International Journal on Artificial Intelligence Tools
Vol. 26, No. 2 (2017) 1730001 (28 pages)

DOI: 10.1142/S0218213017300010

Behavior Trees for Computer Games

Yoones A. Sekhavat

Faculty of Multimedia, Tabriz Islamic Art University
Hakim Nezami Square, Azadi Blvd, Tabriz, 51647-36931, Iran
sekhavat@tabriziau.ac.ir

Received 9 August 2016
Accepted 17 November 2016
Published 12 April 2017

Although a Finite State Machine (FSM) is easy to implement the behaviors of None-Player Characters (NPC) in computer games, it is difficult to maintain and control the behaviors with increasing the number of states. Alternatively, Behavior Tree (BT), which is a tree of hierarchical nodes to control the flow of decision making, is widely used in computer games to address the scalability issues. This paper reviews the structure and semantics of BTs in computer games. Different techniques to automatically learn and build BTs as well as strengths and weaknesses of these techniques are discussed. This paper provides a taxonomy of BT features and shows to what extent these features are taken into account in computer games. Finally, the paper shows how BTs are used in practice in the gaming industry.

Keywords: Behavior tree; computer games; decision making; non-player character.

1. Introduction

Activities associated with Artificial Intelligence (AI) in computer games are designed for behaviors of Non-Player Characters (NPC). Defining a logic for various entity characters in computer games is an important issue in the design of a game that specifies how these characters behave and react against different conditions. Providing an easy to use AI component that is general enough to provide a game designer the ability to specify complex behaviors is crucial.

Rule-based systems are the simplest and fairly limited to control the behavior, in which the behavior of a NPC is a simple function of present conditions. Generally, a rule-based system consists of a knowledge-base and a set of if-then rules available to an AI agent. Rules are examined against knowledge-base to check if conditions are met. Rules are triggered and executed when the conditions are met.[1] For example,

the motion of a ghost in Pac-Man[a] game can be modeled by a rule-based system, in which the agent goes ahead whenever possible. When the way is blocked, the agent selects actions turning right, turning left, and reversing. In backward chaining rule-based systems, the agent starts with a goal in the knowledge-base and tries to fire a set of rules such that these rules lead to that goal. Although this system is easy to understand and implement, it is almost impossible to model complex behaviors using rule-based systems.

Finite State Machines (FSM) by adding a notion of *state* make it possible to handle the complexity of behaviors. FSMs are generally used to represent the behaviors of a NPC in terms of a set of states and transitions between states. A FSM is a directed graph in which each node represents a state. A NPC can only be in one of the states. An edge in this graph corresponds to an event that triggers a change from one state to another state. Unlike rule-based systems, the behavior of an NPC is a function of the current state of the NPC as well as the present conditions. Scalability is the main problem of FSM. Today's games are complex games that require a large number of states. Although this problem can be partially mitigated using a hierarchical FSM (a.k.a., HFSM), it is still difficult to manage a large number of states.[2] Efforts to use sophisticated decision making techniques such as fuzzy logic and neural networks have not resulted in considerable successes as these techniques are hard to get them working right.[1]

Behavior Tree (BT) is a practical scalable solution to implement decision making, which is a hierarchical structure of behaviors. An agent running a behavior tree performs a depth-first search on the tree to find and run the lowest level leaf node. Scalability, expressiveness and extensibility are the main advantages of BT in comparison to FSM.[3] Unlike explicit transitions from one state to another (like FSM), each node of a tree indicates how to run its children. In a FSM, every state must know the transition criteria for every other state, while BTs are stateless. As a result, there is no need to know previously running nodes to decide what behaviors should be executed next. The expressiveness of BTs come from the use of different levels of abstraction, implicit transitions and arbitrarily complex control structures for composite nodes. In FSM, once a switch from one state to another state is conducted, since the source state is not saved, it is a one-way control transfer. On the other hand, in BTs, transitions between states are achieved through calls and return values passed between tree nodes, which is known as a two-way control transfer.

BTs are extensible as it is possible to start from a base algorithm and add extra functionalities. Making changes to a behavior trees is much easier than FSM. First, since creating and editing behavior trees are performed using a visual editor, maintaining a behavior tree is simply conducted by drag and drop of nodes. Such visual tools make BTs much more accessible and understandable to non-programmers.

[a]http://pacman.com

Table 1. Classification of research issues for behavior trees in computer games including supporting papers in each category.

Basics of BTs	Formal Representation of BTs	Behavior Tree Semantics
	[1], [2], [3], [4], [7], [8], [9], [10], [11]	[13], [14], [15], [16], [17], [18], [19]
Forming BTs	Automatic Techniques	Manual Techniques
	Machine Learning: [20], [21], [22], [23], [24], [25], [26]	Facilitating tools: [36], [37]
	Case-Based Reasoning: [27], [28], [29]	
BT Extension	Modification in the Structure of BTs	Augmenting BTs with Additional Knowledge
	[38], [39], [40], [41], [42], [43], [54], [55], [56], [44], [45], [50], [52], [53], [80], [3], [46]	[47], [48], [49], [58], [51], [57], [59], [60], [61], [28], [78], [79]
BT Application	[29], [62, [63], [3], [64], [65], [66], [68], [69], [21], [70], [59], [71], [10], [72], [73], [74], [75], [14], [76], [77]	

However, adding a new node or changing a state in a FSM requires changing the transitions between states, that is not easy with growing the number of states. Second, unlike FSM that requires considering all states and transitions to make a change in the state of an agent, in a BT, this is performed just by modifying the tasks around or adding a new parent. Moreover, BT is more flexible than FSM. The only way to run two different states with FSM is creating two separate FSMs. However, simultaneous execution of two behaviors can be easily handled in behavior trees using parallel nodes. An AI designer may also want to prevent to run two tasks at the same time. For example, suppose you have two different tasks that play a sound effect and they could potentially play the sound effect at the same time. There is no way in FSM to prevent the simultaneous running of these two sound effects. However, adding guard nodes to behavior trees (a semaphore task that allows only one active instance) makes it possible to play only one sound effect at a time.

This paper reviews various issues regarding BTs in computer games. As shown in Table 1, this paper classifies the study of BTs in four basics categories including *Basics of BTs, Forming BTs, BT Extension* and *BT Application*. In the basics of BTs, formal definitions and semantics of behavior trees in computer games from different perspectives are discussed. In the context of forming behavior trees, automatic and manual techniques to facilitate the forming of behavior trees are reviewed for NPCs in computer games. In terms of BT extension, techniques to modify the structure of behavior trees as well as methods to augment behavior trees with

additional knowledge to increase the expressiveness of behavior trees are discussed. A list of features are proposed by which the strengths and weaknesses of these techniques are compared. Finally, this paper shows how BTs are used in practice to implement decision making in research prototypes and commercial games.

Algorithms designed for sequential decision making problems with multiple goals are discussed in several surveys[4] including probabilistic inference techniques for scalable multi-agent decision making,[5] and decision making with dynamic uncertain events.[6] However, none of these techniques takes into account the specific aspects of decision making for NPCs in computer games.

2. Behavior Tree Basics

BTs make it possible to control characters in computer games and to define hierarchies of decisions and actions.[7] Behavior trees are also crucial to implement believable agents[8] and to control groups of characters in an interactive way.[9] This section discusses the structure and the semantics of behavior trees. Then, a list of features is propped to compare different BTs.

2.1. *Behavior tree structure*

As a classic definition,[10] a Behavior Tree (BT) is a directed tree including a set of nodes and edges. The root of a BT is a node without parents. On the other hand, nodes without children are the leaves of this tree. In a basic BT, a non-leaf node can be a *selector* node or a *sequence* node. Selectors are used when we aim to find and execute the first possible child that can run without failure. A selector node succeeds once any child is performed successfully. In a sequence node, all nodes are evaluated sequentially. A sequence node succeeds only if all children are performed successfully. On the other hand, a leaf node can be an *action* or a *condition*. Action nodes include playing an animation, changing the state of a character, or any activity that changes the state of the game. On the other hand, a condition node is generally used to test some values. Tests for proximity, testing the state of a character and testing the line of sight are examples of condition nodes. A condition returns success if the condition is met; otherwise, returns failure.

An example of a behavior tree including a selector node (i.e., root represented by ?), two sequence node (i.e., two children of the root represented by $\rightarrow$), a condition node (Is door open?) and 4 action nodes (move into room, move to the door, open door and move into room) are shown in Fig. 1. Using the depth-first search to run the behavior tree in this figure, the first node selected to run is "is door open?" if the door is open, it returns success. Then, the sequence task moves on to its next child, which is "move into room". If this action succeeds, the whole tree returns a success and execution is done. In the case if the door is closed, or "move into room" returns failure, the second sequence node is tried staring from the "move into door". The tree will return success if all children of the second sequence node return success.

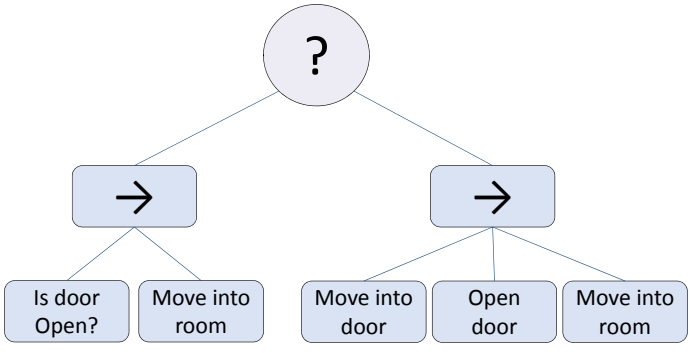

Fig. 1. A behavior tree including selector, sequence, condition and action nodes.

In addition to the core nodes, behavior trees are also extended with *decorator* and *parallel* nodes. Adapted from the "Decorator" design pattern in software engineering, a decorator node in behavior tree is a node that has a single child task. A decorator node can be used for filtering which makes a decision to allow a child behavior to run "until fail". A decorator node may also be used to limit the number of runs. On the other hand, parallel nodes provide some sort of concurrency in BTs. A parallel node is used when there are actions that must be run concurrently with other actions (e.g., playing a move animation while actually moving from one place to another). Some commercial products such as RAIN[b] are also extended with *Priority* node (that allows setting priority both in run time and start time) and *Custom Action* node (that allows defining a decision specific to a game).[11] A node in a behavior tree can be as simple as checking the value of a variable, or as complex as executing an animation. Arbitrary conditions and actions can be combined together to form a behavior tree.

Sharing data between tree nodes is an important feature of BTs that makes it possible to handle complex behaviors. In the case of the lack of data sharing between nodes, we may form a big behavior tree with separate branches for each option, which can be wasteful. The best approach to implement this option is by decoupling the tasks from the data that behaviors need. This can be achieved by employing an external repository to save all data that a behavior tree needs, which is called "blackboard". A blackboard can store any data that can be queried by any task node.

Once a behavior tree is executed, the root of the tree is ticked in each time step of the control that results in propagating down the tree. When this tick reaches a leaf node, some states or variables of the BT are affected. Once a node representing a task is executed, the return value is *success*, *fail* or *running*. The root node branches down to some nodes until leaf nodes are reached.

[b]http://rivaltheory.com

2.2. *Formal model of behavior trees*

Formally, a BT is defined as a triple $T_i = \{f_i, r_i, \Delta t\}$, in which i the index of the tree, f_i:$R_n \rightarrow R_n$ is the right hand side of an ordinary difference equation, Δt is a time step and $r_i : R_n \rightarrow \{Running, Success, Failure\}$ is the return status.[3] The execution of a behavior tree T_i is defined as a standard difference equation:

$$x_{k+t}(t_{k+1}) = f_i(x_k(t_k)), \quad \text{where } t_{k+1} = t_k + \Delta t\,.$$

In this formula, a condition is a node that does not return *running* state. An action behavior in this formula is a behavior tree T_i that has no subtree. A sequence operator allows combining two or more BTs in order to form a complex BT, which is formally defined as $T_0 = Sequence(T_1; T_2)$. To run T_0, the system first tries to run T_1. The second child runs if T_1 returns success. The sequence T_0 returns Success if all children are succeeded. A selector operator, denoted $T_0 = Selector(T_1; T_2)$, is a composition of two BTs such that once T_0 is executed, the system first tries to run child T_1 as long as it returns running or success. T_2 is executed only if T_1 fails. T_0 returns failure if T_1 and T_2 are failed. The node types Sequence, Selector, Action and Condition behave according to Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4, respectively, where $Tick(child(i))$ represent triggering the algorithm, and N is the number of children of that node.

In original sequence node of behavior tree algorithms, for an arbitrary child Action A_j that is ticked at time t_k, it is necessary that $\{X_1(t_k) \in S_1 \wedge \cdots \wedge$

Algorithm 1: An implementation of Selector node

```
for i = 1 to N do
    state = Tick(child(i));
    if state == running then
        return running
    if state == success then
        return success
end
return failure;
```

Algorithm 2: An implementation of Sequence node

```
for i = 1 to N do
    state = Tick(child(i));
    if state == running then
        return running
    if state == failure then
        return failure
end
return success;
```

Algorithm 3: An implementation of Action node

```
if X_n(t) ∈ Success then
    return Success
if X_n(t) ∈ Failure then
    return Failure
if X_n(t) ∈ Running then
    U_n(t) = γn(X_n(t))
    return Running
```

Algorithm 4: An implementation of Condition node

```
if X_n(t) ∈ Success then
    return Success
if X_n(t) ∈ Failure then
    return Failure
```

Algorithm 5: A modified implementation of Selector

```
for i = runIndex to N do
    state = Tick(child(i));
    if state == running then
        runIndex = i;
        return running
    if state == success then
        runIndex = 1;
        return success
end
runIndex = 1;
return failure;
```

$X_{j-1}(t_k) \in S_{j-1}\}$. Accordingly, in a selector node, for an arbitrary child Action A_j that is ticked at time t_k, it is necessary that $\{X_1(t_k) \in F_1 \wedge \cdots \wedge X_{j-1}(t_k) \in F_{j-1}\}$. Although this may be useful in many cases, there are situations in which we need to remember which nodes have already returned success or failure. This is used to not check them (and tick them accordingly) again on the next iteration. Instead of storing success or failure statuses of nodes, a variable representing the child that has most recently returned running can be employed.[12] Every time the selector or sequence returns success or failure, this variable is initialized. Algorithms 5 and 6 show different implementations of sequence and selector nodes, respectively, using this modification.

Behavior trees allow abstraction, where at design time, an AI designer creates basic behavior trees representing simple behaviors. Such basic behaviors can be specializations of more general behaviors developed for one or more NPCs. To

Algorithm 6: A modified implementation of Sequence

```
for i = runIndex to N do
    state = Tick(child(i));
    if state == running then
        runIndex = i; return running
    if state == failure then
        runIndex = 1;
        return failure
end
runIndex = 1; return success;
```


provide re-usability in the design of a tree, behaviors can be parametrized (hard-coded at design time or dynamically at run time). For example, in the "Take" behavior, the taken object can be specified as a parameter to provide re-usability.[60] Since behavior trees represent the behavior of NPCs, an execution context is created for trees at run time. This context includes a set of variables consisting attributes of the game state that can be accessed by NPCs. Different actions of an agent can change these variables, representing the need for having a separate context for each tree assigned to an agent.

2.3. *Behavior tree semantics*

Different semantics and definitions have been used for behavior trees in computer games including the semantics of behavior trees, in which a tree is translated into Communicating Sequential Processes (CSP).[13] However, in the case of complex states, it becomes very complicated. In formal semantics for behavior trees,[14] CSP[15,16] is used as a formal notation to represent interaction between concurrent processes. This new language, which is called CSP_σ is extended with state-based constructs and a message passing facility similar to publish/subscribe models of communication.[17] The semantics in this formal notation is used to develop an automated analysis of system behavior, simulation and model checking. In a formal definition for translating complex requirements to behavior trees,[14] a formal semantics for BTs using CSP is proposed, in which state-based constructs are employed. This technique improves the precision of behavior tree models and results in a better understanding of modeling decisions, while resolving ambiguities and inconsistencies. This technique can also be used in the automated analysis of system behaviors.

In a technique to transform a basic behavior tree model to UML state machines,[18] a framework is proposed that makes it possible to use a rich variety of relations that capture non-functional aspects of requirements as extra annotations to nodes in a behavior tree. This technique provides a tool to go from natural-language system requirements to model driven engineering. In A behavior tree notation to improve the modeling and understanding requirements,[18] rules are stated in natural

Table 2. A summarized list of features of behavior trees extracted in this paper as well as supporting works.

Feature	Details	Supporting Works
Automatic generation	Automatic techniques	[20], [21], [22], [23], [24], [25], [26], [27], [28], [29]
Support utility based AI	Utility based BTs	[38], [39], [40], [41], [42], [43], [54], [55], [56]
Enhanced states	Supporting internal states	[47], [48], [49], [58]
	Persistent states	[58]
Behavior tagging	Behavior tagging	[57]
	Hinted execution BTs	[59]
Social and emotional	Social territoriality	[46]
	Emotional factor	[51]
Re-usability	Modularity	[54], [58]
	Planning integration	[57], [6], [16]
	BT library	[40], [41], [42], [15], [16], [64], [52], [57], [6]
Runtime support	Non-blocking actions	[44],[72], [55], [37], [16], [30], [4], [6]
	Dynamic tree	[7], [16], [35], [21], [65], [4], [15], [6], [30]
	Infinite execution	[44], [6], [72], [16], [30]
	Live editing	[7], [20], [21], [16], [22], [26], [44], [45]
	Dynamic variables	[5], [6], [7]
Flexible Nodes	Node with several parents	[44], [6]
	Component reasoner	[47], [19]
	Timed behavior nodes	[37]
Flexible variables	Global variables	[57], [15], [44], [6], [72], [37], [74], [16], [30], [4], [6]
Multiple agents	Multiple agents	[57], [6], [72], [55], [37], [74], [30], [6]
Enhanced editor	Visual editor	[29], [30]
	Visual debugging	[2], [23]
Parametrization	Agent-centric parametrization	[7], [2], [44], [51]

language. In a process algebra for capturing the constructs of behavior trees,[14] due to the complexity of operational semantics, desired properties such as the composition of parallel behavior trees are not considered. In order to provide a meaning to behavior trees, a language called Behavior Tree Process Algebra (BTPA)[19] is proposed that provides operational semantics, while allowing to define a mechanical translation of behavior trees into BTPA. This allows providing a meaning for synchronization and message passing. However, this architecture provides a small set of primitives from which complex behaviors can be simulated.

2.4. *Feature comparison*

In eleven features for behavior trees,[12] only part of them are supported in different behavior trees. These features can be used to evaluate different behavior trees.[12] Although various behavior tree models are proposed and implemented in the context of computer games, different features of these trees are not generalized to be exploited in other research works. NodeCanvas compared the features of major behavior trees based on 22 features.[c] These features are mostly from implementation view and user interface design. Based on the structure of behavior trees and considering the behavior tree features[12] and NodeCanvas, this paper came up with a comprehensive list of BT features that can be used to evaluate and compare existing works. A summarized list of BT features as well as supporting works are shown in Table 2.

As shown in Table 2, features such as automatic generation of BTs, reusability, runtime support of BTs, flexible variables and multi-agent behavior trees are widely supported features in computer games. These are the basic features that must be supported and taken into account to develop new behavior tree components. On the other hand, supporting persistent states, behavior tagging, hinted execution BTs, supporting social and emotional factors, visual editing and debugging, and supporting component reasoner are features that are rarely considered in behavior trees for computer games. The rest of features including supporting utility-based AI, enhanced states, flexible nodes and parametrization are complementary features that enhance the application of behavior trees. This classification not only provides a benchmark to compare behavior tree components, but also provides a road map to develop new behavior tree components for future research prototypes and computer game industry.

3. Learning and Forming Behavior Trees

This section reviews techniques that have been proposed to form behavior trees for computer games. A taxonomy of techniques to form behavior trees including the main features, strengths and the weaknesses of them are shown in Table 3.

[c]http://nodecanvas.com/comparison

Table 3. A taxonomy of techniques to form behavior trees including the main features, strengths (+) and the weaknesses (−) of these techniques.

Genetic algorithms	+ Automatically evolving BTs + Different genetic algorithm can be used to each part − Initial set of BTs are required − Genetic algorithms does not necessarily result in better behavior tree in all cases
CBR	+ Allows to extract behaviors from a knowledge-base + Nodes are enhanced with querying functionality + behaviors are considered as patterns that can be reused − Performance of reasoning is the main problem of this technique
Q-learning	+ Makes it possible to decide when is the right time to execute AI logic + allows debugging BTs + Allows to optimize behavior trees + This technique cannot be used to make plans − This technique suffers from performance issues
Learning examples	+ Quickly reacts to changes at runtime + The generated BT summarizes a large amount of expert knowledge + Employs hierarchical machine learning to model player behavior + The tree can make non-deterministic decisions based on samples − The initial set of behaviors significantly changes the final tree
BT generation	− This is still the game programmer who should implement programming classes − The scalability as well as the complexity of communications between the game world and the behavior tree authoring tool are unanswered questions

3.1. *Automatic generation of BTs*

An efficient design of behavior trees is an important issue in computer games, which is generally performed by experienced experts. There is little research into automated manipulation or improvement of an initial behavior tree implementation. This section reviews automatic techniques to generate BTs.

Genetic Algorithms. There have been efforts to automatically design behavior trees using evolutionary algorithms. Behavior trees of DEFCON game[20] are generated based on an initial population of trees, where genetic operators are used to produce improved trees. Starting with some manually created trees, this technique first encodes possible behaviors for different parts. Then, different genetic programmings run for each part, that consequently result in generating new behaviors from the original set. In this technique, genetic algorithms is used to evolve BTs in an exploratory process, in which a fitness function is used for evaluating evolved behavior trees. In using genetic programming to dynamically evolve behavior trees[21] (which is used in Super Mario), the syntax of possible solution is determined by a context-free grammar.[21] In a modified version of this approach,[22] an AND-OR tree structure is used to different layers of selector and sequence nodes. In learning

```
<BT>     ::= <BT> <Node> | <Node>
<Node> ::= <Condition> | <Action>
<Condition>::= if(obstacleAhead) then <Action>;
| if(enemyAhead) then <Action>;
<Action> ::= moveLeft; | moveRight; | jump; | shoot;
```

Fig. 2. An Illustrative grammar for simple approach to a generic shooting game.[21]

domain-specific planners from example plans,[23] domain-specific planners are used to solve specific planning problems. In spite of promising results, Genetic Algorithms do not necessarily result in better behavior tree in all cases.[21–23]

In Grammatical Evolution,[24] which is a grammar-based form of genetic programming, the syntax of a possible solution is employed to map binary strings to syntactically correct solutions. In the genotype-to-phenotype mapping in grammatical evolution, variable-length strings are evolved using genetic algorithms that are used to select production rules from a grammar. After evaluating the program, its fitness is used by the evolutionary algorithm. An example of an Illustrative grammar for a simple approach to a generic shooting game[21] is shown in Fig. 2.

Meta-heuristic evolutionary learning algorithms are better than grammatical evolution since they provide a natural way of manipulating BTs and applying genetic operators.[25] They tested their technique on "Mario AI" benchmark to simulated autonomous behavior of the characters. BT crafting algorithm can be initialized with a BT that is created manually,[26] where behaviors can be used to control flying robots, while the BT is evolved in every experiment.

As an example of a mapping process given the grammar in Fig. 2, and the string (4, 5, 3, 6, 8, 5, 9, 1), the final program will be moveRight; if (enemyAhead), then shoot. More specifically, the first value (4) is used to select from the start symbol $\langle BT \rangle$ based on the formula $4\%2 = 0$. Thus, the first expression, which is $\langle BT \rangle$ $\langle Node \rangle$ is selected. Accordingly, using $5\%2 = 1$, $\langle BT \rangle$ is replaced by $\langle Node \rangle$, that consequently results in $\langle Node \rangle \langle Node \rangle$. In the same way, $\langle Action \rangle \langle Node \rangle$ is generated from $3\%2 = 1$, and moveRight; $\langle Node \rangle$ is generated from $5\%2 = 1$.

Case-Based Reasoning. In Case-Based Reasoning (CBR) technique,[27] prior experience is employed to solve new problems. For example, CBR[28] is used to extract behaviors from a knowledge base. In this technique, behavior nodes are enhanced with querying functionality. Being in a state allows the node to query the knowledge base for cases of similar states that are visited in the past, and consequently, selecting the best behavior. In the manual editing of behavior trees for Non-Player Characters (NPCs),[29] the main idea is that behaviors typically occur in patterns that can be reused in future designs. In this technique, CBR is used to retrieve and reuse behaviors that are already represented as behavior trees. In this technique, the behavior of a NPC is built at run time using CBR.

CBR can be used in BTs to extract behaviors from a knowledge base.[21] In this technique, CBR is known as solving new problems based on prior experience.[27] In this method, an NPC in a particular state will cause the node to query the knowledge base for similar states that are already processed in order to load an appropriate behavior.

Q-Learning. In Q-learning behavior trees,[2] AI designer is guided by indicating when is the right time to execute branches of AI logic. This technique also facilitates the AI design procedure by debugging, analyzing and optimizing behavior trees. Originally, in the Q-Learning[30] algorithm, a table of values are generated and stored representing the utility of taking an action in a state. In this algorithm, there are associations between different states and rewards. Rewards are feedback to state-action pairs, where utility estimates are improved. In the training phase, this algorithm saves Q-values for visited states. Updates are conducted based on this formula:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_a(Q(s',a')))$$

where $Q(s,a)$ and $Q(s',a')$ are the q value of the current state and the successor state-action pairs, respectfully. α is the learning rate parameter representing how much the new data affects the previous data in q value. r represents the rewards for the successor state. In this formula, γ is the discount factor specifying the importance of future rewards. Determining which action is selected for an agent is performed using a greedy policy, in which the action which is estimated as the best action is selected.[2,31,32] Although learning probabilistic behavior models,[31,32] require little domain knowledge, they cannot be used to build plans. Deepest sequence nodes from an initial behavior tree can be extracted, where such nodes are employed in an off-line Q-learning phase to generate a Q-value table.[31] Then, subtables from this table are generated. The highest valued states for the action are extracted into Q-Condition nodes. Then, condition nodes are replaced with the Q-Condition nodes. The final step is reconstructing of the organization of BT through sorting children nodes based on maximum Q-value.

Learning from Examples. Domain knowledge can be learned from examples of behaviors,[33] which can quickly react to changes at runtime. The knowledge learned is shown in the form of a behavior tree. Formally, given a set of examples of a high-level task, $\{E_1, E_2, \ldots, E_n\}$, in which an example is a sequence of cases $E_i = (C_{i1}, C_{i2}, \ldots, C_{im})$, where a case is an observation and action pair $C_{ij} = (O_{ij}, A_{ij})$, a similarity metric between pairs of observations and pairs of actions, $M(O_{ij}, O_{kl}) \in [0,1]$, and $M(A_{ij}, A_{kl}) \in [0,1]$. This technique aims to extract policies that predict the next action given previous cases and the current observations. This way, the behavior tree can show and summarize a large amount of expert knowledge from this behavior. The procedure starts with identifying areas of commonality within action sequences, as they show common sub-behaviors. To achieve this goal, a maximally

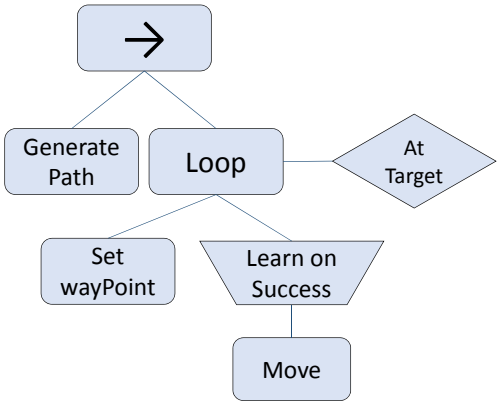

Fig. 3. Refining the behavior tree using the learning technique.[35]

specific BT is generated using samples of case sequences. Then, the BT is pruned and reduced by merging common patterns of actions. This continues until no new pattern is found.

Automatically modifying behavior trees designed by AI designer can be taken into account in order to display characteristics of human players.[34] This starts with a deterministic behavior tree in a multi-player online role-playing game. This technique learns behavior trees that employ human player traces as well as a human-designed tree to cover the variations. In particular, hierarchical machine learning approaches are used to model the behaviors of player. In this technique, a given deterministic behavior tree is adapted to cover a range of observed human behaviors. To this end, an agent controlled by the behavior tree in the game is synchronized with observations from a human player traces. The agent is continuously updated by observations. The generated tree makes it possible to produce most of the observed behaviors, while storing contextual observations representing the conditions for each variation. This way, the tree can make non-deterministic decisions based on the samples.

Machine learning can be used to refine a single behavior within the context of other behaviors.[35] This is implemented using a decorator node (a node used to control and modify return values). Given the behavior tree in Fig. 3, when the Move node returns Success (representing that the agent has traversed the wayPoint), a learning routine is called on a modified Waypoint behavior. In the training phase, an agent is initialized with this tree and data about traverses of players. Quadruple (L, E, F, M) shows the learning method, where L is the set of possible locations, E is the error function (between current location set by the behavior l_1 and the current location actually traveled l_2), F returns a set of environmental features and M is a learning model. When the model is trained, the set of wayPoint behaviors is perturbed based on the probabilities learned by M. This way, more human-like behavior is created.

3.2. *Manual BT generation*

Behavior Tree Authoring. Authoring high-quality behaviors for NPCs is an important research issue in computer games in which games benefit from the adoption of AI techniques.[36] However, since authoring behaviors for NPC requires related skills in design and programming, the duty of behavior generation process is restricted to individuals who have a certain expertise, which limits exploiting the ideas of talented people who are not familiar with these expertises. This problem has resulted in techniques that allow individuals without technical skills to author and design NPC behaviors in an easy way. In creation of behavior trees by novice users,[36] a digital authoring tool is proposed by which novice users can employ to design the behaviors of NPCs.

In AIPaint,[37] which is a tool to design and implement game independent behavior trees, users can create and edit behavior trees using natural sketching interface overlaid on the game world. As a proof of concept, AIPaint is used to recreate the behaviors of classic Pac-Man ghosts as well as a computer soccer game. In AIPaint, game independence is achieved by using Java interface classes to communicate with the game world. However, this is the game programmer who should implement these Java interface classes. On the other hand, the game must provide AIPaint the world state information using feature-value pairs. The scalability of this approach as well as the complexity of communications between the game world and the behavior tree authoring tool are the biggest unanswered questions in AIPaint.

4. Extensions to Behavior Trees

Behavior trees are widely extended from different perspectives to address the various requirements of decision-making in computer games. This section reviews different variations of behavior trees in computer games literature. A taxonomy of variations in the structure of BTs as well as strengths and weaknesses of them are shown in Fig. 4

4.1. *Modification in the structure of BTs*

BTs and Utility-Based AI. In a technique to combine behavior tree and utility-based AI,[38] an architecture is proposed to design realistic bots for military simulators. The most important requirement of a military simulator is designing an enemy acting like a human enemy. In this framework, each level of a behavior tree corresponds to a different state, where utility-based AI[39,41,40] is used to decide how the state goes in from one transition in a level to another. In addition to the selector node in the original behavior tree that chooses one event at a time, reasoned nodes are used to calculate the probability of many events at a time.[38] Then, on the basis of their values, one event is chosen. Reasoner is a complex composite that dynamically calculates probabilities, which on this basis the choices are made.[42] In this setting, instead of a single correct choice, there may exist multiple choices, where

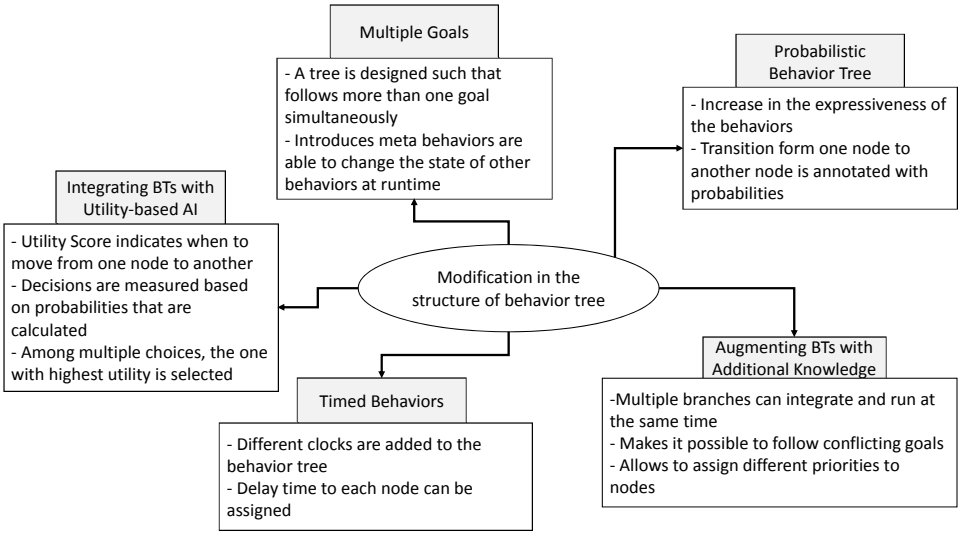

Fig. 4. A taxonomy of variations in the structure of BTs.

one of which is selected based on external factors. The probabilities of options are given based on $f(x) = e^{-x}$, where $x = \alpha * \mathit{threat} + \beta * \mathit{health} + \gamma * \mathit{ammo}$, and α, β and γ are normalizing factors. The response is aggressive as this function is not a linear function. In the case of having more health and ammunition, much more aggressive actions are taken. As an example, suppose once an enemy is detected, the AI goes into the *attack* mode including *fire* ($f(x)$) and *go to cover* ($1 - f(x)$). Increasing the threat results in increasing the probability of going to *go to cover* option. Computing the probabilities of each option is based on computing utility functions. Given u_i denoting the option i, the probability of option i is calculated as $P = u_i / \sum_{i=1}^{n}(u_i)$.

Behavior trees are extended with utility-based AI,[43] in which scores (ranging from 0 to 100) are assigned to each action using utility-based AI. There is a threshold that results in interrupting the current action execution when the score of an action exceeds this threshold. This results in suspending an action to make it possible for the execution of urgent actions.

Component Reasoner,[54] which is a modular and hierarchical decision-making technique, has some similarities with the structure of Behavior Tree (BT). In addition to supporting many decision-making approaches, Component Reasoner relies primarily on Weight-Based Reasoners such as a utility-based technique. This technique is used in video games such as Zoo Tycoon 2 franchise,[d] Iron Man,[e] and Red Dead Redemption.[f] Originally, behavior trees use boolean discriminators to run

[d]https://www.bluefangsolutions.com/
[e]https://www.gameloft.com
[f]http://www.rockstargames.com

selectors. Although this is simple to develop, examining a complex situation may not be possible before making a decision.[55] There are cases in which a more complex approach to decision-making is required, while making it possible to keep simplicity in simple decision-making situations. This was a motivation for a technique[54] to propose a framework that keeps the hierarchy and modularity of behaviors trees, where it is possible to use complex decision makers when it is appropriate. Reasoners[54] instead of selectors can be used to make such decisions.

While selectors employ simple logics such as following the first valid option or fixed probabilities (that are computed at design time) for decision-making, a reasoner allows using any arbitrarily complex logic. This way, highly deterministic behaviors can still use a BT-style selector, while in the cases it is required to compute relative advantages of several possibilities, Component Reasoner is used. In the cases when AI agents need to learn from past results, Genetics-Based Machine Learning[56] techniques can be employed.

Timed Behaviors. BTs can be extended by a time notation[44] (based on the concepts used in timed automata[45]). This is conducted in a way that different clocks are added to behavior trees such that they work simultaneously. The clocks include a constraint indicating how long the process can delay after taking an action. The authors also provided a formal semantics for timed behavior trees. In a timed BT, standard BT nodes (e.g., selection and sequence) are augmented with clock reset (for state realization), clock guard (that restricts the timing of a transition from one location to the next), and invariant (a constraint on how long the process can delay after taking an action).

Probabilistic Behavior Trees. Current syntaxes for behavior trees do not have an option to represent probabilistic behaviors.[50] In a technique to extend BTs,[50] probabilistic behavior is expressed, which results in increasing the expressiveness of behaviors. This technique, which is called probabilistic timed Behavior Trees (ptBTs), makes it possible for AI designers to model timed as well as probabilistic behaviors. In this technique, transitions are annotated with a probability indicating the probability of taking place of that transition. In particular, each node is associated with an optional probability (between 0 to 1). Probabilistic branching is used in addition to concurrent branching. The sum of probabilities in the child nodes is less than or equal to 1. In this setting, given p as the sum of probabilities of a node, no transition is taken with the probability of $1 - P$.

Multiple Goals. Because behavior trees are designed to control the behaviors of a single unit, using them to simultaneously processing of multiple goals is complicated.[52] In a game agent (EISBot),[52] ABL (A Behavior Language)[53] is used as a reactive planning language that includes a set of agents, where each agent must deal with a set of goals to achieve (e.g., Fig. 5). In order to achieve goals, agents choose and execute a behavior from behaviors collection, where a behavior is a set

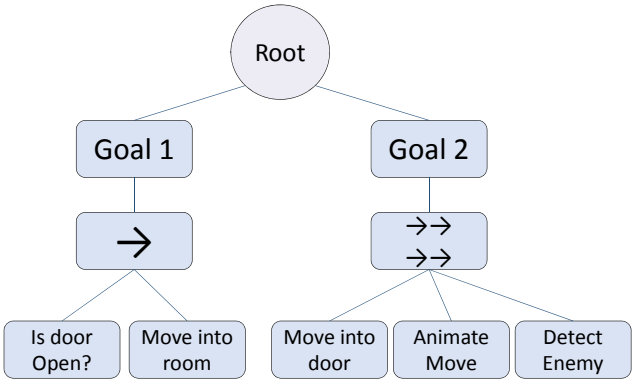

Fig. 5. A tree with multiple goals. Goal 1 and Goal 2 are continued with sequential and parallel behaviors, respectively.

of preconditions specifying if this behavior can be run in the current conditions. This technique supports meta behaviors, which are known as behaviors that can change the state of other behaviors at runtime. This makes it possible for an agent to deal with multiple goals simultaneously, while allows communications between behaviors. At runtime, all of the goals of an agent are stored in an active behavior tree, where in each execution, one of the open leaf nodes (which is a behavior that follows a goal and includes component steps) are executed (i.e., scripts to run actions or to compute).

In an architecture to control the behaviors of AI agents,[80] responsive, collaborative, interruptible and resumable behaviors based on behavior queues are supported. In this technique, behaviors are assigned to roles in order to encapsulate behaviors into components that evolve dynamically, depending on environmental conditions.

Stochastic Behavior Trees. In a technique to compute performance measures of plans using BTs,[3] the concept of Stochastic Behavior Trees is introduced to interpret the interaction of a BT node with its children in terms of a Discrete Time Markov Chain. This way, computations for performances are propagated from one level to the next level in the BT. Finally, the reliability of a BT (that is a composition of primitive actions) is computed. This is reported in terms of the success probability and time to success.

Integration with Social Territoriality. Social territoriality can be integrated with behavior trees, in which multiple branches can integrate and run at the same time.[46] In this architecture, different branches of a node may follow goals that are conflicting. In this architecture, tree nodes have a priority, where nodes with higher priority can subsume lower priority nodes, which makes it possible to immediately respond critical events. The subsumption allows handling complex behavior trees in terms of horizontal layers of goals at different levels of abstraction. In computer

games, this technique results in the responsiveness and the continuity of motions with controlling where to look, stand, and move, while playing a right animation.

4.2. *Augmenting BTs with additional knowledge*

A summary of the techniques to enhance BTs by augmenting them with additional knowledge is shown in Table 4. Main features of these techniques as well as strengths and weaknesses of them are shown in this figure.

Table 4. A summary of the techniques to enhance BTs by augmenting them with additional knowledge.

Genetic Algorithms
+ Integration of FSM and BTs
+ Adding parameter to parent behavior to store the state of the behaviors
− Requires additional time manager to manage long-term missions
Emotional Factor
+ BT designer can indicate which behaviors can be affected by emotional factors
+ Emotional selector node indicates which child node to run based on emotion values
− Extra processing time
Behavior Tagging
+ Employs the concept of behavior tag to maintain specific factors
+ Employs the concept of behavior message that reused to inform other nodes regarding the changes in the behaviors
Hinted Execution Behavior Trees
+ Makes it possible to create and test new features in a plug-in fashion
+ Hint is a piece of information that tell AI what to do
Dynamic Behavior Trees
+ Some nodes hold queries instead of actual behaviors, where at runtime, query nodes are replaced with actual behavior nodes
+ Allows to make abstract behavior trees
Self Validating Behavior Trees
+ Allows to detect faulty BTs at design time
+ Adding parameter to parent behavior to store the state of the behaviors
− Not applicable on existing BT design tools

Augmenting with Internal States. There are behavioral cases that cannot be performed using standard behavior tree implementations.[47] They attribute this problem to the lack of internal states in standard BTs, that do not allow initialization and termination of tasks. To address this problem a technique[47] is proposed that integrates state machines within behavior trees. This technique is compatible to existing behavior tree libraries.[48,49] In this technique, state machines are integrated into arbitrary behavior trees. This way, the full power of state machines are available for AI designer in addition to standard features of BTs. In this technique,

state machine are clocked, where there are delays in the control flow. Time events are used in this technique for long term mission statements.

Classic behavior trees suffer from the problem that all states in a behavior are lost once the behavior is removed.[58] One solution would be adding a separate permanent component to an NPC, where the behaviors can access to this component. One alternative general solution is allowing to store persistent data class specific to the behavior. However, the hard-coded structure of the behavior trees do not provide modularity options such as programming. To address the problem of two NPCs of the same type that behave differently, a parameter can be added to the parent behavior to store the states of behaviors.[58]

Emotional Factor. Emotion is an important factor for human in decision-making, that allows choosing among alternative solutions.[51] Although emotions are integral parts of decision-making, current behavior trees are not integrated with emotions. Behaving in an emotional way is key to have a natural human behavior. In a technique to extend behavior trees with emotions,[51] an AI designer indicates which behaviors can be affected by emotional factors. They argue that time discounting, risk perception, and planning are affected by emotions. They proposed a new selector composite called *emotional selector*, in which emotions are used to change the priorities of child nodes at runtime. In the emotional selector, the values for time, risk, and planning are computed and attached to each child of the emotional selector. The overall weight of the child node i of emotional selector node is computed as: $W_i = \alpha.W_{risk,i} + \beta.W_{time,i} + \gamma.W_{plan,i}$, where α, β and γ are importance factors of the parameters. After sorting child nodes of the emotional selector node based on increasing weights W_i, a probability $prob_i$ to every child node i is assigned such that: $prob_i = a(1-a)i-1)$, where $0.5 \leq a < 1$.

Behavior Tagging. In order to handle complex variations of behaviors, a behavior organizing structure similar to behavior trees is proposed.[57] In this technique, behavior tags and behavior messages are used to manage behaviors of virtual characters. The idea for behavior tag is to maintain specific factors in the behavior in terms of tags, which are ignored when a tree is customized. On the other hand, behavior messages are used to inform the changes regarding the behaviors. Instead of representing this in a static behavior tree structure, messages are dynamically issued to a specific behavior node during the execution of the behavior tree.

Hinted Execution Behavior Trees. Hinted Execution Behavior Tree (HeBTs)[59] is proposed as an extension to BT that provides more controls for developers to create and test new features in a plug-in fashion. In this technique, a hint is a piece of information tells the AI what to do. However, the AI itself decides if it should accept the suggestion. HeBTs allow generating low-risk prototypes of behavior trees in a way that allows teams to try new ideas at final stages of development without hard changes in the code.

Dynamic Behavior Trees. A technique to extend behavior trees is proposed called Dynamic Behavior Trees (DBT),[28] in which some nodes hold queries instead of actual behaviors. A DBT is generated at runtime through substituting query nodes with actual behaviors. This allows an AI designer to specify high level behaviors. This way, properties of the desired behavior for a given state is specified without behavior implementation. In this technique, new basic behaviors are automatically selected such that they best fit to designers' specifications.

Self-Validating Behavior Trees. In an architecture for a BT that employs modules with reflection capabilities,[60] it is possible to detect faulty BTs during the design of a behavior tree. This provides an extra check to design BTs. A similar method is also used at runtime such that the reflective components is used before BT execution. In the application of parameterization in behavior trees,[61] BT tasks are functions with parameters in comparison to classic behavior trees, where tasks are nonparametric. In this technique, a subtree is encapsulated with an exposed parameter interface by a lookup node.

A model reducing technique for behavior trees[78] can be applied before model checking for the purpose of reducing time and memory resources at runtime. This model reducing technique is developed based on the slicing[79] technique, which is originally used to help understanding and debugging programs.

5. Behavior Tree in Practice

BTs are widely used in the game industry[29,62,63] since their introduction for the video game industry. Major published games as well as major game engines such as Unity, Unreal Engine, and CryEngine provide facilities to define and use behavior trees. Behavior trees make it possible to design intelligent behaviors for NPCs in computer games.[3] They are useful to control complex multi mission agents,[63] while allow to formally verify mission plans of NPCs.[64] They also allow to compute execution times activities.[65]

RAIN[g] and Behav[h] are two important commercial behavior trees developed for Unity game engine. UNREALKISMET, that is integrated in the Unreal Development Kit game editor,[i] and flow-graph editor in CryENGINE[j] are two examples of visual scripting tools to model the behaviors in computer games through some variation of data flow diagrams without programming.

Wide variety of games including Halo 2[66] and Spore,[68] and GTA[69] use behavior trees. In Halo 2, which is a first-person shooter game in which the player fights groups of aliens, behavior tree is used to control behaviors including search, combat, flight, self-preservation, idle, charging, fighting, and guarding. This tree has a maximum depth of four, including leaf nodes that execute concrete behaviors.

[g]http://angryant.com/behave/
[h]http://rivaltheory.com/wiki/behaviortrees/behaviortreeeditor
[i]https://www.unrealengine.com/
[j]http://cryengine.com/

Reactive behaviors are enabled using stimulus behaviors. In this game, events received from the game react by inserting stimulus nodes into the behavior tree at runtime. In the next execution, the new stimulus behavior is executed. Although this makes it possible to react to rare events without a need to check conditions, the behavior tree becomes hard to understand. Memory usage in Halo is limited because of using a shared static structure for the behavior tree. This prevents each character to have its own AI.

In an evolutionary technique for behavior trees,[20] a technique is proposed to develop a competitive player that outperforms DEFCON game's original player in a majority of cases. Grammatical Evolution[21] was employed to evolve behavior trees for the Mario Game. In Darmok,[70] a technique to build a repository of behaviors is employed from which behaviors are selected. In the decision cycle during updates, finished steps are checked, and then the world state is updated. Finally, the next step for the next action is selected. In an adaptive behavior tree[59] to modify and update the behavior tree based on knowledge gained from the player during playing, BTs are employed to generate levels tailored to the player. This makes it possible to decrease the difficulty of the game to accommodate casual players.

Since the opponents in computer games often move in some kind of continuous space, it can be represented by a Hybrid Dynamical System (HDS).[71] In the formalized behavior trees in computer games,[10] relations between BTs and HDS are defined to use in Unmanned Aerial Vehicle guidance (UAV). In classic HDS, transitions are one-way transfers of control, (similar to the goto-statement in the programming), while the implicit and two-way transfer of control in behavior trees are more similar to function calls. Behavior trees are used to improve Unmanned Aerial Vehicle guidance.[10] In this technique, BTs are used as a Hybrid Dynamical Systems (HDS), in which state transitions are represented in a tree structure. This architecture is used in the control architecture of many automated in-game opponents. As shown in this paper, any HDS can be written in terms of a BT.

In a behavior tree based control to control the decision-making processes for robot soccer,[72] a hierarchical behavior tree is proposed to simplify the decision-making process for soccer robots. This is an example of uncertain environment, where a priori knowledge of environment and movement of other robots are absent or partial. They also proposed a hierarchical behavior tree based algorithm to enhance path and improve navigation of soccer robots. In the BTs proposed in this work, complex states are easily defined and designed in a modular way such that incremental development of behaviors are possible, while makes it easy to maintain and extend the control system. BTs notation and model checking were used to evaluate system safety.[73] They argue that since verifying the logic of system safety designs is not trivial for humans (due to large details and scenarios), using this notation makes it possible to automate such analysis.

Behavior trees are used to address the problem of representing requirements of a system to be developed such that it is readable by clients and expert modelers.[74,75] In this technique, BT is a graphical representation that allows representing a range

of constructs, states, synchronization, message passing, concurrency, choice and iteration control structures. A requirement can be represented using a behavior tree, where each node is annotated with a requirement, allowing traceability back to the original informal requirements.[14] In this technique inconsistencies, redundancies, incompleteness, and ambiguities are identifiable. Such tree can be considered as a common language that is used to validate the process. In this framework, behavior tree is used as a systematically structured representation of the system. A technique[76] is proposed that exploits behavior tree notation to allow the integration of requirements and describing the integration steps.

Behavior trees can be used to show functional requirements of software as an easy to understand structure.[77] In this method, each requirement is translated to a behavior tree, where behavior trees can be integrated into a single behavior tree, which is known as design behavior trees (DBTs). This representation makes it possible to extract component-based design of the system. When a system is evolved, each version is shown using different behavior trees. Merging algorithms can be used to compare different DBTs that results in generation of an evolutionary design behavior tree (EvDBT). An EvDBT includes data regarding all the compared DBTs, where the evolution information is represented.

BRAINS[k] is an open-source behavior tree component employed in different types of games including action, tactical, and stealth games. This component exploits UDK AI and Navigation, Unreal script for Behavior Tree Hierarchies, and Kismet as a Visual Editor. This BT is currently used in SanctityLost. *Think* is an AI behavior tree editor that can be configured and used for different types of projects. AI designers can create their own custom behavior tree nodes with custom properties. Such user defined nodes are integrated to create behavior trees. This makes it possible to reuse tree nodes for future projects. The behavior tree can be exported as XML, text, C++ code and Unreal Script code.

BST[l](Behavior Selection Tree) is a tool to create and manage behavior trees in CryENGINE3. AI agents have their own BST, which is used to choose a behavior based on game conditions. Brainiac[m] is a visual behavior tree design tool that makes it possible to build behavior trees employing drag and drop of nodes. This tool allows exporting the behavior trees in XML format. The game designer writes her own code to generate logic for the trees.

ComBE[n] is a behavior tree editor that allows an AI designer to simultaneously edit the trees using textual and graphical items. RAIN's behavior tree[o] supports simple to complex decision-making process. This component provides a graphical editor to design behaviors and add different actions, animations, and sensing features. This behavior tree component is widely used in Unity game engine.

[k]https://www.epicgames.com/
[l]http://www.cryengine.com
[m]http://www.brainiac.codeplex.com/
[n]http://www.oskarvanrest.github.io/ComBE/
[o]http://www.rivaltheory.com/

The pervasiveness of mobile devices as well as the growth in the technologies of application development is driving games market faster than before. Revenue from games is now surpassing revenue from movie sales.[81–83] Entering to this market requires employing powerful AI tools to make advanced games. Behavior trees by providing facilities to define complex behaviors make it possible to design believable non-player characters.

6. Conclusion

Behavior trees have become essential to design, control and monitor of NPCs in computer games. The paper presents a survey of literature for techniques to automatically and manually generate behavior trees. This paper also reviews various techniques to extend and enhance the structure of behavior trees with the aim of increasing the modularity, reusability, dynamic control of trees, and flexibility in the design of behaviors. This paper presents a list of features to compare BTs and classifies BTs based on these features. This classification provides a benchmark to compare behavior tree research prototypes as well as commercial products. The paper also surveys how and to what extend BTs are used in practice in computer games.

References

1. I. Millington, *Artificial Intelligence for Games*, ed. J. Funge (Elsevier Science and Technology, 2009).
2. R. Dey and C. Child, QL-BT: Enhancing behaviour tree design and implementation with Q-learning, in *2013 IEEE Conference on Computational Intelligence in Games (CIG)* (IEEE, August 2013), pp. 1–8.
3. M. Colledanchise and P. Ogren, How behavior trees modularize robustness and safety in hybrid systems, in *2014 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2014)* (IEEE, September 2014), pp. 1482–1488.
4. D. M. Roijers, P. Vamplew, S. Whiteson and R. Dazeley, A survey of multi-objective sequential decision-making, *Journal of Artificial Intelligence Research* (2013).
5. A. Kumar, S. Zilberstein and M. Toussaint, Probabilistic inference techniques for scalable multiagent decision making, *Journal of Artificial Intelligence Research* **53**(1) (2015) 223–270.
6. M. Kalech and S. Reches, Decision making with dynamic uncertain events, *Journal of Artificial Intelligence Research* **54**(1) (2015) 233–275.
7. G. Robertson and I. Watson, A review of real-time strategy game AI, *AI Magazine* **35**(4) (2014) 75–104.
8. P. Rizzo, Goal-based personalities and social behaviors in believable agents, *Applied Artificial Intelligence* **13**(3) (1999) 239–271.
9. R. A. Rodrigues, A. de Lima Bicho, M. Paravisi, C. R. Jung, L. P. Magalhaes and S. R. Musse, An interactive model for steering behaviors of groups of characters, *Applied Artificial Intelligence* **24**(6) (2010) 594–616.
10. P. Ogren, Increasing modularity of UAV control systems using computer game behavior trees, in *AIAA Guidance, Navigation and Control Conference* (Minneapolis, MN, August 2012).

11. C. Bennett and D. V. Sagmiller, *Unity AI Programming Essentials* (Packt Publishing Ltd, 2014).
12. A. Marzinotto, M. Colledanchise, C. Smith and P. Ogren, Towards a unified behavior trees framework for robot control, in *2014 IEEE Int. Conf. on Robotics and Automation* (*ICRA*) (IEEE, May 2014), pp. 5420–5427.
13. K. Winter, Formalising behaviour trees with CSP, in *IFM* (April 2004), pp. 148–167.
14. R. J. Colvin and I. J. Hayes, A semantics for behavior trees using CSP with specification commands, *Science of Computer Programming* **76**(10) (2011) 891–914.
15. C. A. R. Hoare, Communicating sequential processes, *Communications of the ACM* **21**(8) (1978) 666–677.
16. B. Roscoe, *The Theory and Practice of Concurrency* (1998).
17. P. T. Eugster, P. A. Felber, R. Guerraoui and A. M. Kermarrec, The many faces of publish/subscribe, *ACM Computing Surveys* (*CSUR*) **35**(2) (2003) 114–131.
18. S. K. Kim, T. Myers, M. F. Wendland and P. A. Lindsay, Execution of natural language requirements using state machines synthesised from behavior trees, *Journal of Systems and Software* **85**(11) (2012) 2652–2664.
19. R. Colvin and I. J. Hayes, A semantics for Behavior Trees (2010) (No. SSE-2010-03), pp. 1–26.
20. C. U. Lim, R. Baumgarten and S. Colton, Evolving behaviour trees for the commercial game DEFCON, in *Applications of Evolutionary Computation* (Springer Berlin Heidelberg, 2010), pp. 100–110.
21. D. Perez, M. Nicolau, M. O'Neill and A. Brabazon, Evolving behaviour trees for the Mario AI competition using grammatical evolution, in *Applications of Evolutionary Computation* (Springer Berlin Heidelberg, 2011), pp. 123–132.
22. A. Champanard, Understanding the second-generation of behavior trees. http://aigamedev.com/insider/tutorial/second-generation-bt. (2012).
23. E. Winner and M. Veloso, Distill: Learning domain-specific planners by example, in *Proc. of the Int. Conf. on Machine Learning* (2003), pp. 800–807.
24. M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, Vol. 4 (Springer Science and Business Media, 2012).
25. M. Colledanchise, R. Parasuraman and P. Ögren, Learning of Behavior Trees for Autonomous Agents (2015), arXiv preprint arXiv:1504.05811.
26. K. Y. Scheper, S. Tijmons, C. C. de Visser and G. C. de Croon, Behavior Trees for Evolutionary Robotics, *Artificial Life* (2016).
27. J. L. Kolodner, An introduction to case-based reasoning, *Artificial Intelligence Review* **6** (1992) 3–34.
28. G. Flórez-Puga, M. Gómez-Martín, B. Diaz-Agudo and P. Gonzalez-Calero, Dynamic expansion of behaviour trees, in *Proc. of AIIDE* (2008), pp. 36–41.
29. G. Flórez-Puga, M. A. Gómez-Martín, P. P. Gómez-Martín, B. Díaz-Agudo and P. A. González-Calero, Query-enabled behavior trees, *IEEE Transactions on Computational Intelligence and AI in Games* **1**(4) (2009) 298–308.
30. C. J. C. H. Watkins, Learning from delayed rewards (Doctoral dissertation, University of Cambridge) (1989).
31. E. W. Dereszynski, J. Hostetler, A. Fern, T. G. Dietterich, T. T. Hoang and M. Udarbe, Learning Probabilistic Behavior Models in Real-Time Strategy Games, in *AIIDE* (September 2011).
32. G. Synnaeve and P. Bessiere, A Bayesian model for plan recognition in RTS games applied to StarCraft (2011), arXiv preprint, arXiv:1111.3735.
33. G. Robertson and I. Watson, Building behavior trees from observations in real-time strategy games, in *2015 Int. Symp. on Innovations in Intelligent SysTems and Applications* (*INISTA*) (IEEE, September 2015), pp. 1–7.

34. E. Tomai and R. Flores, Adapting in-game agent behavior by observation of players using learning behavior trees, in *Proc. of the 9th Int. Conf. on the Foundations of Digital Games* (*FDG 2014*) (2014).
35. E. Tomai, R. Salazar and R. Flores, Simulating aggregate player behavior with learning behavior trees, in *Proc. of the 22nd Annual Conf. on Behavior Representation in Modeling and Simulation* (July 2013).
36. M. Mehta and A. Corradini, An Approach to behavior authoring for non-playing characters in digital games, in *Proc. of the 2014 Mulitmedia, Interaction, Design and Innovation International Conference on Multimedia, Interaction, Design and Innovation* (ACM, June 2014), pp. 1–7.
37. D. Becroft, J. Bassett, A. Mejía, C. Rich and C. L. Sidner, AIPaint: A sketch-based behavior tree authoring tool, in *AIIDE* (September, 2011).
38. S. Jadon, A. Singhal and S. Dawn, Military simulator — A case study of behaviour tree and unity based architecture (Chicago, 2014), arXiv preprint arXiv:1405.7944.
39. D. Mark, K. Dill and A. I. Engineer, Improving AI decision modeling through utility theory, in *Game Developers Conference* (2010).
40. J. Laird and M. VanLent, Human-level AI's killer application: Interactive computer games, *AI Magazine* **22**(2) (2001) 15.
41. K. Dill and D. Mark, Embracing the dark art of mathematical modeling in AI, in *Game Developers Conference* (2012).
42. M. van Lent and J. Laird, Developing an artificial intelligence engine (1998).
43. M. N. M. Othman and H. Haron, Implementing game artificial intelligence to decision making of agents in emergency egress, in *2014 8th Malaysian Software Engineering Conference* (*MySEC*) (IEEE, September, 2014), pp. 316–320.
44. R. Colvin, L. Grunske and K. Winter, Timed behavior trees for failure mode and effects analysis of time-critical systems, *Journal of Systems and Software* **81**(12) (2008) 2163–2182.
45. J. Bengtsson and W. Yi, Timed automata: Semantics, algorithms and tools, in *Lectures on Concurrency and Petri Nets* (Springer Berlin Heidelberg, 2004), pp. 87–124.
46. C. Pedica and H. H. Vilhjálmsson, Lifelike interactive characters with behavior trees for social territorial intelligence, in *ACM SIGGRAPH* 2012 Posters (ACM, August 2012), p. 32.
47. A. Klöckner, Behavior Trees with Stateful Tasks, in *Advances in Aerospace Guidance, Navigation and Control* (Springer International Publishing, 2015), pp. 509–519.
48. H. Elmqvist, F. Gaucher, S. E. Mattsson and F. Dupont, State machines in Modelica, in *Proc. of 9th Int. Modelica Conference* (Munich, Germany, September 2012), pp. 3–5.
49. A. Klöckner, F. van der Linden and D. Zimmer, The Modelica behavior trees library: Mission planning in continuous-time for unmanned aircraft, in *Proc. of the 10th Int. Modelica Conference* (2014), Vol. 96, pp. 727–736.
50. R. Colvin, L. Grunske and K. Winter, Probabilistic timed behavior trees, in *Integrated Formal Methods* (Springer Berlin Heidelberg, January 2007), pp. 156–175.
51. A. Johansson and P. Dell'Acqua, Emotional behavior trees, in *2012 IEEE Conf. on Computational Intelligence and Games* (IEEE, September, 2012), pp. 355–362.
52. B. G. Weber, P. Mawhorter, M. Mateas and A. Jhala, Reactive planning idioms for multi-scale game AI, in *2010 IEEE Symp. on Computational Intelligence and Games* (*CIG*) (IEEE, August, 2010), pp. 115–122.
53. M. Mateas, Interactive drama, art and artificial intelligence (2002).

54. K. Dill and L. Martin, A game AI approach to autonomous control of virtual characters, in *Proc. of the 2011 Interservice/Industry Training, Simulation, and Education Conference* (2011).
55. K. Dill, Embracing declarative AI with a goal-based approach, *AI Game Programming Wisdom* **4** (2008) 229–238.
56. G. A. Harrison and E. W. Worden, Genetically programmed learning classifier system description and results, in *Proc. of the 9th Annual Conference Companion on Genetic and Evolutionary Computation* (ACM, July 2007), pp. 2729–2736.
57. L. Li, G. Liu, M. Zhang, Z. Pan and E. Song, BAAP: A behavioral animation authoring platform for emotion driven 3D virtual characters, in *Entertainment Computing* (*ICEC 2010*) (Springer Berlin Heidelberg, 2010), pp. 350–357.
58. G. Alt, The suffering: A game AI case study, in *Nineteenth National Conference on Artificial Intelligence Challenges in Game AI Workshop* (2004), pp. 134–138.
59. S. Ocio, Adapting AI behaviors to players in driver San Francisco hinted-execution behavior trees, in *Proc. of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2012).
60. D. Llansó, M. A. Gómez-Martín and P. A. González-Calero, Self-validated behaviour trees through reflective components, in *AIIDE* (2009).
61. A. Shoulson, F. M. Garcia, M. Jones, R. Mead and N. I. Badler, Parameterizing behavior trees, in *Motion in Games* (Springer Berlin Heidelberg, 2011), pp. 144–155.
62. A. Champandard, Getting started with decision making and control systems, *AI Game Programming Wisdom* **4** (2008) 257–264.
63. R. Palma, P. A. González-Calero, M. A. Gómez-Martín and P. P. Gómez-Martín, Extending case-based planning with behavior trees, in *FLAIRS Conference* (March, 2011).
64. R. E. Fikes and N. J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* **2**(3) (1972) 189–208.
65. X. Wang, Learning by observation and practice: An incremental approach for planning operator acquisition, in *ICML* (July 1995), pp. 549–557.
66. D. Isla, Handling complexity in the Halo 2 AI, in *Game Developers Conference*, Vol. 12 (March 2005).
67. D. Isla, Building a better battle, in *Game Developers Conference* (San Francisco, 2008).
68. C. Hecker, My liner notes for spore/spore behavior tree docs (2009), www.chrishecker.com (2007).
69. A. Champandard, Behavior trees for next-gen game AI, in *Game Developers Conference*, Audio Lecture (December 2007).
70. S. Ontanón, K. Mishra, N. Sugandh and A. Ram, On-line case-based planning, *Computational Intelligence* **26**(1) (2010) 84–119.
71. M. S. Branicky, Introduction to hybrid systems, in *Handbook of Networked and Embedded Control Systems* (Birkhäuser Boston, 2005), pp. 91–116.
72. R. H. Abiyev, N. Akkaya and E. Aytac, Control of soccer robots using behaviour trees, in *2013 9th Asian Control Conference* (*ASCC*) (IEEE, June 2013), pp. 1–6.
73. P. Lindsay, K. Winter and N. Yatapanage, Safety assessment using behavior trees and model checking, in *2010 8th IEEE Int. Conf. on Software Engineering and Formal Methods* (*SEFM*) (IEEE, September 2010), pp. 181–190.
74. Z. Liu and H. Jifeng, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Series on Component-Based Software Development (World Scientific Publishing Co., Inc., 2006).

75. C. Smith, K. Winter, I. Hayes, G. Dromey, P. Lindsay and D. Carrington, An environment for building a system out of its requirements, in *Proc. of the 19th IEEE Int. Conf. on Automated Software Engineering* (IEEE Computer Society, September 2004), pp. 398–399.
76. K. Winter, I. J. Hayes and R. Colvin, Integrating requirements: The behavior tree philosophy, in *2010 8th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)* (IEEE, September, 2010), pp. 41–50.
77. L. Wen, D. Kirk and R. G. Dromey, A tool to visualize behavior and design evolution, in *Ninth Int. Workshop on Principles of Software Evolution: in conjunction with the 6th ESEC/FSE joint meeting* (ACM, September 2007), pp. 114–115.
78. N. Yatapanage, K. Winter and S. Zafar, Slicing behavior tree models for verification, in *Theoretical Computer Science* (Springer Berlin Heidelberg, 2010), pp. 125–139.
79. M. Weiser, Program slicing, in *Proc. of the 5th Int. Conf. on Software Engineering* (IEEE Press, March 1981), pp. 439–449.
80. M. Cutumisu and D. Szafron, An architecture for game behavior AI: Behavior multi-queues, in *AIIDE* (2009).
81. Y. A. Sekhavat, Nowcasting mobile games ranking using web search query data, *International Journal of Computer Games Technology* (2016).
82. Y. A. Sekhavat and P. Abdollahi, Can Google Nowcast the market trend of Iranian mobile games?
83. Y. A. Sekhavat, KioskAR: An augmented reality game as a new business model to present artworks, *International Journal of Computer Games Technology* (2016).