Programming Languages and Translators

COMS 4115

Ecosystem Modeling Language
EcoMod Language Reference Manual

Vika Kanchakouskaya

June 26, 2007

1. Introduction

EcoMod is a language that helps simulate simplified models of ecosystems for the purposes of observing, studying, and experimenting with such models. EcoMod syntax resembles that of C++ or Java, making it easy to pick up for programmers familiar with those languages. However it is domain-specific and has build-in high-level constructs, which make ecosystem modeling easier than with general purpose programming languages. This manual describes the rules and the conventions of EcoMod and can be used as a reference by EcoMod programmers.

The following conventions are used in this manual to describe the rules of the language.

*	is used to denote one or more of the preceding token
< >	are used to enclose a group of tokens
[ ]	are used to enclose an optional token or group of tokens

2. Lexical Conventions

There are 5 types of tokens in EcoMod: keywords, identifiers, constants, operators and punctuation. White spaces denoted by space, tab, newline, and carriage return characters serve as separators between tokens and are otherwise ignored. At least one white space character is required to separate keywords, identifiers, constants and operators.

2.1 Comments

Comments follow C++ or Java style single line comment convention: all characters between // and new line or carriage return are considered comments and are ignored by the compiler. There is no support for multi-line comments, and // has to be placed at the beginning of every line intended as a comment.

2.2 Keywords

The following items are reserved as EcoMod keywords and may not be used otherwise:

if	*int*	*float*	*System*	*States*
else	*string*	*return*	*Object*	*Habitat*
for	*bool*		*Qualities*	
while	*true*		*Transitions*	
goto	*false*		*Action*	

EcoMod is a case sensitive language, and upper and lower case letters are considered different. It should be noted that domain-specific keywords like *System* and *Object* start with an upper case letter to differentiate them from general keywords like *for* and *else.*

2.3 Identifiers

Identifiers are sequences of alphanumeric characters or underscores starting with a letter. EcoMod keywords cannot be used as identifiers. E.g. *Whale, piglet, human_being* and *zebra2* are valid identifiers, while *_frog, 4rest,* and *Object* are invalid.

2.4 Constants

There are 3 types of constants: integers, floats and strings.

Integers are sequences of digits and can only be decimal.

Floats consist of integer part, decimal point, and fractional part. Integer and fractional parts are sequences of digits and both are required to form a valid floating constant.

Strings are sequences of characters enclosed in double quotes. A double quote can be included into a string by escaping it with '\' .

2.5 Operators

EcoMod supports a number of arithmetic, relational and logical operators.

Arithmetic operators:	+ - / * %
Relational operators:	< > <= >= == !=
Logical operator:	&& ||
Initialization operator:	=
Dot operator:	.

The meaning and the precedence of the operators is the same as in C++ and Java. Dot operator is used to select a member of an Object or a System, similar to selecting a member of a class in C++/Java.

2.6 Punctuation

Punctuation is used to delimit other language tokens. The following punctuation symbols are supported in EcoMod

;	statement delimiter, denotes end of statement
[]	used for array indexing
{}	delimiter for a block of statements
()	delimiter for arguments to a function
“”	string delimiter

3. Types

EcoMod supports the following storage types

int	32 bit integer
float	floating point number with fractional part
string	a string of characters
bool	boolean type with values true or false
Object	type for objects inhabiting the ecosystem

4. Statements

In EcoMod statements are executed sequentially, unless stated otherwise. This section describes the types of statements supported in EcoMod

4.1 Expression statements

This is a most common type of statement. Expressions are usually assignments or function calls and should be separated by semicolons. It is a good practice to put one expression per line in an EcoMod program.

4.2 Conditional statements

The following types of conditional statements are supported:

*if (expression) { statement; * }*

*if (expression) { statement; * } else {statement; * }*

In each conditional statement expression is evaluated, and if non-zero or true, the block of statements is executed, otherwise, if *else* clause is present, the block of statements following *else* is executed. The *else* ambiguity is resolved according to C language convention: else is bound to the last elseless *if*.

4.3 Loops

EcoMod supports the traditional while and for loops of the following format:

*while (expression) { statement; * }*

for (expression1; expression2; expression3) {statement;}*

In the *while* loop, the expression is repeatedly evaluated in a loop and the block of statements is executed as long as the expression is non-zero or true. In a *for* loop, *expression1* is loop initialization; *expression2* is a test performed before each iteration, which determines if the block of statements is going to be executed; *expression3* is executed after each iteration.

4.4 Return statements

return expression;

Return statements are used to return the value of the expression to the caller of the function.

5. Functions

Due to the fact that EcoMod is a domain specific language that uses FSM to describe the states that the objects are in, there are several types of functions supported by the language.

5.1 Transitions

Transitions are functions that check on a specific condition associated with them and change the state in which the object is in, if the condition holds true. Transitional functions are implicitly defined as part of Object definition in the following form:

*FSM_name { < state : if (expression) goto state; > * }*

FSM_name denotes the name of the group of states and transitions that are part of one Finite State Machine. This is used to provide support for multiple FSM's per Object. Transitions are never called directly, and their evaluation can be triggered by calling updateFSM() built-in member function as follows:

Object_name.updateFSM(FSM_name);

Examples of transition definitions and calls are provided at the end of this document.

5.2 Actions

Actions are special function that are accessible by the users and provide the user interface for interacting with the members of the *Habitat* and for manipulating their behavior. Actions don't have return values and are defined with *Action* keyword:

*Action identifier ([parameter list]) { <statement> *}*

5.3 Standard functions

EcoMod also supports standard C++ or Java -like functions, which should be defined in a traditional way including return type, function name, list of arguments, and function body enclosed in { }. Such function are private to Objects and can only be accessed from the body of the Object declaration.

5.4 Built-in functions

Add(Object)	- member of *System*, adds a copy of the *Object* to the global *Habitat* array
Remove()	- member of *Object*, removes the *Object* from *Habitat*

Size()	- member of *Habitat*, returns the array size
updateFSM()	- member of Object, call all transitional functions and updates the states
onExit()	- member of a State, if defined – is implicitly called when Object leaves the state
onEntrance()	- member of a State, if defined – is implicitly called as Object enters the state
print()	- prints constants to stdout

6 Additional EcoMod Language Specifications and Summary

An EcoMod program generally consists of two parts: *definition* and *main*.

6.1 Definition Part

Definitions consist of defining *System* and *Objects* that inhabit it. There can only be one *System* per EcoMod program. System has a built-in *Habitat* array that contains all the objects inhabiting the *System. Objects* that are defined within the body of the *System* definition are added to *Habitat* automatically. Additional *Objects* can be inserted via *Add(Object)* function call.

Object definition should be nested into *System* definition. The first release of EcoMod only supports one type of *Object* per *System*, consequently there can be only one nested *Object* definition; however multiple instances of that *Object* can be added to the system with the help of *Add(Object)* function. *Objects* may possess certain *Qualities*, which can be depicted with the help of *Qualities* keyword. *Qualities* are a set of significant attributes, such as age, sex, defense strategy, health, etc. The behavior of the objects is modeled via FSM representation, with *States* and *Transitions*. Every state has built-in functions *onExit()* and *onEntrance()*, which may or may not be defined. *Actions* are *Object* members that provide a public interface for *Object* manipulation.

6.2 Main

Main part of the program in enclosed in *void main()* function. This is the part of the program where simulation takes place. *main()* function with void return type and no arguments should always be part of an EcoMod program.

7. Sample EcoMod program

```
// This very simple program defines an experimental humanity
// which is originally inhabited by one person.
// 100 years of humanity development is simulated
// in order to observe how its population grows.
// We make a simplifying assumption that a person
// is able to regenerate independently.
System Humanity
{

  // nested Object declaration
  Object Human
  {
     // Qualities are member variables of the  Object
    Qualities:
      int age = 0;
      int avgNumChildren = 2;
      bool hasChildren = false;

    States:
      // FSM name followed by states comprising it
      development = child, adult, aging, dead;

      // Transitions must be defined for every pair of states that can be
      // transformed from one to another with help of Transitions keyword
     Transitions:
        development {
         child:
           if age >= 16 goto adult;
         adult:
           if age >= 45 goto aging;
         aging:
           if age >= 80 goto dead;
         death:
          // onEntry() is defined for State death
          //the Object removes itself from Habitat
```

```
            onEntry { Remove();}
        }

  Action Regenerate()
  {
   // checks what the current state of FSM
   // and if the Human regenerated already
   if (development == adult && hasChildren == false)
   {
      for (int i=0; i<avgNumChildren; i++)
      {
        Humanity.Add(Human);    // copies Human and adds it to Habitiat
      }
      hasChildren = true;
    }
   }
 }
}

// Main part of the program
void main()
{

  int period = 100;

  for(int year =0; year < period; year = year+1)
  {
    for (int index = 0; index < Habitat.Size(); index = index+1)
    {
      Habitat[index].age++;
      Habitat[index].updateFSM(development);
      Habitat[index].Regenerate();
    }
  }
  print("Humanity has ");
  print(Habitat.Size());
  print(" people\n");
}
```