CμLOG

An Entity Interaction Simulation Language

John Demme (jdd2127)
Nishant Shah (nrs2127)
Devesh Dedhia (ddd2121)
Cheng Cheng (cc2999)

Columbia University

September 23, 2008

1 Overview

CμLOG is a logic programming language and it resembles Prolog in many aspects. The fundamental method of finding solutions is very similar to Prolog but CμLOG's way of representing it is much more familiar to the end user. Additionally, CμLOG has provisions for entity communication. As this language is going to be used for simulating real life motions, we strongly emphasize that the program learn and forget data/rules/information at run-time. For this, similar to "assert" and "retract" of Prolog we have introduced two functions called "learn" and "forget."

In CμLOG there exist no specific data structures like you would see in Java or Python, however rules and facts can be added to the program dynamically, which allows programs to remember data in a much more natural way since the data simply becomes part of the running code.

The semantics of CμLOG are similar to Prolog, but the syntax is heavily modified to look more like C. One of the most unique features about it is entity linking and protection, which can be used to simulate entity communication. To run a useful application on CμLOG you will need to have an environment entity (defining the environment and game) and at least one agent, to provide a subject to be simulated. We link entities to the environment and the access to data in all entities is dynamic-permission-based. Data access can help in an environment when all agents have a common goal and data encapsulation can also be useful when the goals are different.

Each agent program communicates with other programs to find out more information about other agents or its own status in the environment. The simulator has all the information about each agent. This feature provides for interesting inter-program communication and at the same time protects when necessary data. Examples of this are discussed in a subsequent section.

Another feature of CμLOG is that it has a powerful library of its own and also can support third party libraries. The library supports functions like range, and move which are very useful for simulating any environment. These functions are available in a library and can also be complemented by the user to suit his needs. The default library will always be included in the program without any specific code but a third party one would have to be explicitly included in the program in the following way:

@attach$< abc.xx >$

2 Introduction to Logic Programming

Logic programming is a kind of computer programming using mathematical logic. Specifically, it is based on the the idea that of applying a theorem-prover to declarative sentences and deriving implications. Compared with procedural languages, logic programming solves the problem by setting rules with which solutions must fit. We can represent logic programming by the formula:

$$Facts + Rules = Solutions$$

Logic programming languages provide several important advantages:

- Logic programming languages are inherently "high level" languages, allowing programmers to specify problems in a declarative manner, leaving some of all of the details of solving the problem to the interpreter.
- The structures–both programming structures and data structures–in both prolog and CμLOG can be very simple- such as facts. The relationship between code and data is also of note. CμLOG uses the Von Neumann style (vs. Harvard architecture) wherein data is code. It is therefore possible (and inherently necessary) for programs to be introspective, and self-modifying. In other words, it is easier for programs to learn and adapt.

There are several different logic programming languages. Among these languages, Prolog is the most popular one, which was developed in 1972 by Alain Colmerauer. In prolog program, there are facts and rules. The program is then driven by a question. It is widely used in artificial intelligence and computational linguistics.

3 Applications

The language CμLOG we have designed will be used for communication between agents and an environment, as well as to determine behavior of said entities. Every agent program communicates with the environment program through a simulator. The simulator runs a CμLOG logic solver and interpreter which functions on a set of rules defined and modified by the environment and agents then provides solutions representing the actions to be taken by the agents.

The environment is grid based and defined by a CμLOG program. It potentially includes obstacles and a goals which the agent must reach, however the game is defined almost entirely by the environment program. Every object (i.e. agents, walls, switches, goals) in the environment is defined by grid positions. The environment specifies the representations of the entities to the simulator. The simulator re-evaluates the object rule during each turn when it renders the grid, so the contents of the grid can be dynamically defined based on the state of the simulation or the contents of the program (which can be changed by the program.) For example based on the grid position of the agent the environment might remove or insert a wall. The agent program decides the next move based on previous moves and obstacle data.

The simulation of the agent program is also turn based. Each time the agent makes a move it sends its new coordinates to the simulator. The new coordinates become part of the simulation's rules which are exposed to the environment when it is solved to render the scene.

The simulator discussed could be modified to be used for flight simulation of several agents in a grid based airspace. Alternatively, it could be used in a real environment like the movement of *pick and place* robots in a warehouse. The language could be used to define the warehouse environment and agent programs for robots.

4 Example Code

Several examples are now given. They are not complete, and only intended to give a gist of the language's syntax and semantics.

The environ1.ul program defines a 15x15 grid as well as several wall locutions. The simulator doesn't know anything about a "wall" or a "goal", but gets symbols for each grid point by solving the "object" rule. The environment interacts with the simulator primarily via the object rule. The "repr" rule is also used to tell the simulator what file should be associated with each symbol. This file could be an image (to display on the grid) or an agent program to run, starting in that grid.

The agent1.ul program is a pretty simple program which attempts to reach the "goalObject" without running into anything else. To do this, it uses very simple graph-search algorithms to find a valid path to the goalObject, or–alternatively–a grid square which it has not been to yet. This sort of searching algorithm is very simple in CμLOG due to it's logical nature. Indeed, it depends on the simple search which is used internally to solve programs. The agent also attempts to communicate it's knowledge of the environment to any other agents it encounters.

5 Purpose

The language proposed here is ideal for entity interaction and simulation for a number of important reasons:

- Generic- games are defined completely by the environment application.
- Composable- individual behaviors can be written simply and easily, then combined to obtain high-level actions and reasoning.
- Declarative- programmers can specify what they want entities to do rather than how
- Controlled Communication- data in the system is frequently made up of nearly-atomic bits of data many of which can be used both on their own and composed as complex data. This means that subsets and smaller pieces of data can be communicated between entities without loosing meaning.
- High-level libraries- due to the flexibility of the language, high-level algorithms–such as path-finding–can be easily implemented in libraries, allowing further, domain-specific intelligence to be written in the programs.

Program 1 A sample CμLOG environment programming

```
/*
  environ1.ul
  The environment being operated in is the list of the
  simulator's facts, then the facts and rules below
*/

// This is a sample 15x15 environment
size(15,15);

@attach <geometry.ul>;

// A wall segment at (5,5)
wall(5,5);

// A wall segment from (1,10) to (5,10)
wall($X,$Y) {
    0 > X >= 5;
    Y == 10;
}

// A wall that only appears when an agent is at (1,2) or (1,4)
wall(1,3) {
    {
        object(1, 2, agent1) |
        object(1, 4, agent1);
    }
}

// A wall that only appears when an agent is at (2,2) or (2,4),
//  but stays there after the agent leaves
wall(2,3) {
    {
        object(2, 2, agent1) |
        object(2, 4, agent1);
    }
    learn( wall(2,3) );
}

/* An invisible switch appears at (3,3) and dissolves the wall
   at (2,3) when the agent steps on it */
object(3, 3, switchObject) {
    object(3, 3, agent1);
    forget( wall(2,3) );
}

// There is a "wallObject" at x,y,
//  iff we have defined a wall there
object($x, $y, wallObject) {
    wall($x, $y);
}

// The objective is at (15,15)
object(15, 15, goalObject);

// These are the icons for each object
repr(wallObject, "pix/wall.png");
repr(switchObject, "pix/switch.png");
repr(goalObject, "pix/goal.png");

// Agent success if it reaches (15, 15)
finish(SuccessAgent1) {
    object(15, 15, agent1);
}

finish(SuccessAgent2) {
    object(13, 15, agent2);
}

// Fail the simulation if the agent hits a wall
finish(Failure) {
    object($x, $y, agent1);
    wall($x, $y);
}

// Load agent1
repr(agent1, "agent1.sl");

//Place at (1,1) then forget about the agent,
// so the simulator will take over agent management
object(1, 1, agent1) {
    forget(object(1, 1, agent1));
}

viewRange($x, $y, $viewer, $obj, $rangeMax) {
    object($ViewerX, $ViewerY, $viewer);
    range($x, $y, $ViewerX, $ViewerY, $range);
    0 <= $range <= $rangeMax;
    object($x, $y, $obj);
}

viewAccessRule(agent1);
access (viewAccessRule) {
    //How far can agents see?
    // This is defined in geometry.ul
    view($x, $y, $viewer, $obj) {
        viewRange($x, $y, $viewer, $obj, 1);
    }
}

repr(agent2, "agent2.ul");

comsAccess(agent1) {
    comsRange(agent1, agent2, 5);
}
access(comsAccess) {
    peers(agent1);
    peers(agent2);
}


/*
  Symbols used here to interact with the simulation:

  finish (Reason) - Is the simulation over?

  repr (symbol, filename) - What should the simulator
          use to represent this symbol?

  object(X, Y, symbol) - Is there an object identified by
          'symbol' at (X, Y)?

  view(X, Y, Viewer, Object) - Used by agents to look at
          the environment

  Library functions used:

  range(X1, Y1, X2, Y2, Range) - true if the distance between
                (X1,Y1) and (X2, Y2) is Range
*/
```

Program 2 A sample CμLOG agent

```
// agent1.ul: Sample agent program


//Sample routine to recall last move
lastCoord($X1, $Y1) {
    moveNum($N);
    myCoord($N-1, $X1, $Y1));
}

//Sample routine to remember moves
storeCoord() {
    moveNum($N);
    learn( myCoord($N, $X, $Y) );
    forget(moveNum($N));
    learn(modeNum($N + 1));
}

//Move routines resolve coordinates for a
// direction or vise-versa
move($X1, $Y1, $X2, $Y2, Up) {
    $X1 == $X2;
    $Y1 + 1 == $Y2;
}

move($X1, $Y1, $X2, $Y2, Down) {
    $X1 == $X2;
    $Y1 - 1 == $Y2;
}

move($X1, $Y1, $X2, $Y2, Left) {
    $X1 - 1== $X2;
    $Y1 == $Y2;
}

move($X1, $Y1, $X2, $Y2, Right) {
    $X1 + 1 == $X2;
    $Y1 == $Y2;
}

//Find and remember all my local peers
action($dir)
{
    {
        $env.peers($p) |
        myPeers($p);
    }
    $p != $this;
    // Tell all my peers about all the
    //  walls I know about
    memObj($Xo, $Yo, wallObject);
    $p.learn( object($Xo, $Yo, wallObject) );

    // Remember all my friends
    learn( myPeers($p) );
    false;
}

//Learn everything I can see, and store my coordinate
action($dir) {
    $env.view($ox, $oy, $this, $obj);
    learn( memObj($ox, $oy, $obj) );
    storeCoord();
    false;
}

//Solution solver base case
solution($Xp, $Yp, []) {
    memObj($Xp, $Yp, goalObject);
}

//Find a path to goal
solution($Xp, $Yp, [$Dir | $Rest]) {
    //Does $Rest get us to goal?
    solution($Xn, $Yn, $Rest);
    //Would we run into anything at the new coordinates?
    !memObj($Xn, $Yn, _);
    //If no, then resolve the direction
    move($Xp, $Yp, $Xn, $Yn, $Dir);
}

//Find a coordinate where we haven't been
explore($Xp, $Yp, []) {
    //Valid explore goal if we haven't been here before
    !myCoord($Xp, $Yp, _);
}

//Find a path to an unexplored tile
explore($Xp, $Yp, [$Dir | $Rest]) {
    //Does $Rest get us closer to a place we haven't been?
    explre($Xn, $Yn, $Rest);
    //Would we run into anything at the new coordinates?
    !memObj($Xn, $Yn, _);
    //If no, then resolve the direction
    move($Xp, $Yp, $Xn, $Yn, $Dir);
}

//If we know a solution to goal, use it
action($dir) {
    solution($X, $Y, [$dir | _]);
}

//If not, use a path to an unexplored square
action($dir) {
    explore($X, $Y, [$dir | _]);
    //What would the new coordinates be?
    move($X, $Y, $Xn, $Yn, $dir);

}
```