Unifying Behavior Trees and Logic Programming[⋆]

Samuel Hill[1,†], Ian Horswill[1,†]

[1] Northwestern University

Abstract

It has been shown in previous studies that logic programming can be used to create performant large-scale character simulations. In these simulations the action-selection mechanism for characters either uses a simple decision tree or some utility-based action selection such as needs-based AI from The Sims or the simple action selection found in Talk of the Town (TotT).

While this is useful for one-shot action selection that occurs entirely within a single game tick, modern games often have need for more advanced action selection. Finite-state machines are sometimes used for this task, but behavior trees, their hierarchical cousin, are more commonly used today. Behavior trees are a popular technique for action selection in games but manually implementing them in logic programming is a cumbersome process and the resulting code is difficult to read.

In this paper, we show that standard behavior trees can be efficiently compiled into TED, a bottom-up logic programming language, in a way that allows the two systems to inspect one another's internal state. For example, triggering conditions for BT nodes can be logical assertions, and logical rules can be conditioned on the states of behavior tree nodes. The compiled version is performant and supports parallel execution. We demonstrate this on a simulation involving hundreds of agents.

Keywords

Behavior Trees, Logic Programming, Parallel Execution [1]

1. Introduction

It has been argued that logic programming is an immensely useful paradigm for certain styles of games such as large-scale character simulations as well as certain tasks such as simulating social practices [1] and forward simulation of character actions [2], [3]. Previous research on logic programming for LSCS has used utility-based action selection, both in the style of Ryan's Talk of the Town [3] and The Sims' needs-based AI [4]. These are stateless mechanisms that make techniques such as subgoaling difficult to implement. However, many games today use behavior trees (BTs) to implement more stateful action selection for NPCs, buildings, and other agentive forces.

First introduced with *Halo 2* [5], further popularized by its use in *Spore*, and brought to the forefront of game development AI techniques with Alex Champanard's tutorial on their use [6]. For the past two decades BTs have been both heavily used by industry and researched in-depth by the AI community. While other techniques exist such as GOAP or reactive planners are also popular, BTs are arguably the dominant action selection system in modern game development [6], [7].

Even though BTs are already rather declarative – often being described visually with graphical inputs rather than in written code – there hasn't been any work (that we are aware of) on using behavior trees in logic programming languages. Although one might make an existing BT system work with an existing logic program via a foreign-function interface, a more natural integration that allowed them to freely interoperate would be much more useful, at least for the logic programming system.

In this paper we describe a methodology to compile BTs into a logic programming language and demonstrate their performance in a simulation involving hundreds of agents. The goal is to allow BTs to interoperate with the base language (in this case Simulog/TED) with large numbers of

[⋆] *Joint AIIDE Workshop on Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies, November 10-11, 2025, Edmonton, Canada.*
[1†] These authors contributed equally.

samuelhill2022@u.northwestern.edu (S. Hill); ian@northwestern.edu (I. Horswill)

characters with a frame rate of at least 60fps. An interesting property of our implementation is that it supports multi-core execution, making it an appealing choice for modern processors.

2. Related Work

Many games have used behavior trees, from major titles such as *Halo 2* [5], and *Tom Clancy's The Division* [8], to indie titles such as *Project Zomboid* (written in Java using JBT) [9]. Games such as *Alien: Isolation* have open source tools to edit their BTs on GitHub [10] which is itself based on an tool called Brainiac Designer for visualizing and editing BT's generally [11]. Major game engines now offer BT tools directly with Unity Behavior [12] and Unreal's AI Controller Behavior Tree option [13].

Several surveys of BTs have been written to try and explain how they are used and what capabilities and techniques are yet to be as heavily adopted by commercial games (something Kim McQuillian dubs the Commercial-Academic Gap) [6], [7], [14]. These surveys cover a gamut of usages for BTs from textbook NPC AI [15] to more novel systems like the Interactive Behavior Trees created for interactive narratives [16]. As far as we can tell, there has not been any work on the usage of BTs in a logic programming language.

There has been some use of logic programming in game development. A proof-of-concept game was developed recently that used Prolog and an ontology coded in OWL to play Wumpus [17]. Nelson's *Inform 7* language [18], [19], which is the current most popular IF language, can be seen as a hybrid declarative/procedural language combined with an upper ontology specialized to interactive fiction. *MKULTRA* [20] was written primarily in Prolog, save for the graphics and UI code. The Lume system [21] was also Prolog-based. Versu [1] used a custom logic programming language, Praxis [22], as did *City of Gangsters* [23]. TED [2], the target language into which Simulog compiles, was used in *Rise of Industry 2* [24].

While not technically logic programming, several games have used production systems or other rule-based systems. These include *The Sims 3* [25] used a rule-based system to script the interactions between situations, personality traits, and actions available to a given character [26]. Several other systems have used forward-chaining rule-based systems such as *Comme Il Faut* [27], [28], the social simulation engine (implemented in JavaScript) upon which *Prom Week* [29] was built, and the *Ensemble Engine* [30], *CiF's* successor. *Façade's* [31], [32] internal working memory included a forward-chaining production system.

3. Implementation

To implement Behavior Trees in a logic programming language, we chose Simulog/TED as our base. TED is a version of Datalog that is written in C#. Simulog is a language built on-top of TED that facilitates the management of temporal entities (people, places, events, etc.). These languages are embedded in the C# language, and so the code fragments that follow will have the syntax of C#. Because TED and other Datalog-like languages implement predicates internally as tables, we will often use the terms "predicate" and "table" interchangeably, adopting whichever term is clearest in a given context.

In this section we will describe the system and how it compiles into TED code. There are many implementations of BTs, supporting a variety of APIs, naming conventions, and expectations/contracts. We will follow Champandards' [33] terminology and basic functionality. Our system implements behaviors, actions, composites, selectors, sequences, and filters. This is a subset of the components commonly found in modern systems – missing explicit implementations for return statuses, conditions, decorators, parallels, monitors, or active selectors – but we believe this is sufficient to demonstrate the feasibility of compiling to a logic programming language.

3.1. Behaviors

All nodes inside of the behavior tree are called Behaviors and are of the same C# base type. This ensures all nodes have a name for debugging purposes, a field to indicate when they want to run (used by actions – leaf nodes – to indicate when they should be selected and by composites to act as filters/preconditions), and a generate function that is used to build the tree. This base type also allows us to create lists of child nodes more simply in C#'s strong typing. Behaviors are not a component themselves, acting only as the unifying base type for all nodes in our BTs.

3.2. Composite nodes

Internal nodes have children and so are referred to as composite nodes. Composite nodes share a base type with functionality to manage child nodes. These features include the set of predicates used to manage a node (discussed shortly), a list of child nodes, and functions which compile the behavior tree to logic programming code. These generation functions are virtual: different kinds of composite nodes have different generators.

The set of tables (predicates) that all internal nodes have are:

- Select[agent, child], which holds when agent has selected node child.
- NeedToSelect[agent], which holds when agent needs to select a child of this node.
- FailedToSelect[agent], which holds when agent tried to select a child of this node but could not find an applicable one.
- FailedToSelectDelayed[agent], which holds when agent failed to select on the *previous* tick.

Evaluation of the tree (selection) in a given clock tick proceeds in parallel, from root to leaves. However, if a node terminates or fails on a given tick reevaluation must then be delayed to the following tick – hence the need for the FailedToSelectDelayed predicate.

The logic for our Select predicate is established inside of the generate functions, but some logic is static across all composites. The logic governing FailedToSelect is simply:

```
FailedToSelect.If(NeedsToSelect[who], !Select[who, __]);
```

which says that this node failed to select another node if it needed to select a node for someone, but a node was not selected. For connections to other composite nodes our NeedToSelect predicate is governed by two rules; one is:

```
child.NeedsToSelect.If(this.Select[who, child]);
```

which says that a child nodes NeedsToSelect predicate is true for the agents (who) that were Selected for that child. The other rule is:

```
parent.NeedsToSelect.If(child.FailedToSelectDelayed[who]);
```

which says that a parent node NeedsToSelect for agents for whom a child failed on the previous tick.

3.2.1. Selector

Selectors are a standard type of node implemented by nearly all behavior tree systems. For a given agent, they choose the first child node that WantToRun for that agent. This generated code depends on the number of children but roughly looks like the following:

```
Select.If(NeedsToSelect[who], FirstOf[And[child1.WantToRun, btNode==child1],
                                      And[child2.WantToRun, btNode==child2], ...]);
```

which helps make clear why one wants a compiler to generate this rather than writing it by hand.

To create selector nodes, we can use a constructor of the form:

```
OurSelector = Selector("OurSelector", parentNode);
```

which makes a new selector node called OurSelector that is a child of parentNode. We can also define a WantToRun goal condition for this node by either saying

```
OurSelector.ChooseIf(wantToRunGoal);
```

or by changing the constructor (which is what we use for selector nodes in the examples that follow)

```
OurSelector = Selector("OurSelector", parentNode, wantToRunGoal);
```

3.2.2. Sequence

Sequence nodes are another type of node implemented by nearly all behavior tree systems. When selected, they run their children in sequence, first to last. Sequence nodes in our implementation maintain their state in the AgentSequenceIndex which specifies what child in this sequence each agent is currently running. The nodes Selection rule is defined by:

```
Select[who, child].If(NeedsToSelect[who], AgentSequenceIndex[who, index - 1]);
```

where index is coming from the loop over the child nodes, and index - 1 is referring to the previous node in sequence. This assigns an individual to the next node in sequence if that agent needs to select for this node. The state is maintained with the following logic:

```
AgentSequenceIndex.Add[who, index].If(Select[who, child]);
```

which updates the agents assigned index as dictated by Select (Add here has an overwrite flag enabled so it will update the keyed who instead of adding another row). There is additional logic to handle those who are not yet in sequence and those that have already been in sequence, but the description of how these problems are handled would be cumbersome.

Of note is the fact that the conditions for progressing in the sequence are determined by both the AgentSequenceIndex and NeedsToSelect tables. This may be somewhat confusing, but because of the rules that both composite nodes (which you have seen) and action nodes (found in the next section) have regarding NeedsToSelect, the above logic is sufficient for a sequence node. When a child node either finishes or fails to select, the next node in sequence will be selected.

Sequence nodes are created like selector nodes, e.g. OurSequence = Sequence("OurSequence", parentNode);, with an optional WantToRun field that can be filled in, again, either directly or with the matching constructor.

3.2.3. Filters

As mentioned earlier, any composite node can have preconditions added by setting the WantToRun field to some goal that acts as a filter. Composites, however, are a base type and do not have their own generate function as they are not meant to be used directly. Due to this fact, only

Selector and Sequence nodes can be filtered although any future composite nodes would be able to as well (provided their generate code uses it).

3.3. Actions

Actions (i.e. leaf nodes) are governed by a set of predicates that indicate when something is Active as well as when it has Started and Finished. The logic for choosing an action node comes from its Active predicate, with rules similar to the connection between composite nodes:

```
action.Active.Add[who, true].If(Select[who, action], action.WantToRun);
```

which says that an action should become active if it was selected and it wants to run, and:

```
FailedToSelect.If(Select[who, action], !action.Active.Add[who, true]);
```

which says that an action was failed to be selected if it was selected but not activated. Actions (leaf nodes) can indicate that they want to run (i.e. that some condition has been met that indicates whether this node should continue processing) like all nodes, but this is needed for leaf nodes of selectors.

While the selection logic is generated automatically based on the connecting parents and WantToRun conditions, the logic governing when an action is finished must be described by declaring its termination conditions using action.DoneWhen(conditions) or by adding conditions manually via action.Finished.If(conditions) rules. This finished predicate is then used to pass agents back up the node's parent's need to select predicate: parent.NeedsToSelect.If(child.Finished);. One can also end an action without finishing – e.g. in the event of the character dying part way through some action –action.AbortWhen(conditions); to both stop the action and not percolate back up the tree.

The above logic is sufficient to encode when an action should start and end, but it does not directly use the Started predicate mentioned earlier. Instead, the Started predicate's conditions detect when someone has been added (or re-added) to an action node. In effect, this means that the Started predicate is just a signal for when an action has started that doesn't require monitoring the status of the Active predicate manually.

Actions are created similarly to other node types, e.g. OurAction = Action("OurAction", parentNode, wantToRunGoal).

3.4. Root

The root node of a BT is a special case selector node in our implementation, adding initialization logic and logic for introducing new agents to the tree as they're created. Root nodes are created by the behavior tree constructor: OurBT = BehaviorTree("OurBT", agent, btNode); – where both agent and btNode are a Simulog variables of the types Var<AgentType> and Var<Behavior<AgentType>> respectively – which returns a root node that can be assigned as the parent of the next level of our tree. In addition to the plain constructor mentioned above, there is a constructor that allows you to assign a condition that will be used to populate the tree initially:

```
OurBT = BehaviorTree("OurBT", agent, btNode, initializeCondition);
```

In addition to being able to initialize a BT with agents for tick 1 of the simulation, agents can be loaded into the simulation as it runs by setting the load conditions OurBT.Load.If(loadingCondition).

To finish the creation of a BT, simply call OurBT.End() after all nodes related to this tree have been created and their logic established. This call navigates the tree that has been created and generates the connecting logic that has been mentioned in previous sections automatically, compiling the BT into logical predicates. This architecture for generating the BT is reminiscent of

Alex Champandard's simple example BehaviorTreeBuilder [33, p. 74] but with the added flexibility of being able to break up the creation of the BT with other relevant logic or game code.

4. Code Examples

The following is the simple behavior tree that governs plant growth in our testbed simulation of dwarves farming and harvesting plants. The first portion is Simulog code that establishes the existence of plants, creating new plants each tick until they have a population of 200 and ending them when they have been Harvested. The logic behind harvesting is controlled by another BT in this simulation for the dwarfs.

```
Plants = Exists("Plants", plant);
Plants.StartWhen(Plants.Population[count], count < 200, NewPlant[NamePlant, plant]);
Harvested = Predicate("Harvested", plant);
Plants.EndWhen(Harvested[plant]);
```

The second portion defines a BT that is initialized with the initial Plants population and that has one child node of the root – our growth sequence node.

```
PlantGrowth = BehaviorTree("PlantGrowth", plant, plantBTNode, Plants.Initially[plant]);
PlantGrowth.Load.If(Plants.Start[plant]);
GrowthSequence = Sequence("GrowthSequence", PlantGrowth, True);
```

The next portions define actions for each of the 4 stages of growth, each with random durations per plant, and the ability to be harvested at any point in the progression. Due to the actions in this sequence all being essentially run by some timer that determine how long to wait before growing – and because this simple action type was a common enough pattern – we created a special case action type called TimerAction that manages the Finished conditions by internally using Simulog timers. The timer predicate only needs a clock tick delta (in this case a floating point representing the number of seconds in real time) to be assigned in the timer goal (the third argument in its constructor).

```
SproutingStage = TimerAction("SproutingStage ", GrowthSequence,
                              RandomFloatInRange[3.0f, 10.0f, clockTicks])
                   .AbortWhen(Harvested[plant]);
SeedlingStage = TimerAction("SeedlingStage ", GrowthSequence,
                              RandomFloatInRange[5.0f, 15.0f, clockTicks])
                  .AbortWhen(Harvested[plant]);
SaplingStage = TimerAction("SaplingStage", GrowthSequence,
                              RandomFloatInRange[10.0f, 20.0f, clockTicks])
                 .AbortWhen(Harvested[plant]);
```

From here we get to the final growth stage and the end of this BT. The adult tree growth stage does not have a timer or finished condition as it is intended for plants to stay at this size until harvest.

```
AdultTreeStage = Action("AdultTreeStage", GrowthSequence).AbortWhen(Harvested[plant]);
PlantGrowth.End();
```

With this we have the logic for plants existing, growing, being able to be harvested at any point in that growth, and because of the harvest, stop existing. The code governing growth is managed by the BT which is only 11 lines of code when formatted normally in an IDE with 120 characters per line.

In the same simulation we have another BT that governs the actions of our dwarfs (NPC sims a-la Dwarf Fortress) which is primarily built around a selector node – as opposed to the sequence-oriented BT for plant growth. While the root node is a selector node and as such, we could add all actions described in the following sections directly to this node, we instead build these actions off another selector node that is the child of our root. The first portion of our BT establishes the root and a child much the same as the sequence example from earlier. Not included are the Dwarfs existent and the logic regarding their start and end.

```
Root = BehaviorTree(dwarf, dwarfBTNode, Dwarfs.Initially[dwarf]);
Root.Load.If(Dwarfs.Start[dwarf]);
CoreSelector = Selector("CoreSelector", Root, True);
```

Next, we have the harvest action which selects dwarves to go harvest plants if there are more than 5 unclaimed plants (UnclaimedPlantsCount logic is not shown, but it is based on a set of predicates that manage which plants have someone going to harvest them, i.e. are claimed). As well, the action is finished when the dwarf has finished harvesting the plant (again, not shown is the logic for FinishedHarvesting, but it is based on a set of predicates that manage when the dwarf arrives at a plant and harvests it). The Harvested predicate from the plant growth BT is also established here.

```
Harvest = Action("Harvest", CoreSelector)
        .ChooseIf(UnclaimedPlantsCount[result], result > 5)
        .DoneWhen(FinishedHarvesting[dwarf, plant]);
Harvested.If(FinishedHarvesting[dwarf, plant]);
```

With logic that is very similar to the harvest action, we have the work in shop action. This node selects dwarves to go work in an unclaimed workshop as needed, and to finish with this action once they have finished working in the workshop.

```
WorkInShop = Action ("WorkInShop", CoreSelector)
        .ChooseIf(UnclaimedWorkshopsCount[result], result > 1)
        .DoneWhen(FinishedWorkingInShop[dwarf, workshop]);
```

Finally, we have a very simple idle action that acts as a default fallback when harvesting and working in a shop are both unavailable. Dwarves stop idling with a 30% chance each tick, at which point they are sent back to CoreSelector to try and assign them to harvest or work in a shop again (only falling back to idle if those didn't want to run).

```
Idle = Action ("Idle", CoreSelector, True).DoneWhen(Prob[0.3f]);
Root.End();
```

This establishes the core logic for action selection in our dwarves, with interoperation between Simulog predicates that govern existence and between the two BTs with predicates that each can use in its conditions. The code that we have shown for the dwarves BT is only 12 lines, and while not including a handful of predicates that manage the claiming of resources and the movement to those resources, this is relatively compact and (in our view) a rather natural way of defining BTs.

5. Performance

We made a simple simulation to demonstrate a few hundred agents of various types all running behavior trees to determine their actions. In these simulations we have 1000 dwarves (a la Dwarf

Fortress) moving around a map, cutting down trees from a selection of 200 at any given time, and working at some of the 50 workshops that get built as resources are harvested. Both the dwarves and the trees are running BTs to do action selection. When running uncompiled and in the unity editor on a M3 Pro MacBook Pro we are still able to achieve an average frame time of 12.86ms for the TED code, only a portion of which is the BT execution time as can be seen in Figure 1.

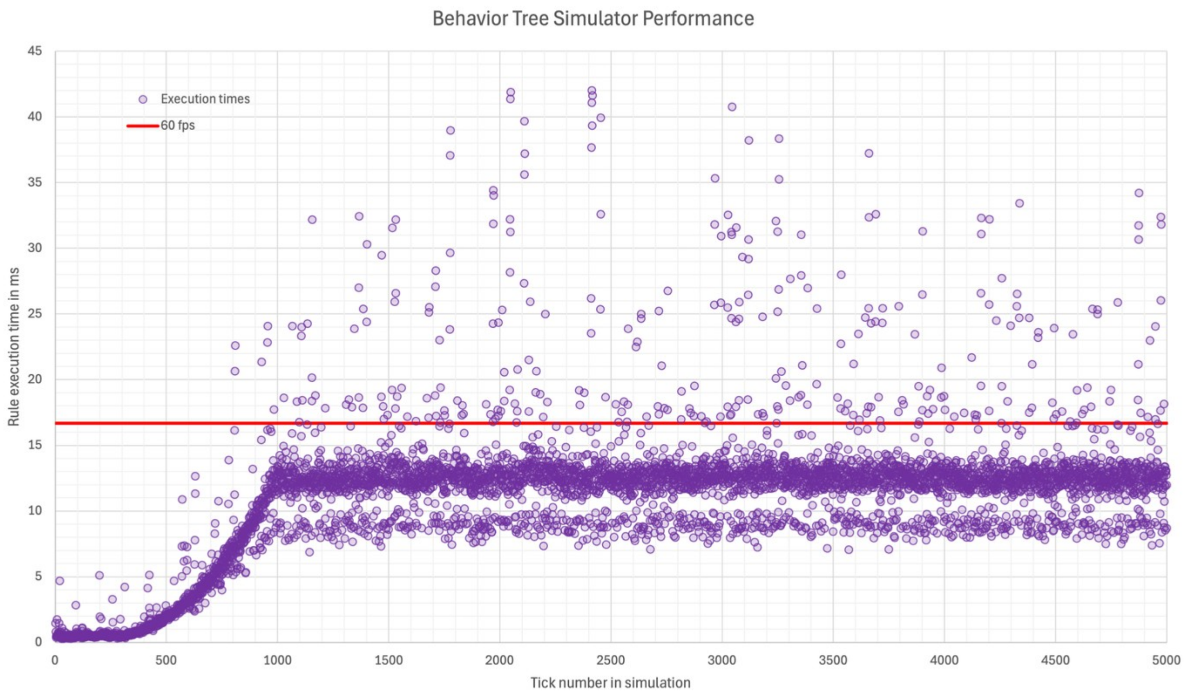

Figure 1: Behavior Tree Simulator performance in the first 5000 ticks.

This result of an average of 77 fps is only if there were no additional CPU tasks for things like moving the agents on screen, but even with such code running (and having done no optimization passes on these portions of the code) we still achieve 20.48ms average frame time or approximately 50 fps. Given the performance increases that we have seen from compiling TED alone, we have little doubt that the system we have demonstrated today could run faster, with more characters, and/or with more complexity in the action selection. Not only does this run 1000+ characters performantly, but the code that created this BT and manages all of the logic for it is only 80 lines.

6. Limitations

The main limitations that we have found with this system are to do with tick delays. There is a one tick delay when you fail on some action, which is inherent to the way a non-recursive datalog works. The FailedToSelect predicate is updated based on the values from NeedToSelect but would itself like to add to that predicate.

As well, upward propagation through the tree happens only at one level per tick. This is due to the delay added on FailedToSelect, but we believe that this could be eliminated by using static analysis to determine which nodes to fail up to. A simple rule like fail up to the first sequence node and if none then the root, would allow for one tick propagation to any level – provided it is not nested in sequence nodes.

7. Future Work

Our main addition that we have planned is to add a means for one tick traversal back up the BT. This was mentioned in the limitations section, and we believe that the issue of waiting multiple ticks to percolate back up the BT is solvable with some simple static analysis. Additionally, we would like to add support for other node types such as decorators or monitors.

8. Conclusion

We have shown that behavior trees can be naturally compiled into logic programming languages in a natural way. Although not a transformation one would want to do manually, it allows declarative specification of node behavior using logical queries and allows the logic program to interrogate the state of the behavior tree in an equally declarative manner.

The system is quite performant, allowing hundreds of agents to be simulated simultaneously. Moreover, it transparently supports multicore execution, allowing it to take advantage of modern CPU architecture without the programmer having to think about locking or scheduling issues.

The result is a system that makes it relatively easy to build fast, large-scale simulations using very compact, declarative specifications.

9. Declaration on Generative AI

No GenAI tools or services were used in the preparation of this work (neither in the code nor the writing).

References

[1] R. Evans and E. Short, "Versu—A Simulationist Storytelling System," *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 2, pp. 113–130, Jun. 2014, doi: 10.1109/TCIAIG.2013.2287297.

[2] I. Horswill and S. Hill, "Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2024. doi: 10.1609/aiide.v20i1.31866.

[3] S. Hill and I. Horswill, "An Executable Ontology for Social Simulation," in *Workshop on Experimental Games in AI (EXAG-23)*, University of Utah, Utah, USA, Oct. 2023.

[4] S. Hill and I. Horswill, "5000 Characters at Video Frame Rate using Declarative Programming," *CEUR Workshop Proc.*, vol. 3926, 2024.

[5] D. Isla, "Handling Complexity in the Halo 2 AI," Game Developer's Conference 2005. Accessed: Aug. 25, 2025. [Online]. Available: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

[6] K. McQuillan, "A Survey of Behaviour Trees and their Applications for Game AI," Jun. 2017, Accessed: Aug. 25, 2025. [Online]. Available: https://www.academia.edu/33601149/A_Survey_of_Behaviour_Trees_and_their_Applications_for_Game_AI_A_Survey_of_Behaviour_Trees_and_their_Applications_for_Game_AI

[7] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A Survey of Behavior Trees in Robotics and AI," 2020, *arXiv*. doi: 10.48550/ARXIV.2005.05842.

[8] J. Gillberg, "AI Behavior Editing and Debugging in 'Tom Clancy's The Division.'" Accessed: Aug. 25, 2025. [Online]. Available: https://gdcvault.com/play/1023382/AI-Behavior-Editing-and-Debugging/

[9] C. Simpson, "Behavior trees for AI: How they work." Accessed: Aug. 25, 2025. [Online]. Available: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

[10] M. Filer, *OpenCAGE/BehaviourTreeEditor*. (Aug. 25, 2025). C#. OpenCAGE. Accessed: Aug. 25, 2025. [Online]. Available: https://github.com/OpenCAGE/BehaviourTreeEditor

[11] learno, *learno/Brainiac-Designer*. (May 30, 2024). C#. Accessed: Aug. 25, 2025. [Online]. Available: https://github.com/learno/Brainiac-Designer

[12] "About Unity Behavior | Behavior | 1.0.12." Accessed: Aug. 25, 2025. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.behavior@1.0/manual/index.html

[13] "Behavior Tree in Unreal Engine - Quick Start Guide | Unreal Engine 5.6 Documentation | Epic Developer Community," Epic Games Developer. Accessed: Aug. 25, 2025. [Online]. Available:

https://dev.epicgames.com/documentation/en-us/unreal-engine/behavior-tree-in-unreal-engine---quick-start-guide
[14] W. Zijie, W. Tongyu, and G. Hang, “A Survey: Development and Application of Behavior Trees,” in *Communications, Signal Processing, and Systems*, vol. 654, Q. Liang, W. Wang, X. Liu, Z. Na, X. Li, and B. Zhang, Eds., in Lecture Notes in Electrical Engineering, vol. 654. , Singapore: Springer Singapore, 2021, pp. 1581–1589. doi: 10.1007/978-981-15-8411-4_208.
[15] I. Millington and J. D. Funge, *Artificial intelligence for games*, 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2009.
[16] M. Kapadia, J. Falk, F. Zünd, M. Marti, R. W. Sumner, and M. Gross, “Computer-assisted authoring of interactive narratives,” in *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, San Francisco California: ACM, Feb. 2015, pp. 85–92. doi: 10.1145/2699276.2699279.
[17] S. Lapeyrade and C. Rey, “Non-Player Character Decision-Making With Prolog and Ontologies,” presented at the IEEE Conference on Games, IEEE Press, 2023. doi: DOI: 10.1109/COG57401.2023.10333221.
[18] G. Nelson, “NATURAL LANGUAGE, SEMANTIC ANALYSIS AND INTERACTIVE FICTION,” 2006.
[19] G. Nelson, *Inform 7.* (2006).
[20] I. Horswill, “Postmortem: MKULTRA, An Experimental AI-Based Game,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 14, no. 1, Art. no. 1, Sep. 2018, doi: 10.1609/aiide.v14i1.13027.
[21] S. Mason, C. Stagg, and N. Wardrip-Fruin, “Lume: A System for Procedural Story Generation,” in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, San Luis Obispo California USA: ACM, Aug. 2019, pp. 1–9. doi: 10.1145/3337722.3337759.
[22] R. Evans, “Introducing Exclusion Logic as a Deontic Logic,” in *Deontic Logic in Computer Science*, vol. 6181, G. Governatori and G. Sartor, Eds., in Lecture Notes in Computer Science, vol. 6181. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–195. doi: 10.1007/978-3-642-14183-6_14.
[23] SomaSim, *City of Gangsters.* (2021). Chicago.
[24] M. Viglione, *Rise of Industry 2.* (2025). SomaSim, LLC.
[25] Maxis, *The Sims 3.* (2009). Maxis.
[26] R. Evans, “AI challenges in Sims 3,” *Artif. Intell. Interact. Digit. Entertain.*, 2009.
[27] J. McCoy, M. Treanor, B. Samuel, N. Wardrip-Fruin, and M. Mateas, “Comme il Faut: A System for Authoring Playable Social Models,” *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 7, no. 1, pp. 158–163, Oct. 2011, doi: 10.1609/aiide.v7i1.12454.
[28] J. McCoy, M. Treanor, B. Samuel, B. Tearse, M. Mateas, and N. Wardrip-Fruin, “Comme il Faut 2: a fully realized model for socially-oriented gameplay,” in *Proceedings of the Intelligent Narrative Technologies III Workshop*, Monterey California: ACM, Jun. 2010, pp. 1–8. doi: 10.1145/1822309.1822319.
[29] J. McCoy, M. Treanor, B. Samuel, A. A. Reed, N. Wardrip-Fruin, and M. Mateas, “Prom week,” in *Proceedings of the International Conference on the Foundations of Digital Games*, in FDG ’12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 235–237. doi: 10.1145/2282338.2282384.
[30] B. Samuel, A. A. Reed, P. Maddaloni, M. Mateas, and N. Wardrip-Fruin, “The Ensemble Engine: Next-Generation Social Physics,” in *Proceedings of the Tenth International Conference on the Foundations of Digital Games*, Jun. 2015.
[31] M. Mateas and A. Stern, *Façade.* (2005). [Online]. Available: https://www.playablstudios.com/facade
[32] M. Mateas and A. Stern, “Façade: An Experiment in Building a Fully-Realized Interactive Drama,” 2003.
[33] A. J. Champandard and P. Dunstan, “Chapter 6: The Behavior Tree Starter Kit,” in *Game AI Pro: Collected Wisdom of Game AI Professionals*, 0 ed., S. Rabin, Ed., A K Peters/CRC Press, 2013, pp. 73–91. doi: 10.1201/b16725.