Dynamic Expansion of Behaviour Trees *

Gonzalo Flórez-Puga, Marco Gómez-Martín, Belén Díaz-Agudo and **Pedro A. González-Calero**
{gflorez,marcoa}@fdi.ucm.es and {belend,pedro}@sip.ucm.es

Abstract

Artificial intelligence in games is typically used for creating player's opponents. Manual edition of intelligent behaviors for Non-Player Characters (NPCs) of games is a cumbersome task that needs experienced designers. Our research aims to assist designers in this task. Behaviours typically use recurring patterns, so that experience and reuse are crucial aspects for behavior design. The use of hierarchical state machines allows working on different abstraction levels, sharing transitions and reusing pieces from the more detailed levels. However, the static nature of the design process does not release the designer from the burden to completely specify each behaviour. Our approach applies Case-Based Reasoning (CBR) techniques to retrieve and reuse stored behaviors represented as hierarchical state machines (actually, behaviour trees). In this paper we focus on dynamic retrieval of behaviours taking into account the world state and the underlying goals to select the most appropriate state machine to guide the NPC behaviour. The global behaviour of the NPC is dynamically built in run time querying the CBR system. We exemplify our approach through a serious game, developed by our research group, with gameplay elements from First-Person Shooter (FPS) games.

Motivation

Finite State Machines (FSMs) are widely used as the main technology when creating the artificial intelligence of non-players characters (NPCs), because of their efficiency, simplicity and expressivity. Different implementations of FSMs have been published (Rabin 2002b; 2002a; Tozour 2004; Rosado 2004; Yiskis 2004) though almost every game includes a custom implementation in its own specific language, such us C++ or LUA (Ierusalimschy, de Figueiredo, and Celes 2006). FSMs have also influenced the definition of script languages used in games and virtual environments such as UnrealScript (used in Unreal Tournament) and Linden Scripting Language (used in Second Life).

Nowadays, however, gamers demand more believable characters, and FSMs have some drawbacks that force designers and programmers to subvert it. For example, pure FSMs are not Turing complete, so developers must add extra functionality that is not easily understood; additionally, they do not allow reusability of logic in different contexts and therefore designers end up duplicating states (redundancy) or adding a great number of complex transitions.

The evolution of FSMs are Hierarchical Finite State Machines (HFSMs) (Harel 1987). HFSMs allow designers to group together a set of states (the so called *super-states*) that have common transitions. Designers may create transitions to the super-states instead of applying it to each state, reducing redundancy. They are also able to group super-states, creating a hierarchy. HFSMs, however, do not allow designers to reuse states for different situations because transitions are hard-coded in them. The solution is the use of hybrid HFSMs or the more general Behavior Trees (BT). This technique moves transitions to external states so the states become self-contained. Every state therefore encapsulates some piece of logic transparently and independently of the context. With this new schema, states have no transitions, and may be seen just as behaviours.

Even though HFSMs and behavior trees make easier the creation of NPCs, it still takes a lot of time to wire it up because of the number of states or behaviours (Halo 2 had an average of 60 different behaviours arranged in 4 layers (Isla 2005)).

Nowadays there is yet another drawback. As the development of a game takes more and more time, the number of base behaviors available to designers greatly changes during the project life time. The static design process may result, for example, in a certain race that is not using certain behaviours just because they were created after the development of such race. Avoiding this problem requires a great amount of communication between designers and continuous revision of previously developed complex behaviours.

In this paper we propose Dynamic Behaviour Trees (DBT) as an extension to behaviour trees where some nodes in the tree store queries instead of actual behaviours. A DBT is expanded in run-time by substituting query nodes with actual behaviours through a similarity-based retrieval process. The main point of this approach is to let the designer of a high level behaviour to specify the properties of the desired behaviour for a given state, without committing to a particular behaviour implementation. This way, as new basic behaviours appear, they can be automatically selected in those situations where they best fit to designers' specifications.

*Supported by the Spanish Ministry of Science and Education (TIN2006-15202-C03-03 and TIN2006-15140-C03-02)

Formalization

In order to clarify the details of our approach in this Section we will provide a formalization of the elements that we consider in the representation of Dynamic Behaviour Trees.

In a very general settlement we propose to have:

- A set E of entities in the world. Entities have a type and can refer to NPCs, the player, or passive entities.
- A set T of entity types organized in a taxonomy through the $is-a$ relation.
- A set B of behaviours that entities can execute. Behaviours are organized in a taxonomy through the $is-a$ relation.
- A set A of attributes that can be used to describe entities. Attribute values can be numbers or symbols.

Each individual behaviour can be implemented through primitive actions or through behaviour trees. We denote as P the set of available primitive actions and as BT the set of behaviour trees in a particular game.

Every behaviour implementation is described through a behaviour $b \in B$, a set of variables, and a set of variable constraints. Variables are used to parameterize the behaviour implementation, allowing to express, for example, that a primitive action implements the behaviour $attack$ to a given entity $?target$. Every variable in a behaviour implementation have to be bound to entities in E for that implementation to be runnable. Every behaviour implementation is parameterized with at least the variable $?this$ which represents the NPC who is executing the behaviour.

Variable constraints describe the applicability conditions for a given behaviour implementation in terms of the attributes of the entities that should be used to bind its variables. Variable constraints may represent, for example, that a given attack behaviour is best suited for $?target$ of type $slow-enemy$ of for a $?target$ whose $aggressive$ attribute is 0.7. Additionally, applicability conditions for behaviour implementations can refer to preferred values for global variables, such as the current difficulty level of the game or the perceived boredom of the player.

A state s in a DBT is described through the set of behaviours B_s that the entity in that state is concurrently executing. Behaviours in a state can be references to actual behaviour implementations (primitive actions in P or behaviour trees in BT) or behaviour queries that when run will retrieve the most similar behaviour implementation according to the designer specification and the current situation in the game.

A query is essentially a partial description of a behaviour implementation that may include the desired behaviour $b \in B$, along with a number of variables and variable constraints using the same vocabulary used for describing applicability conditions of behaviour implementations.

Knowledge Modeling

According to previous section, our approach requires a rich knowledge model about the game. Mainly, about behaviours, entities and attributes to describe entity properties.

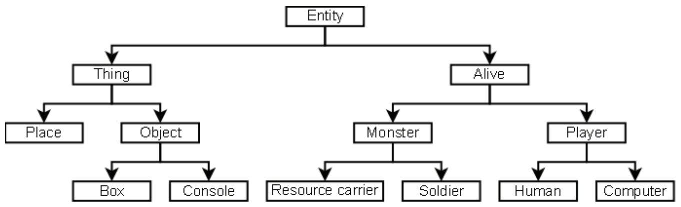


Figure 1: Entity Types Taxonomy

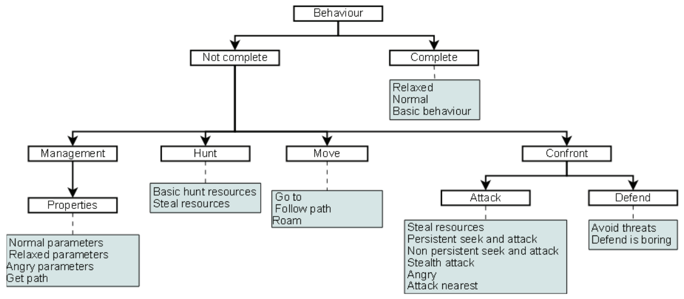


Figure 2: Behaviour Taxonomy

The use of this knowledge rich model is a strength for our approach, as it allows describing, retrieving and reusing behaviours. However, it could be also considered a weak point regarding the knowledge acquisition and representation effort. We strongly support this knowledge intensive approach because this model is reusable through different games of the same genre, such as the First-Person Shooter (FPS) genre that we study in this paper.

We have started by modeling the game terminology from scratch, although being influenced from existing sources. The Game Ontology Project (Zagal et al. 2005) is a framework for describing and analyzing games of all the genres. They have proposed a formalization abstracted from the analysis of many specific games. The project aims to develop a game ontology that identifies the important structural elements of games and the relationships between them, organizing them hierarchically.

We organize the knowledge within two main taxonomies: Entity Types (see an excerpt in Figure 1) and Behaviours (see an excerpt in Figure 2).

Behaviour taxonomy classifies and allows the annotation of the individual behaviour implementations that can be retrieved and reused. We use a basic classification of FPS oriented behaviours to manage resources, to confront other entities, either attacking or defending, to move or transport other entities, and to hunt or chase another entities.

As we have already described, entity types are used to characterize world entities, either NPCs, the player or passive entities. The type of a certain entity constraints the behaviour selection, as they characterize the suitable entities to bind its variables. The entities that participate in a certain behaviour are also constrained by attributes. The domain model also includes properties or attributes to characterize entities and the world. Basically, we have been influenced here by NPC basic abilities of role-play games such as NeverWinter Nights. These are the basic components of characters that alter how you interact with the world.

Although this vocabulary is extensible, we cite here example of properties for characters like: strength, constitution, intelligence, wisdom, aggressiveness, defensiveness, alarm or intuition. Difficulty is an example of a property used to describe the world itself. Each property or ability scores of a set of numeric or symbolic values (like little, medium, high). There are properties applicable to all the entities and there are properties specific for each type of entity in Figure 1. For example, aggressiveness level is a feature only applicable to Alive Entities.

Behaviour Retrieval

In this Section we describe the proposed method for behaviour retrieval using CBR techniques. The description is presented through examples taken from behaviours in JV^{2}M[1], a serious game we have developed to teach the inner workings of the Java Virtual Machine, using gameplay elements from First-Person Shooters (Gómez-Martín et al. 2007).

The context of the example query is presented in Figure 3. It corresponds to the behaviour "Steal resources (entity)". The objective of this behaviour is to obtain resources by stealing them from another entity, specified by an input parameter, *entity*. The restrictions over the value of this parameter are that it can only be of type *PLAYER*, that is, the NPC can only steal the resources from a player entity. The adjoint table shows the attributes that describe the state of the world in two different instants, which will give way to two different queries. Each attribute belongs to a variable, whose scope can be global, like $?world$ or $?player$, or local, like $?this$ (the entity that runs the behaviour) or $entity$ (the input parameter, not referenced in the table).

When the NPC begins executing the state machine, he needs to expand the state "Approach and attack (entity)" into another state machine or a primitive behaviour. The expansion implies executing the query associated and retrieving a behaviour from the case base. The behaviour retrieved by the query is then executed using the received parameter, *entity*.

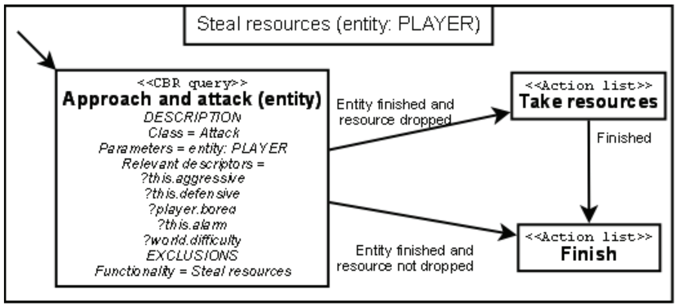
To guide the retrieval process, the designer of the behaviour gives a description of the query using four elements:

1. Functionality: this is the class of the behaviour to be retrieved in the Behaviours taxonomy. In this case, the value assigned to this parameter is "Attack". Figure 2 shows the Behaviour taxonomy with all the classes and several examples of instances of behaviours associated to them.

2. Parameters: the designer can add parameters to the query that will be passed to the retrieved behaviour. The parameters can be the ones received by the current behaviour or any of the global variables.

3. Relevant descriptors: here, the designer can select which variables and attributes of the Game State are relevant for the query and will be used in it.

[1]`http://gaia.fdi.ucm.es/projects/javy/`



GAME STATE		
Variables	**Value (t_0)**	**Value (t_1)**
`?this`		
aggressive	0.3	0.7
defensive	0.1	0.3
alarm	0.2	0.5
`?player`		
bored	0.6	0.3
`?world`		
difficulty	0.5	0.5

Figure 3: Query context

4. Exclusions: in this section the designer can specify behaviours he wants to exclude from the candidates before querying. The behaviours described here will never belong to the set of results of the query.

From this description, the system builds the query using the instance values for the parameters and the values of the variables of the relevant descriptors at the time the query is built. This query is used to retrieve the most appropriate behaviour from the case base.

Given the Game State and the query elements from Figure 3 we obtain two different queries, that can return different behaviours, based on the changes of the state:

	Query(t_0)	**Query(t_1)**
Class	Attack	
Parameters	{(entity, PLAYER_INSTANCE)}	
?this.aggresive	0.3	0.7
?this.defensive	0.1	0.3
?this.alarm	0.2	0.5
?player.bored	0.6	0.3
?world.difficulty	0.5	0.5
Exclusions	Steal resources	

For this example we will use a small sample of the case base. Table 1 gathers the cases and descriptions we will be using. The descriptors used for the cases are very similar to the ones used to describe the query:

1. Case number: is used to identify the case in the case base.

2. Parameters: is the set of parameters received by the behaviour, along with the restrictions of type of each one of them. The type is obtained from the taxonomy in Figure 1. The case can only be retrieved if the type of the

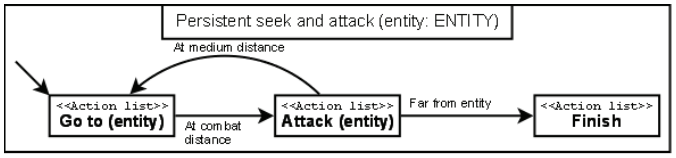

Persistent seek and attack (entity: ENTITY)
At medium distance
<<Action list>> Go to (entity)
At combat distance
<<Action list>> Attack (entity)
Far from entity
<<Action list>> Finish

Figure 4: Persistent seek and attack

parameter in the query is compatible with the type of the parameter in the case description.

3. Classes: indicates to which classes of behaviours belong the case on the behaviour taxonomy (Figure 2).
4. Descriptors: is a set of restrictions declared over the local and global variables. The values of the descriptors can be symbolic or numeric.

The retrieval process consists in obtaining the most appropriate behaviour from the case base, based on the query data. To achieve this goal we compare the query with the descriptions of the behaviours in the case base using a similarity function. The similarity function we are using is:

$$\text{sim}\,(Q,C) = \begin{cases} \bullet\ \ C \in Q.\text{exclusions} \Rightarrow 0 \\ \bullet\ \ (p,v) \in Q.\text{parameters} \wedge \\ \quad (p,D) \in C.\text{parameters} \wedge \\ \quad v \notin D \Rightarrow 0 \\ \bullet\ \ \text{otherwise} \Rightarrow w \cdot \text{sim}_{atr}\,(Q,C) + \\ \quad (1-w) \cdot \text{sim}_{fun}\,(Q,C) \end{cases}$$

$$\text{sim}_{atr}\,(Q,C) = \sum_{a \in A(Q,C)} [w_a \cdot \text{sim}_{loc}\,(Q.\text{descriptors}(a), C.\text{descriptors}(a))]$$

$$A(Q,C) = Q.\text{descriptors} \cap C.\text{descriptors}$$

$$\text{sim}_{loc}(Q_a, C_a) = 1 - \frac{|Q_a.\text{value} - C_a.\text{value}|}{\text{size}_a}$$

$$\text{sim}_{fun}\,(Q,C) = \begin{cases} Q.\text{class} \in \text{super}\,(C.\text{classes}) \Rightarrow 1 \\ \text{otherwise} \Rightarrow 0 \end{cases}$$

where $A(Q,C)$ is the intersection of the sets of descriptors of Q and C, size_a is the size of the interval of valid values for an attribute a, and super (s) is the set of classes that subsume the set of classes s. The weight w is a value in the interval $[0,1]$. Similarly, each w_a is a value in the interval $[0,1]$, and $\sum_{a \in A} w_a = 1$.

For this example we will use the weights $w = 0.5$ and $w_a = 0.2\ \forall a \in A$. The results of the similarity measure are shown in Table 2. Results show how changes in the Game State can lead to changes in the most suitable behaviour.

The execution goes as follows: it begins in the state "Approach and attack (entity)" from behaviour "Steal resources (entity)" (Figure 3). This state must be expanded, so we use the query(t_0). As we can see in Table 2, the result of this query is the behaviour "Persistent seek and attack (entity)" (Figure 4), as it shows the greatest similarity to the query. The execution continues with this behaviour in the state "Go to (entity)". Upon reaching this point, suppose that there is a substantial change in the Game State, in the instant t_1, while still executing the behaviour "Persistent seek and attack (entity)". To adapt the behaviour to the new conditions we query the case base again with the query(t_1) in the state "Approach and attack (entity)" with the new values of descriptors. We obtain a new behaviour, "Attack nearest (class)" (Figure 5), that is more suitable to the new Game State. We use this newly recovered behaviour to expand the CBR state. As we can see, the recovered behaviour has another expandable state, that will be expanded, in its moment, with another behaviour.

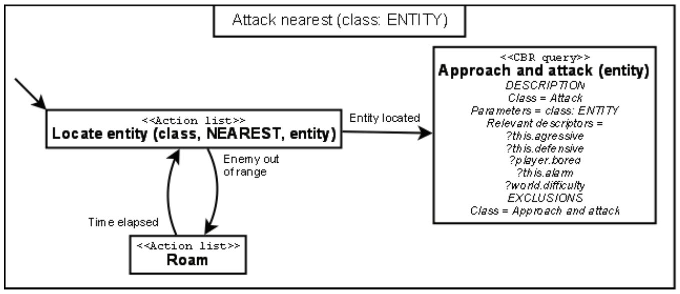


Figure 5: Attack nearest

An especially important issue related with the use of CBR is identifying the moment when there has been enough changes in the Game State so it is necessary to launch again the query. Doing it too frequently can cause unstable behaviours and can degrade the overall performance of the game, since the query processing can be very time consuming. Not enough frequency may cause that the behaviour being executed is not the most appropriate for the current situation.

Execution

Our behaviours are placed in the context of a running application that works in a loop which continuously evaluates the changes of the virtual environment and renders the next frame. At a given moment within the loop, this executes the routines that define the behaviour of the NPCs. In a multi-thread scenario, there are different approaches (Garcés 2006), but in their minimal expressions all of them end up calling a routine that implements the decision of the NPC.

Every NPC has a dynamic behaviour tree that represents its state. However the entire tree is not allocated in memory but just the small numbers of behaviors actually running (the current leaf, its parent and so on up to the root of the tree). The current leaf always contains the primitive behaviour the NPC is executing, while the inner nodes (including the root) represents complex and high-level behaviours such as those showed in Figure 3. There are two different kinds of complex behaviours in the inner nodes:

- Behaviour implementations explicitly named by the designer. Those implementations may refer to primitive actions or finite state machines (FSM) with a current state and a set of *transitions* with associated conditions that, when satisfied, fired them.
- A behaviour implementation acquired through a query described in the current state of the parent node.

```
proc IA_Tick(Event e)
   n = root_node;
   while n != empty_node do
      if n is a FSM then
         if check_transition(n, e) then
            tear_down(n);
            spawn(next_state(n, e);
         endif
      else if n is a query then
         q = build_query_description(n)
         other_behaviour = query(q);
         if other_behaviour != n then
            tear_down(n);
            spawn(other_behaviour);
         end
      endif
      n = child_node(n);
   end while
end proc
```

Figure 6: Pseudocode of the IA routine

Game-loop is instructed to call the AI routine in two different moments: when the current primitive behaviour is finished or when some important event occurs (such us a collision with another entity). In order to simplify our explanation we will consider the end of the primitive behaviour as an event.

The AI routine must visit every node in the behavior tree to check if the event fires a change in its behaviour child. Depending on the kind of node, that will consist on two different actions:

- When the behaviour is encoded in a FSM, the event is checked against all the conditions that define the transitions of the current state. If one of them is fired, the routine must tear the child node down (and recursively all its descendants), and set the new state.
- When the behaviour is a query, the context that originated the retrieval of that behaviour may have changed due to that event. To determine whether that is the case, the query must be performed again and the behaviour retrieved is compared with the current behaviour. When different, the current behaviour must be torn down in the same way described before, and the retrieved behaviour becomes the current behaviour.

Most of the time, events reported by the game engine will not fire any behaviour change and the NPC will continue with its primitive behaviour. When the main loop informs the routine about the end of it, the leaf node will be destroyed and the parent node will have a transition executed. The general process appears in Figure 6.

When a new behaviour is activated (because of a transition or a change in the result of the query) it has to be executed. Depending on the kind of behaviour the process differs:

- When the behaviour consists of a primitive action, it just invokes the game engine to make it executed. As it has been previously described, the engine will report the end of the execution of the action.
- When the behaviour is a FSM, it starts recursively executing its initial state.
- When the behaviour is a query, it builds the description according to the description and the global state of the NPC. Using that description, it retrieves the new behaviour and recursively spawn it.

```
proc spawn(Behaviour b)

   if is_primitive_behaviour(b) then
      game_engine_execute(b)
   else if is_FSM(b) then
      create_child(initial_state(b))
      spawn(b)
   else if is_query(b) then
      q = build_query_description(b)
      new_b = query(q);
      create_child(new_b);
      spawn(new_b);
   else
      error "Unknown behaviour!";
   endif
end proc
```

Figure 7: Pseudocode of the IA spawn routine

The general process may be seen in Figure 7.

Related Work and Conclusions

Alternatives to FSM have been proposed in the last years seeking to solve the scalability problem that bigger developments with higher requirements pose for game. Goal-oriented formalisms provide a declarative way for designers to express NPC behaviour, allowing for an underlying search algorithm to explore a potentially huge solution space. Hierarchical planning have been applied both in academic (Hoang, Lee-Urban, and Muñoz-Avila 2005) and industrial systems (Gorniak and Davis 2007), and it seems to point towards the next generation of more believable NPCs. Nevertheless, it still has to be found the way to educate game designers to be able to write domain descriptions and plan preconditions and postconditions. Furthermore, designers have to understand that a richer AI implies some degree of emergent behaviour, where not everything can be a hundred per cent predicted before hand.

We propose the use of dynamic behaviour trees as a middle point between fully specified behaviour trees and search-based goal oriented formalisms. Through DBT a designer can reuse low level behaviours without actually knowing in advance every possible implementation for a given functionality, but being able to specify the features of the desired behaviour, using a domain language. The ontology used to describe behaviour queries and implementations is the contract between high level and lower level behaviours, and it will evolve as the set of behaviours grows.

As a future work we want to improve the knowledge model. We are borrowing ontologies about real life, like Military strategic and tactic knowledge or military terms. We also plan to explore the use of Description Logics reasoning mechanisms to support the definition and evolution of the domain ontology used to describe behaviours.

References

Garcés, S. 2006. Strategies for multiprocessor AI. In *AI Game Programming Wisdom III*. Charles River Media. 65–76.

Gómez-Martín, P. P.; Gómez-Martín, M. A.; González-Calero, P. A.; and Palmier-Campos, P. 2007. Using metaphors in game-based education. In chuen Hui et al., K., ed., *Technologies for E-Learning and Digital Entertainment. Second International Conference of E-Learning and Games (Edutainment'07)*, volume 4469, 477–488.

Gorniak, P., and Davis, I. 2007. Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters. In Schaeffer, J., and Mateas, M., eds., *AIIDE*, 14–19. The AAAI Press.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3):231–274.

Hoang, H.; Lee-Urban, S.; and Muñoz-Avila, H. 2005. Hierarchical plan representations for encoding strategic game ai. In Young, R. M., and Laird, J. E., eds., *AIIDE*, 63–68. AAAI Press.

Ierusalimschy, R.; de Figueiredo, L. H.; and Celes, W. 2006. *Lua 5.1 Reference Manual*. Lua.org.

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

Rabin, S. 2002a. Enhancing a state machine language through messaging. In *AI Game Programming Wisdom*. Charles River Media. 321–330.

Rabin, S. 2002b. Implementing a state machine language. In *AI Game Programming Wisdom*. Charles River Media. 314–320.

Rosado, G. 2004. Implementing a data-driven finite-state machine. In *AI Game Programming Wisdom 2*. Charles River Media. 307–318.

Tozour, P. 2004. Stack-based finite-state machines. In *AI Game Programming Wisdom 2*. Charles River Media. 303–306.

Yiskis, E. 2004. Finite-state machine scripting language for designers. In *AI Game Programming Wisdom 2*. Charles River Media. 319–326.

Zagal, J.; Mateas, M.; Fernandez-Vara, C.; Hochhalter, B.; and Lichti, N. 2005. Towards an ontological language for game analysis. In *Proceedings of the Digital Interactive Games Research Association Conference (DiGRA 2005), Vancouver B.C., June, 2005. Included in the Selected Papers volume.*

Case	Parameters	Classes	Descriptors
1	<<**HFSM**>> **Steal resources**		
	{(entity, PLAYER)}	Attack Hunt	?this.aggressive = 0.8 ?this.defensive = 0.3 ?this.alarm = 0.4 ?player.bored = 0.7 ?world.difficulty = 0.6
2	<<**HFSM**>> **Non-persistent seek and attack**		
	{(entity, ENTITY)}	Attack	?this.aggressive = 0.3 ?this.defensive = 0.1 ?this.alarm = 0.7 ?player.bored = 0.4 ?world.difficulty = 0.2
3	<<**HFSM**>> **Persistent seek and attack**		
	{(entity, ENTITY)}	Attack	?this.aggresive = 0.5 ?this.defensive = 0.1 ?this.alarm = 0.7 ?player.bored = 0.6 ?world.difficulty = 0.5
4	<<**HFSM**>> **Stealth attack**		
	{(entity, ENTITY)}	Attack	?this.aggresive = 0.5 ?this.defensive = 0.3 ?this.alarm = 0.7 ?player.bored = 0.5 ?world.difficulty = 0.6
5	<<**HFSM**>> **Basic hunt resources**		
	{(entity, ENTITY)}	Hunt	?this.aggresive = 0.4 ?this.defensive = 0.0 ?this.alarm = 0.3 ?player.bored = 0.5 ?world.difficulty = 0.4
6	<<**HFSM**>> **Angry**		
	{(entity, ENTITY)}	Attack	?this.aggresive = 0.8 ?this.defensive = 0.0 ?this.alarm = 0.7 ?player.bored = 0.6 ?world.difficulty = 0.7
7	<<**HFSM**>> **Attack nearest**		
	{(entity, ENTITY)}	Attack	?this.aggresive = 0.5 ?this.defensive = 0.4 ?this.alarm = 0.5 ?player.bored = 0.4 ?world.difficulty = 0.3

Table 1: Case base

Query	**Similarity**						
	C_1	C_2	C_3	C_4	C_5	C_6	C_7
t_0	0.00	0.90	**0.93**	0.89	0.45	0.87	0.88
t_1	0.00	0.88	0.91	0.93	0.39	0.89	**0.94**

Table 2: Query results