New NPC design

Sami Mäkelä

August 31, 2002

1 Introduction

The first phase in Kokatrix NPC design was to design a library that can be used to implement NPC behavior. The next phase is to find a way for NPCs to reason about their behavior. Third phase could include something like learning and other more advanced stuff.

To implement reasoning, a design codenamed Linguistic Turn has been made. First step in Linguistic Turn is to define a language of events. This language enables symbolic reasoning because it describes the world in a syntactic way. Next we can define "semantics" for the language. The semantics is based on propositions and rules. Semantics can now be used to model the world. Different events will change the state of the model, based on rules. Also based on the rules, the NPCs can find out the results of their actions.

2 Propositions and rules

The atoms of knowledge are some values or pairs of atoms. The syntax for pairs is (a, b). Notation: $(a, b, c) = (a, (b, c)) = a(b, c) = [a][b]c$. The meaning of pairing is undefined: it's just an ordered association. A *state* is a multiset of atoms. 1 denotes empty state.

The product $a \otimes b$ means that there are two distinct substates, a and b. The rule $e : a \multimap b$ means that event e is associated with a state change that removes a from the state and adds b to state.

An *agent* has a state and a set of rules: $r_1; r_2; \ldots$. There are three kinds of rules: ones that describe the actions of the agent, ones that describe the actions of other agents and ones that describe the actions of environment, events. The actions of others are selected by other agents and the actions of environment are selected randomly.

The agent can also have *desires*, which tell how good a substate is. With these desires, the total desires of states can be calculated. The *gain* of an action or a sequence of actions is the difference of the end and the beginning state of the actions. The actions can have associated *efforts*, which tell how much effort does executing this action cause. If the gain of using a sequence of actions is smaller than effort, these actions won't be used. To accomodate the effects of

events and actions of others, the actions can have *probabilities*, which tell what is the probability of an action to succeed before an event disturbs it.

The state includes the knowledge and beliefs of an agent. There are no clasical reasoning steps about a static world: reasoning steps are actions that change the knowledge.

2.1 Syntactic sugar

$$[s](a;b) = [s]a;[s]b$$
$$[s](ev : a \multimap b) = evs : [s]a \multimap [s]b$$
$$[s](a \otimes b) = [s]a \otimes [s]b$$

$$\mathbf{section}\, s\{a;b\} = \mathbf{section}\, s\{a\};\mathbf{section}\, s\{b\}$$
$$\mathbf{section}\, s\{e : a \multimap b\} = e : s \otimes a \multimap s \otimes b$$

2.2 Models and contexts

A *model* is some information about the rules of a situation. To use a model for a specific situation, it can be instantiated to form a *context*. An NPC has several different contexts that have information. These contexts can be handled as one big knowledge base. Contexts are stateful objects. They have methods (actions) and spontaneous events. A model is a class for contexts.

Contexts can be represented using pairs. For example $[id1](building \otimes [id2](door \otimes open))$ means that context $id1$ includes state $building \otimes (id2, door) \otimes (id2, open)$.

2.3 Associated information

Beliefs can have associated information that tell how the belief should be handled. The priority and reliability of a belief can be encoded in this information. Low-priority states might become forgotten after some time. Rules have associated events or actions.

3 Goals and planning

A *goal* is a (sub)state that the NPC wants to accomplish. A goal includes the desired state and also a model and a way to instantiate the model into a context. The desired state is a state in this context. $b \vdash g$ means that the agent has beliefs b and tries to achive goal g. The goals are always in some context, just like atoms. Planning can also be done in the contexts without having explicit goals. The desires of states can be calculated, and best one is selected as goal.

To accomplish a goal the agent must construct a *plan*. A plan is a tree or graph, where there are preferred actions for every anticipated state. An action in the plan returns a value, like **success** or **failure**. Based on this value, the

rest of the plan is selected. If there is no plan for the value, more planning is needed.

The general method to accomplish a goal is:

1. Find/retrieve context.
2. Plan using the context.
3. Execute the plan.

It might be possible to do these phases lazily.

A plan in a context includes an event handler which handles the events that have rules in that context. Event handler modifies the knowledge base using the rules. The events can interrupt actions of the agent causing them to immediately return with a value that tells what happened.

Plan execution task needs to do some checking:

- When starting new subgoal, check that state is correct.
- Check if the goal is finished, skipping.
- Check if the plan is still valid: If the goal makes the transition it promised, does that help it to achieve state that was wanted?

The actions in the plan might also have to use planning. Eventually atomic goals are reached. They need no planning.

3.1 Events

Actions can be selected whenever the preconditions hold, but any event might happen before an action has succeeded. Events and actions have *probabilities*. For example if there are two actions two actions with probabilities 1 and 2, and an event with probability 2, then if the player selects the first action, it has success probability 1/3, and if the player selects the second action, it has success probability 1/2.

3.2 Examples

When entering building, the initial goal is $[building](unknown \multimap inside)$.

- Context: *building*.
- Goal: *inside*.
- Rule: $enter_door(X) : door(X) \otimes outside \multimap door(X) \otimes inside$.
- Rule to find more information: $check : unknown \multimap door(door_1) \oplus (door(door_1) \otimes door(door_2)) \oplus 1$.

There are also other rules related to the building.

In this context, $door_1$ is a *name*. It can refer to an actual object. Perhaps it doesn't. The reference is bound when doing the action *check*.

If there is an atom $[id](building \otimes door(door_1))$, a possible subgoal is $enter_door(door_1)$. This goal includes the context and the desired state $[id][door_1]inside$. When this state has been achieved, the subgoal returns and the transition associated with the action in the goal is applied.

3.3 Skill system

In a typical skill system, the characters or items have different kinds of attributes. Then skill success or damage is calculated based on these attributes. So the effect is a function of just the relevant attribute of the involved objects.

But there might be objects that are better with some special kind of objects. For this, one solution is multiple dispatch. In multiple dispatch, the effect is depends on the types of objects that are involved in addition to the relevant attributes. Actually the type can be seen as a special attribute. If a subtype of a supertype has different kind of behaviour from the supertype, and an object is of the subtype, then the effect of subtype should be selected. But because this is multiple dispatch, there might be a situation where $A :> A'$, $B :> B'$, and effects $a : A * B' \to C$ and $b : A' * B \to C$, when it's not clear which effect to select. So, to implement multiple dispatch, there needs to be a list of all cases. These cases are then tried in order. It is clear that this approach does not have the modularity of original approach. There could just as well be a big function that returns the value of the skill check by using arbitrary computations on the involved objects.

Our solution is to split the skills and skill checks and into small pieces. A complete skill can be seen as a model. The skill of an agent is then a context. It is probably most simple to specify skill contexts by just giving the possible plans.

In skill contexts, the actions of agents alternate. Each action sends event to other agent. The actions are $\oplus$-actions, so the randomness in skill checks is reduced to selection of an event from the list of outcomes. More complex randomness can be caused from executing several actions. Multiple dispatch like stuff can be made by having special events that get information about the other agent, or by sending special damage etc. events.

4 Implementation

4.1 Features

The implentation needs to have the following features:

- Storing knowledge in *knowledge base*.
- Changing knowledge based on *rules*.

- Match event and return all possible actions.
- Planning that returns the best action in given situation.

4.2 Graphs

A fast way to implement planning is using graphs. A model can include a finite graph constructed by the rules of the model. This graph includes all the possible states as nodes. The graph has two kinds of transitions: events and actions.

A rule like $a \multimap b \oplus c$ can be implemented using an extra state, where there are two events, one for b and one for c. There can also be actions that are controlled by other players.

First, the desires for the states are calculated. These are the *real values* of the nodes. The *virtual values* of the nodes tell how close they are to good states. The virtual value of a node is for every other state, the maximum of the desire of the state and desires of every other state minus the average effort of actions needed to get to the other state.

The value of a general node x is calculated like this:

Probability of events:
$p_e = \sum p_{e_n}$
Probability of actions of a player:
$p_{a_n} = \sum p_{a_{nm}}$
Total probability for player actions:
$p_a = \sum p_{a_n}$
Total probability:
$p_{total} = p_e + p_a$
A random event happens:
$e = \sum p_{e_n} t_{e_n}$
An actor tries an action:
$a_{nm} = 1/p_{total} * (p_{a_{nm}} t_{a_{nm}} + (p_{total} - p_{a_{nm}})x)$
Actor selects best action for him:
$a_n = \max_n(a_{n1} \ldots a_{nm})$
Total value of the node:
$x = \max(d_x, 1/p_{total} * (p_e e + \sum p_{a_n} a_n))$

In the formula, d_x is the real value of the node x. $\max_n$ selects the best value for player n. The above formula doesn't handle $\oplus$-cases, but they could be handled by setting $t_{a_{nm}} = \sum p_{a_{nmk}} t_{a_{nmk}}$. Then $t_{a_{nmk}}$ represents the k'th outcome of the m'th possible action of player n. It is calculated $t = y_t - e_t$, where e_t is the effort and y_t is the target node.

Calculating the exact value for each node in the graph is hard because that would mean solving a linear system with max operations. The solution can be estimated by guessing a value for each node, and then calculating new guess by evaluating the right hand side of the formula, replacing the values of the nodes with guesses. Because the bigger nodes affect the smaller nodes, the estimation

is done in best-first search order. After few iterations, good enough estimation should be achieved.

5 Using the planning in the game

The engine is parameterized by the event language. The language includes actions as a subset. Event language also includes the spoken language of the game.

Integration steps:

- Full language of events is defined.
- The knowledge engine is made from the adventure experiment.
- Knowledge engine is loosely integrated to the system.
- Knowledge engine is used for event handling.
- Knowledge engine is used for planning.

Modifying and accessing the knowledge base of an agent area actions in the script. When the main script of an agent comes to a point where planning is needed, it will use the planner to generate a plan. Then the plan can be executed using a plan execution -script.

5.1 Plot engine

The graph mechanism can be used for the plot engine. The plot engine is an actor that generates events so that the game goes according to a plot. Plot can be described in a non-linear way using a graph. There are goals which can be achieved using the plot. Goals can change because of the actions of players and a new plot will be generated. The engine keeps track of the game state and makes plans if the state becomes bad. NPCs can be used by the plot engine to help the players.

6 Advanced AI

6.1 Learning

In the system, there are to different kinds of learning. Using the graphs, it's possible to change the probabilities and efforts of actions. Harder kind of learning is learning new context and the effects of events in them.

6.2 Memory

Memory contains symbols and associations between them. For example symbol "number" is associated to "1", "2", "3" ... and symbol "month" to "january", "february".... Associations are directed, sometimes the weight of the association is smaller backward than forward, for example "sunday", "saturday", "friday"... In addition to single associations between symbols, there are associations between sets of symbols, for example "month" and "11" is associated to "november", and "month" and "number" is associated to "12" or "30". Perhaps these associations can be derived from single associations.

Associations and symbols are got from the environment. When an association is recalled, if it's useful, it's weight becomes bigger, if it's useless or harmful, it's weight becomes smaller.

6.3 Better planning

Splitting the actions to several different contexts is somewhat artificial. To get the best plans, all actions should be in one big context, and then all possibilities could be calculated.

7 Conclusion

The system can now do knowledge, planning and reacting to environment.

Now we can encode situations instead of plans. The NPCs understand situations and can make plans to achieve their goals. Planning is also efficient when the situations can be described using a finite number of states.

There are several advantages over hardcoded plans:

- Behavior can vary because of the planning mechanism.
- Encoding a situation is more efficient than encoding all the possibilities if there are many different goals.
- The NPC always knows what it is doing.

Advantage of hardcoding is more flexibility. A goal task is just a specific case of hardcoded task. For different goals, there might be different planners.