Self-Validated Behaviour Trees through Reflective Components *

David Llansó, Marco Antonio Gómez-Martín and Pedro A. González-Calero
{llanso,marcoa,pedro}@fdi.ucm.es

Abstract

Developing the AI for non-player characters in a video game is a collaborative task between programmers and designers. Most of the times, there is a tension between the freedom that designers require to include their narrative in the game and the effort required from programmers to debug faulty AI specified by good story tellers who are not programmers.

In this paper is presented an architecture for building the AI of an NPC that extends the component-based approach, which represents the functionality of an entity as a collection of functionality-specific components. By associating an action in a behaviour tree with a collection of components, and equipping those components with some reflection capabilities, we are able to identify faulty behaviour trees at design time.

Introduction

Behaviour trees (BTs) are a new method for creating the artificial intelligence of NPCs. The framework of BTs provides us with a set of "bricks" that can be used to create the tree by adding internal nodes and leafs. Usually the internal nodes are *composite* nodes that define the strategy of the execution of their children. Common composite nodes are sequences, selectors or parallels.

One of the main advantages of having BTs is that they explicitly represent the behaviour of the entities. In that sense, they can be treated as data and therefore they can be analysed in order to extract information both during design (or creation) time and during its execution.

As BTs are an intuitive method for describing behaviours, both programmers and designers are able to build them with the purpose of creating the AI of the NPCs (Isla 2008). A BT may be stored in an external file that is read during the execution of the game, having a data oriented architecture that reduces the edit-compile-run cycle.

In order for designers to build these files, usually some support tools are created in the form of tree editors. Usually these tools provide designers with a set of composite nodes and basic actions that can be used to create the entire tree. A designer may create its structure and define the parameters of every node (such as the failure policy of a composite node or the number of milliseconds to wait in a `Wait` action).

During the creation of the game, all these built BTs are assigned to different races, or concrete NPCs, by using the map editor. Our proposal is to enhance the tool to check if this assignment is correct. In this way, designers would have an extra check that would aid them when creating BTs and NPCs. By adding reflective capabilities to the components, which provide an entity behaviour, we are able to detect at design time if the NPC would be able to perform the selected behaviour. As the check would be done *before* the execution of the BT, designers would be more confident about the correct link between NPCs and BTs.

The rest of the paper runs as follows. Next Section describes the two techniques we are extending: component-based game entities, and behaviour trees. Next Sections present the core of the contribution by describing the idea of self-validated behaviour trees, a model of execution of primitive actions in BTs through collections of components, and the reflective capabilities required on components. Then a detailed example is provided before concluding the paper.

Background

Components

Traditional object-management systems are often based on inheritance hierarchy where all different kinds of entities derive from the same base class. Classes, directly derived from the base class, are usually also abstract and they represent a split in the tree between classes with different functionalities. During the game's development, some decisions are made about how to split the tree but, due to changeable nature of video games, sometimes those decisions could turn into bad decisions in the future.

For example, if we had a traditional inheritance tree as in Figure 1 and we wanted to add a new different type of enemy called `CHumanEnemy`, with the same qualities as the player such as driving vehicles, we would have to botch the tree to allow it, giving most of the `CPlayer` class content to `CActor` class. This kind of decision would cause that our tree becomes increasingly top heavy and it would cause also that classes at the bottom had some unnecessary qualities.

It would be worse if we wanted to allow the `CBreakableDoor` class which would be able to be dam-

*Supported by the Spanish Ministry of Science and Education (TIN2006-15202-C03-03)

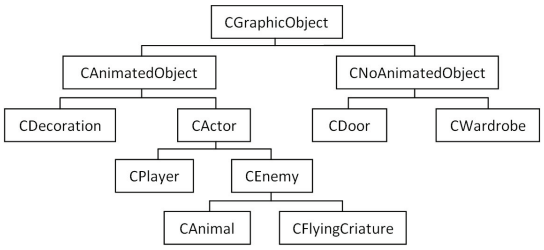

Figure 1: Traditional inheritance tree.

```
<BluePrints>
    <entity type="HumanEnemy">
        <component type="AnimatedGraphic"/>
        <component type="Physic"/>
        <component type="BTExecuter"/>
        <component type="Move–To"/>
        <component type="DummyBehaviours"/>
        <component type="Carry"/>
        <component type="Skills"/>
        <component type="Use–Vehicle"/>
    </entity>
    ...
</BluePrints>
```

Figure 2: `CHumanEnemy` entity built by components.

aged and to be destroyed like actors. If we wanted to allow it in the tree we would have to botch the tree again because if it inherited from `CActor` class it could not be opened like regular doors but if it inherited from `CDoor` class it could not be attacked and destroyed.

Some of the consequences of this extensive use of class inheritance are, among others, an increase in the compilation time (Lakos 1996), a code base difficult to understand and big base classes. To mention just two examples, the base class of Half-Life 1 (Valve Software 1998) had 87 methods and 20 public attributes while Sims 1 ended up with more than 100 methods.

Due to all these problems developers tend to use a different approach, the so called component-based systems (West 2006; Rene 2005; Buchanan 2005; Garcés 2006). Instead of having entities of a concrete class, which define their exact behaviour, now each entity is just a component container where every functionality, skill or ability that the entity has, is implemented by a component. From the developer point of view, every component inherits from the `IComponent` interface whilst an entity becomes just a list of `IComponents`. In this way, the creation of new entities could be done data-oriented using external files such us simple XML files.

As an example, the previous `CHumanEnemy`, which highlighted the disadvantages of the traditional inheritance trees, would become an entity (Figure 2) sharing some components with the `CPlayer` entity: those components that defines the abilities that `CPlayer` had and we wanted to reuse through inheritance.

Components may require some external information to configure themselves at the beginning of their execution. For example, the `AnimatedGraphic` component shown in Figure 2 would need at least the name of the file with the 3D model and animations. All this information is usually stored in the map file of every game level and it is given to the components during their initialization.

As the components are now generic objects with a common interface independent of their functionality, the usual method invocation is not enough. A piece of code calling a method like `MoveTo()` cannot take place, because no such method even exists. What an entity has now is a *component* (`Move-To` in Figure 2) that is able to move this entity from one point to another, but externally this is just a `IComponent` indistinguishable from other.

The communication must be therefore performed in a different way, using message passing. Every `IComponent` is viewed as a communication port that is able to receive and process messages. A message is just a piece of data with an identification and some optional parameters . So, the components have a method like `handleMessage()` that is called externally to send the piece of information to it; depending on the concrete component, the message can be ignored or processed accordingly. In this scenario, entities play the role of the broadcaster of messages. Both internal components and external modules may send messages to the entity that are automatically distributed among all its components.

An example of working could be one in which the entity in Figure 2 was attacked. When another entity attacked this entity the `Physic` component would be informed by the Physic engine. Consequently, the `Physic` component would send a `Collision` message with the other collided entity as a parameter. So the entity would transmit the message through their components and the components that have the ability to carry out that message would accept and store it. Specifically, in this example, only the `Skills` component can carry out it. Thus then, during the processing time of the `Skills` component, it would check if the collision was referred to an attack and, in this case, the component would evaluate if the entity was wounded and the degree of damage taking into account some skills of both entities such as attack and defence abilities or strengths. If the attack caused any damage the `Skills` component would update its values (such as the life factor) and it would send a `Wounded` (or `Death`) message through the entity. Then, other components such as `Graphic` component would accept and store it for future processing in which it would play the corresponding wounded (or death) animation.

Behaviour Trees

According to the number of papers dedicated to the subject in the editions 3 and 4 of the AI Game Programming Wisdom (Rabin 2006; 2008), Behaviour Trees (BTs) are the technology of choice for designing the AI of non-player characters (NPCs) in the game industry.

BTs are proposed as an evolution for hierarchical finite state machines (HFSMs) intended to solve its scalability problems by emphasizing behaviour reuse (Isla 2005). Nodes in a BT represent behaviours, where an inner node is

a composite behaviour (corresponding to an abstract state in a HFSM) and a leaf in the tree represents an action (corresponding to a concrete state in a HFSM). To promote reusability, the states in a BT (i.e., the behaviours) do not include the conditions that lead to transitions. Those conditions are represented as guards for the behaviours so that the same behaviour can be used in different contexts with different guards. To further promote reusability, behaviours may be parametrized, so that in a particular context parameters are bound to actual values in the map. In this way, a node in such a BT is represented through: a behaviour (be it composite or a primitive action); bindings for the parameters of that behaviour; and a guard condition that must be true, during the execution time of that behaviour, to be activated.

From the different models of execution for BTs, we have chosen one where a BT has an active branch, going from the root to a leave, of behaviours being executed. Every tick of the game, some guards may get evaluated and some behaviours may finish, be it successfully or with failure, eventually leading to the expansion of a new active branch in the tree.

Although more complex types of composites are described in the literature of BTs, for the goals of this paper, we only require three types of composites: sequences, static priority list and dynamic priority list. A sequence composite behaviour executes its children in the order they are defined, succeeding when every children succeeds and terminating with failure whenever one of the children fails. The children behaviours of a sequence are not guarded by conditions. A static priority list is a composite node that evaluates its children guards in order and activates the first child whose guard is true. A dynamic priority list, in its turn, re-evaluates the guards of its children with higher priority (the first child being the one with highest priority) than the active one, and switches to a higher priority child whenever possible.

Self-Validated Behaviour Trees

As we have described earlier, a BT describes the actions that an avatar should executed within the environment using a tree hierarchy. BTs are then assigned to concrete entities either during design time or once the game is in execution.

Most of the implementations of BTs distinguish between two different kinds of failures of the basic actions:

- Failures prior to the execution: these errors appear *before* the beginning of the execution of the action. The action ends without having made any changes to the world. A typical example is the failure of a `Move-To` action because there is no path to the target position. Usually, the solution involves local replanning.
- Failures during execution: before the execution, the action checks the environment and it establishes that the action could completely perform the task. However, something changes in the virtual environment and the task cannot be completed (i.e. this errors are due to the ever changing nature of the world). When these failures happen the actions have already made changes to it. One example is the failure of a `Move-To` action because the path is suddenly blocked, and there is no other way to reach the target. When these failures take place, the response usually looks for other alternative in a different part of the tree.

We can, however, extend the kind of errors that can be found in a BT. Generally speaking, a BT fails when their actions cannot be executed (let us assume that the failure of composite or internal nodes depends on the success or failure of their children that, in the end, are basic actions). Therefore, it makes sense to carefully analyse new conditions that may cause an action to fail:

- The NPC assigned to the BT does not have the ability to execute the action. An example would be an action that follows a path within the environment but requires the NPC to fly. If it did not have that ability, the action would fail. This is an intrinsic limitation of the entity, not a failure of the action itself because of the environment.
- The NPC has the ability to execute the action but it cannot do it under the concrete conditions created by the designer. This is related to the *parameters* of the actions instead of to its nature. For example, an entity might be able to carry objects but the particular object specified on the BT could be too heavy. Once again, the failure is due to an inherent limitation of the NPC.

So, in general, let us say that there are two kinds of errors: those due to the intrinsic nature of the entity that would try to execute the actions (the extended kinds of failures), and those that are related to the state of the environment (the previous).

Our proposal is focused on the extended kind of errors due to the intrinsic nature of the entity that ended up trying to execute the BT.It tries to identify them as soon as possible in order to check, during the design time, if there are some guarantees about the success of the execution of the BT. With this technique, designers would have an extra check that would aid them when creating BTs and NPCs. As the check would be done *before* the execution of the BT, designers would be more confident about the correct link between NPCs and BTs.

When a designer assigned a generic BT to an entity, the first kind of errors can be checked: the association would success if the NPC has all the abilities required by the BT actions. Due to the representation of entities as a list of components, and having into account that components represents somehow the abilities of the entity it is easy to know if the avatar would be able to execute every action: the ability/-component that was the one responsible for that action must be looked for.

Though this approach is really easy to implement (see next section), it is quite coarse-grain and its precision is not good enough.

Therefore, a fine-grained approach should consider the general abilities (or general intrinsic limitations) of the entities but also the limitations due to the parameters of the actions. This is also possible when the parameters have been *hard coded* in the BT during the design time. As it is shown in the next section, we would need to instantiate the component responsible for executing the action and to query it in order to know if the entity had some limitations that would prevent it from executing that action.

Behaviour Tree Executer

When we think about the implementation of the AI of an entity using a BT, the obvious approach would be to create a new component that was able to execute a given BT over that entity. In Figure 2 this component was called `BTExecuter`.

This new component could be seen as an interpreter of BTs. It is meant to read the properties of the entity during initialization time and load the BT file specified on them. It must be able also to handle different messages for changing the current BT in execution; this would be useful when designers wanted to hard coded a change on the behaviour for the sake of the gameplay.

The composite nodes that compound the BT would be executed accordingly by the component. It would track the branch of the BT being executed and would check the condition of the open nodes periodically. When the flow of execution reached a leaf where a basic action resides, it would have to perform it over the entity it belonged to.

This basic actions (such as a `Move-To` action) could be carried out in two different ways. The first approach would consist on running it autonomously by sending messages in every tick over the other components of the entity. For example, in a `Move-To` action the `BTExecuter` component might send update messages of positions in every tick.

In the second approach the `BTExecuter` component would not take the responsibility of executing any of the basic actions but it would delegate its execution to other components that the entity should own. In that sense, the atomic actions that appear in the leaves of the tree would be carried out by other components (such as `Move-To` component or `FollowPath` component). Our `BTExecuter` would become a director of the execution that sent messages with the action information and waited the confirmation messages indicating that executions of these actions had ended, in order of having the BT execution moves forward.

Our proposal is based on this second way. As shown in next sections, if actions were carried out by different components, abilities and skills of the entity would be explicit because BT actions and components that can carry out those actions could be linked. So it would give us the possibility to applied inference mechanisms that would help us detecting possible execution fails of BTs associated to specific entities during the design time.

let us expose an example in which the `BTExecuter` component had to move the entity (Figure 2) from one point to another. This component would send a `Move-To` message, with some parameter like the target position, to the entity. The entity would transmit the message through their components and the `Move-To` component would accept and store it. Then the `Move-To` component during its processing time would find the correct path for the movement and it would send `ChangePosition` messages periodically, with parameters such as the new position or the movement type, through the entity. Components such as `Physic` component or `Graphic` component would accept and store it for future processing in which they would change positions of the physic and the graphic entities and would play the corresponding animation. Finally, when path was completed, the `Move-To` component would send a confirmation message that the `BTExecuter` component would receive to continue the BT execution.

Reflective Components

The implementation of the process of validating the BT is based on the components. If the entity is specify in terms of components and bearing in mind that a component can be seen as an *ability* that the entity has, it makes sense to try to identify the failures related to the inherent nature of the entities using such description. In order to do that, we will see as starting point the explicit knowledge that is available both in the list of components and in the BT, which contains the list of actions needed to execute the behaviour.

According with the classification of failures that is listed in the previous section, the implementation has to cope with two different limitations: those inherent to the actions and those related to the parameters given to it.

For the first ones, and taking into account that every action is performed by a component, the easy (and naïve) approach would be to make direct associations between actions in final nodes of BTs and components which were able to execute these actions.

As an example, let us suppose a BT that defined a character behaviour that consisted of patrolling from one point to another and, when another character is perceived, it shoots it with a gun. The final BT would have (together with composite nodes and the condition node related to the perception of another entity) a `Move-To` and an `Attack` action. The test would check if the entity assigned to the BT had a `Move-To` component for the patrol and an `Attack` component for shooting at the enemy.

To implement this idea it would be enough to have a table of pairs, a BT action with a list of components that carry out the above-mentioned action. So to validate a BT assignation would suffice to check that, for each action, the entity had at least one component of the component list associated with each action in the table.

Unfortunately, though the idea is easy to implement, it would not be precise enough, because it would give the designer false positives. In more specific cases, the entity would not be able to perform the behaviour while our implementation assured that it would.

There are two different reasons why this could happen. Some components could be associated to BT actions, but they were not always able to carry out all these kinds of actions, either because they needed the collaboration of other components which would be not in the entity or because the component was not able to correctly execute the action with its associated parameters.

For example, if a BT had an action that should make an entity fly from one point to another, the entity might have a `Move-To` component, but at the same time the `AnimatedGraphic` component of the same entity could not be able to play a flight animation. When the `Move-To` component broadcast the `ChangeAnimation` message with the "`flight`" parameter, nobody would be able to perform the action, and therefore the BT would not be suitable for the entity which it wanted to be associated with.

On the other hand, a component that, let us say, allows the entity to take other objects may form part of the entity. When the component was created, it would take some information from the map, such us the maximum weight that entity could carry. If the AI of the entity decided to take some object, it would send a `Take` message with a parameter specifying the entity that had to be taken. If the BT sent the message with an entity whose weight exceeded the maximum load the entity might carry, the message would not be successfully handle by any component.

In order to manage both kinds of errors, we have extended the `IComponent` interface, so that the components can be asked about their ability for performing actions in the BT. Those consults are made by asking the components if they are able to handle a concrete message according to their configuration. So they will be queried by using the messages that BT actions generate during execution time to give instructions to the entity and they include specific parameters of BT actions. Considering the previous examples, we will use these new method to check if there are some component able to process a `Go-To` message that requires the entity to fly, or a `Take` message with a concrete entity.

The proposed implementation consists of when a BT was assigned to an entity during the design time, the final BT actions would be gone through, one by one, asking the entity, through the `canEntityCarryOut()` method, if an action, with its parameter, could be executed. The method would ask the components sequentially about their ability to execute this action, until a component returned true or the list had ended.

The specific components would be those responsible for implementing the `canComponentCarryOut()` method, declared in the interface of the component, reporting which actions can be performed. Depending on the component, it would automatically return true or it would check if their attributes allowed the action to be executed (for example comparing the maximum weight with the one of the entity that is supposed to take). More complex actions would require the component to recursively consult other components about their ability to execute primitive actions such as pick the flight animation. Figure 3 shows the pseudo-code of the implementation.

It is obvious that components should be initialized to receive messages and to check their abilities but full initializations would not be necessary. For example, the `Graphic` component, during the execution time, would create a graphic entity into the graphic engine from the model specified to the component but it would not be necessary during the design time and the component would only need the animation names to know which types of animations would be able to play.

Example

Let us suppose that a game where avatars need wood for their subsistence is being developed. Designers would create BTs such as the one shown in Figure 4 that would represent a behaviour of "wood harvesting". It would be compounded by a composite node that executes in sequential order the actions of its children. Avatars would get in an excavator

```
class IComponent {
  virtual bool canComponentCarryOut
            (Message m){return false;}
  ...
};

class Entity {
  bool canEntityCarryOut(Message m) {
    for each Component c in components {
      if (c.canComponentCarryOut(m))
        return true;
    }
    return false;
  }
  ...
};

bool check(Entity e, BT bt) {
  for each Action a in bt.actions {
    Message m = a.getMsg();
    if (!e.canEntityCarryOut(m))
      return false;
  }
  return true;
}
```

Figure 3: Pseudo-code of the implementation

shovel and use it to go to a known wood source, load wood, drive to camp and unload wood there.

During the design time, this BT could be associated to the entity described in Figure 2. To validate this association, the `check()` method (previous section) would extract all the basic actions of the BT, find out which messages they would eventually send to the components and check if they would be handled by any of the components.

In our example, actions related to excavator shovel, `Get-In` and `Drive-To`, in which the entity tries to get in the vehicle and to drive, could be carried out using the `Use-Vehicle` component listed in Figure 2 and `Move-To` action is carried out by `Move-To` component.

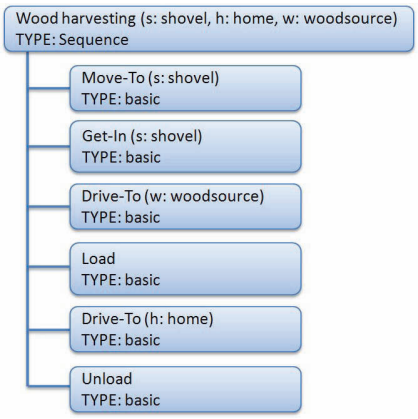

Figure 4: Behaviour Tree 1 of "wood harvesting" type.

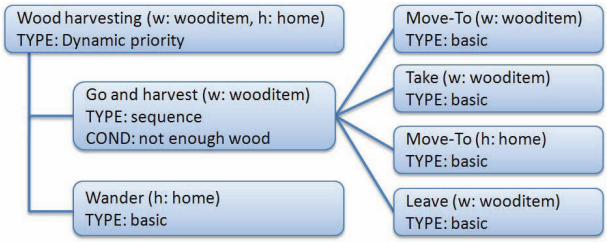

Figure 5: Behaviour Tree 2 of "wood harvesting" type.

These components would ensure that entity's `Graphic` component was able to play the corresponding animations.

However, `Load` and `Unload` wood actions would fail. These actions could not be carried out by any component of the entity. Although a `Use-Vehicle` component existed, it would be generic for all kinds of vehicles and it would permit us to execute actions such as driving, but it would not permit specific actions like how to use a shovel. This would be the reason why this BT could never be associated with this entity, if we did not add a new specific component for the use of an excavator shovel: not all the basic actions of the BT could be executed (i.e. the entity would not have some of the abilities required by the basic actions).

In this situation, designer would be warned about the impossibility of the entity to execute that behaviour. He could then think about extending the entity with the new abilities or using a different BT for the same goal. In that sense, he could create the new behaviour shown in Figure 5. This behaviour is a composite of dynamic priority list type with two nodes: the first one is a sequence that harvests a piece of wood when there is not enough wood at home, and the second one is a dummy behaviour that makes the entity wander around home in other case. As it is explained before, a dynamic priority list re-evaluates the guards of its children periodically. In that sense, the first node would have the condition of not having enough wood at home as its guard, therefore the `BTExecutor` would choose this node only when the condition became true; while it kept on being false, the `Wander` node would be selected. This behaviour would have a parameter that allows the designer to choose the actual piece of wood that would be fetched. He might decide to specify it during design time or precede it with another behaviour that would select it in real time.

To validate the association, `Wander`, `Take`, `Move-To` and `Leave` actions, with their parameters, should be gone through by asking the entity if they can be executed, in the same way as it was explained in the previous example. During the design time the association could be validated because `Take` and `Leave` actions could be carried out by the `Carry` component, `Wander` action by `DummyBehaviours` and `Move-To` action by the `Move-To` component.

If the designer specified the parameter of the behaviour, i.e. the piece of wood to be taken, the check would move a step forward. Instead of just test if all the abilities needed would be available, it can also check if the actions could be performed with the actual parameters. In that sense the behaviour could be ruled out because of the piece of wood specified in the association was too heavy for the entity. During the check of the `Take` action, the `Carry` component would ask the entity if it was able to generate enough force to pick up the piece of wood. If the piece of wood was too heavy, both the `Skills` component which is responsible for generating forces, and the rest would communicate to entity the impossibility of producing enough force, so the `Carry` component would not be able to execute `Take` action and, because of this, the entity would not be able to carry out the BT.

Conclusions and Future Work

In this paper we have presented an architecture that combines BTs and components with some reflection capabilities in order to identify faulty BTs at design time. In this way, designers will have an extra check that will aid them when creating the AI for the NPCs. This is becoming important because designers are more and more involved in the association between AIs and entities. For example, using Kismet, the gameplay editor used in the Unreal Engine, they are able to assign a concrete behaviour to an entity placed on the game map.

We envision the use of similar techniques in real time during the execution of the game. In that sense, the reflective components may be used prior to the execution of the BT. The approach may be useful when the BT is automatically generated using other mechanisms such as planners or CBR (Flórez-Puga et al. 2008).

References

Buchanan, W. 2005. *Game Programming Gems 5*. Charles River Media. chapter A Generic Component Library.

Flórez-Puga, G.; Gómez-Martín, M. A.; Díaz-Agudo, B.; and González-Calero, P. A. 2008. Dynamic expansion of behaviour trees. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. Palo Alto, USA: AAAI Press.

Garcés, S. 2006. *AI Game Programming Wisdom III*. Charles River Media. chapter Flexible Object-Composition Architecture.

Isla, D. 2005. Handling complexity in the Halo 2 ai. In *Game Developers Conference*.

Isla, D. 2008. Halo 3 - building a better battle. In *Game Developers Conference*.

Lakos, J. 1996. *Large Scale C++ Software Design*. Addison Wesley.

Rabin, S., ed. 2006. *AI Game Programming Wisdom 3*. Charles River Media.

Rabin, S., ed. 2008. *AI Game Programming Wisdom 4*. Charles River Media.

Rene, B. 2005. *Game Programming Gems 5*. Charles River Media. chapter Component Based Object Management.

Valve Software. 1998. Half life.

West, M. 2006. Evolve your hierarchy. *Game Developer* 13(3):51–54.