**Introduction to Behavior Trees**

What is a behavior tree? How does it work and what is its role in game AI?

[](http://harrypotter.wikia.com/index.php?title=Whomping_Willow&image=Whomping_Willow_PA-jpg)

Welcome to a series of blog articles about my experiment (read: stumbling around) of marrying data-oriented, memory-streamlined behavior trees with a second representation to ease creation and modification during development. I write it to document my findings and decisions and to ask for your invaluable feedback to build a BSD licensed BT toolkit that is truly useful.

Article Updates

* March 10, 2011 – Added a [reference to the second article](http://altdevblogaday.com/2011/03/10/shocker-naive-object-oriented-behavior-tree-isnt-data-oriented/) in my behavior tree experiment blog series.
* March 05, 2011 – Posted [this article](http://bjoernknafla.com/introduction-to-behavior-trees) on my own blog [bjoernknafla.com](http://bjoernknafla.com/), too.
* March 02, 2011 – Reader eric wrote a [fantastic behavior tree feature analyzation](http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/#comment-1136) in the comments section. Don’t miss it!
* February 24, 2011 – added a reference section to the end of the article with additional references not found in the text.
* February 24, 2011 – added a section to show the advantages of behavior trees over finite state machines based on a [question](http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/#comment-959) by snake5.

Background

Behavior trees (BTs) are deployed by more and more game AI programmers to implement reactive decision making and control of the virtual creatures entrusted to them as [AiGameDev.com’s](http://twitter.com/aigamedev) Alex Champandard notes in his [retrospective for 2010 and outlook for 2011](http://aigamedev.com/open/editorial/2010-retrospective/).

What is a behavior tree and how does it tick

My view and understanding of behavior trees has been fundamentally shaped by [Alex Champandard’s](http://twitter.com/aigamedev) [tutorials](http://aigamedev.com/search/?tx_ttnews%5bswords%5d=behavior+tree&tx_ttnews%5bscats%5d%5b%5d=13&tx_ttnews%5bscats%5d%5b%5d=18&tx_ttnews%5bscats%5d%5b%5d=19&tx_ttnews%5bscats%5d%5b%5d=20&tx_ttnews%5bscats%5d%5b%5d=17&tx_ttnews%5bscats%5d%5b%5d=27) and online masterclasses on[AiGameDev.com](http://aigamedev.com/). Alex behavior tree definition is very elaborate and detailed. Other great online resources about behavior trees in games are [Damian Isla’s Gamasutra article about the AI in Halo 2](http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml), Max Dyckhoff’s [presentation about *Decision Making and Knowledge Representation in Halo 3*](http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/nips07.pdf), and Ricard Pillosu’s [*Coordinating Agents with Behavior Trees*](http://staff.science.uva.nl/~aldersho/GameProgramming/Papers/Coordinating_Agents_with_Behaviour_Trees.pdf) slides about their use in Crysis. [Joost van Dongen](http://twitter.com/#!/JoostDevBlog)blogs about the role behavior trees play in *Swords and Soldiers* [here](http://joostdevblog.blogspot.com/2010/12/ai-in-swords-soldiers-part-1.html) and [here](http://joostdevblog.blogspot.com/2011/01/ai-in-swords-soldiers-part-2.html).

Factually, behavior trees aren’t just trees but directed acyclic graphs (DAGs). A node can have multiple parents which allows reuse of sub behavior trees and therefore modularization. I’ll stick with the not-quite correct *tree* term as it is the one most widely used.

Behavior trees replace the often intangible growing mess of state transitions of finite state machines (FSMs) with a more restrictive but also more structured traversal defining approach. Behavior trees are formed by hierarchically organizing behavior sub-trees consisting of nodes. Visiting a node, respectively the sub-tree it roots, means running it according to its semantics. Execution of a node or a sub behavior tree results in an (aggregated) return state, e.g.:

* the node is *ready* to run before it is called,
* it finished with with a *success* state,
* the behavior has not finished *running* yet and should be called again during the next simulation step to eventually finish,
* it *failed* cleanly without side effects, or
* an *error* occurred that had side effects to explicitly deal with.

To update a behavior tree it’s always traversed beginning with its root node in depth first order. Each traversed node affects traversal. Selector nodes select one of their child nodes (if available) to traverse next. After the first child node and its associated sub-tree has been traversed the node is re-run to react to the child return status so it can decide if traversal should go up to its own parent node or if it might select another child node to traverse next.

Inner nodes explicitly control traversal semantics, e.g.:

* On each traversal *priority selectors* check which child to run in priority order until the first one succeeds or returns that it is running. One option is to call the last still running node again during the next behavior tree update. The other option is to always restart traversal from the highest priority child and implicitly cancel the last running child behavior if it isn’t chosen immediately again.
* *Sequence selectors* run one child to finish after the other. If one or multiple fail the whole sequence fails, too. Without a reset or without finishing the last child node a sequence stores the last *running* child to immediately return to it on the next update.
* *Loops* are like sequences but they loop around (hah, who would have thought!) when reaching their last child during their traversal instead of returning to their parent node like sequence node do.
* *Random selectors* randomly (hah again) select which of their child nodes to visit. A running node is visited again until it finishes.
* *Concurrent* nodes visit all of their children during each traversal. A pre-specified number of children needs to fail to make the concurrent node fail, too. Instead of running its child nodes truly in parallel to each other there might be a specific traversal order which can be exploited when adding conditions (see below) to a concurrent node because an early failing condition prevents its following concurrent siblings from running.
* *Decorator* nodes typically have only one child and are used to enforce a certain return state or to implement timers to restrict how often the child will run in a given amount of time or how often it can be executed without a pause.

Leaf nodes are classified as:

* *Actions* which finally implement an actors or game world state changes, for example to plan a path and move on it, to sense for the nearest enemies, to show certain animations, switch weapons, or run a specified sound. Actions will typically coordinate and call into different game systems. They might run for one simulation tick – one frame – or might need to be ticked for multiple frames to finish their work.
* *Conditions* check that certain actor or game world states hold true. If a sequence node has a condition as one of its children then the failing of the condition will prevent the following nodes from being traversed during the update. When placed below a concurrent node, conditions become a kind of invariant check that prevents its sibling nodes from running if a necessary state becomes invalid.

Actions and conditions get filtered access to the actor and the world state through an actor specific blackboard of collected or pre-computed values passed to them during an update step.

There be dragons – an example behavior tree and its traversal

I didn’t truly grasp all the details of the behavior tree update traversal until I started to implement it for myself. To shorten this process for you let’s take a look at an example how the traversal of a behavior tree for a dragon might work. Here is the simple example behavior tree:

0. Root *priority selector*

1. Guard treasure ***concurrent selector***

1.1. Is thief near treasure detectable?  ***condition***

1.2. Make thief flee (or eat him/her) **behavior**

2. Rob more treasures *sequence selector*

2.1. Choose a castle to fly to! ***action***

2.2. Fly to castle! ***action***

2.3. Fight (and eat) guards

2.4. Is still strong enough to carry treasure home? ***condition***

2.5. Take gold! ***action***

2.6. Fly home! ***action***

2.7. Put newly robbed treasure to possessed treasure! ***action***

3. Post pictures of hoard to Facebook ***behavior***

Each of the child nodes (and of the child nodes child nodes, and of the… you know the drill) might be another behavior defining sub-tree. For this example I only examine child node 1. and 2. in detail.

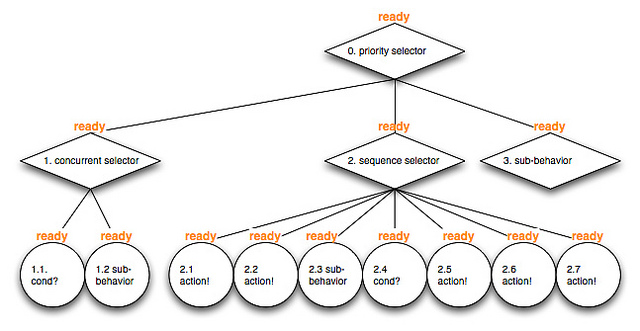
The root of the example tree is a priority node (with id 0).

*Guard treasure* is a concurrent node with two children. The *Is thief near treasure detectable?* condition only returns *success* if a trespasser can be sensed near the treasure, otherwise it fails. Keep care to not only test if a thief is near the hoard but also if the dragon is in a position to detect the robber. If the dragon is currently away to organize more gold it can’t see thieves to abandon its raid. Recall that a priority selector might always check its highest prioritized child behaviors first instead of sticking to a *running* child.

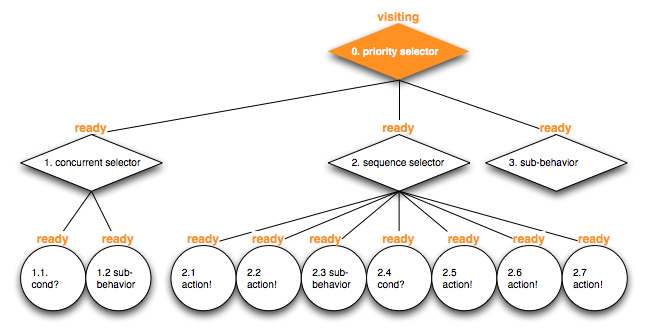
The *Make thief flee* behavior might be a sub-tree or might be an action to activate the games combat system.

Back to the children of the root priority node – its second child *Rob more treasures* is a sequence node. *Choose a castle to fly to!* is an action that selects a castle to raid and searches for a path to it. Via *Fly to castle!* the dragon flies along the calculated path until it reaches the castle. These are the nodes I cover in the example traversal. The others should be pretty self explanatory by their names.

On beginning the traversal all nodes are *ready* for execution.

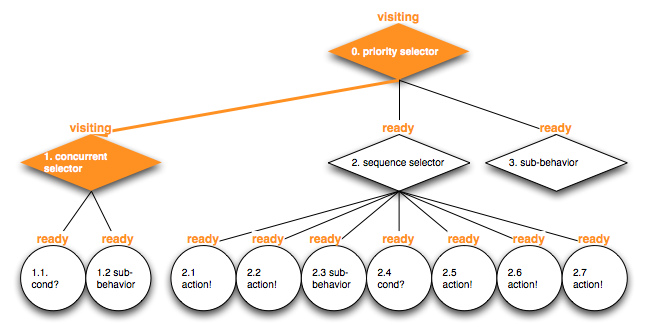
[](http://www.flickr.com/photos/bjoernknafla/5472728646/)

Let the first behavior tree update begin – it visits and runs the *ready* to run root priority node 0.

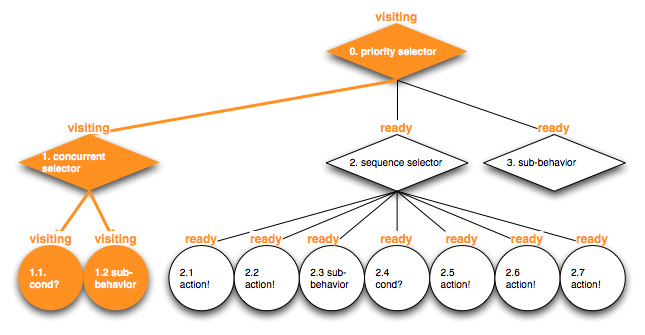
[](http://www.flickr.com/photos/42150256@N02/5472137229)

The priority node checks its child nodes from the first to the last until a child node returns a *success* or *running* state. It fails if no succeeding or *running* child node can be found.

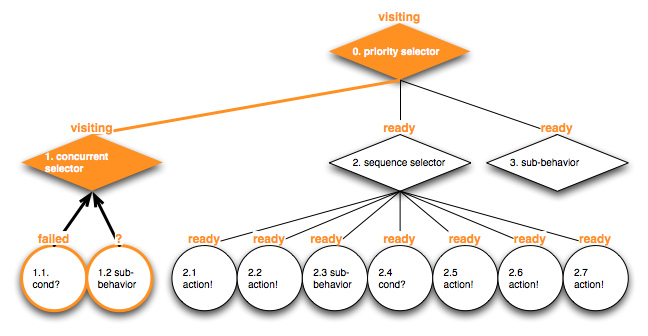
The first priority child behavior with id 1. is a *ready* concurrent selector.

[](http://www.flickr.com/photos/42150256@N02/5472137577)

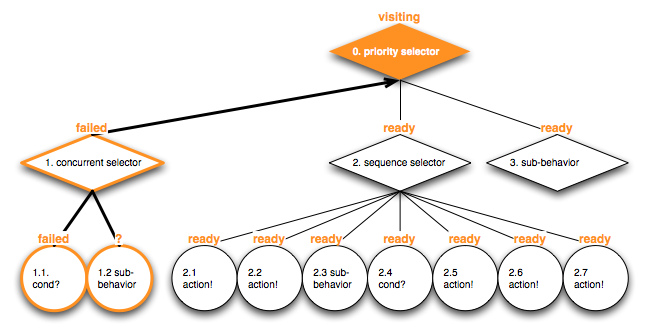
To traverse it all of its child nodes are visited concurrently.

[](http://www.flickr.com/photos/42150256@N02/5472137985)

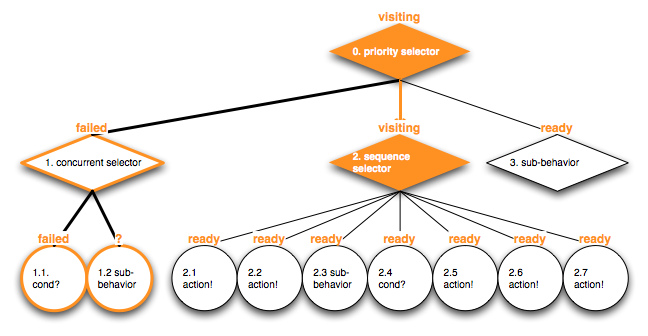
Its *Is thief near treasure detectable?* condition node child 1.1. fails because no trespasser is visible.

[](http://www.flickr.com/photos/42150256@N02/5472730162)

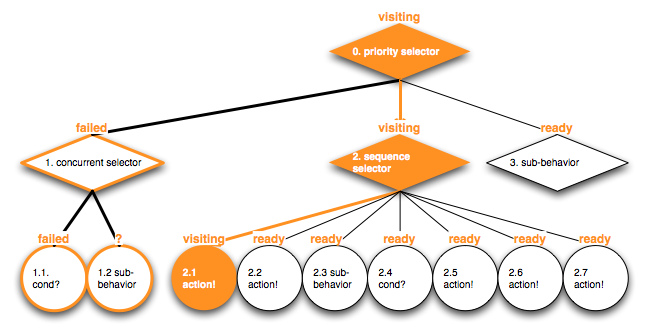
Therefore the whole concurrent *Guard treasure* node 1. fails.

[](http://www.flickr.com/photos/42150256@N02/5472138859)

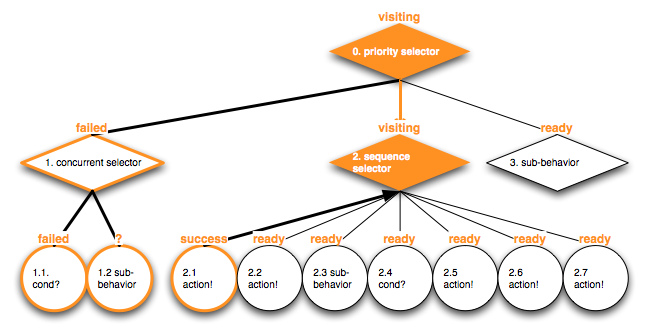
Priority node 0. tries to run the next child in its priority order. Child 2. *Rob more treasures* is *ready* and therefore visited.

[](http://www.flickr.com/photos/42150256@N02/5472139285)

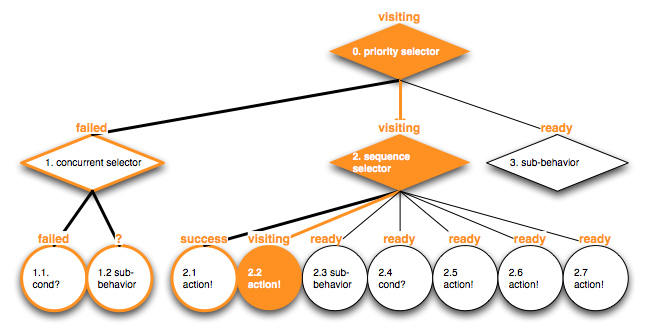
The sequence node starts by steering traversal into its child node 2.1. *Choose a castle to fly to!*.

[](http://www.flickr.com/photos/42150256@N02/5472731500)

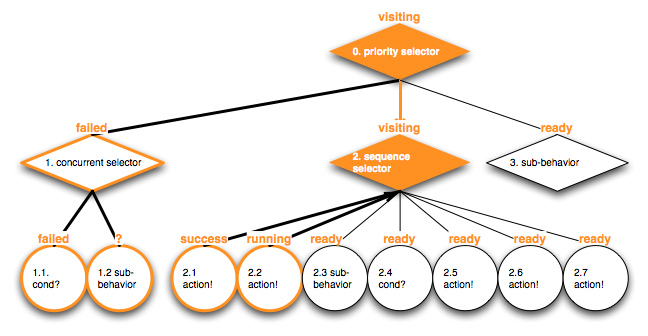
A target castle has been chosen successfully.

[](http://www.flickr.com/photos/42150256@N02/5472140207)

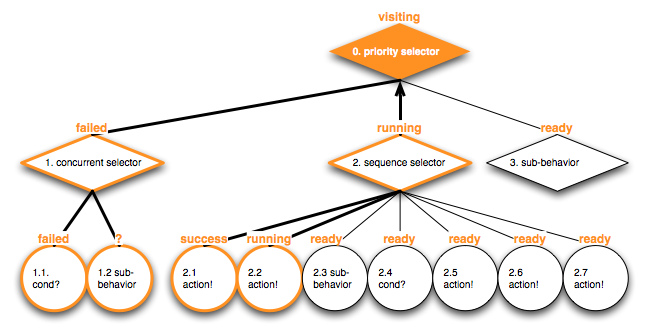
The next child in the sequence is executed – node 2.2. *Fly to castle!*.

[](http://www.flickr.com/photos/42150256@N02/5472140543)

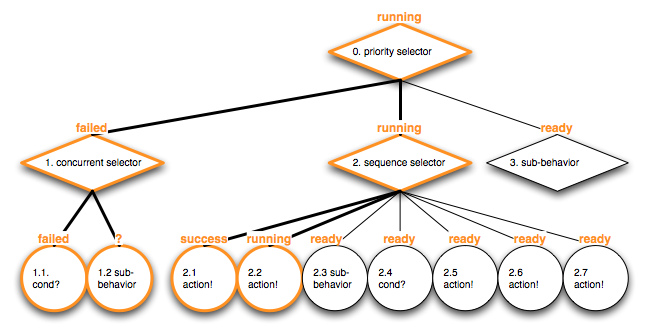
As the castle is far, far away it can’t be reached in this update step so it returns *running*.

[](http://www.flickr.com/photos/42150256@N02/5472732682)

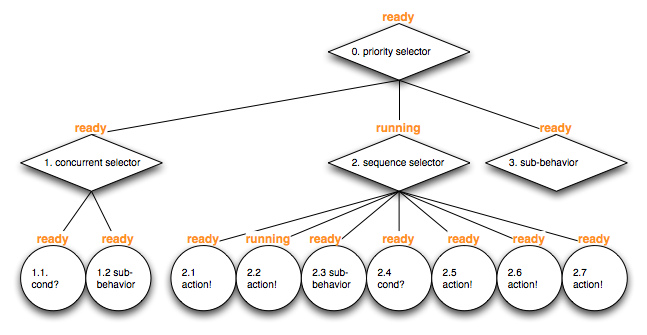
The sequence selector 1. can’t precede further and also returns *running* to its parent node.

[](http://www.flickr.com/photos/42150256@N02/5472141313)

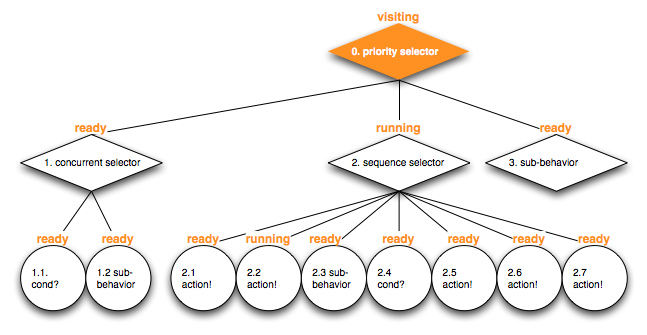
The priority selection root node bails out with a *running* state, too, it has found a running child node so it doesn’t need to check lower priority child nodes to traverse. This update traversal is done.

[](http://www.flickr.com/photos/42150256@N02/5472733384)

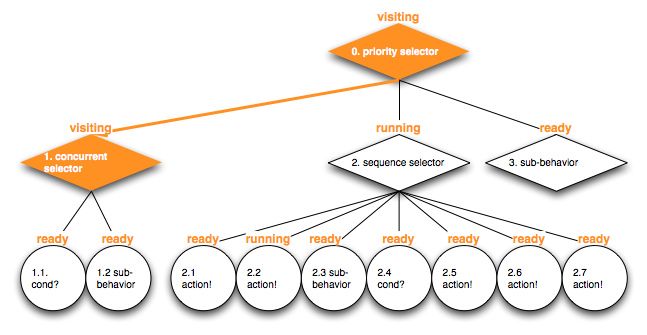
All non-running nodes are set to be *ready* before the next traversal.

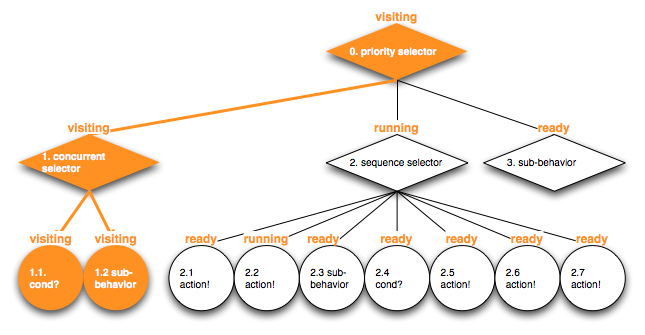
[](http://www.flickr.com/photos/42150256@N02/5472733716)

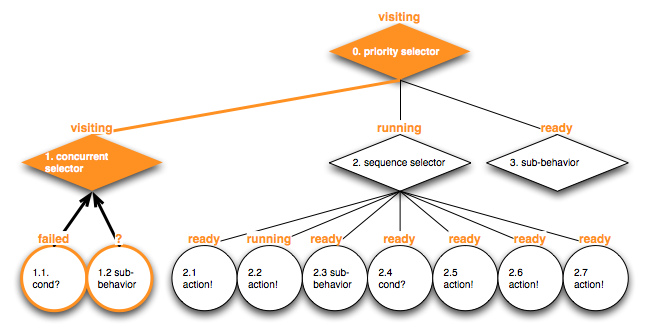
During the next simulation step the behavior tree is updated again and traversal restarts at the root node.

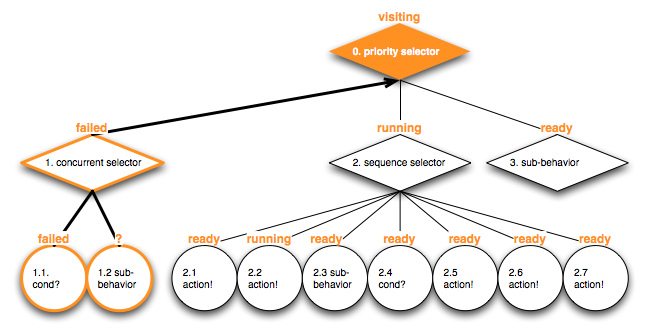
[](http://www.flickr.com/photos/42150256@N02/5472734060)

It checks its first child node which subsequently still fails because the dragon can’t see a thief nearby its treasure.

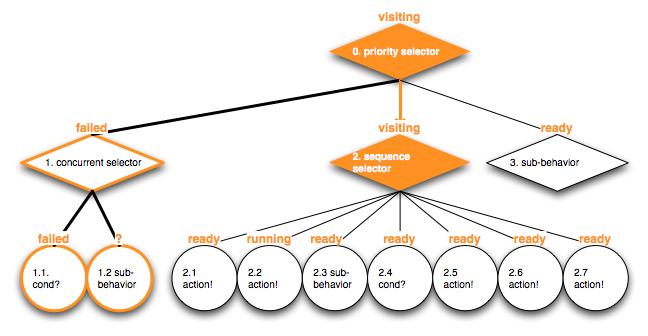
[](http://www.flickr.com/photos/42150256@N02/5472734424)

[](http://www.flickr.com/photos/42150256@N02/5472734816)

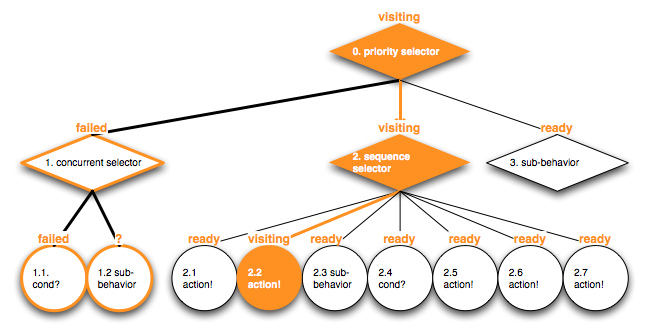
[](http://www.flickr.com/photos/42150256@N02/5472143445)

[](http://www.flickr.com/photos/42150256@N02/5472143815)

Therefore traversal reaches the next priority node child *Rob more treasures* which hasn’t reset its *running* state and still stores which of its children was the last one executing because no higher-priority sibling has run so it wasn’t reseted.

[](http://www.flickr.com/photos/42150256@N02/5472736120)

The *Rob more treasures* sequence node has stored that it last visited its child action 2.2. *Fly to castle!* – which might succeed, or keep running, or fails during this update step.

[](http://www.flickr.com/photos/42150256@N02/5472736498)

I’ll stop bombarding you with behavior tree traversal diagrams for now.

Behavior trees vs. finite state machines – fight!

Why replace finite state machines (FSMs) with behavior trees?

The transitions between FSM states give a finite state machine creator great freedom – and enough rope to grow into an intangible mess, as they proliferate and are harder and harder to follow and understand. Hierarchical finite state machines (HFSMs) help a bit in this regard though.

However, the main point where behavior trees shine, is in their clear – even if a bit restricted – vocabulary in comparison to FSMs. For me it is easier to focus on what I want to express while I often feel a bit lost and uncomfortable where to start with complex FSMs. Constraints are good in this case – they function like guidlines for my thinking.

Selectors force me to think and view problems in a very specific way. Ad-hoc transition structures of finite state machines become explicit.

In addition, conditions and actions as atomic building blocks plug-able into selectors, guide me to make them more reusable. An action or even a whole behavior sub-tree can appear at multiple places in a tree. Sspecial conditions can be grouped with them and guard behaviors to fit into the specific place in the tree. If there are actions to fight enemies with impressive combat styles, then by combining them with conditions, the actions can “rely” on the actor being near the enemies and on having enough room to use the specific combat style.

Alex Champamdard often makes the point to think about actions, behavior trees and sub-trees as goals to reach. The tree is formed around these goals to check that they are reachable, to run through the right steps to approach them, and to ensure that there aren’t other, higher priority goals, that need the actors immediate attention.

Because of their explicit structured, behavior trees also enable the memory organization I am working on, and therefore allow a very efficient traversal – something I can’t envision with FSMs right now (though I haven’t thought about it in depth).

That’s it – what’s next?

So much for this post. I hope that I could give you a good introduction of what behavior trees are and how they conceptually work. In my next blog post I will motivate why a memory streamlined behavior tree representation makes sense, how I envision such a data-oriented behavior tree to look like, and, I will start with the first test-driven implementation of behavior tree functionality.