Trained Behavior Trees: Programming by Demonstration to Support AI Game Designers

Ismael Sagredo-Olivenza, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín, and Pedro Antonio González-Calero

***Abstract*—Programming by demonstration (PbD) has a straightforward application in the development of the artificial intelligence (AI) for nonplayer characters (NPCs) in a video game: a game designer controls the NPC during a training session in the game, and thus demonstrates the expected behavior for that character in different situations. Afterwards, applying some machine learning technique on the traces recorded during the demonstration, an AI for the NPC can be generated. Nevertheless, with this approach, it is very hard for the game designer to fully control the resulting behavior, which is a key requirement for game designers, who are responsible for putting together a fun experience for the player. In this paper, we present trained behavior trees (TBTs). TBTs are behavior trees (BTs) generated from traces obtained in a game through PbD. BTs are a technique widely used for AI game programming that are created and modified through special purpose visual editors. By inducing a BT from a PbD game session, we combine the ease of use of PbD with the ability to fine-tune the learned behavior of BTs. Furthermore, TBTs facilitate the use of BTs by game designers and promote their authoring control on game AI.**

***Index Terms*—Behavior trees (BTs), decision trees, knowledge acquisition, machine learning.**

I. Introduction

GAME development is a multidisciplinary task that involves expertise from different areas with different backgrounds, mainly artists, programmers, and game designers. This paper specifically concentrates on the creation of behaviors for nonplayer characters (NPCs), which is part of the game artificial intelligence (game AI). The observed behavior of NPCs in a video game is the result of the collaboration between artificial intelligence game programmers and level designers. AI game programmers develop a number of configurable components and systems, which level designers then use to put together a dramatic and fun experience for the player.

Game designers envision the way their characters behave and are used to express themselves using high-level descriptions such as "high general alertness (very nervous, thoroughly investigates every little disturbance, hard to circumvent, hard to shake off once he sees the target)."[1] Following the specifications given by designers, game AI programmers develop perception systems, behavior representation, path finding and decision making algorithms that conform with the AI system that game designers use to put together levels in the game.

AI game programmers and designers must work in close collaboration. Depending on the particular workflow of a given studio, designers may be in charge of just designing, tweaking parameters and testing the NPC behavior, or being responsible for programming part of the behavior through scripting, visual languages, or special purpose tools. Some solutions to AI problems are easier for a nontechnical designer, but might take more implementation time; other solutions might require technical expertise to use but allow for flexibility and can take less time to create. Therefore, assessing the strengths of the design team is crucial to choose the best combination of AI tools and techniques for a particular project of a given studio [1].

In this paper, we present trained behavior trees (TBTs), a technique that was initially conceived for designers without technical expertise, although our experiments indicate that designers with a computer science degree can also find TBTs useful. TBTs are behavior trees (BTs) generated from traces obtained in a game through programming by demonstration (PbD).

BTs have been proposed as an alternative to finite-state machines (FSMs) when programming games AI for complex decision making and orchestrating nontrivial NPC behavior. BTs have been used in AAA commercial games, for example in *Halo 2* (Microsoft, 2004) [2] or *Tom Clancy's The Division* (Ubisof, 2016) [3]. FSMs have a long tradition as game AI programming tools [4], but they also have serious limitations when dealing with complex behaviors since the number of transitions can exponentially grow leading to a combinatorial explosion [5].

BTs, nevertheless, have their our drawbacks regarding authoring control for game designers. Whereas FSMs are easily grasped without technical expertise, BTs include concepts from general purpose programming languages that require technical expertise to be fully exploited. Concepts used in BTs such as the difference between parallel execution and iteration in a game loop require a good understanding of how a game engine works [6]. Actually, BTs were first introduced as a tool for AI game programmers [2].

Manuscript received January 16, 2017; revised January 19, 2017 and September 17, 2017; accepted October 20, 2017. Date of publication November 21, 2017; date of current version March 15, 2019. This work was supported by the Spanish Ministry of Science and Education under Grant TIN2014-55006-R. *(Corresponding author: Ismael Sagredo-Olivenza.)*

The authors are with the Universidad Complutense de Madrid, Madrid 28040, Spain (e-mail: isagredo@ucm.es; pedrop@fdi.ucm.es; marcoa@fdi.ucm.es; pedro@fdi.ucm.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TG.2017.2771831

[1]This is the description of one of the NPCs in *FarCry* (Ubisoft, 2004).

In our previous work, we have already demonstrated that nontechnical game designers can build simple BTs. And since BTs facilitate the work at different levels of abstraction, we have defined a methodology to support the collaboration between programmers and designers, by letting designers build high-level BTs as the composition of lower level subtrees built by programmers [7].

With TBTs, we intend to extend the range of BTs that can be created by game designers without requiring technical expertise. We propose the use of PbD (which can also be found as learning from demonstration or learning by imitation in the literature) to let designers create an NPC behavior just by playing the game. This process is accomplished in two steps, first, game traces are recorded with the designer controlling the NPC to be programmed, and then those traces are processed to generate a BT that can serve to control the same NPC in the final game. In addition, the resulting BT can be further edited with a BT visual editor. Such a BT generated from game traces is what we call a "trained BT." Therefore, by inducing a BT from a PbD game session, we combine the ease of use of PbD with the ability to fine-tune the learned behavior of BTs.

In order to evaluate our approach, we asked a number of graduate game design students, with different technical backgrounds, to apply the approach by training an NPC and evaluating the observed behavior of the NPC controlled by the induced BT. As we will show later, the experiments demonstrate that game designers, regardless of their technical background, can effectively train complex BTs.

The rest of this paper is organized as follows. Section II describes BTs and some results about the importance of having technical expertise when designing them. Section III describes TBTs and the proposed algorithm used to obtain a BT from the game traces. Section IV shows the results obtained when using TBTs, and finally Section V presents related work and some conclusions.

II. Behavior Trees

The first documented use of BTs for AI game programming is in the *Halo* video game franchise developed by Bungie Studios and published by Microsoft between 2001 and 2010 [2], [8]. Since then, they have become popular as a substitute for FSMs when developing nontrivial NPC behavior. Several popular game engines today include BT visual editors and execution engines, either natively or via extensions. *Unreal Engine* includes a BT editor to create BT Assets that can be later executed from a blueprint graph. CryEngine incorporates its modular BT editor that also allows building a BT and describing it through XML. And although *Unity3D* does not provide native BT support, there are different extensions available at its asset store, such as NodeCanvas and BehaviorBricks (BB), that support them.

BTs improve scalability over FSMs by removing explicit transitions between states and replacing them using internal nodes with different selection strategies when executing their children. The tree structure provides abstraction and reusability. A subtree (maybe just a leaf) models a goal (for example, "go to a location," "attack an enemy," or "look for ammo"). When a subtree is activated, it can run during many game cycles, and eventually it will finish with a succeed or fail condition. This termination status is used by the parent nodes in the tree hierarchy to decide about their own end condition or to activate any other child.

Leaf nodes can run tasks that are of two different types, which are as follows.

1) *Actions*: They are tasks that modify the game environment when they are executed. Some examples can be "shoot," "go to," or "open the door."
2) *Conditions*: They are tasks that check the state of the environment and return succeed or fail depending on that state. They can be considered the replacement of the conditions that fire transitions in FSMs and are the main tool for internal nodes to come up with decisions. Examples are "Have I enough ammo?" "Is the player in sight?" or "Is the door closed?"

There are many different internal nodes described in the literature [9] and tools implement different subsets. A popular subset is the one including the following.

1) *Sequences*: They execute their children in order, one by one, creating a linear control flow. The node succeeds if all of its children succeed, and fails as soon as any of them fails.
2) *Selectors*: They also execute their children in order, but as soon as any of them succeeds, the selector also succeeds. Only if all the child nodes have failed, does the selector fail. They are useful for decision making, where a set of options are tried in a given order.
3) *Priority selectors*: They constitute an improved version of selectors. Every child node has an associated condition. The priority selector (or, for brevity's sake, priority) launches the execution of its first child (from left to right) whose condition is true. But instead of ruling out the execution of the previous siblings, the priority node will continuously monitor their conditions, so if any of them becomes true, it will abort the execution of the child with less priority and start running the new one immediately.
4) *Parallels*: They run all their children concurrently. The ending policy may be configured to behave as a sequence, a selector or using majority vote when all its children have finished. Notice that the leaf nodes reachable from a parallel should allow that concurrence. Care should be taken to avoid, for example, two "Go to" tasks being executed at the same time.
5) *Decorators*: They are special nodes that only have one child. They can extend the functionality of the other node. For example, the repeat node runs its child a number of times (or even indefinitely).

Fig. 1 shows a partial view of an example BT that would control an enemy NPC patrolling a facility in the game. The root node is a priority selector that first tries to shoot the player, if within range, then tries to chase them, if within sight, and defaults to patrolling the area if none of the previous conditions is true. Higher priority conditions are periodically tested so that the NPC will stop patrolling as soon as the player is seen.

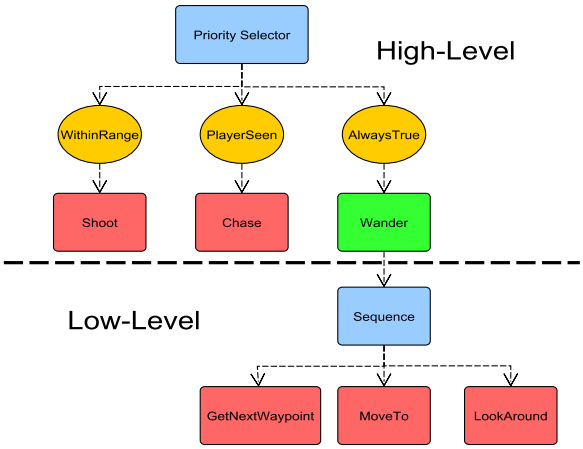

Fig. 1. Example showing high- and low-level behaviors.

A. BB for Programmers and Designers

BTs were initially proposed as a tool for AI game programming in *Halo 2* (Microsoft, 2004) [2] developed by Bungie Studios. In the AI workflow of Bungie Studios for that game, designers did not build the BTs, and they could just tweak sets of parameters, and define styles for the NPC behavior. Given a BT built by the programmers, a style is defined by disabling certain actions in the tree. For example, defensive styles do not allow charge or search behaviors; aggressive styles do not allow self-preservation; and noncombatant style would not allow any combat behaviors at all, instead allowing only idle or retreat behaviors. Styles also allow the designer to skew some of the parameters controlling behavior one way or the other, for example, allowing characters to flee more easily in a cowardly style.

Our main goal with TBTs is to bring the expressiveness of BTs to game designers without technical expertise. This work falls within a long term effort to facilitate the collaboration between programmers and designers when building game AI, that includes the use of reusable behaviors [10] or the explicit domain modeling of actions and entities in a game [11], and which also includes the development of BB.

BB[2] is a runtime engine and visual editor tool for creating BTs in *Unity3D* developed by PadaOne Games[3] (a spinoff company from the Complutense University of Madrid). BB is available in the Unity Asset Store and, as of this writing, located in the top three free Unity editor extensions for visual scripting.

BB has two key features that promote the division of labor and collaboration between game programmers and designers, which are as follows.

1) Parameterized reusable subtrees. One of the primitive actions that users may locate as terminal nodes is the subbehavior node, that allows programmers and designers to split their BTs in multiple abstraction layers. A low-level BT may be parameterized and called from different nodes of a high-level BT. The runtime engine lies on top of a blackboard shared by all nodes being executed and acts as a place to pass input parameters to a subbehavior or to get their results back to high-level BTs.
   We use this feature for splitting the creation of a complete behavior in low-level BTs (created by programmers) and high-level BTs (built by designers).
2) Late binding of tasks. The implementation of a task, that can be an action, a condition, or a subtree, is chosen at runtime when the BT is executed. This way, a task can include a reference to another task, using an id that represents the goal requested, such as ATTACKTOENEMY. This goal could be provided by different concrete tasks (such as SHOOT, HITWITHSWORD, or CHARGEAT). If one of the leaf nodes has a task referenced by other tasks, before the execution, when the BT is instantiated and configured for a specific NPC, a binding with the correct task is needed. The correct task will be selected among those available, depending on the skills of the NPC (in the example of ATTACKTOENEMY, depending of the type of weapon being equipped).
   Notice that with this mechanism, we can do a hierarchy of tasks. This feature is particularly interesting because it is the foundation for building TBTs. Thanks to it, as will be seen in Section III, the designer specifies the goal to be learned in a special node, and the system finds those tasks linked to this goal, in order to be able to offer them as execution options at the time of learning.

The combination of parameterized reusable subtrees and late binding of tasks supports the collaboration between programmers and designers by allowing behaviors to be easily decomposed into different layers of abstraction. Low-level behaviors can be defined as those that can be reused in other behaviors and that manage the concepts closer to scripting languages such as: target selection, route following, or coverage finding. High-level behaviors can be defined as those behaviors describing the general behavior of the NPC. When the collection of low-level behaviors is complete enough, the creation of BTs that manage an entire NPC can be easily done in terms of those behaviors. An example is shown in Fig. 1, where the high-level BT of an NPC uses a priority selector as its root node and relies on two primitive actions, Shoot and Chase, and a subbehavior, Wander, available also as a BT.

With this division, we can easily split the creation of an NPC behavior into two different tasks, high-level and low-level behavior creation. Designers are in charge of the former, whereas programmers create the latter.

Moreover, using BB programmer's responsibility is twofold. On one hand, they must create the low-level primitives using the scripting language of the underlying engine (in the case *Unity3D*); and, on the other hand, they must implement with the BB editor the low-level behaviors as BTs.

The responsibility of designers is the creation of the high-level behaviors, a task that they can successfully perform since it is closer to the designer language. Our initial hypothesis was that as BB supports this separation of labor, it would be a good approach to split behavior into high and low levels, and have designers without technical expertise building BTs only in the top of the abstraction level.

[2]BB http://bb.padaonegames.com/
[3]PadaOne Games http://www.padaonegames.com/

In this context, we ran different experiments with users with different programming skills creating BTs. We first confronted them with a programming test to assess their skills. Later on, they created some high-level behaviors only using BB, that we also evaluated. The results, described in [7], show some evidence of the existence of a correlation between the result of the programming test and the result of the behavior creation exercise.

The test consisted of 20 questions about elementary programming concepts such as sorting, array manipulation, and control flow sentences. The BT exercise was evaluated based on the behavior created by the users and its performance when it was running.

The correlation results were not very high (0.525 using a Pearson product moment correlation), but diving into the results, we observed that there was a clear difference between users that had some programming knowledge and those who had none. In this context, we discovered that designers without technical expertise can create some high-level behaviors. However, experimental results also showed that there were some users that were not able to create any complex BT and, moreover, some BTs were out of reach for every designer without technical expertise.

We initially conceived TBTs as a way to increase the range of BTs that can be built without requiring technical expertise.

III. Trained Behavior Trees

As explained in Section I, designers and programmers must work together to build character behaviors. But this is not always easy. Designers and programmers usually have opposing interests and concerns, that affect even the way in which they communicate. One way to reduce the communication error and development time consists on easing the way behaviors are specified. Designers have a plethora of alternatives to create them without the tedious necessity of writing code, mainly using visual editors to create FSMs, BTs, or data flow charts. However, the use of these behavior editors requires certain technical knowledge that not all designers possess.

To solve this problem, TBTs intend to help nontechnical designers create behaviors in a more intuitive way, just by playing the game as the NPC would do. The process starts with the creation of a minimal BT using the provided BT editor. This initial BT contains a special node called a trainer node (TN). This node can be trained by a designer in a pure game session where he simulates the intended behavior of the NPC. The knowledge acquired in this training session is later used at run-time when the player executes the game. The TN is replaced by a machine-generated subbehavior that, when reached, selects the task that best fits the actual state of the game according to the traces stored previously by the TN during the training session. This computer-generated behavior is called a TBT and thanks to it, the designer can create behaviors without programming.

For this approach to work, some basic tasks should be available, so that complex behaviors can be built using them. These basic tasks are implemented by programmers. In essence, we propose the following methodology: the designer starts with the definition of the behavior of the game NPCs using very high level specifications, for example, natural language, tables, rules, diagrams, etc. Using this description, designers and programmers identify and select the basic tasks of the NPC. These tasks are implemented by the programmers and later used by designers as bricks (low-level primitives) when building the behaviors using the BT editor. If the designers reach a dead end and is not able to express themselves using BTs, it can try to generate one by demonstration placing a TN on the BT. After the training phase, designers may test the generated behavior in a regular game session to check if it fits its expectations and repeat the training phase or fine-tune the behavior via its inferred BT.

Fig. 2. TowoT screenshot.

Incorporating our TBTs into a game development process requires the existence of a training phase that, using the same interactions as the game being developed already has, allows the designer to control the NPC. In order to facilitate the task, the system offers an application programming interface (API) to control the TN. To give an idea of the entire process, we will describe how it was used in the creation of our game prototype called TowoT.

TowoT is a tower defense game developed by the authors. It combines tower defense with third-person action elements. As a tower defense, every level of the game consists on two stages: tower deployment first, and defense afterwards. First, the player has to place a number of towers that will serve to protect certain places in the level, once the enemies deploy. While in pure tower defense games similar to *Plants vs Zombies* (PopCap Games, 2009), players keep on placing towers when new enemies arrive, in TowoT, an action-tower defense game such as *Orcs Must Die!* (Robot Entertainment, 2011), the player concentrates on actually fighting the enemies. In addition to the static towers with diverse defensive and offensive abilities, the player is helped by TowoT, a big robot that moves slowly but fires heavily. TowoT constantly consumes energy, and it must periodically recharge itself in charging stations to avoid get exhausted.

Fig. 2 shows a screenshot from the game, with Jacob (the hero) in the foreground looking toward TowoT defending the spaceship from an approaching enemy. The goal of the game is to collect uridium, a valuable mineral that is extracted using refineries that are built on planets where the ore abounds. This must be accomplished by keeping both the refinery and the spaceship (the Core) safe from enemies.

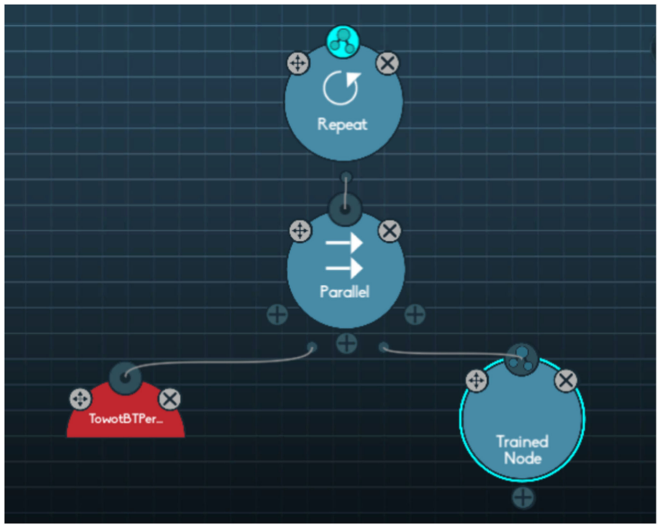

Fig. 3. BT used to train the TowoT behavior.

TABLE I
ATTRIBUTES STORED DURING THE TRAINING PHASE

	Energy/life	Risk level [0, 4]
Jacob	JACOBLIFE [0, 500]	JACOBRISKLEVEL
TowoT	TOWOTENERGY [0, 100]	–
Core	CORELIFE [0, 1000]	CORERISKLEVEL
Refinery	REFINERYLIFE [0, 500]	REFINERYRISKLEVEL

Fig. 3 shows the BT used to train TowoT. In this BT, there is a task named *TowotBTPerception* that reads the TowoT perception and modifies the blackboard values of the BT. The second task is the TN that runs in parallel and that is in charge of storing the learning data.

During the training phase, designers must play the role of the TowoT AI, replacing the TN of this BT. The user interface will show the game state as perceived by the *TowotBTPerception* node (environment box, top-right corner of Fig. 2). It consists of the energy (life level) and risk level of the four main player entities in the game: TowoT, core, refinery, and the player themselves. Table I summarizes the parameter names and their value ranges. Risk levels ([0, 4]) saturate when the fourth enemy is reached.

When the designer wants to instruct TowoT to follow a primitive (low-level behavior), he pauses the game and the interface allows them to choose one of those actions created by programmers in advance (left-hand side of Fig. 4). In TowoT, the options available are: TOWOTRECHARGE, PROTECTCORE, PROTECTJACOB, and PROTECTREFINERY. The TN will pair together the current game state and the chosen action as an indication of the desired final global behavior. This relationship is stored not only when the designer chooses a new action, but also during game cycles where the same action is kept.

After training, BB will generate a BT inferred from the observed data (the TBT) and, at runtime, will replace the TN with it.

Fig. 4. Interface of the training phase implemented in TowoT.

Designers can then test the learned behavior and confirm that it is correct. If they are not satisfied with the final result perceived in game mode, the following three different alternatives are available:

1) forget the last learning episode and retrain the TN;
2) fine-tune the machine-generated BT, using the BB editing tools; and
3) if designers missed some primitive actions during the training phase to control the NPC, they can ask a programmer to implement them and retrain the TN using them.

In order to incorporate our trained behavior trees into any game development process, designers must be able to pause the game and instruct the NPC to execute one of the available actions during the training process. At the same time, the TN must sample the environment and serialize the state. All these run-time requirements become a specific scene in the Unity 3D project. In order to ease the use of TBTs, it exposes an API so training scenes can be created for virtually any game. Specifically, the game must be able to perform the following.

1) Pause and restart the TN and the game: when the TN is paused, the node executes a callback to inform the game about the set of tasks available in that moment. Therefore, programmers should create a user interface that allows designers to select the task to execute.
2) Change the behavior: while the node is paused, the game can change the task that will be executed in the next node update.
3) The value of the internal parameters of the BT, that are not accessible in other circumstances, can be watched by the game with the aim to help the designer when they select the task to run in the training as well as detects errors in the game phase.

As said before, when running in game mode, the TN is replaced with the machine-generated TBT. The process to infer the behavior starts with a decision tree created with the C4.5 algorithm [12]. We use the C4.5 C# implementation included in the library Accord.Net[4] configured with the default setup values given by this implementation. The resulting decision trees have

[4]http://accord-framework.net

TABLE II
EXAMPLE OF RULES GENERATED FROM A DECISION TREE

1. TOWOTRECHARGE ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL ≤ 0.5
2. PROTECTJACOB ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL $> 0.5 \wedge$ CORERISKLEVEL ≤ 0.5
3. PROTECTJACOB ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL $> 0.5 \wedge$ CORERISKLEVEL > 0.5
4. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $\leq 0.5 \wedge$ REFINERYRISKLEVEL ≤ 2.5
5. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $\leq 0.5 \wedge$ REFINERYRISKLEVEL $> 2.5 \wedge$ JACOBRISKLEVEL ≤ 0.5
6. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $\leq 0.5 \wedge$ REFINERYRISKLEVEL $> 2.5 \wedge$ JACOBRISKLEVEL > 0.5
7. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL > 1.5
8. PROTECTCORE ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL $\leq 1.5 \wedge$ JACOBRISKLEVEL ≤ 1.5
9. PROTECTCORE ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL $\leq 1.5 \wedge$ JACOBRISKLEVEL > 1.5

TABLE III
EXAMPLE OF RULES SIMPLIFIED

1. TOWOTRECHARGE ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL ≤ 0.5
2. PROTECTJACOB ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL > 0.5
4. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL ≤ 0.5
7. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL > 1.5
8. PROTECTCORE ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL ≤ 1.5

TABLE IV
EXAMPLE OF FINAL RULES

1. TOWOTRECHARGE ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL ≤ 0.5
2. PROTECTJACOB ← TOWOTENERGY $\leq 33.2 \wedge$ JACOBRISKLEVEL > 0.5
4. PROTECTREFINERY ← (TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL ≤ 0.5) $\vee$ (TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL > 1.5)
8. PROTECTCORE ← TOWOTENERGY $> 33.2 \wedge$ CORERISKLEVEL $> 0.5 \wedge$ REFINERYRISKLEVEL ≤ 1.5

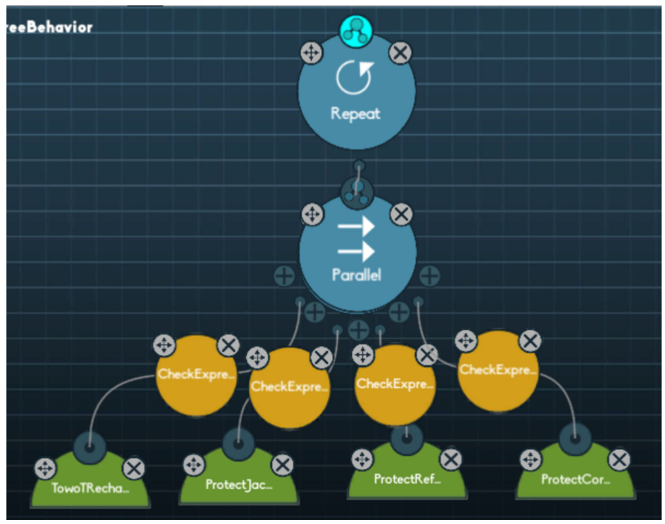

Fig. 5. BT inferred using the training data.

an average validation error of 14.88%. We then generate rules from the learned trees in order to make them more readable and easier to understand.

The decision tree is converted into rule clauses, traversing its nodes with a depth-first search so it is flattened. The head of the clauses (left part) is the action that will be executed and the body of the clause (right part) is the condition that must be fulfilled. These conditions are built as the conjunction of every internal node between that leaf and the root. An example of some rules extracted from the tree are shown in Table II.

We need one rule for each task, but the decision tree generates some redundant rules. For example rules 2 and 3 can be condensed because all terms, but those ones regarding CORERISKLEVEL are the same, and, when OR'ed, the different terms become a tautology. The system is capable of finding these redundancies and condensing them automatically. To carry out this simplification, the algorithm looks for an intermediate node that has two leaf nodes with the same action. Then, the system evaluates whether the condition covers all of the available values of the attribute. In this case, as both sides of the nodes reach the same action, the intermediate node can be replaced by a leaf node with this action. The process is repeated until no more simplifications are possible. This greedy algorithm is fast and simplifies the rules enough.

Once reduced, the simplified rules are shown in Table III.

To generate the tree, we join the rules with the same task with an OR operator. The final rules are show in Table IV.

Rule 4 could be simplified even further:

4. PROTECTREFINERY ← TOWOTENERGY $> 33.2 \wedge$ (CORERISKLEVEL $\leq 0.5 \vee$ REFINERYRISKLEVEL > 1.5

although currently BB is unable to do that.

Finally, we create a BT with a parallel node that executes one of the tasks available. To ensure that the behavior is not executed until its condition is satisfied, each task is protected with a guard node that checks the condition extracted from the rules (see Fig. 5).

IV. EXPERIMENT

The aim of the experiment is to test whether the system is useful for game designers and if they are able to make behaviors using our tool.

As described earlier, we use an in-house developed game (TowoT) as a testbed. The main NPC to train is TowoT, the robotic character that accompanies the player. We measure the system's performance at creating behaviors using an automatic metric by conducting a user survey.

The experiment was carried out by 30 students in a master's program on game design. The participants were not students of the authors but we were aware that they knew the BT formalism. They were developing their master's degree final projects where they had to deal with the design and programming of NPCs.

Fig. 6. Enemy waves.

Their backgrounds were diverse, from computer engineers to artists, journalists, physicists, and architects. About 57% of the users had good programming skills and came from software-related degrees, whereas 43% had medium or low programming skills.

A. Experimental Setup

The experiment was carried out as follows. After a short presentation of the game and the structure of the experiment, we asked participants to play a simplified version of the game where a mining tower (the refinery) and a charging station had already been put in place.

The minimap shown in Fig. 2 presents the charging station as a petrol pump icon and the mining tower as an oil extraction tower, both near its center. The yellow spot shown at the bottom marks the position of the spaceship.

In the experiment, designers played the same level several times. A level in our game, as in many other tower defense games, is split into several enemy waves (three in our case) with increasing difficulty, due to the growing number of enemies as the level progresses. Fig. 6 shows the state of the minimap in particular moments of the three waves, where the small red icons represent the enemies. The first wave comes from the enemy generator in the upper left corner of the map, the second wave from the enemy generator placed in the bottom-left, and in the third wave, enemies appear simultaneously from upper left, upper right, and bottom-right. Enemies randomly select which element to attack with certain probabilities. The target most likely to be attacked is the core, then the mining tower, and finally, the charging station. The experiment was divided into two episodes, both with a training phase and a game phase. The level was the same in both and the difference was the strategy used by the designers while defining the behavior of the NPC. In both episodes, during the training phase, designers had to control both the player and the TowoT. During each training phase, between 2000 and 3500 game state–action pairs are collected.

Later, in the game phase, the system uses the traces recorded to create a behavioral model using a decision tree and automatically exporting the equivalent BT that replaces the TN. All this process is transparent to the user. They just had to control the player while TowoT moved autonomously, guided by the inferred TBT.

As mentioned in Section III, TowoT had only four possible behaviors: protect the spaceship, protect the mining tower, protect Jacob, and recharge itself; they were previously created by programmers and taken as building blocks. When protecting the spaceship or the mining tower, TowoT goes to the vicinity of the spaceship or the tower and moves around. When protecting Jacob, TowoT chases the player's avatar, and when recharging, it moves to the charging station to refill its energy. In any of those states, TowoT will fire at any enemy entering its firing range.

Since we wanted to obtain comparable results, we asked all the designers running the experiment to train the TowoT in the same way, following the same strategy. In the first episode, they should play using a defensive strategy: place the TowoT to defend the spaceship, make the player defend the mining tower, and take care to recharge the TowoT between enemy waves. The second episode consisted of playing in a more aggressive way (offensive strategy): ask TowoT to protect Jacob, and use Jacob to wait for the enemies close to their spawn point, since combined fire from Jacob and TowoT would destroy most of them as they emerge from the generator. For the final wave where enemies came from different points (see Fig. 6, right), they had to split forces, with TowoT defending the spaceship and Jacob the mining tower. Each training phase took about 4–5 min and was used to create the traces that were used in the game phase afterwards.

The description of each trace consists of a set of relevant parameters (the game state described in Section III) and the task executed in each moment. The information is gathered from time to time, after a customizable time frame (100 ms in the experiment). As described in Section III, the game state was visible to designers during the training (upper right corner of Fig. 2).

The game ends with defeat if the refinery or the main ship (core) is destroyed and also if Jacob is shot down. It ends with victory if all the enemies are eliminated. Victory or defeat has no impact on the quality of the behaviors generated because it also depends on what the player does.

At the end of each episode, users rated the quality of the behavior learned, comparing what they had instructed TowoT to do during the training phase and what it really did in the game phase. Once the experiment was completed, users were also asked to fill out a survey related to their global satisfaction using this methodology.

B. Experimental Results

As said before, designers were asked to first train TowoT and then play the level again, using the same strategy, whereas TowoT was controlled by the newly created TBT. TowoT should have a behavior analogous to that in the training phase, because both executions would be quite similar. The scores awarded by users (from 1 to 5) subjectively rated that the behaviors trained and learned were similar and do not rated if the behaviors were good or bad (if they won or lost). We also compared both training and game phases calculating the Levenshtein distance between two vectors created with the sequence of actions performed by TowoT on each phase. This distance is then normalized using the vector lengths and used to calculate the vectors' (phases) similarity.

TABLE V
AVERAGE AND STD. DEVIATION OF THE SIMILARITIES BETWEEN THE TRAINING PHASE AND RANDOM/TBT AI

	Average	Std. deviation
Episode 1		
Random AI similarity	0.237	0.114
TBT AI similarity	0.858	0.082
User scores	3.77	1.01
Episode 2		
Random AI similarity	0.234	0.152
TBT AI similarity	0.717	0.121
User scores	3.19	1.28

Obviously, TowoT behavior would not exactly match between executions, because usually in a game, some random ingredients exist. For example, an enemy can attack the base or the refinery with some probability, or the player actions do not take place at exactly the same time. Under these circumstances, it would be difficult to get an idea about the quality of the similarity values because a perfect match (that would receive a 1 value) is nearly impossible. To solve this issue, we also run the episodes using a random AI and calculated the same similarity with it to have a baseline for comparison.

Table V summarizes the results of the comparison between the actions chosen by the designers during the training phases and both the random and TBT AI. The average similarity in both episodes is more than three times better than the random AI although it is worse in the second one, where designers were asked to play with an offensive strategy. We blame the additional complexity of this strategy for this lower result. Although this could cast some doubts about the use of our technique for complex behaviors, when compared with other machine learning alternatives, TBTs provide an extra advantage. Whereas neural networks or genetic algorithms are black-box techniques, TBT is an editable BT that designers, or programmers, can examine, understand, and fine-tune to circumvent the complex behavior learning difficulties.

Although the random baseline AI provides some insights about the quality of the comparison function, we also wanted to check that the similarity values calculated with the Levenshtein distance were reliable. We compared the calculated similarity with the users' ratings using a Pearson product-moment coefficient between both. The correlation value was 0.68, which shows some evidence that both measures are related. Fig. 7 shows the regression line obtained between them.

The last result from the experiment comes from the survey performed at the end. We asked the students the following questions.

1) Do you think it is easier to implement the behavior using TBT or a conventional BT editor?
2) Do you think you could train a behavior in a game using this tool without the loss of the character control?

The responses of the two questions are summarized in Table VI. In total, 87.9% of interviewees think that it is easier to make behaviors with TBTs than using a standard BT editor and 76.2% think that they could use this tool to make behavior without the loss of the character control. This is important because we feel that designers are usually afraid that behaviors created with machine learning techniques can produce unpredictable behaviors.

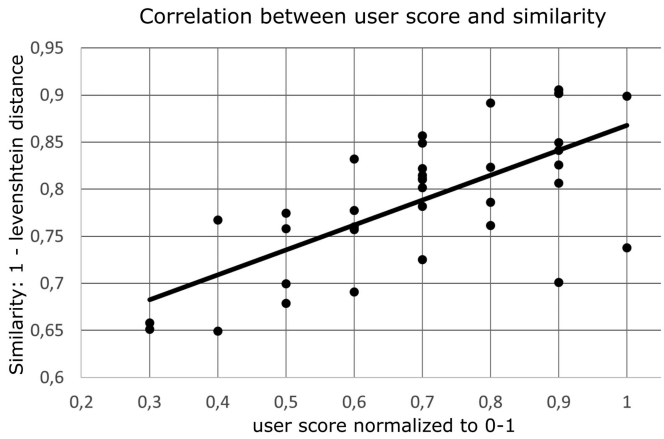

Fig. 7. Comparing similarity and score.

TABLE VI
RESULTS OF QUESTIONS 1 AND 2

Answer	Question 1	Question 2
Yes	87.9%	76.2%
No	12.1%	23.8%

V. CONCLUSION AND FUTURE WORK

We have presented a novel approach to facilitate the collaboration among programmers and game designers in game development. Designers can be seen as end-user programmers that need to build a very complex software system in close collaboration with their programmer partners. TBTs combine PbD and BTs to put in the hands of game designers the tools for creating complex behaviors for NPC in games.

Significant work has been done under the label PbD or learning from demonstration [13] in games. PbD is a subfield of machine learning that studies how to perform a task by observing the behavior of an expert while doing the task that the system has to learn. When used for NPC behavior creation in games, designers play the game while controlling the character they want to define and the system uses the information gathered in those play sessions to automatically build the behavior. A proven approach is to apply case-based reasoning (CBR) to decide which actions should be performed based on the actual state of the world and the actions recorded in the training [14].

PbD has been used in the literature for building AIs in multiple contexts, such as the one reported by Floyd *et al.* [15] that uses it for training agents to play in the RoboCup Simulation League by observing other teams playing. Another example is [14] that describes its use in a real-time strategy (RTS) game named *Darmok 2*. They manage to learn plans whereas designers

play the game, and then they use these plans to make automatic decisions, controlling the game with an autonomous agent, using CBR. Other examples of PbD and CBR are the one presented by Rubin and Watson [16] in the creation of a poker playing agent, and the AI created by Jaidee *et al.* [17] for coordinating multiple agents in Wargus, an RTS game created for research purposes.

PbD has been combined with other techniques such as hidden Markov models [18] where it is used to learn a model of the behavior of agents in real-time strategy games. Finally, a comparative study between PbD and other techniques employed to imitate human plays such as dynamic scripting [20] and neuroevolutionary [21] using *Super Mario Bros* (Nintendo, 1985) as a test bench is given in [19].

The main drawback of the reviewed related work is that the goal is to produce a good quality AI for the NPC in the game, but they are essentially black-box techniques that do not bother to explain to the designer the model learned by the system. In general, this is not a problem, because the main goal is to have a good decision system, but when the main goal is the applicability of these techniques in the game industry, it is crucial that the designer has complete control over the behavior learned, since designers are responsible for putting together a fun experience for the player. In this context, techniques such as CBR or other black-box techniques like neural networks do not provide enough control to game designers.

Another aspect that distinguishes our approach as especially adequate for professional applications is the use of BTs that are state-of-the-art technology in use in the video game industry. Previous work has combined BTs with other techniques such as computational biology [22] or Q-learning [23]. Both approaches are interesting to build behavior combined with BTs and furthermore, both produce good results, but also generate opaque behaviors for the designer. In [24], a grammatical evolution system is used to evolve BT structures. BTs in this case are either evolved to both deal with navigation and react to elements of the game, or used in conjunction with a dynamic A* approach. Some authors [24] point out as a distinguishing feature of their approach that the resulting solutions are human readable, and easy to analyze and fine-tune, addressing a concern of the game industry regarding evolutionary approaches. Nevertheless, with their evolutionary approach, they can only produce simple navigation and reactive behavior.

To the best of our knowledge, there are not any previous experiences in combining BTs and PbD or using decision trees to create subbehaviors on the fly. The proposed combination of techniques has a number of benefits, which are as follows.

1) Designers have more control on the AI implementation. This combination of techniques provides additional tools for the designer to control the implementation of NPC behavior and therefore the user experience.
2) It is easy to use without technical expertise. Game designers can create the behavior for NPCs just by playing the game while controlling them.
3) It allows the use of BTs for nontechnical designers. BTs are state-of-the-art technology for programming complex behavior in games, but some game designers (especially those with little technical knowledge) can have some difficulties when creating behaviors using them. TBTs propose a BT to the designer based on his game traces, and therefore greatly facilitate the creation of the tree.
4) It is also useful for technical designers. Our previous experiments have shown that game designers with programming skills also find TBT useful. It supports rapid iteration of NPC behavior and facilitates the exploration the different alternatives.
5) Supports the collaboration between programmers and designers. BTs are a good framework to integrate portions of behavior as subtrees. In the same tree controlling a given NPC, we can find subtrees produced by both designers and programmers.
6) Closing the gap between industry and academia. PbD are much more popular in academia than in the game industry. TBTs are a step toward bridging this gap by bringing an academic technique, PbD, into an industrial one, BTs.
7) Closing the gap between industry and academia. PbD are much more popular in academia than in the game industry. TBTs are a step toward bridging this gap by bringing an academic technique, PbD, into an industrial one, BTs.

Nevertheless this approach also has some drawbacks, which are as follows.

1) Scalability: as shown in our experiments, if the complexity of the behavior grows, PbD gets worse results when imitating the intended behavior. This may require further training and become too cumbersome.
2) The implementation of the training environment: for each game where we want to use this system, programmers must implement a training environment where designers can control the NPCs in the training phase. This environment requires an extra development effort.

Regarding future work, we plan to explore the ability of other machine learning techniques to produce BTs, turning them into white-box tunable tools. In [25], a new functional formulation of BTs is proposed, and using this formulation, they demonstrate that BTs generalize decision trees, indicating how this opens up possibilities of learning BTs.

We need to further evaluate the impact of using TBTs in terms of productivity in game production. We plan to make an A/B test with designers that have technical expertise and have a control group producing BTs just with a visual editor, and an experimental group using TBTs.

We also plan to test TBTs to train more complex behaviors for other types of games. In particular, we are interested in extending the approach to train a set of coordinated NPCs such as those appearing in first-person shooter games. BTs are specially suited for specifying this type of coordinated behavior and it is an interesting challenge to extend our approach to such a scenario.

ACKNOWLEDGMENT

The authors would like to thank the editor and anonymous reviewers for their constructive comments, which helped them improve the manuscript.

REFERENCES

[1] B. Pfeifer, "Creating designer tunable AI," in *AI Game Programming Wisdom 4*, S. Rabin, Ed. Newton Centre, MA, USA: Charles River Media, 2008, ch. 1.3, pp. 27–57.

[2] D. Isla, "Handling complexity in the Halo 2 AI," in *Proc. Game Developers Conf.*, 2005. [Online]. Available: https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

[3] P. D. Drew Rechner, "Blending autonomy and control: Creating NPCs for Tom Clancy's The Division," in *Proc. Game Developers Conf.*, 2016. [Online]. Available: https://www.gdcvault.com/play/1023056/Blending-Autonomy-and-Control-Creating

[4] S. Rabin, "Implementing a state machine language," in *AI Game Programming Wisdom*, S. Rabin, Ed. Newton Centre, MA, USA: Charles River Media, 2002, ch. 6.5, pp. 314–320.

[5] A. J. Champandard, "10 Reasons the age of finite state machines is over," 2007. [Online]. Available: http://aigamedev.com/open/article/fsm-age-is-over/. [Accessed on: Jun. 19, 2017].

[6] A. J. Champandard, "Behavior trees for next-gen AI," in *Proc. Game Developers Conf., audio lecture*, 2007. [Online]. Available: http://aigamedev.com/open/article/behavior-trees-part1/

[7] I. Sagredo-Olivenza, M. A. Gómez-Martín, and P. A. González-Calero, "Supporting the collaboration between programmers and designers building game AI," in *Entertainment Computing—ICEC 2015*, vol. 9353, K. Chorianopoulos, M. Divitini, J. B. Hauge, L. Jaccheri, and R. Malaka, Eds. New York, NY, USA: Springer-Verlag, 2015, pp. 496–501.

[8] D. Isla, "Halo 3—Building a better battle," in *Proc. Game Developers Conf.*, 2008. [Online]. Available: https://www.gdcvault.com/play/497/Building-a-Better-Battle-HALO

[9] A. J. Champandard, "Getting started with decision making and control systems," in *AI Game Programming Wisdom 4*. Newton Centre, MA, USA: Charles River Media, 2008, ch. 3.4, pp. 257–264.

[10] G. Flórez-Puga, M. A. Gómez-Martín, P. P. Gómez-Martín, B. Díaz-Agudo, and P. A. González-Calero, "Query enabled behaviour trees," *IEEE Trans. Comput. Intell. AI Games*, vol. 1, no. 4, pp. 298–308, 2009.

[11] D. Llansó, M. A. Gómez-Martín, and P. A. González-Calero, "Self-validated behaviour trees through reflective components," in *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE 2009*, C. Darken and G. M. Youngblood, Eds. Palo Alto, CA, USA: AAAI Press, 2009.

[12] J. R. Quinlan, *C4.5 Programs for Machine Learning*. Amsterdam, The Netherlands: Elsevier, 2014.

[13] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "On-line case-based planning," *Comput. Intell.*, vol. 26, no. 1, pp. 84–119, 2010. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8640.2009.00344.x

[14] S. Ontanón and A. Ram, "Case-based reasoning and user-generated artificial intelligence for real-time strategy games," in *Artificial Intelligence for Computer Games*. New York, NY, USA: Springer-Verlag, 2011, pp. 103–124.

[15] M. W. Floyd, B. Esfandiari, and K. Lam, "A case-based reasoning approach to imitating robocup players," in *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, D. Wilson and H. C. Lane, Eds. Palo Alto, CA, USA: AAAI Press, 2008, pp. 251–256. [Online]. Available: http://www.aaai.org/Library/FLAIRS/2008/flairs08-064.php

[16] J. Rubin and I. D. Watson, "On combining decisions from multiple expert imitators for performance," *Proc. Int. Joint. Conf. Artif. Intell.*, 2011, pp. 344–349.

[17] U. Jaidee, H. Muñoz-Avila, and D. W. Aha, "Case-based goal-driven coordination of multiple learning agents," in *Case-Based Reasoning Research and Development*. New York, NY, USA: Springer-Verlag, 2013, pp. 164–178.

[18] 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 *Proc. 7th AAAI Conf. Artif. Intell. Interact. Digit. Entertainment*, 2011, pp. 20–25.

[19] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis, "Imitating human playing styles in Super Mario Bros," *Entertainment Comput.*, vol. 4, no. 2, pp. 93–104, 2013.

[20] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, "Adaptive game AI with dynamic scripting," *Mach. Learn.*, vol. 63, no. 3, pp. 217–248, 2006.

[21] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evol. Intell.*, vol. 1, no. 1, pp. 47–62, 2008.

[22] G. Robertson and I. Watson, "Building behavior trees from observations in real-time strategy games," in *Proc. Int. Symp. Innov. Intell. Syst. Appl.*, 2015, pp. 1–7.

[23] R. Dey and C. Child, "QL-BT: Enhancing behaviour tree design and implementation with Q-learning," in *Proc. IEEE Conf. Comput. Intell. Games*, 2013, pp. 1–8.

[24] M. Nicolau, D. Perez-Liebana, M. O'Neill, and A. Brabazon, "Evolutionary behavior tree approaches for navigating platform games," *IEEE Trans. Comput. Intell. AI Games*, vol. 9, no. 3, pp. 227–238, Sep. 2017.

[25] M. Colledanchise and P. Ögren, "How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees," *IEEE Trans. Robot.*, vol. 33, no. 2, pp. 372–389, Apr. 2017. [Online]. Available: https://doi.org/10.1109/TRO.2016.2633567

Authors' photographs and biographies not available at the time of publication.