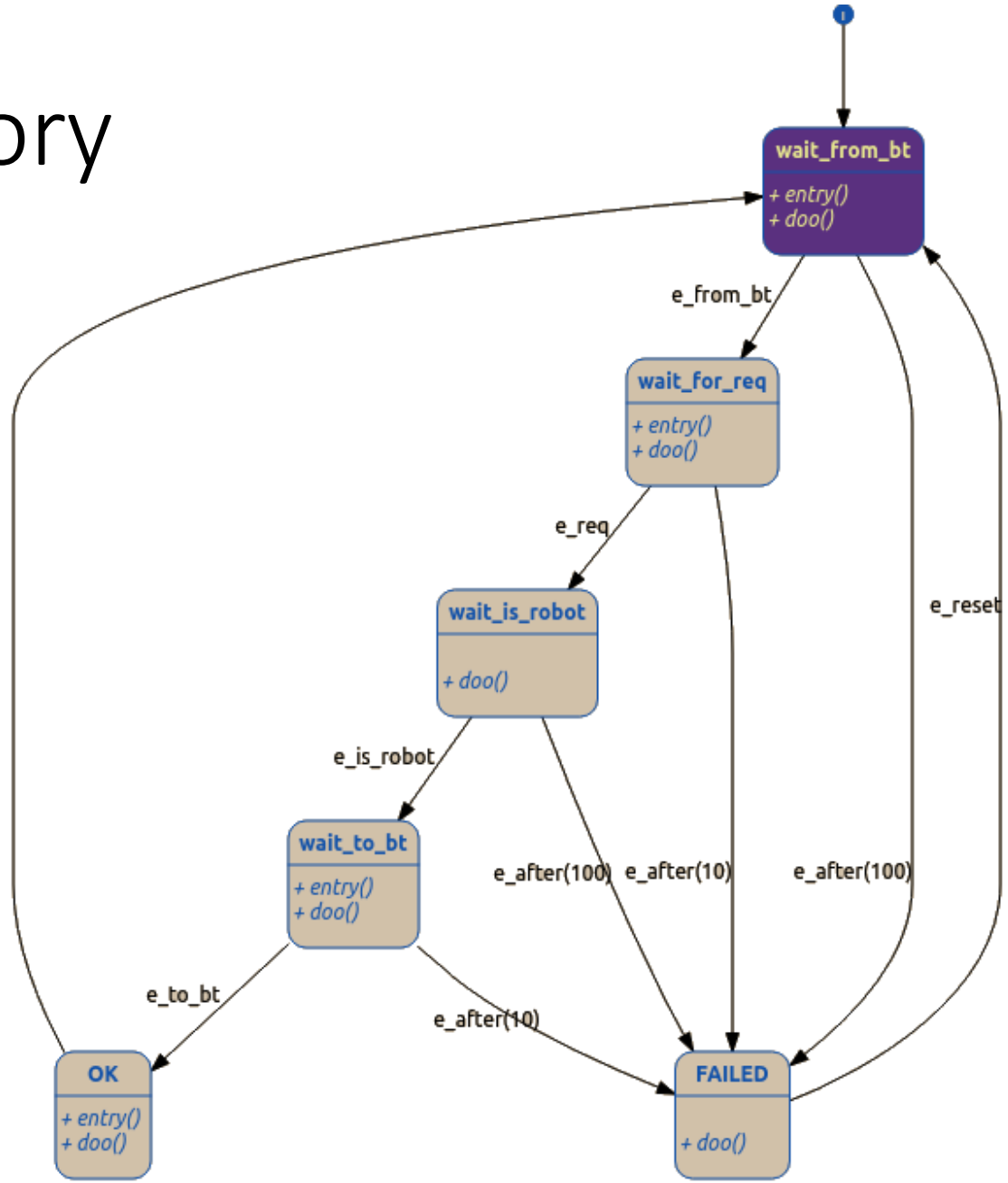
The monitors are utilities to verify the skills are working correctly.

Right now they perform a high level check and simply verify the skills will succesfully terminate its job within a certain amoungh of time.

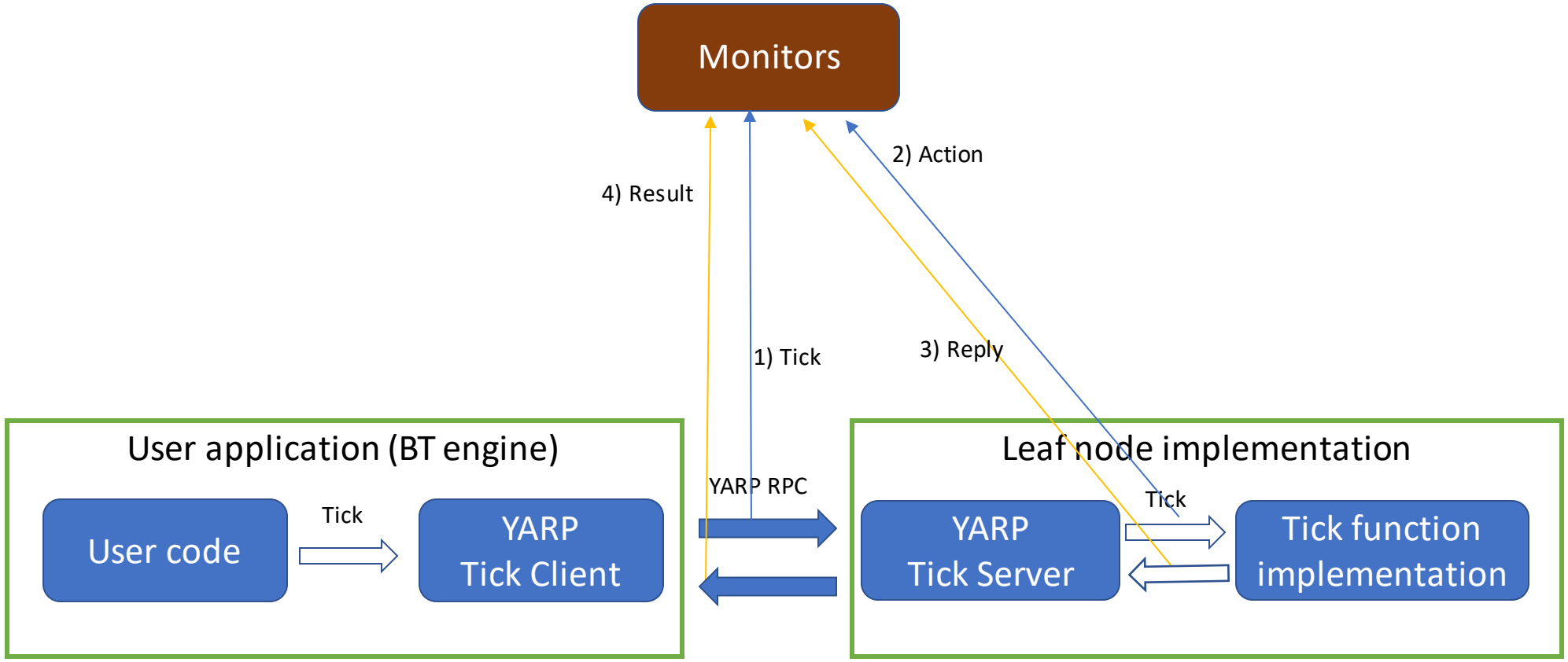
They do so by making sure the Tick is correctly propagated from the BT to the environment via the skill, and way back to the BT.

Basic monitoring - theory





Basic monitoring - implementation



The engine

Dependency:

The Behavior Tree engine is based on the library BehaviorTree.CPP , which has a optional dependency the ZeroMQ library used to communicate with the Groot GUI.

The engine

It uses a custom XML format as Behavior Tree description. The syntax is described in the BehaviorTree.CPP documentation page. Example:

```
<BehaviorTree ID="BehaviorTree">
    <Fallback name="Bottle Found">
        <Condition name="Bottle  Found" ID="CheckCondition" serverPort="/blackboard" flag="found" target="bottle"/>
        <Action name="Find Bottle"  ID="GoToLocation_cl" serverPort="/navigation_module" target="findBottle"/>
    </Fallback>
</BehaviorTree>

<TreeNodesModel>
    <Condition ID="CheckCondition">
      <input_port name="flag">           Boolean condition to be checked on target.  </input_port>
      <input_port name="serverPort">  Name of the YARP port to connect to.       </input_port>
      <input_port name="target">         Name of the target to test.              </input_port>
    </Condition>
    <Action ID="ComputePose_cl">
      <input_port name="resources">    List of resources required by this action, if any.</input_port>
      <input_port name="serverPort"> Name of the YARP port to connect to. </input_port>
      <input_port name="target"  default="InvPose"> Pose to compute </input_port>
    </Action>
</TreeNodesModel>
```

The engine

Create a new leaf

```
class BtCppClient : public BT::ActionNodeBase,        // inherit from BehaviorTree_cpp library
{
  // It is mandatory to define this static method, from behavior tree CPP library
  static BT::PortsList providedPorts()
  {
    return { BT::InputPort("target",   "Name of the target this action is referring to, if any."),
          BT::InputPort("resources", "List of resources required by this action, if any."),
          BT::InputPort("serverPort", "YARP Port Name to connect to.")
    };
  }
}
```