Project Report: Automatic Behavior Customization & Variation of Non-Player Characters

Kyle Li

School of Computer Science, McGill University, Montreal Qc H3A 2T5, Canada
yifan.li@mail.mcgill.ca

Abstract. Non-Player Chracters(NPCs) are becoming more important a part in the nowadays video games, especially in Role-Playing Games(RPGs), in which game designers are trying to make the bahavior of the NPCs believable and interesting. There are a number of ways to cutomize NPC behvaior. Hector Gomez-Gaucha and Federico Peinado[1] proposed a model for customization depending on the players temperament. Ontologies[2] and Case-Based Reasoning[3] are used as the reasoning model for adaption. Once we have various behaviors for each single NPC, we can go to the next level and consider how they behave when they are together. There are a lot of things to consider when modeling group behavior, and it is a much harder problem than the singlular case. The solution is to have relations and variations among the group. In my project I am going to automate this process by model transformation. The modeling tool I used is statechart in Atom3[4]

1 Introduction

One of the major goals of video games is evoking player's emotion through various interactions in the game. Current game design is much focused on making the interaction challenging and interesting to keep the player wanting to explore more. However, players have different temperaments, thus will be affected by interactions in a different way. It is naive to assume that one particular NPC behavior is interesting to everyone.

In the paper by Hector Gomez-Gaucha and Federico Peinado, they introduce Keirsey's theory[5] as a simplified model to describe human temperament. Keirsey's theory says that a person's temperament is a combination of four basic temperaments - Artisan, Guardian, Idealist, and Rational. This information for a particular player is retrieved before the game session. Once the player's information is known, they use Ontologies and Cased Based Reasoning(CBR) to reason about that information, and calculate the most appropriate customization.

The customization of every single NPC is not enough. In a racing game, you do not want all the opponents to have exactly the same strategy. If you are playing a RPG, it is going to be boring if all the NPCs behave exactly the same way. In those situations, a group of NPCs is somewhat related or having

a common goal, but making them behave exactly the same does not look very realistic. Variations must be done within a group to generate each group member a little differently from others, thus a more lively group.

2 Current Approaches for Game Customization

Most RPGs nowadays employ a conversation tree and the interaction are state-controled. Some games offer NPC customization but require manual configuration. Blade Runner is an RPG that is replayable by employing different NPC behavior every time the game is played, even though the auto-customization is not based on the particular player. Whereas some action game such as Max Payne, can adjust the NPC skills automatically based on the player's skill level. Fable, another RPG, rates the player's action as good or bad, and change NPC behavior based on player's reputation. This is truly dynamic, but rather simple to judge everything as either good or bad.

It is becoming common that intelligent components of games are developed as a piece of software by itself, and get linked to game engines through middleware. Some of these intelligent systems include: Zocalo, I-Storytelling, and KIIDS which uses CBR and Ontologies.

3 The example Scenario

I will use a fantasy RPG Never Winter Nights(NWN) to show the customization process. The player's avatar in the game is called Drax. The scenario we are going to customize is that after Drax defeat evil monsters and returned to his castle, his servant is trying to welcome and comfort the master.

Fig. 1. Servant: Welcome home, sir. [Repeating fast worship movements] Drax: Hum, I feel so tired... Please, servant, prepare the bath. Servant: Immediately, sir. [Running to the bathroom] Example dialogue at the 13th level of politeness

in figure 1, there are a number of things that could be customized. By default, the servant will fall to the ground and the default text will show. The servant will then start fast worshiping animation. This could be a funny animation to some players. But might not be appropriate to players with a serious temperament. It could be even offensive to players from a certain culture. There is no single NPC behavior that is good for every player, but rather various behavior for various player types. To do that, we need models to describe the player types.

4 CBR and Ontologies

Ontology is a term from philosophy that describes entities in a particular domain. One can specify individuals, classes, properties, roles as Ontology entities. I will use Ontologies to describe all the facts in our domain such as player tempraments, customization actions, and possible adaptions.

CBR defines a case base. Each case is a problem-solution pair. The problem describes what temprament the player has. The solutioin is the command to change NPC behavior so it looks more interesting to that player. This could be done either in game native commands or some scripts developed for the game. CBR represents the knowledge that we have about customization.

I will integrate the knowledge and facts together by defining both problems and solutions as entities of Ontologiy and build knowledge intensive CBR(KI-CBR). After we build the case base, the whole customization process follows like this:

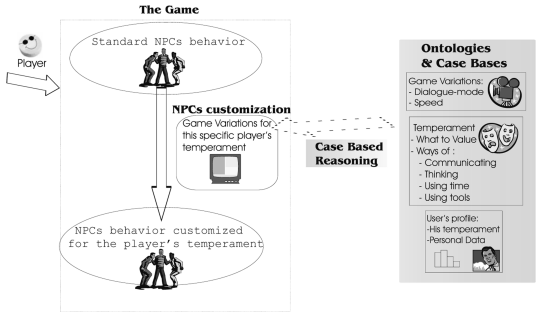

Fig. 2. Elements of the automatic NPC customization model

The model starts with a player playing the game which has a standard NPC behavior. The CBR cycle will retrieve a case that is similar to that of the player. The customization is performed to make NPC more similar to the player. Then if the distance between the retrieved case and the player is more than the threshold, the case will be adapted to make the NPC behavior more suitable. The Ontologies of different entities are discribed next.

5 Temperament Ontology

Based on David Keirsey's theory, which is very widely applied in psychology, each person has a unique proportional combination of the four basic temperaments. In this report I will use a unique combination as the new case: Artisan 10%, Guardian 10%, Idealist 30%, and Rational 50%. Usually one of the temperaments is predominant, which means the person will behave like that type most of the time. Rational is predominant in our case.

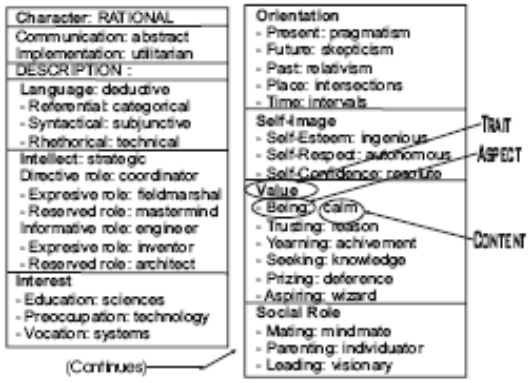

Fig. 3. The basic temperament Rational and its traits

Above is the description of Rational basic type. The description has several traits. Each trait has several aspect. Each aspect has a value. Each trait defines how people behave in each aspects of that trait. For example, the Value trait has an aspect Being, which has value calm. That means Rational people value most or like to be calm.

6 Variation Ontology

Variations are all the possible changes for customizing NPC behavior. For example if we want to implement one assertion: "a polite NPC with slow movements makes you feel calm", then there are two aspects in this customization: an NPC's politeness and speed of movements.

The customization is carried out either by game native commands such as changing the value of a varialbe which decides the level of politeness of NPCs, or by performing scripts which are developed for the game. Both of these are described in hasActivationCommand and hasActivationParameters traits.

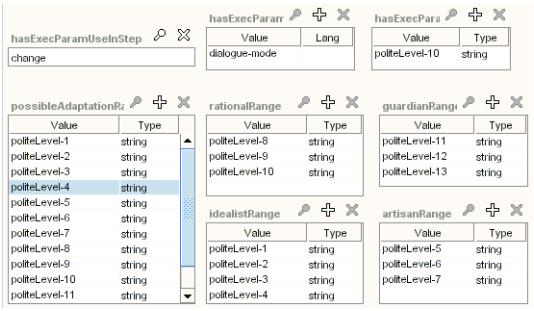

Fig. 4. ExecParam of a variation concept and ranges to be adapted

7 Mapping Temperaments into Variations

Each variation may affect some traits of a specific temprament. This is captured by relating affectToTraits trait of Variation to a particular trait in a Tempra-ment. In the figure below, affectTotraits has value ValueBeingCalmX, which means it is related to the Value trait of Rational temperament. Now we can prmote the calmness by performing the changes defined in the Variation.

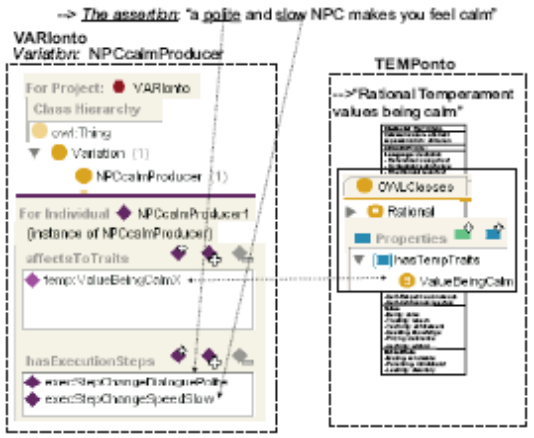

Fig. 5. An assertion as a variation related to a specific trait of a temperament

Also notice that the same variation affects different temperaments in a to-tally different way. For example, the same dialogue change affects Value trait of Rational temperament in a different way to the same trait of Artisan tem-perament. This is becase Rational people appreciate to be calm while Artisan people appreciate to be excited. This is tackled by having different parameters for different temperaments.

8 Player Ontology

This is where the player profile is stored. Player Ontology has 3 traits. hasTemperamentProportions defines what combination a player has. hasUserType defines the best match from the case base. hasUserAdaptions defines the adaption needed to adapt the user type to a specific player.

9 CBR Process

First we get the description of the player by Keisey's questionnaire before the game session. Suppose we get a player description like this: Artisan 10%, Guardian 10%, Idealist 30% and Rational 50%. From the case base, we get the most similar case e.g. Artisan 10%, Guardian 30%, Idealist 30% and Rational 30%.

To measure the similarity between the two cases, the sum of the differences between each components is calculated:

$$\sum_{T=Artisan}^{Rational} (\%NewCase_T - \%RetrievedCase_T) * Correction_T$$

A CorrectionFactor is used here to make sure the sum is large as long as the predominant temperament is very different, even if the rest of the temperaments are similar.

The model reuses the retrieved case by adapting it. In our example, Atisan stays the same, Guardian must be increased by 20%, Idealist stays the same, Rational must be 20% less. The model uses the possibleAdaptionRange trait of Variation to perform the adaption. For example, to make the case 20% more Rational, we look up the list of values in the adaption range of Rational. This list of values goes from 0% to 100%. We use the new proportion of Rational as an index to get the corresponding element in the list. After the adaption, the modified variation is the new case solution and will be executed. 6 shows the same scenario after customization. The text is adjusted to be more polite and the animation of the servant is different. This is because the game knows the player is Rational predominant, so the game politeness has increased from 13th level to 8th level.

The last step is to get the feedback from the player after the game session to decide if the new case is a useful one, and store usefel ones in the case base for future reuse.

10 Variation Generation

Now we have the model to generate decent behavior for each individuals. Even though each of the NPCs could behave properly, how to generate their behaviors when they are acting as a group? Group behavior appears a lot in modern video games. In a First Person Shooting game, there are often groups of enemies. In

Fig. 6. Servant: Welcome to the castle, sir! Everybody has heard about your courage in the battlefield. Drax: I am tired because of the battle, servant. Prepare my bath. Servant: At your command, my lord. [Saluting him] A perfumed bath is ready for you. Example dialogue at the 8th level of politeness

a racing game, there are groups of opponent cars. In social simulation games, there are friends, families, and different social communities. These groups are either with the player or against the player. It is the uniqueness of different group members that make the game interesting and challending. Some times you will have neutral groups such as NPCs walking on the streets in RPGs, they are there to make the game look realistic and interact with the player in a interesting way. But if all of them behave similarly, then it is not going to be so realistic. In all of these situations, the group members are related in a certain way or share some common goals. But we should generate each members a little differently without affecting the group goals.

There are a number of ways to do this, but they are all centered around producing minor random variations among group members while keeping the main goal or common actions among the group. One thing we could do is to start with defining high level variations. For instance, if we want to get variations among walking pedestrains, we could first define a pedestrain's acting mode, such as normal, happy, angry, or polite. For each of the acting mode, we could define it as a combination of low level acting components. These lower level components must be implemented in the environment controller. For example, a normal pedestrain has only one walking component, a happy pedestrain has a walking component and a smiling component, an angry pedestrain has a walking component and shows a bad face, whereas a polite pedestrain has a walking component and yield to others whenever there is a conflicting situation.

To make things more interesting, one can combine those high level modes once we defined them. For instance a pedestrain could be happy and polite at the same time by having all the components in both modes. The result is that a smiling pedestrain that yield to others all the time. In this process we must

also enforce our rules such that for example a pedestrain can not be happy and angry at the same time.

11 Variation Generation Model

We could generate these variations by programming. But the drawback of programming approach is that it is not efficient and intuitive. We must implement every level from bottom to top, then evaluate our code and hope to observe a good result. If the result does not look good, we must identify where things have gone wrong, and go back to the first step to correct it. The evaluation could be time consuming because it involes compiling and testing for every cases.

Another approach is modeling. We could model every components by an appropriate model such as statechart. This could let us design our logic at a higher level and not concern about detail implementations. Modeling is also visual, by which it is easy to describe our behavior flow. I did not create a seperate meta model for this project because statechart is already very good at describing concurrent components by orthogonal components. So each basic component gets mapped to an orthogonal statechart component. These small statecharts become the elements that we can play with. All the combinings are then defined as model transformations. In the transformation rules, one can identify those basic statecharts and add one component to another according to predefined rules. To start the transformation, we need an initial statechart. A good choice is the statechart that describes the common goal of the group members e.g. walking for pedestrains, speed up for player's opponent cars. The advantage of modeling is that we could stop at any intermediate level and evaluate how good our transformation is. It also scales well by defining a 'producing' rule that keeps adding instances.

To illustrated the process, I implemented a bouncing ball environment using python. All the necessary graphical commands are defined as the interface between the environment and the statechart. The statechart holds a reference to the environment which is going to be called on every statechart action to perform varioius tasks. This environment is relatively easy and not representing the complexity of generating real NPC behaviors, but it is good enough to show you the idea. A ball could have very limited bahaviors, such as bouncing, changing colors, changing shapes, and getting blown away. These are defined as four basic components and implemented in my python environment. I used statechart in Atom3[4] as my modeling environment. The initial bouncing statechart is defined in 7.

There is a initial state. When it receives an event 'start', it moves to moving state, which keeps bouncing the ball every 0.01 second thereafter. The 'start' event also passes in a 'ctl' reference of the environment controller. Moving command is going to be function calls on this reference every 0.01 second.

The transition action for moving state is defined in 8. It calls ctl.bounce(b, 0.01) to update the ball's position. Please execute 'ControlSingle.py' to see the animation.

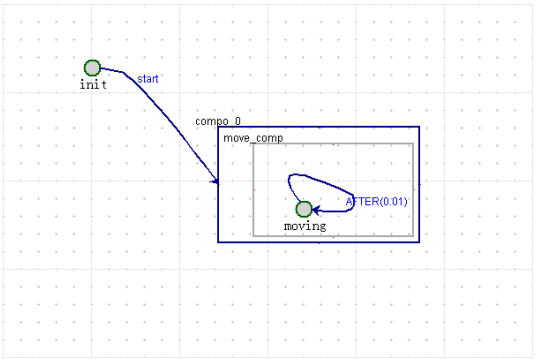

Fig. 7. bouncing ball statechart

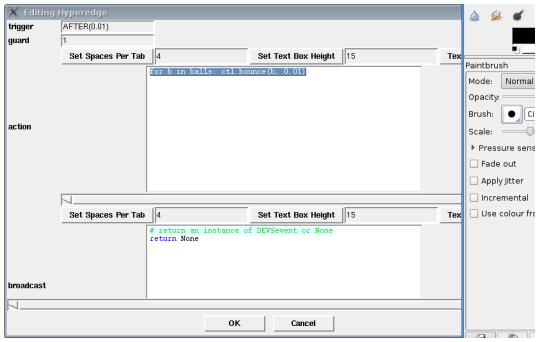

Fig. 8. moving action definition

X

To make a bouncing ball change colors, I am going to define a transformation rule in the Transformation Toolbar of Atom3. On the rule's left hand side, I am going to recognize the statechart I just defined and add another orthogonal component that changes the ball's color.

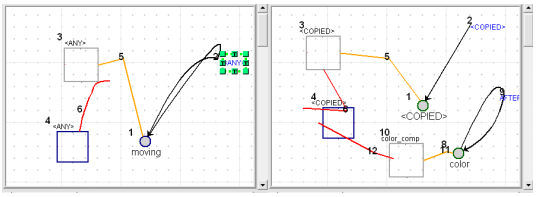

Fig. 9. transformation rule to add a color component

The left statechart of 9 is the original statechart. On the right hand side, we keep the original, and add an orthogonal color component. We must copy over all the actions, guards, and event triggers of the original statechart, and create the new component with one default state. The trasition action of the new component is defined to be:

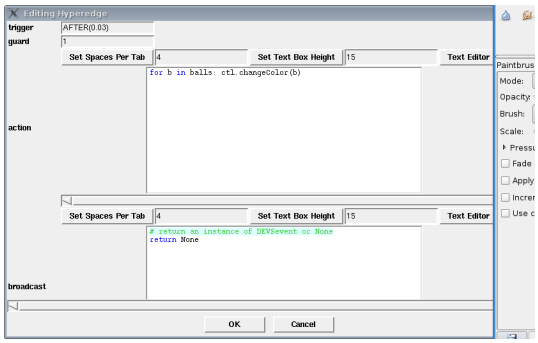

Fig. 10. action definition of the color component

This rule is then applied and we get a statechart in 11 that describes a color changing bouncing ball. Observe that the moving and the coloring are two orthogonal components of the same composite state:

I defined another rule to change the ball's shape. It is exactly the same as the color component rule except the different action for the new component. After it is applied, the statechart in 12 has three components.

Please execute 'ControlColorShape.py' to see the animation.

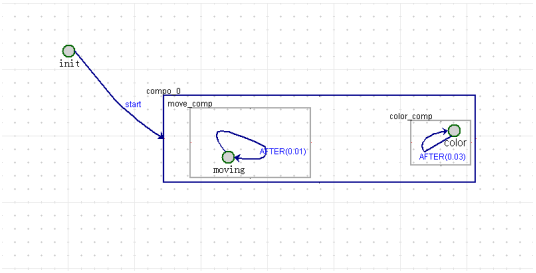

Fig. 11. color changing bouncing ball statechart

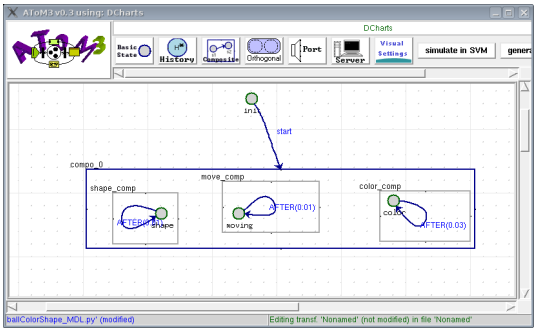

Fig. 12. color/shape changing bouncing ball statechart

I defined three more components that blow the ball to the right, left, and up. These are also done with similar graph grammar but different actions. After they are applied, I get a statechart with six components:

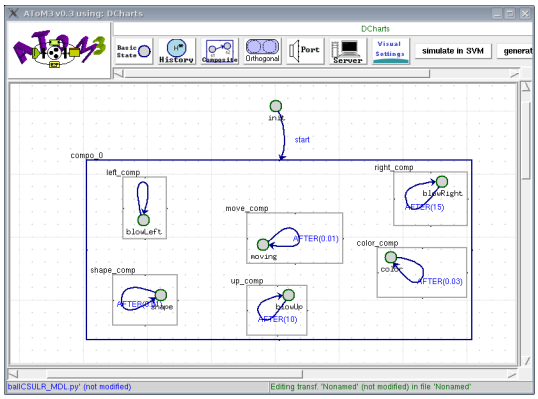

Fig. 13. full featured bouncing ball statechart

Now all the features have been added. The ball will periodically change its color and shape. Every once in a while it also gets blown away to the left, right or up. Please execute 'ControlCSULR.py' too see the animation.

The last component is to product group of balls. There are several ways to do productions. One is to define a transformation rule that makes a initial moving statechart like 7 from nothing. This statechart is going to be the basis for the second bouncing ball and will undergo a series of transformations to make another ball that is hopefully different from the current one. But for bouncing balls, it is enough to make one producing component like in statechart 14. Every time the component is producing a new ball in the environment, the initial parameters are different. The statechart keeps track of all the balls created so far, and the set of balls that have each particular component. A component is going to perform actions only on those balls available to it. Due to time constraints, I initialized every ball as full featured. But I used random parameters in each components when actions are performed on different balls. I tested the model on a group of 300 balls, and it looks fairly various and realistic. Please execute 'ControlAll.py' too see the animation.

12 Conclusions

I introduced a model that integrates CBR and Ontology to customize NPC behavior according to player's temperament. I used Never Winter Nights as an example to illustrate the process.

Then I discussed modeling group behavior of NPCs. To make group behavior look good, we want to have variations among group members. I used two levels

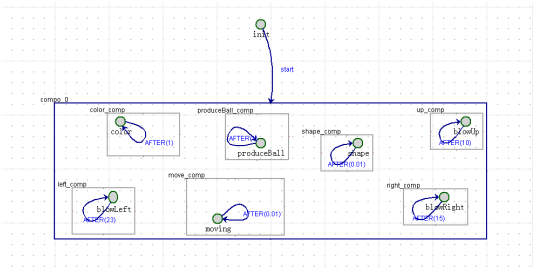

Fig. 14. the bouncing ball is changing color and shape, and every few seconds gets blown away in left, right, or up direction. The producing component creates a new ball every 0.1 second.

to represent variations. A high level defines the general mode that the NPC acts in. Each mode consists of different combinations of low level components. We can make complex behaviors by combining different modes. We also need to define combining rules here to make sure conflicting modes can not be combined e.g. a logical person can not be happy and angry at the same time.

I used statechart to generate various behaviors. I started by defining all components as simple statecharts. Then using model transformation, I combined those statecharts to different orthogonal components in a composite state. Of course the transformation could be done in a various way and must be rule based.

The future work would be adding more complex components to model more interesting behaviors. Also the transformation must be constrained so that we do not violate the rules. This is easily implemented as transformation guards or rules-based transformation system. For instance in rule-based transformation system we could define transformation trees so that after each step, only those that are not conflicting with the current components appear in the next step. The rules must be implemented in the graph grammar to generate more sensible groups. This probably requires a more complex environment than bouncing balls. Also random variations must be implemented. This means that in the graph grammar, there must be a way to randomly decide which component to add, so that we could produce a group of nature but random NPCs.

References

1. Gomez-Gaucha, H., Peinado, F.: Automatic Customization of Non-Player Characters Using Players Temperament. Depto. Ingeniera del Software e Inteligencia Artificial Universidad Complutense de Madrid, Spain
2. : http://ontology.buffalo.edu/smith/articles/ontologies.htm
3. Aamodt, A., Plaza, E.: Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. (1994)
4. : http://atom3.cs.mcgill.ca/
5. Keirsey, D.: Please Understand Me II: Temperament, Character, Intelligence, 1st Ed.,. Prometheus Nemesis Book Co.