# Experience-Based Design of Behaviors in Videogames

Gonzalo Flórez Puga, Belén Díaz-Agudo, and Pedro Gonzalez-Calero

Department of Software Engineering and Artificial Intelligence,
Universidad Complutense de Madrid, Spain
gflorez@fdi.ucm.es, {belend,pedro}@sip.ucm.es

**Abstract.** Artificial intelligence in games is usually used for creating player's opponents. Manual edition of intelligent behaviors for Non-Player Characters (NPC) of games is a cumbersome task that needs experienced designers. Amongst other activities, they design and integrate the new behaviors with the virtual environment, in terms of perception and actuation over the environment. Behaviors typically use recurring patterns, so that experience and reuse are crucial aspects for behavior design. In this paper we present a behavior editor (eCo) using Case Based Reasoning to retrieve and reuse stored behaviors represented as hierarchical state machines. In this paper we focus on the application of different types of similarity assessment to retrieve the best behavior to reuse. eCo is configurable for different domains. We present our experience within a soccer simulation environment (SoccerBots) to design the behaviors of the automatic soccer players.

## 1 Introduction

Artificial Intelligence for interactive computer games is an emerging application area where there are increasingly complex and realistic worlds and increasingly complex and intelligent computer-controlled characters. Interactive computer games provide a rich environment for incremental research on human-level AI behaviors. These artificial behaviors should provide more interesting and novel gameplay experiences for the player creating enemies, partners, and support characters that act just like human players [1].

The edition of intelligent behaviors in videogames (or simulation environments) is a cumbersome and difficult task where experience has shown to be a crucial asset. Amongst other activities, it implies identifying the entities which must behave intelligently, the kind of behaviors they must show (e.g. helping, aggressive, elusive), designing, implementing, integrating and testing these behaviors in the virtual environment.

Designing new behaviors could be greatly benefited from two features that are common in most of everyday videogames. First of all, modularity in behaviors. That means complex behaviors can be decomposed into simpler behaviors that are somehow combined. Second, simpler behaviors tend to recur within complex behaviors of the same game, or even in different games of the same genre.

For instance, in a soccer game “defend” could be a complex behavior that is composed of two simpler behaviors like “go to the ball” and “clear”; meanwhile “attack” could be composed of “go to the ball”, “dribbling” and “shoot”. Both features are useful to build new complex behaviors based on simple behaviors as the building blocks that can be reused.

In this paper we describe our ongoing work developing a graphical behavior editor that is able to store, index and reuse behaviors previously designed. Our editor (eCo) [6] is generic and applicable to different games, as long as it is configured by a game model file. The underlying technologies of eCo are Hierarchical Finite State Machines (HFSMs) [8] and Case Based Reasoning (CBR). In this paper we focus on the similarity assessment and retrieval processes and give some ideas about our future work on reuse.

HFSMs are appropriate and useful tools to graphically represent behaviors in games, which provide a suitable starting point to automatically generate the code that implements the behavior and that will be integrated in the game [4]. HFSMs facilitate the modular decomposition of complex behaviors into simpler ones, and the reuse of simple behaviors. The eCo behavior editor provides with a graphical interface which allows the user to manually create or modify behaviors just by “drawing” them. Using a CBR-based module, the user can make approximate searches against a case base of previously edited behaviors. Both technologies work tightly integrated. Initially, the case base is empty, so all the editing has to be done via the manual editing (graphic) tools. Once there are enough cases in the case base, new behaviors can be constructed by retrieving and reusing the stored ones.

First, in Section 2, we introduce some general ideas on behavior representation and present the approach followed by the eCo behavior editor. In Section 3 we show a small example of application of the editor to a simulation environment: SoccerBots. Section 4 describes the CBR module integrated in the editor focusing in the different ways of computing similarity. Finally, in Section 5 and 6, we present related work, future goals and conclusions.

## 2 Modelling Reusable Behaviors

In general terms, the execution of a computer video game can be viewed as the continuous execution of a loop of perceiving, deciding the behavior, acting and rendering tasks. The behavior for each NPC basically decides the set of actions or reactions performed by the controlled entity, usually in relation with its environment. In a computer game or simulation, each entity gathers information about its environment using a set of sensors, which could be compared to the senses of the living beings. Depending on this information, the entity performs certain actions, using a set of actuators. In general, the set of sensors and actuators is unique for all the entities of a game and is different for each game or simulation environment, although there will be similarities between games of the same genre. For example, sensors in a first-person-shooter (FPS) game will give access to the position, the steering, the health, the visibility of other entities or

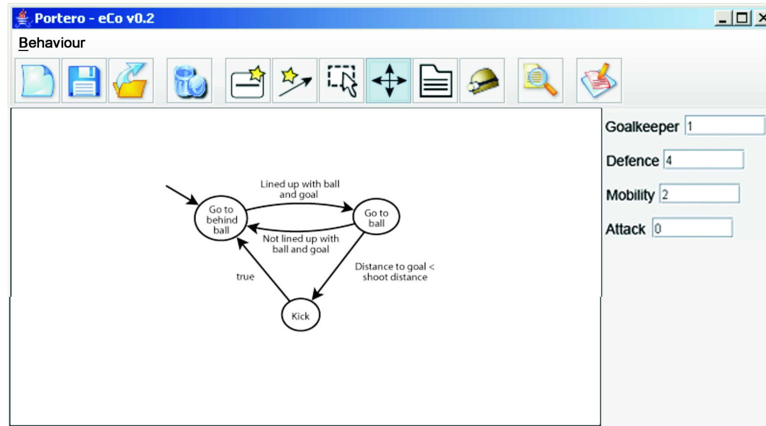



**Fig. 1.** Example of a HFSM

the remaining fuel of a vehicle. Regarding the actuators, the entity can shoot, look at or go to a place, talk to other entities, among others.

Several suitable techniques exist for the representation of behaviors. Due to its expressive power and simplicity, Finite State Machines (FSMs) is one of the most popular techniques. FSMs have been used successfully in several commercial games, like Quake [2], and in game editing tools, like Simbionic [7]. A FSM is a computation model composed of a finite set of states, actions and transitions between states. Simple states are described by the actions or activities which will take place in a state and the transitions point out changes in the state, and are described by conditions formulated over the sensors. One of the drawbacks of the FSMs is that they can be very complex when the number of states grows. To prevent this situation, we used Hierarchical Finite State Machines (HFSMs), which are an extension to the classic FSMs. In a HFSM , besides a set of actions, the states can contain a complete HFSM, reducing the overall complexity and favoring its legibility [8].

We have developed eCo, a game designer oriented tool that represents behaviors using HFSMs. The main module offers a graphical editor to manually "draw" the state machine representing a certain behavior. It includes tools for loading, saving and importing the behaviors from disk, drawing and erasing the nodes and edges, and specifying their content (actions or subordinate state machines, and conditions respectively). Once the behavior is complete, it is possible to use the code generation tool to generate the source code corresponding to the behavior. This tool uses the structure of the state machine together with the information in the *game model* to generate the source file. As the game model and the source file required are usually different for each game, the code generator will also be unique for each game.

The *game model* is a configuration file that describes some details of a game or a simulation environment. Each game model is an XML file, which includes the information about sensors and actuators, and a set of descriptors. The sensors

and actuators are obtained from the game API. Descriptors are the attributes used by the CBR module to describe the behaviors and retrieve them from the case base. The descriptors are obtained through the observation of the characteristics of the different behaviors that exist in the domain of the game.

Every manually designed behavior is stored and indexed and, as behaviors tend to recur, there is a CBR module that allows retrieving and reusing behaviors previously stored. We use XML files to store the cases. Each case in the case base represents a behaviour using the following components:

– Attributes: descriptors that characterize different properties of the behavior. The attributes are different for each game, although similar games (e.g. games of the same genre) will share similar attribute sets. The designer specifies as many attributes as necessary in the game model.
– Description: textual description of the behavior used to fine tune the description given by attributes.
– Enclosed behaviors: specifies which behaviors are hierarchically subordinated. This allows the user to retrieve behaviors which include a specific set of sub-behaviors or actuators.

Next we describe an example using a Soccer simulation environment.

## 3 SoccerBots Example

As we have already mentioned, the behavior editor described in Section 2, and the CBR system that we are describing in Section 4, are independent of any specific game. However, for the sake of an easier exposition we are explaining the basic ideas using a simple game. SoccerBots is a simulation environment developed by Tucker Balch, where two teams play in a soccer match. Simulation time, behavior of robots, colours, size of field, and many other features are configured from a text file. Basically, rules are similar to those from Robocup.

The first step in using eCo to generate behaviors for the SoccerBots environment is to define the game model with the information about sensors, actuators and CBR descriptors of the SoccerBots simulation environment. In the SoccerBots API we can find sensors like getBallX, getBallY, which checks the X, Y position of the ball, getBallR, which checks its distance, and getBallT, which checks its angle. Some examples of actuators (i.e. actions that robots can take) are kick, which kicks the ball, setSpeed(int), which changes the speed of the robot, or setSteerHeading(int), which changes the direction the robot is facing.

As we stated before, the descriptors are obtained through the observation of the characteristics of the different possible behaviors. We used four numeric descriptors to characterize SoccerBots behaviors, namely *mobility* is the ability to move all over the playfield; *attack* is the ability of the robot to play as an attacker; *defence* is the ability of the robot to play as a defender; and *goalkeeper* is the ability of the robot to cover the goal. Next section describes how to deal with these and others ways of describing behaviors in the CBR system.

## 4 CBR for experience based behaviour design

Case Based Reasoning is specially well suited to deal with the modularity and reuse properties of the behaviors; it assists the user in the reuse of behaviors by allowing her to query a case base. Each case of the case base represents a behavior. By means of these queries, the user can make an approximate retrieval of behaviors previously edited, which will have similar characteristics. The retrieved behaviors can be reused, modified and combined to get the required behaviors.

Initially, the case base is empty, so all the editing has to be done via the manual editing (graphic) tools. Once there are enough cases in the case base, new behaviors can be constructed by retrieving and adapting the stored ones. The number of cases necessary in the case base to obtain relevant results will vary from game to game, depending on the complexity of the descriptors and the heterogeneity of the behaviors that can be constructed for that particular game. In the example of the Soccerbots environment, we began with a small case base composed of five cases, and made it grow until we obtain reasonable results for the queries. This happened with a case base of 25 cases. To analyse the goodness of the results of the queries we adopted a subjective criteria but we should work about other more quantifiable criteria.

There are two kinds of queries: functionality based queries and structure based queries. In the former, the user provides a set of attribute-value parameters to specify the desired functionality for the retrieved behavior. In the latter, a behavior is retrieved, whose composition of nodes and edges is similar to the one specified by the query.

### 4.1 Functionality based retrieval

The most common usage of the CBR system is when the user wants to obtain a behavior similar to a query in terms of its functionality. The functionality is expressed by means of a set of parameters, which can be any (or all) of the descriptors of the cases presented in Section 2 (i.e. the attributes, the textual description and the enclosed behaviors).

The parameters that form the query are used to describe the behavior, and are closely related to the game model. The more differences exist between two games, the more different the associated behaviors are and, hence, the parameters used to describe them. The eCo editor provides a query form, showed in figure 2, for the user to enter the parameters of the query.

To obtain the global similarity value between the cases and the query, the similarity of the numeric and symbolic attributes is aggregated with the similarity due to the textual description of each behavior. The user can select the most appropriate operator to combine them in the query form. Some examples of operators could be the arithmetic and the geometric mean or the maximum.

The user enters the query using a form and (s)he can also select the similarity measure used to compare them to the ones in the case base. Descriptor based similarity is based on standard similarity measures here, like the normalized difference value for numbers.

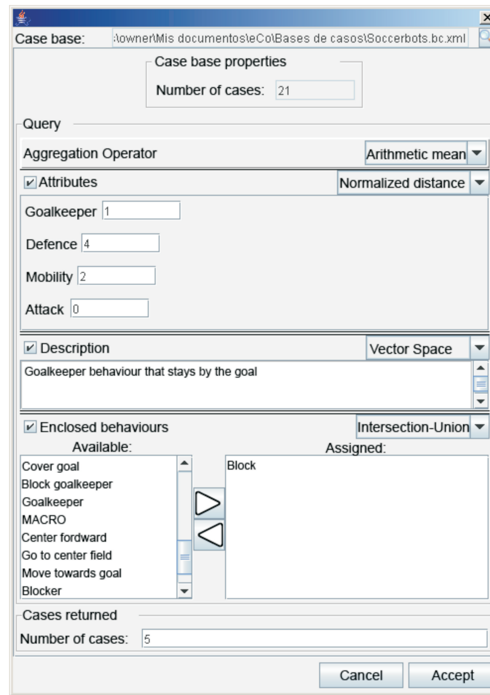



**Fig. 2.** Functionality based queries

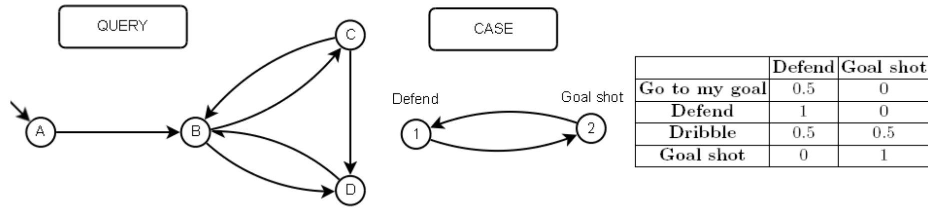
To easy the querying process, the user can use a textual description that is used to fine tune the query by including in it characteristics not considered by the attributes. Each case is described by a short textual description of the represented behavior. For instance, in the previous example, the user is requesting a behavior that stays near the goal. This descriptor was not included in the game model, as it is not relevant for most of the behaviors. Instead, the textual description is used. In the current version, we use the vector space model [10] to compute the similarity measure between the text descriptions.

### 4.2 Structure based retrieval

There are cases in which the behaviour designer knows the general structure of the state machine (i.e. the distribution of the nodes and edges and the generic functionality of them). In these cases, it would be easier and faster for the designer if he could "draw" the state machine and let the editor find a similar state machine in the case base.

Finite state machines are directed graphs, so we can compare them using any of the existing techniques in the literature. In the left part of figure 3 there is an example of a query for a Soccerbots behaviour.

Entering this data, the retrieved state machine would be similar to the query in terms of its shape, but the behaviour it implements could be any. Hence, we




| | Defend | Goal shot |
|---|---|---|
| Go to my goal | 0.5 | 0 |
| Defend | 1 | 0 |
| Dribble | 0.5 | 0.5 |
| Goal shot | 0 | 1 |

**Fig. 3.** Query and case for structure based retrieval and similarity between nodes

need to allow the behaviour designer to point out the desired functionality of the retrieved state machine and then, compare the desired functionality with the functionality implemented in the nodes of the state machines in the case base.

The functionality of the drawn nodes is expressed linking each node to a functionality query (see Section 4.1) that the user must build to expresses the desired behaviour that should be contained in the node. The linked functionality queries are compared to the descriptors in the nodes of the behaviours in the case base during the query process. In the aforementioned example, and for the sake of simplicity, instead of expliciting the whole functionality query, we will use a descriptive name to express it. Thus, for instance, the user could link node $A$ to a behaviour whose desired functionality is “Go to my goal”. To do this (s)he must build a functionality query that expresses this and link it to the node. For the examples we will consider the following linking of the nodes:

- A: “Go to my goal”.
- B: “Defend”.
- C: “Dribble”.
- D: “Goal shot”.

Our approach to these *structure based queries* is to use the drawing facilities of the editor to “draw” the state machine (the behaviour pattern) and then assign functionality based queries to the nodes, which will show the functionality of each node. Figure 4 shows the query editor for the structure based queries. In the left pane the user can draw a behaviour pattern and in the right pane he can specify the desired functionality of the retrieved behaviour by entering a functionality query. Additionally, each node can be linked to another functionality query, as we have already mentioned, to tune up the search.

In the next section we review different techniques to calculate labelled graph similarity and how they can be applied to our specific problem.

## Graph similarity

The graph similarity problem is an issue that has been approached in several

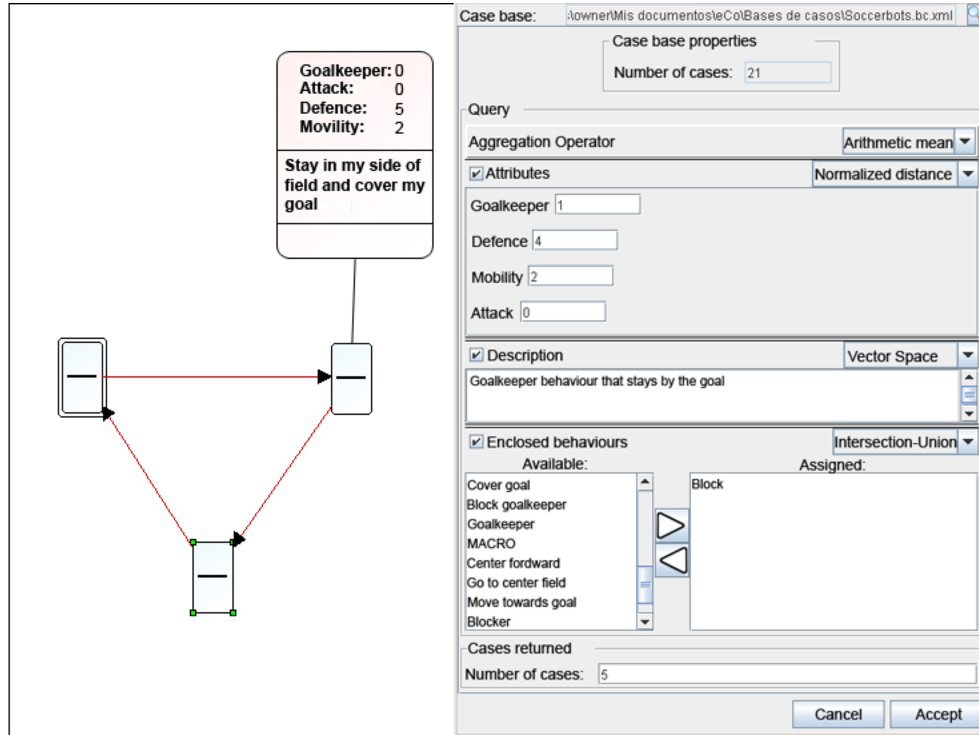



**Fig. 4.** Structure based query editor

different ways in the literature. Each approach has its own advantages and disadvantages. In the following paragraphs we review some of them and explain how we adapted them to solve our current problem, the labelled graph similarity.

*First approach*

Bunke and Messmer's approach [3] is based in the calculation of the weighted graph edit distance, a generalization of the string edit distance [11]. They define a set of edit operations (namely, adding a node (A), deleting a node (D) and editing the label of a node (E), and adding an edge (A'), deleting an edge (D') and editing an edge(E')). Each operation has an associated cost ($C_A$, $C_D$, $C_E$, etc.). Using different sets of cost values will lead us to different results. The edit distance (*dist*) is the minimum cost among all sequences of edit operations that transform the source graph into the target graph. The distance can be converted into a similarity measure by defining a function that uses the distance, like:

$$\text{sim}(G_1, G_2) = [1 + dist(G_1, G_2)]^{-1}$$

For instance, for the example in figure 3, valid sequences of edit operations are:

$$S_1 = \{D(A), D(C)\}$$
$$S_2 = \{D(A), D(B), E(C)[\text{Dribble} \rightarrow \text{Defend}], A'(D, C)\}$$
$$S_3 = \{E(A)[\text{Go to my goal} \rightarrow \text{Goal shot}], D(C), D(D), A'(B, A)\}$$

| $C_1 = 2 \cdot C_D$ | $C_2 = 2 \cdot C_D + C_E + C_{A'}$ | $C_3 = 2 \cdot C_D + C_E + C_{A'}$ |
|---|---|---|

Intuitively, if $C_E$ and $C_{A'}$ are greater than 0, the sequence $S_1$ has the lowest cost, and therefore, is the edit distance.

The sequence associated to the edit distance contains the operations needed to transform one graph into the other, and hence, it can be used to perform the adaptation of the retrieved behaviour later.

In the worst case, the complexity of the computation of the graph edit distance is exponential in the size of the underlying graphs, although it can be speeded up using heuristics and bound techniques.

This approach considers the labels in the nodes and edges of the graphs. Continuing with the former example, in the second sequence we deleted nodes A and B, and added an edge from D to C. After doing this edit operations, the resulting graph is equal in shape to the case graph, but still differs from it in the labels, so we have to use one edit operation to change the label on node C.

One of the limitations of this approach is, as we can see in the example, that all the node editing operations have the same cost ($C_E$) regardless of the labels contained in the nodes. For instance, sequence 2 and sequence 3 have the same cost, but the behaviours in nodes C (Dribble) and 1 (Defend) are more similar than the ones in nodes A (Go to my goal) and 2 (Goal shot). In our approach, as we will see later, we use a cost function. This function takes into account the similarity of nodes in edit operations.

*Second approach*

The approach followed by Champin and Solnon in [5] is based on the definition of correspondences between nodes of the source and target graph.

Each graph $G$ is defined by a triplet $\langle V, r_V, r_E \rangle$ where $V$ is the finite set of nodes, $r_V$ is a relation that associates vertices with labels, and $r_E$ is a relation that associates pairs of vertices (i.e. edges) with labels. $r_V$ and $r_E$ is called the set of features of the graph. A correspondence $C$ between $G_1$ and $G_2$ is a subset of $V_1 \times V_2$, that associates, to each vertex of one graph, 0, 1 or more vertices of the other.

Given a correspondence $C$ between $G_1$ and $G_2$, the similarity is defined in terms of the intersection of the sets of features ($r_V$ and $r_E$) of both graphs with respect to $C$:

$$
\begin{aligned}
&\mathit{descr}\left(G_1\right) \cap_C \mathit{descr}\left(G_2\right) = \\
&\qquad \left\{(v,l) \in r_{V1} \middle| \, (v,v') \in C \wedge (v',l) \in r_{V2}\right\} \cup \\
&\qquad \left\{(v',l) \in r_{V2} \middle| \, (v,v') \in C \wedge (v,l) \in r_{V1}\right\} \cup \\
&\qquad \left\{(v_i,v_j,l) \in r_{E1} \middle| \, (v_i,v_i') \in C \wedge \left(v_j,v_j'\right) \in C \wedge \left(v_i',v_j',l\right) \in r_{E2}\right\} \cup \\
&\qquad \left\{\left(v_i',v_j',l\right) \in r_{E2} \middle| \, (v_i,v_i') \in C \wedge \left(v_j,v_j'\right) \in C \wedge (v_i,v_j,l) \in r_{E1}\right\}
\end{aligned} \tag{1}
$$

$$
\mathrm{sim}_C\left(G_1,G_2\right) = \frac{f\left(\mathit{descr}\left(G_1\right) \cap_C \mathit{descr}\left(G_2\right)\right) - g(\mathit{splits}(C))}{f\left(\mathit{descr}\left(G_1\right) \cup \mathit{descr}\left(G_2\right)\right)}
$$

Where *splits* is the set of vertices from $V_1 \cup V_2$ which are associated with 2 or more vertices by $C$. The total similarity value is the maximum similarity value of all the possible correspondences:

$$\mathrm{sim}\left(G_1, G_2\right) = \max_{C \subseteq V_1 \times V_2} \left\{\mathrm{sim}_C\left(G_1, G_2\right)\right\}$$

The complexity of this problem is, again, exponential in the number of vertices of the graphs being compared, but the use of heuristics and bounding functions can accelerate the search.

This approach is more sensible to the similarity of the labels in the edges. On the other hand, the possible values when comparing one label with another (whether it is a node or an edge label) can only express if they are identical or not. We need a way to compare, not only the shape of the behaviours but also their functionalities and, in the scenario we are dealing with, its uncommon to find two nodes or two edges which have exactly the same labels, so we will need some way to relax this comparison.

*Third approach*

The similarity measure proposed by Wang and Ishii in [12] is also based in the definition of correspondence relations between the nodes of the two graphs.

This method doesn't use the intersection, but an algebraic formula to obtain the final similarity measure. As in the previous approach, the similarity degree of two graphs $G_1$ and $G_2$ is the maximum similarity of $G_1$ and $G_2$ over all the possible correspondences:

$$\mathrm{sim}\left(G_1, G_2\right) = \max_{C} \left\{\mathrm{sim}_C\left(G_1, G_2\right)\right\}$$

and the similarity of $G_1$ and $G_2$ over the correspondence $C$

$$\begin{aligned}
\mathrm{sim}_C\left(G_1, G_2\right) =& \frac{F_n + F_e}{M_n + M_e} \\
F_n =& \sum_{n \in V_1} \frac{W(n) + W(C(n))}{2} \cdot \mathrm{sim}\left(n, C(n)\right) \\
F_e =& \sum_{e \in E_1} \frac{W(e) + W(C(e))}{2} \cdot \mathrm{sim}\left(e, C(e)\right) \\
M_n + M_e =& \max\left(\sum_{n \in V_1} W(n), \sum_{n \in V_1} W(C(n))\right) + \max\left(\sum_{e \in E_1} W(e), \sum_{e \in E_1} W(C(e))\right)
\end{aligned}$$

where $W$ is the weight of a node or an edge.

For this approach, the labels in the nodes and edges are single variables or constants, and their similarity is defined by the following functions:

- For nodes, if the value represented for the constant or variable in both nodes is the same, then the similarity is 1, and 0 in any other case.

– For edges, if the source and target nodes of the edges are related by $C$ and the labels are equal, then the similarity is 1; if the labels are different, the similarity is 0.5 and is 0 in any other case.

In this case we can change this similarity function so we can obtain a more descriptive value. We use a functionality based similarity function (Section 4.1) to compare the descriptors of the nodes. As with the previous techniques, the complexity of this one is also exponential and its also possible to reduce the search space by the use of heuristics and bounding techniques.

**Our approach**

Our approach to the similarity problem in finite state machines is based in both the structure of the state machine and the labeling in the nodes. The labels associated to the nodes are used to express the functionality of the behaviours contained in them.

In our implementation we allow the user to select any of the three techniques explained before to obtain the similarity measure in the structure based retrieval.

*First approach*

This approach is based in the calculation of the edit distance between two graphs. The distance is obtained as the sum of the operations needed to transform one graph into the other.

The cost assigned to each edit operation determines the final distance. In our approach, we are considering the costs of edit operations, not as constants, but as functions defined over the source and target nodes or edges. This way, we can express the intuitive idea that changing one label for another is cheaper in cost if the labels are more similar. For instance, the cost of the edit operation $E(C)[\text{Dribble} \rightarrow \text{Defend}]$ is:

$$cost(E(C)[\text{Dribble} \rightarrow \text{Defend}]) = C_E \cdot (1 - \text{sim}(\text{Dribble}, \text{Defend}))$$

where Dribble and Defend are the labels of the nodes (actually, the labels are the functional descriptors of the behaviours, but we used these descriptive names to simplify the example) and the sim function is the similarity function used in functionality based retrieval in Section 4.1.

We also impose the following restrictions on the possible values of the cost functions, so the results of the distance function are reasonable:

1. $C_E \leq C_A + C_D$ and $C_{E'} \leq C_{A'} + C_{D'}$
   This means that editing the label of a node is cheaper than an addition and a deletion of the same node with different labels.
2. $C_A = C_D$ and $\text{sim}(X, Y) = \text{sim}(Y, X)$
   These two restrictions give symmetry to our distance measure.

For instance, to obtain the similarity between the query and the case in Figure 3, if we use the costs $C_A, C_D, C_E, C_{A'}, C_{D'}, C_{E'} = 1$, and the sequences:

$$S_1 = \{D(A), D(C)\}$$
$$S_2 = \{D(A), D(B), E(C)[\text{Dribble} \rightarrow \text{Defend}], A'(D, C)\}$$
$$S_3 = \{E(A)[\text{Go to my goal} \rightarrow \text{Goal shot}], D(C), D(D), A'(B, A)\}$$

The distances are:

$$d_1 = 2 \cdot C_D = 2$$
$$d_2 = 2 \cdot C_D + C_E \cdot (1 - \text{sim}(\text{Dribble}, \text{Defend})) + C_{A'} = 2 + 0.5 + 1 = 3.5$$
$$d_3 = 2 \cdot C_D + C_E \cdot (1 - \text{sim}(\text{Go to my goal}, \text{Goal shot})) + C_{A'} = 2 + 1 + 1 = 4$$

As we can see, the result of $d_2$ is better than $d_3$ because the labels *Dribble* and *Defend* are more similar than *Go to my goal* and *Goal shot*.

*Second approach*
This approach is based in the definition of a correspondence between the nodes of the query and the case graphs.

As has been seen in equation (1), in page 9, the intersection with respect to a correspondence $C$ only takes into account the nodes and edges who share identical labels. In the case of finite state machines, it is convenient to consider a more relaxed similarity measure, so we can take into account the nodes that are not equal but similar. To address this problem we add a value $\beta$ to each tuple in the intersection. This value represents the similarity between the labels of the nodes or edges:

$$\begin{aligned} &\mathit{descr}\,(G_1) \cap_C \mathit{descr}\,(G_2) = \\ &\quad \{(v, v', \beta) \mid (v, v') \in C \wedge (v, l) \in r_{V1} \wedge (v', l') \in r_{V2} \wedge \beta = \text{sim}\,(l, l')\} \cup \\ &\quad \{\left((v_i, v_j), (v'_i, v'_j), \beta\right) \mid (v_i, v'_i) \in C \wedge \left(v_j, v'_j\right) \in C \wedge (v_i, v_j, l) \in r_{E1} \wedge \\ &\quad \left(v'_i, v'_j, l'\right) \in r_{E2} \wedge \beta = \text{sim}\,(l, l')\,\} \end{aligned}$$

$$\text{sim}_C\,(G_1, G_2) = \frac{f\,(\mathit{descr}\,(G_1) \cap_C \mathit{descr}\,(G_2)) - g(\mathit{splits}(C))}{F}$$

The similarity function we use is the functionality based retrieval similarity (Section 4.1).

The similarity value $\beta$ is used by the function $f$ to obtain the final similarity value, and the constant $F$ is an upper bound of $f$ that maintains the result in the interval $[0, 1]$. For instance, considering the example in figure 3, and the functions:

$$f(I) = \sum_{\text{for each node n in I}} (f_N(n)) + \sum_{\text{for each edge e in I}} (f_E(e))$$
$$f_N((v, v', \beta)) = \beta$$
$$f_E(((v_i, v_j), (v'_i, v'_j), \beta)) = \beta$$
$$g(S) = |S|$$
$$F = \max\{|r_{V1}|, |r_{V2}|\} + \max\{|r_{E1}|, |r_{E2}|\} = 4 + 6 = 10$$

we can have the following similarity values:

- for $C = \{(A, 1), (B, 1), (C, 2), (D, 2)\}$:

$$\begin{aligned} descr(G_1) \cap_C descr(G_2) = & \{(A, 1, 0.5), (B, 1, 1), (C, 2, 0.5), (D, 2, 1), \\ & ((B, C), (1, 2), 1), ((B, D), (1, 2), 1), \\ & ((C, B), (2, 1), 1), ((D, B), (2, 1), 1)\} \\ splits(C) = & \{(1, \{A, B\}), (2, \{C, D\})\} \\ sim_C(G_1, G_2) = & \frac{(3 + 4) - 2}{10} = 0.5 \end{aligned}$$

- for $C = \{(A, 1), (B, \emptyset), (C, 1), (D, 2)\}$:

$$\begin{aligned} descr(G_1) \cap_C descr(G_2) = & \{(A, 1, 0.5), (C, 1, 0.5), (D, 2, 1), \ ((C, D), (1, 2), 1)\} \\ splits(C) = & \{(1, \{A, C\})\} \\ sim_C(G_1, G_2) = & \frac{(2 + 1) - 1}{10} = 0.2 \end{aligned}$$

To simplify this approach, we can consider only the nodes and edges whose $\beta$ is greater than a certain threshold.

*Third approach*

The third approach is also based in defining the possible correspondences between the graphs being compared. In this case, the calculation includes the comparison of the similarity of labels. To adapt it to our scenario we use the functionality based retrieval similarity function, instead of the one proposed.

As a first approach we give all the nodes and edges the same weight (1). The resulting similarity measure is:

$$\begin{aligned} \text{sim}_C(G_1, G_2) = & \frac{F_n + F_e}{M_n + M_e} \\ F_n + F_e = & \sum_{n \in N_1} \text{sim}(n, C(n)) + \sum_{e \in E_1} \text{sim}(e, C(e)) \\ M_n + M_e = & |N_1| + |E_1| \end{aligned}$$

For the example in figure 3 we can have the following results:

- for $C = \{(A, 1), (B, 1), (C, 2), (D, 2)\}$:

$$sim_C(G_1, G_2) = \frac{(0.5 + 1 + 0.5 + 1) + (1 + 1 + 1 + 1)}{4 + 6} = 0.8$$

- for $C = \{(A, 1), (B, 2), (C, 1), (D, 2)\}$:

$$sim_C(G_1, G_2) = \frac{(0.5 + 0 + 0.5 + 1) + (1 + 1 + 1 + 1)}{4 + 6} = 0.6$$

## 5 Related Work

There exist several tools oriented towards the edition of finite state machines. Most of them are general purpose state machine editors (like Qfsm or FSME), which allow a more or less elastic definition of the inputs and outputs (the sensors and actuators) and the generation of the source code corresponding to the state machine in one or more common languages like C++ or Python. Most of them don't allow the use of HFSMs, nor facilitates the use of CBR or some other tool to favour reusing the state machines.

Regarding game editors, most of them are only applicable to one game or, at the most, to the games implemented by one game engine (as is the case of the Valve Hammer Editor). Besides, the vast majority only allow map edition. The few that allow editing the entity behaviors are usually script based, like the Aurora Toolset for Neverwinter Nights.

Finally, there exist some tools like BrainFrame and, its later version, Simbionic, which are game oriented finite state machine editors. These editors allow the specification of the set of sensors and actuators for the game and the edition of HFSMs using that specification. The HFSMs generated by the editor are interpreted by a runtime engine that must be integrated with the game. Currently, there exist a C++ and a Java version of the runtime engine. There are two crucial differences between our approach and the approach used in Simbionic. First of all, the Simbionic editor doesn't offer any assistance for reusing the behaviors, like the CBR approximate search engine integrated into the eCo editor. And second, to integrate a behavior edited with the Simbionic editor with a game, it is mandatory to integrate the Simbionic runtime engine with the game. On the other hand, the eCo editor can generate the source for behaviors in any language, provided we have implemented the appropriate code generator. Besides, it can generate any kind of file, like image captures, summaries of the behaviors or text files.

## 6 Conclusions and Future Work

In this paper we have described an ongoing work using CBR to design intelligent behaviors in videogames. We have developed a graphical editor based on HFSM that includes a CBR module to retrieve and reuse stored behaviors.

One of the main advantages of our approach is that the editor and the CBR module are generic and reusable for different games. We have shown the applicability in a soccer simulator environment (SoccerBots) to control the behavior of the players. As part of the testing stage and to check the editor applicability we have proposed the integration of the eCo editor with other games with very different nature: SoccerBots is a sports simulator, Neverwinter Nights is a role playing computer game, $JV^2M$ [9] is an action game and AIBO is a real life multipurpose robot) and with different integrating characteristics. For instance, while in $JV^2M$ we define the set of sensors and actuators, it is fixed for the other environments; while Neverwinter Nights is highly event-oriented, the rest of the

environments are basically reactive systems. The eCo behavior editor has been easy to use in the different environments and offers a friendly interface. The editor assists the user in the definition of new behaviors through a CBR module that retrieves previously stored behaviors.

In this paper we have described the current state of the work but there are many open lines of work. We have finished the graphical editor, defined the structure of the cases and the game models, and we have been working on case representation, storage and similarity based retrieval. Current lines of work are automatic reuse of behaviors and learning.

By now, the adaptation process is carried out manually by the user, who receives some assistance from the system. The system evaluates the differences between the values of the attributes in the query and the retrieved case and use them to indicate what nodes should be modified.

In the current version, the learning of the CBR system is totally user guided: the user indicates which cases must be stored in the case base and also enters the values for the descriptors. The set of values for each descriptor is a very subjective matter, so it would be a good idea to automatize this process or make the system suggest some suitable values, using machine learning approaches.

Regarding structure based similarity, we have proposed three different approaches to compare finite state machines. Our next step in this issue will be testing them to determine which is the most suitable approach and for what kind of cases.

The use of HFSM offers many possibilities to reuse and combine pieces of behaviors within other more complex behaviors. We are also working on the definition of an ontology about different games genres to be able to reuse behaviors, vocabulary and sets of sensors and actuators between different games of the same genre. This way we can promote the reuse of behaviors, even among different games, while making easier the use of the editor, since the user doesn't need to learn the characteristics of the game model for each game.

There exist numerous techniques, besides HFSMs, to represent behaviors, like decision trees, rule based systems, GOAP or Hierarchical Task Networks, for instance. One of the opened investigation lines is the study of the pros and cons of each one of them and the possibility of combining some of them to create the behaviors (for instance, a HFSM in which the nodes were specified by rule systems, Hierarchical Task Networks or some other technique).

Within more complex and human-like current techniques that are used for controlling game AIs (such as big C functions or finite-state machines) will not scale up. But, just as computer game graphics and physics have moved to more and more realistic modeling of the physical world, we expect that game developers will be forced into more and more realistic modeling of human characters.

## References

1. M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick. Machine learning and games. *Machine Learning*, 63(3):211–215, Junio 2006.

2. J. Brownlee. Finite state machines (fsm). Available from http://ai-depot.com/FiniteStateMachines/FSM.html (accessed March 14, 2008).
3. H. Bunke and B. T. Messmer. Similarity measures for structured representations. In *EWCBR '93: Selected papers from the First European Workshop on Topics in Case-Based Reasoning*, pages 106–118, London, UK, 1994. Springer-Verlag.
4. A. J. Champandard. *AI Game Development - Synthetic Creatures with Learning and Reactive Behaviors*. New Riders Games, 2003.
5. P. A. Champin and C. Solnon. Measuring the similarity of labeled graphs. In K. D. Ashley and D. G. Bridge, editors, *5th Int. Conf. On Case-Based Reasoning (ICCBR 2003)*, LNAI, pages 80–95. Springer, June 2003.
6. G. Flórez Puga and B. Díaz-Agudo. Semiautomatic edition of behaviours in videogames. In *Proceedings of AI2007, 12th UK Workshop on Case-Based Reasoning*, dec 2007.
7. D. Fu and R. Houlette. Putting ai in entertainment: An ai authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.
8. A. Girault, B. Lee, and E. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design*, 18(6):742–760, Junio 1999. Research report UCB/ERL M97/57.
9. P. P. Gómez-Martín, M. A. Gómez-Martín, and P. A. González-Calero. *Javy: Virtual Environment for Case-Based Teaching of Java Virtual Machine*, volume 2773 of *Lecture Notes in Artificial Intelligence, subseries of LNCS*, pages 906–913. Springer Berlin / Heidelberg, 2003.
10. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2007. To appear.
11. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
12. Y. Wang and N. Ishii. A method of similarity metrics for structured representations. *Expert Systems with Applications*, 12(1):89–100, 1997.