EMPATH

LANGUAGE REFERENCE MANUAL

Jeremy Posner jp2288@columbia.edu
Nalini Kartha nkk2106@columbia.edu
Sampada Sonalkar ss3119@columbia.edu
William Mee wjm2107@columbia.edu

COMS4115 Programming Languages and Translators

19 October 2006

Introduction

This document is a formal reference for the Empath language, which is designed to model living beings in a simplified form.

Grammar notation

We use standard regular expression for expressing the production rules of the language. Symbols enclosed in single quotes are terminals. All other symbols are non-terminals. * represents zero or more occurrences of the preceding expression and + represents one or more occurrences. | represents a choice between the two expressions on either side of the | character and ? means that the preceding expression is optional. Braces are used to give precedence and denote the scope of an operator.

Lexical conventions

Comments

The // character combination represents a single line comment. For multi-line comments the /* character combination indicates start of comment block and */ character combination indicates end of comment block. The symbol sequence */ cannot appear within a multi-line comment.

Tokens

Tokens include identifiers, keywords, constants, operators, and punctuations.

Identifiers

Identifiers are represented by a sequence of letters and digits starting with an alphabetic character or an underscore character (‘_’). An identifier represents a name for one of the following constructs: variable, function, event, trigger, state.

```
digit      → ( ‘0’..‘9’)
letter     → (’a’..’z’ | ’A’..’Z’)
id         → (’_’ | letter) (’_’ | letter | digit)*
```

Keywords

The following identifiers are reserved keywords and may not be used otherwise:

boolean **entity** **event** **float** **for**

if	**init**	**int**	**label**	**output**	**range**
return	**state**	**string**	**tick**	**to**	**transition**
trigger	**void**	**while**			

The keywords may only be used in the lower case alphabet. The language is case sensitive, so an identifier ‘age’ is distinct from ‘AGE’ or ‘Age’. The Java convention of identifiers starting with lower case alphabet and judicious use of upper case characters as separators is encouraged.

Operators

The language supports a small set of mathematical and logical operators for manipulating entity characteristics. The precedence and associativity rules for these operators are the same as those of mainstream programming languages, specifically C[1], hence we will not redefine them in this manual.

Arithmetic operators	+ - * / %
Shorthand arithmetic operators	+= -= *= /=
Relational operators	< > <= >= == !=
Logical operators	&& ||

Punctuation

The following characters are used for punctuation in the language

;	statement end
,	argument list separator
=	declaration initializer
“ “	string literal
..	range specifier
[ ]	range delimiter or array index delimiter
{ }	function body or block statement delimiter
( )	function parameter list delimiter

Constants

The language defines integer constants, real constants (including only a fractional part, no exponents), boolean constants and string constants

```
Const      → IntConst | RealConst | BoolConst | StrConst

IntConst   → digit +

RealConst  → (digit) * ‘.’ (digit) +
```

[1] For a definition of this precedence, see the C Reference Manual (Ritchie) available online at http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf

```
BoolConst   → ( true | false )

StrConst    → ‘”’ ( letter | digit | ! # % ^ & * ( ) - _ = + ~ ' “ :
              ; ? / | \ { } [ ] , . < > $ )* ‘”’
```

A **“** can be included within a string by escaping it with another “, i.e. “”.

Line Terminators and Whitespace

ASCII spaces, tabs, new line, carriage returns and form feeds are all considered as whitespaces. Whitespace is ignored except as a token separator.

In-built functions

The following function names are not keywords but – similar to the “main” function name in languages like C and Java - if present have a special meaning for the runtime environment:

- onClockTick
- onEntry
- onExit

This is explained in more detail below.

Empath Language Specification

Overview

An Empath program essentially consists of a description of the characteristics of an entity, a finite state machine (FSM) for representing its lifecycle, and a set of options (known as events) available to the user for interacting with the entity. An example piece of Empath code and an accompanying FSM is given in an appendix at the end of this document to make the understanding of this language reference manual easier.

In general, Empath is a C-like language (similar to Java or C#) in its use of curly braces, operators, function definitions etc. The nature of the Empath language necessitates the use of such constructs for performing simple arithmetic and logical operations on entity characteristics; and we did not want to reinvent the wheel in defining an unusual syntax.

Entity Specification

The entity specification gives a name to the entity being modeled and specifies the associated variables (usually characteristics of the entity), functions, states and transitions. The entity specification may also optionally include an embedded specification for one or more of the states of the entity. There can be only one entity specification in an empath program and the program must begin with it. All other constructs of the empath program are embedded in the entity specification block.

```
EntitySpec →      'entity' id ProgramBody
```

State Specification

A state specification is very similar to an entity specification. A *simple state* is one defined as an FSM without any FSMs embedded inside it, and including at most one or more of the special functions onClockTick(), onEntry() and onExit() functions, but no more.

If a state is *complex* (with a single FSM embedded inside it to form a hierarchy), it may include the above mentioned functions, but also a description of the embedded FSM. This distinction is important conceptually, but is not part of the language specification per se. Figure 1 in Appendix describes a dog entity. Here *Dead* is a simple state. The *Puppy* state is a complex state and includes a description of the daily activities of a puppy.

```
StateSpec    →    'state' id ProgramBody

ProgramBody →     ('label' StrConst)?
```

```
'{'
(Decls)? (FuncDef)? (States Trans)? (StateSpec)*
'}'
```

The label keyword is used to specify a string that is passed to the user interface for display purpose. The id is used as an internal handle. Labels are used for states and functions, particularly events.

States

This lists the names of the set of possible states that the entity can be in. It works like a forward declaration for states, and is helpful to the programmer while specifying the transitions. The detailed definition for a state can be defined later in the *StateSpec* block. The initial state (the default state which the entity will be in at the beginning of execution of the program) is marked by including the keyword *init* before the corresponding state name. If the initial state is not thus explicitly marked then it is assumed to be the first state listed. The suggested convention for state names is that they be capitalized however this is not a requirement.

```
States →   'state' (init)? id ( ',' (init)? id) * ';'
```

Transition Specification

A transition specification defines the possible transitions that can occur between the entity's states. Each transition must specify a '*from state*', a '*to state*', a '*trigger*' and, optionally, an action. The trigger may be an expression which evaluates to a boolean value or a trigger function.

```
Trans →     'transition'
            ((id |'*') 'to' id if '(' Condition ')' ('/' FuncCall)?)
            (',' (id | '*') 'to' id if '(' Condition ')'
            ('/' FuncCall)?)* ';'

Condition -> Expr | FuncCall
```

A '*' in place of a '*from state*' is a shorthand operator signifies that a transition can be taken from any of the states in this FSM to a state outside the FSM. In general we allow inter-level transitions from an inner-level state to a higher level state. It would be useful to mention here that the semantics of execution of this FSM are as follows:

- Transitions are evaluated from outside to inside
- A transition triggered at an outer level preempts transitions triggered at an inner level.

Variable Declarations:

```
Decls → Decl Decls | ε
```

```
Decl → Type (id ('=' Const)? ((',' id ('=' Const)?) *) ';'

Type → int | float | boolean | string | range'['(digit+)'..'(digit+)
']'';'
```

Function Definitions

Because of the problem domain which Empath is addressing, in particular its use of FSMs to define the states that an entity is in, there are several different categories of functions. These are described below.

Event: An event is a special type of function which is qualified with the keyword *"event"*. These functions are marked to be visible to the user interface. They provide a means of interaction with the Empath program, and it is necessary to mark a function so that it can be recognized as a user input by the user interface.

Trigger: A trigger is a special type of function which is qualified with the keyword "trigger". These functions have the sole purpose of evaluating if a threshold has been reached and if the corresponding transition will be carried out. As such they are subject to the following restrictions:

- cannot modify characteristics
- return type must be boolean
- can optionally modify variables in local scope but cannot modify those which have global scope

Action: An action is a function defined in the action part of a transition definition. The instructions in an action function will be executed if the trigger corresponding to the transition has been evaluated as true and the transition is fired. An action cannot return a value i.e. the return type of an action must be void. Any function whose return type is 'void' can be an action.

```
FuncCall    →    id '(' (id (, id)*)? ')'

FuncDef     → Func FuncDef | ϵ

Func        → ('trigger' | 'event' | RetType) ('label' StrConst)? id
                  '('(Type id ((',' Type id)*)? ')' '{' FuncBody '}'

RetType     → void | Type

FuncBody    → Statement*

Statement   → Decls
                  | Expr ';'
                  | 'if' '(' Expr ') (Statement | StatementBlock)
                  |  'if'  '('  Expr  ')  (Statement  |  StatementBlock)
                  'else' (Statement | StatementBlock)
                  | 'for' '(' (Expr)? ';' (Expr)? ';'
                        (Expr)? ')' (Statement | StatementBlock)
```

```
                  | 'while' '(' Expr ')' (Statement | StatementBlock)
                  | 'output' '(' Expr ')' ';'
                  | return (Expr)? ';'

StatementBlock → '{' Statement + '}'

Expr          →    '(' Expr ')'
                  | id
                  | Const
                  | FuncCall
                  | id '=' Expr
                  | IncrExpr
                  | ArithExpr
                  | RelExpr
                  | LogicalExpr
                  | 'tick' IntConst
                  | id '@max'
                  | id '@min'

IncrExpr      → id '++'
                  | id '--'

ArithExpr     → Expr '*' Expr
                  | Expr '/' Expr
                  | Expr '%' Expr
                  | Expr '+' Expr
                  | Expr '-' Expr
                  | id '+=' Expr
                  | id '-=' Expr
                  | id '*=' Expr
                  | id '/=' Expr

RelExpr       →    Expr '==' Expr
                  | Expr '!=' Expr
                  | Expr '>' Expr
                  | Expr '<' Expr
                  | Expr '>=' Expr
                  | Expr '<=' Expr

LogicalExpr →      Expr '&&' Expr
                  | Expr '||' Expr
```

Most of the statements and expressions are similar to those in the C or Java language. We describe the new ones:

- output(x): The parameter of this statement is passed to the UI for displaying
- tick n: This expression is useful in the onClockTick() method where on every nth clock tick some characteristic is changed.
- id @max: This expression checks whether a range type variable has attained the maximum value defined in the range.
- id @min: Similarly, this expression checks whether a variable has attained the minimum value defined in the range.

Special Functions

As already mentioned, Empath attaches special meaning to certain function names to provide FSM functionality. These function names and how they are handled are given in the table below

onClockTick	executed for every clock cycle that an entity is in a state
onEntry	executed once when a state is entered
onExit	executed once when a state is exited

Appendix A: Example FSM for Dog Entity

Here we give the FSMs which are described by the example code in Appendix B.

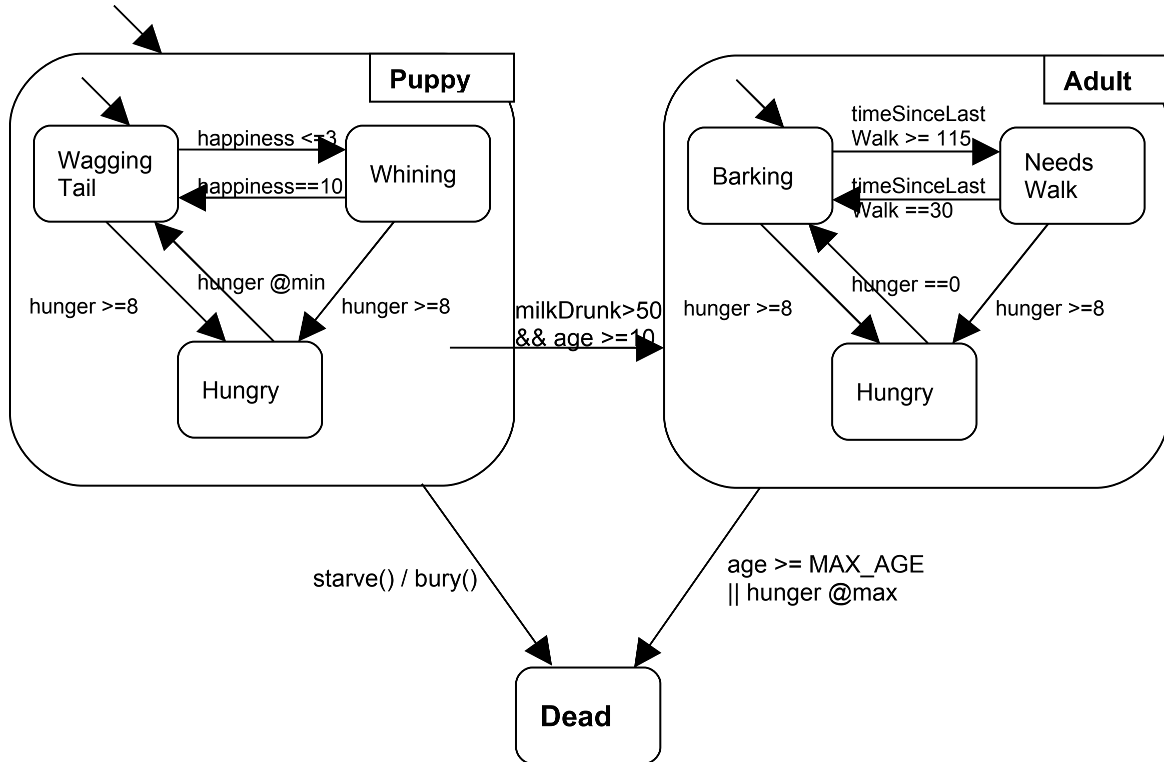

Figure 1: Lifecycle of a Dog entity

Appendix B: Example Empath Code for Dog Entity

```
/* 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 subentities
or substates
*/

entity dog label “Pet Tommy” {
      /* Set of characteristics for the dog entity which are seen
       * within this scope
       * Empath supports the following characteristics
       * int - integer
       * float - floating point value
       * string - text value
       * boolean - boolean value
        *  range  -  bounded  range  of  values  defined  by  two  ints  as
inclusive upper and lower bounds
       */

      string name = “Tommy”;
      float age = 0.0;
      /* a range characteristic */
      range [0 .. 10] happiness = 5;      //range is inclusive
      range [0 .. 100] hunger = 70; //syntatic sugar for range [0..100]

      int MAX_AGE = 20;

      /*  events  are  marked  functions  which  are  visible  to  the  user
interface
       * they offer a means of interaction
       */

      event feed() label "Feed the Dog" {
            hunger=0;
      }

      /* this is a function which is used as a threshold trigger
       * triggers
       *    - cannot modify characteristics
       *    - must return a boolean value
       *    - can optionally modify variables in local scope
       */

      trigger starve() {
            if (hunger>95) {
                  returnValue= true;
            } else {
                  return false;
            }
      }

      /*  this  is  an  action  (i.e.  emission)  which  is  called  when  a
transition is taken. It
       * cannot return a value
```

```
        * any function can be an action
        */
       void bury() {

       }

       /* this is a normal function
        */

       string getName() {
             return name;
       }


       /* "onClockTick" is a special function which is called on every
        * clock cycle.
        */

       onClockTick() {
             if (tick 4) {
                   hunger++;
             } else {
                   age+ = 0.1;
             }

       }


       /* 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.
        *
        * By convention, states are capitalized - this is not enforced
        *
        */

       state init Puppy, Adult;
       state 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 semicolon.
        * Each transition must be defined in the
        * following format –
        * From_state to To_state if (condition) / action
        * the condition can be an inline conditional or a function which
        * returns a boolean value. The condition is always evaluated for
        * each transition so cannot update characteristics.
        * the action is optional and is invoked when the transition is
triggered
        */

       transition Puppy to Dead if (starve()) / bury() ;

       /* range support special operators @max and @min which return
true if the
        * range is currently at its max or min limit respectively
        */
       transition Adult to Dead if (age>=MAX_AGE || hunger @max);
```

```
      /* The following defines the "lifecycle" states which this entity
can
       * be in during its life
       * the init keyword
     */
      state Puppy label "Puppy" {
            int happiness = 10;
            range [0..100] milkDrunk=0;
            onClockTick() {
                  hunger+ = 1;
                  happiness=1;
            }

            /* onEntry() and onExit() are functions which, if defined,
are invoked when
             * a state is entered or exited
             */
            onEntry() {
                  output “Hello world!!! My name is”;
                  output name;
            }
            onExit() {
                  output “I've grown up!!!”;
            }

            event play() {
                  happiness = 10;
            }

            event suckle() {
                  milkDrunk++;
            }

            /* states Whining and Hungry have no further definition */
            state init WaggingTail, Whining, Hungry;

            transition
                  WaggingTail to Hungry if (hunger >= 8),
                  WaggingTail to Whining if (happiness <=3),
                  Hungry to WaggingTail if (hunger @min),
                  Whining to WaggingTail if (happiness == 10),
                  Whining to Hungry if (hunger >=8);
            /* a wildcard * is used to denote from *any* state, and the
target
               state is normally at the parent level
            */
            transition * to Adult if (milkDrunk>50 && age>=10);

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

      /* an adult dog */
      state Adult {
            int timeSinceLastWalk = 0;
            onEntry() {
            }
```

```
		onClockTick() {
			hunger+ = 1;
			timeSinceLastWalk++;
		}

		onExit() {
		}

		event walk() {
			timeSinceLastWalk = 0;
		}

		state Barking , Hungry , NeedsWalk;

		transition
			Barking to Hungry if (hunger >= 8),
			Hungry to Barking if (hunger == 0),
			Barking to NeedsWalk if (timeSinceLastWalk > 115 ),
			NeedsWalk to Barking if (timeSinceLastWalk == 30),
			NeedsWalk to Hungry if (hunger >=8);

		state Barking {
			onEntry() {
				timeSinceLastWalk = 0;
			}
		}
	}

	/* the Dead state */
	state Dead {
		onEntry() {
	/* make the owner feel bad if the dog starved to death */
			if ((hunger @max) && (age<MAX_AGE))
	/* output is understood differently by the different ui's */
				output ("Shame on you!! Your dog died!!");
			else
				output ("Dog died of old age");
		}
	}
}
```