Empath

A Modeling Language for Living Beings

White Paper

Jeremy Posner, Nalini Kartha, Sampada Sonalkar, William Mee

COMS4115 Programming Languages and Translators

26 September 2006

Introduction

The Empath language is designed to allow simplified models of the development of living beings over time. Beings which could be modeled include single entities, like a virtual pet cat, a Tamagotchi-style creature[1] or a simulated human character in a game. In addition, entire ecosystems, such as a simulated garden, beehive or an ocean are possible. The Empath models are based on characteristics which vary over time, as well as states in finite state automata.

Motivation

The Empath language is designed to allow rapid prototyping, development and testing of virtual beings and their interaction with human users. It is a domain-specific language with special constructs which make achievement of these objectives easier and more concise than programming them in a general-purpose language. The language includes necessary constructs to support its goal, such as time flow.

Empath allows users to interact with virtual beings through simple, programmer-defined actions, such as "feed" or "put to sleep." The language also allows the programmer to create growth and decay formulae to define how a being's needs change over time. The system can be used to develop characters similar to those one might find in a Tamagotchi or other virtual pet, as well as the more complex sort of character one might encounter in The Sims. Using Empath, a game designer can quickly answer questions like how robust is a character in a game? how interesting – or fun – is a virtual pet to interact with?

Empath is back-ended by the Java language, and is therefore usable not only on desktop computers, but on smaller hand-held systems as well. Empath supports both the definition of models of living things as well as their execution. By offering a simple API for graphics subsystems to interact with the executing environment, it supports user interfaces of varying complexity.

Empath is targeted at the game and broader entertainment industry. Virtual pets, for example, are big business. Bandai, the company behind the Tamagotchi and Digimon

1 See http://www.tamagotchi.com/ (official site) or http://en.wikipedia.org/wiki/Tamagotchi (good definition)

games, has already sold over 20 million units of its “Tamagotchi Plus”[2], and in 2005 recorded net sales of over US$ 2 billion [3].

Concepts and Definitions

Within the context of Empath, we define an ***entity*** as a simplified model of a living being.

Entities maintain a set of internal ***characteristics*** which indicate important attributes of the being they model. Characteristics might, for example, indicate a hunger level, happiness, age, health level or simply a name. Characteristics can vary in a defined way over time, or as a result of external events (see below). Simple characteristics have a defined type, such as boolean flags, strings, ranges etc. Since characteristics can be hidden from the end user’s view, hidden characteristics can be a powerful means of creating complex internal processes for an entity.

A more complex type of a characteristic is a ***state****.* This is a reference to a finite state automaton, with defined transitions between the states. For an entity modeling a frog, an important state would indicate whether the frog is currently an egg, a tadpole or adult frog. Transitions between states are triggered as a result of a combination of other characteristics which reach defined ***threshold levels***. For example, given the correct combination of warmth and time, the frog's state will change from egg to tadpole. While an entity can only be in one base state, a state can itself then be defined as a set of states, so that within the state of being an adult frog, the entity could be hopping, sleeping, eating or croaking. The concept of states are described in more detail in the section on finite state automata below.

Entities interact with their environment via ***events***. These events can originate from the environment (such as the simple flow of time) or from a human interacting with the entity. Eventually the language may be expanded to support interaction between entities. The event would typically have some affect on one or more of the entity's characteristics; so stroking the virtual cat would increase its happiness level, possibly triggering it into the purring state (or into the scratching state; cats being non-deterministic). An event could be instantaneous (such as a feeding) or could occur over a period of time (such as getting tired over the course of a day).

Finite State Automata

Important characteristics of an entity, such as its lifecycle, are modeled using finite state Mealy automata. A Mealy automaton is a set of states and transitions, with a single defined initial state.

The lifecycle of an entity can typically be modeled as stages in its life, and additional activities in a particular stage. It is necessary to separate these in order to achieve a simplified intuitive design. We have introduced the notion of hierarchy of automata for this purpose. An entire finite state automaton can be embedded in a state of an automaton at a higher level. This nesting can be done to any level, although it is anticipated that 2-4 levels of nesting would be sufficient to describe most entities.

Semantically, when the top level parent state is active, one of the states in the child

2 http://www.bandai.co.jp/e/releases/E2006041201.html
3 http://www.bandainamco.co.jp/en/ir/financial/pdf_bandai/financial/05_financial_co.pdf

automaton is also active. If the parent state is inactive, the child automaton is also inactive. There is no notion of history viz, whenever the parent state is entered, the initial state of the child automaton is activated. Inter-level transitions are disallowed as these are non-modular and ambiguous. The exception to this rule is that the system does not require that all of the embedded automata go to the same depth. For example, an egg state may only have two states (waiting to hatch and hatching), while the upper level state into which the hatched egg transitions may have far more depth of detail.

The execution semantics of the automata is an instantaneous reaction to events to produce output actions and change state. The following are considered valid events for the transition firing:

1. user input events
2. reaching defined characteristic thresholds
3. boolean expressions formed from the above using ‘and’, ‘or’, and ‘not’ operators.

In response to user input events, the appropriate characteristics of the entity are modified.

States have onEntry() and onExit() functions for performing certain initializations and resets while entering/exiting the state. While the state is active, the onClockTick() function is executed in each reaction. When a transition fires, the onExit() of the source state is executed followed by the onEntry() of the destination state.

In a hierarchy, transitions are evaluated inside out and without preemption. A tadpole that has been given food will feed before metamorphosing into an adult frog. Neither transition can be preempted by the other one. The overall semantics of execution is non-preempting. A transition that is enabled must be fired; it cannot be disabled by an event in some other part of the system.

Language Outline

Empath uses syntax based on that of C and Java. As with those languages, whitespace is generally unimportant, and blocks of code are enclosed in braces. Functions can be included from library files, allowing for eventual expansion of the language's capabilities beyond the basic, built-in functions. Basic math functions (addition, subtraction, multiplication, division, modulo) and logic functions (and, or, not, etc.) are evaluated using symbols and syntax rules identical to those of Java. Additionally, flow control operators (if, while, for) are also set up to use syntax identical to that of Java. The idea is to create an environment that is relatively familiar to those who already know Java and C, reducing the learning curve for programmers transitioning to the new language. This syntactic style also has the advantage of being proven as extremely flexible for both simple and complex programming projects.

An example code in draft syntax is given at the end of this document, in Appendix A.

Limitations and Restrictions

We have made several important simplifying limitations for Empath. Although it is feasible for entities to interact with each other, we do not consider this in the initial version of Empath. Furthermore, Empath only supports the use of a single automaton at any one level (although several automata can be nested).

Future Directions

Although the base Empath language includes only limited functionality, there is the potential for programmers to release libraries to allow for significantly increased capabilities. For example, although the language has no built-in capacity for artificial intelligence, the language does include sufficient math support to provide the foundation for a set of AI libraries. Since the language can handle multiple status values for a character, they can be set up to interact with each other in ways that allow for the programmer to define all sorts of hidden behavior, including the sorts of processes necessary for AI. Other libraries might include support for more sophisticated math and interfaces. Because individual beings are so complex, the capacity for creating them with Empath is limited only by the imagination of the programmers.

Another important area of future work would be to allow interaction between entities to be defined in the language. Networked toys, such as the new generation of the Tamagotchi or Nintendo's Nintendogs[4], are increasingly important.

4 http://www.nintendogs.com/

Appendix A: Example Syntax

We have described an example program which models the behavior and interaction with a pet dog. The dog goes through three main states in life, namely the 'puppy' state, the 'adult' state and finally the 'dead' state. There are some characteristic variables associated with the 'dog' entity whose values determine its current state. Transitions from one state to another are described based on the change in variable values as well as based on user interaction. For example, the 'puppy' changes into an 'adult' when its age reaches 10 years. In each of the first two states, there is a further division into sub-states, which are to be modeled. For example, the puppy can be in one of three states – the 'wagging tail' state, the 'hungry' state or the 'needs to be played with' state. Valid transitions between each of these sub-states are described. In particular, variables that are associated with secondary levels in the state hierarchy cannot be accessed at the primary level.

```
/* The following defines the high level entity or creature that is
being simulated. The definition of the entity includes a set of
characteristic variables, a set of events that can occur while
interacting with the entity, a set of states that the entity can
be in, the possible transitions between states as well as any
possible sub-entities or sub-states */

ENTITY DOG
{

     /* Set of global variables applicable to all states for the
     "dog" entity */

     float age = 0.1;
     int hunger  = 0;

     /*  "clock" is a keyword. The value of this variable
     specifies how often the state of the high level entity will
     be re-evaluated. The unit for this is ******/

     clock = 10;

     event feed()
     {
          hunger=0;
     }

     /* "onClockTick" is a keyword. It defines a function which
     will be called automatically every clock cycle. The parameter
     that the user passes must be a whole number and defines how
     often the specified instructions will be executed. In the
     following example for every 4 clock cycles, the age will be
     incremented by 0.1.*/
```

```
onClockTick(4)
{
     age+ = 0.5;
}


/* The following is the set of states that the "dog" entity
can be in. The first state in the list is taken to be the
initial state by default. This list must begin with the
keyword "stateset". The state names must be separated by
commas and the list must be terminated with a semi-colon. */

stateset Puppy, Adult, Dead;

/*  The following is the list of transitions defined between
states of the "dog" entity. This list must begin with the
keyword "transition". The individual transition definitions
must be separated by commas and the list must be terminated
with a semi-colon. Each transition must be defined in the
following format -
From_state to To_state when (condition) */

transitions
Puppy to Adult when (age>=10),
Puppy to Dead when (hunger == 10),
Adult to Dead when (age>=20 || hunger == 10);

/*The following defines the sub-entity "Puppy" and its local
variables, state set, etc.*/

entity Puppy
{
     int happiness = 10;

     onClockTick()
     {
          hunger+ = 1;
          happiness- = 1;
     }

     onEntry()
     {
          print "Hello world!!!";
     }

     event play()
     {
          happiness = 10;
```

```
      }

      stateset WaggingTail, Hungry , Needs_to_play;

      transitions
            WaggingTail to Hungry when (hunger >= 8),
                  Hungry to WaggingTail when (hunger == 0),
                  WaggingTail to Needs_to_play when (happiness <=
      3),
                  Needs_walk to WaggingTail when (happiness ==
      10),
            Needs_walk to Hungry when (hunger >=8);

      state WaggingTail
      {
            onEntry()
            {
                  happiness = 10;
            }
      }
}

entity Adult
{
      int time_to_next_walk = 120;

      onEntry()
      {
      }

      onClockTick(5)
      {
            hunger+ = 1;
            time_to_next_walk- = 15;
      }

      onExit()
      {
      }

      event walk()
      {
            time_to_next_walk = 120;
      }

      stateset Barking , Hungry , Needs_walk;

      transition
```

```
                Barking to Hungry when (hunger >= 8),
                     Hungry to Barking when (hunger == 0),
                     Barking to Needs_walk when (time_to_next_walk
           <= 5 ),
                     Needs_walk to Barking when (time_to_next_walk
           == 120),
                Needs_walk to Hungry when (hunger >=8);

           state Barking
           {
                onEntry()
                {
                     time_to_next_walk = 120;
                }
           }

      }

      state Dead
      {
           onEntry()
           {
                if ((hunger == 10) && (age<20))
                     print ("Shame on you!! Your dog died!!");
                else
                     print ("Dog died of old age");
           }
      }
}
```