CμLOG Reference Manual

An Entity Interaction Simulation Language

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

Columbia University

October 21, 2008

1 Overview

CμLOG is a logic language designed for entity interaction simulation. It uses a syntax similar to C, making it easier on the typical programmer's eyes, and already integrates with some code tools, such as code indenters. One uses the language to provide a set of facts and rules, and the "program" is run by asking a question, which the interpreter attempts to answer using inference based on the fact and rule set. CμLOG is designed for simulation, so typically a simulator will ask a given agent program what it's next action will be. The agent program then uses CμLOG's entity interaction features to gather information about it's environment and decide what to do.

2 Goals

The language presented here attempts to fulfill the following requirements:

- 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.

3 Lexical

```
[' ' '\t' '\r' '\n'] WS
"/*"     OPENCOMMENT
"*/"     CLOSECOMMENT
"//"     COMMENT
'('      LPAREN
')'      RPAREN
'{'      LBRACE
'}'      RBRACE
';'      SEMICOLON
','      COMMA
'+'      PLUS
'-'      MINUS
'*'      TIMES
'/'      DIVIDE
"=="     EQ
"!="     NEQ
'<'      LT
"<="     LEQ
">"      GT
">="     GEQ
'@'      AT
'.' DOT
'['      ARROPEN
']'      ARRCLOSE
'"'      QUOTE
'$'['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* Variable
['0'-'9']+ Number
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* Identifier
```

4 Facts

Facts define factual relationships. They have a very similar syntax to rules, except they have no code block to make them conditionally true. Any query which matches a fact is simply true. Another way to think of facts is as terminal nodes in the solution search.

Each fact is composed of a name, and a comma separated list of parameters, each of which may be a constant, or a variable. Using any variable except the anonymous variable doesn't make much sense in a fact, but is allowable.

```
Example:
  foo(4, symA);  //Foo of 4 and symA is always true
  foo(4, symA, ?); //Foo of 4, symA, and anything (wildcard) is always true
  wall(4, 5); //In an environment might mean: there is a wall present at (4,5)

Grammar:
  Fact -> Identifier ( ParamList );
  ParamList -> Param | ParamList , Param
  Param -> Variable | Number | String | Array
```

5 Rules

Rules define relationships which are conditionally true. They are similar to facts, but instead of ending with a semicolon, they contain have a block, which defines the conditions upon which the rule should be evaluated as true. Another way to think of a rules is as a node in the solution search which may branch, or be a leaf, depending on the contents of the condition block. Each rule is composed of a name, a comma separated list of parameters, and a block.

```
Example:
  foo(4) { bar(5); }  //Foo of 4 is true if bar(5) is true
  foo(4) { bar(6); }  //Foo of 4 is true if bar(6) is true
The two above rules are together equivalent to:
  foo(4) {OR: bar(5); bar(6); }

Grammar:
  Fact -> Identifier ( ParamList ) Block
```

6 Variables

Variables represent a value to be solved for. During rule matching, they will match any value or type, but can be constrained in an associated block. All variables are scoped to the rule, so that variable solutions can be shared between subblocks. Variables are represented by a dollar sign ($) then the variable name. The name must start with a letter, and is composed of letters, numbers, and underscores. There is a special variable called the anonymous variable which is represented simply by a question mark (?). It cannot be referenced in the block, and simply matches anything.

```
Example:
  foo($X, $y, $foo_bar, $bar9, ?) { }

Grammar:
  Variable -> $[a-zA-Z][a-zA-Z0-9_]* | ?
```

7 Blocks

Blocks contain a list of statements (conditions) to determine truth, and specify a reduction method for those statements. Each block will reduce all of its statements using the same reduction method (usually AND or OR), but may contain sub-blocks. If the reduction method is omitted, AND is assumed. The syntax allows for other reduction methods to be allowed (such as xor, or a user-specified method), however the language does not yet support this.

```
Examples:
 {
    foo();
    bar();
 }
 //True if foo and bar are both true.

 {AND:
    foo();
    bar();
 }
 //True if foo and bar are both true.

 {OR:
    foo();
    bar();
 }
 //True if foo or bar are true.

Grammar:
  Block -> { (Identifer:)? StatementList }
  StatementList -> Statement | StatementList Statement
```

8 Statements

Statements are boolean qualifiers which are used inside of blocks. They can be any one of three types: comparisons, evaluations, or blocks. Comparisons are used to constrain variables. N-ary comparisons are supported. Only values of the same type can be compared, and certain comparisons only work on certain types, so comparisons can be used to constrain variables by type. Evals are used to query the program, and have a similar syntax as facts. They can be thought of as a branch in the solution search. Blocks are considered a statement to support sub-blocks. They are evaluated and the reduced result is used. Comparisons and evals are both terminated by semicolons.

```
Examples:
  1 < $X <= $Y < 10;  // A comparison
  range($X, $Y, 7);   // An eval
  !range($X, $Y, 7);   // This must not evaluate to true
  {OR: $X > 10; $X < 0; }  //A sub-block with two binary comparisons

Grammar:
  Statement -> Block | Eval ; | Comparison ;
  Eval -> (!)? Identifier ( ExprList );
  ExprList -> Expression | ExprList , Expression
  Comparison -> Expression ComparisonOp Expression | Expression ComparisonOp Comparison
  ComparisonOp -> EQ | NEQ | LT | LEQ | GT | GEQ
```

9 Expressions

Expressions are used to modify values being passed into comparisons or evals. They are used to modify integers, and supports plus, minus, times, and divide. Typical infix notation and precedence rules are used, and expressions can be grouped with parenthesis.

```
Examples:
  $r - 10 < $X < $r + 10;  // A comparison: $r - 10 and $r + 10 are the expressions
  range($X, $Y, $r / 3);   // An eval- $r / 3 is the expression here

Grammar:
```

```
  Expression -> Number | String | Variable | Expression Op Expression | ( Expression )
  Op -> PLUS | MINUS | TIMES | DIVIDE
```

10 Types

The following types are supported: integers, strings, arrays, symbols, and entities. Strings in CμLOG are currently atomic, so no string processing such as splitting, joining, or searching is supported. They are primarily used for interaction with the rest of the system (printing, specifying files, ect.). Arrays are discussed in detail in the next section. Symbols are simply identifiers. They share the same namespace as rule and fact names, and can only be compared with equals and not equals. Entities are used to represent other programs (typically agents) and are used for interaction. In addition to equals and not equals comparison operators, they support the dot operator for interaction (discussed later.)

11 Arrays

Arrays in CμLOG behave similarly to functional lists, and are matched similiarly as well. They are delimited by [and], and elements are comma separated. The null list (also the tail of the list) is denoted by []. Any number of elements of the array are matched inside of fact and rule declarations. Arrays only appear as fact and rule parameters. The last element listed in [] is an array, and represents the tail of the list. If the elements listed before the last element comprise the entire list, the last element will be simply [].

```
Examples:
  //A set of rules which ensures that all elements are less than $v
  lessThan($v, [$head, $tail]) { $head < $v; lessThan($v, $tail); }
  lessThan(?, []);

  //Matches the list (1, 4, 6)
  foo([1, 4, 6, []]);

  //Invalid
  foo([1, 4, 6]);

  //Prepends $v to array $a
  prepend($v, $a, [$v, $a]);

  //Constructs a list
  list($a, $b, [$a, $b, []]);

Grammar:
  Array -> [ ArrayElems ]
  ArrayElems -> ArrayElem | ArrayElem ArrayElems
  ArrayElem -> Number | String | Variable | []
```

12 Directives

CμLOG supports a special syntax for interpreter directives. This allows programs to interact with the interpreter while avoiding symbol collisions. The syntax is similar to that of a fact's, but an at sign (@) is prepended. Four directives are currently planned: attach, print, learn, and forget. Attach is used to include code from another CμLOG file. Print is used to output strings, and results of searches during runtime. Learn and forget are discussed in the next section.

```
Examples:
  @attach("geometry.ul");
  @print("Hello, world!");
```

```
Grammar:
  Directive -> @ Identifier ( ParamList );
```

13 Program Modification

The two directives learn and forget are used to modify a program at runtime. This is the only way in which CμLOG supports non-volatile storage. Learn is used to add a fact to a program, and forget is used to remove a fact. The synatax for these two directives is special, consisting of the usual directive syntax, except contained inside the parenthesis is a fact definition. Any non-anonymous variables in this fact definition are filled in with solutions found for those variables, and the learn or forget is "executed" once for each solution. They are similar to Prolog's assert and retract.

```
Examples:
  @learn( wall(4,5); );  //Remember that there is a wall at (4,5)
  @forget( agent(8, 10); ); //Forget about the agent at (8, 10)

Grammar:
  Directive -> @ (learn|forget) ( Fact );
```

14 Interaction- The Dot Operator

If a variable or symbol represents another program (entity), then it supports the dot operator. After appending a dot (.) to the reference, one can put an eval, a learn, or a forget, and that action will take place in the other entity's namespace. This can be used to ask for information from another program (such as the environment program or another agent) or to modify the other program–perhaps to teach another agent, to trick a competitor, or to change the operating environment. Future versions of CμLOG could likely support some sort of access rules in the destination program, allowing it to control who is allowed to access what data, and who is allowed to change its program, and how. These access rules could potentially modify any queries or changes, perhaps revealing an entirely fake namespace to the other agent. Such access rules are beyond the scope of CμLOG initially, however.

```
Example:
  $agent.@learn( wall(4,5); );  //Tell agent2 that there is a wall at (4,5)
  env.view($X, $Y, $obj); //Query the environment, find out what is at ($X, $Y)

Grammar:
  DotOp -> Directive | Statement
  Dot -> Variable . DotOp | Identifier . DotOp
```

15 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.

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) {OR:
    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) {
    {OR:
        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);
//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");

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)
{
    {OR:
        $env.peers($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);

}
```