The oxoJoy1994 model

The objective of this model is to explore the observables that are involved in playing / learning to play noughts-and-crosses, and to show how they can be seen as particular instances of observables that are quite generally used in "OXO-like" games. The oxoGardner1999 model adds a visualisation that works for noughts-and-crosses, but does not apply to other OXO-like generalisations.

Mike Joy has provided a wrapper for the three different games that have been modelled, viz, noughts and crosses on 3-by-3 and 4-by-4-by-4 grids, and variant that is played on the 7 point projective plane (whose points are the non-zero boolean triples, and lines consist of triples of points whose boolean sum is the zero vector). The wrapper is in the file game.e:

```
%eden
/* game.e */
writeln("1: oxo, 2: oxo4 (3D), 3: pp7 - please press a number and then return");
s = gets();
switch(s) {
  case "1": include("geomoxo.e"); include("oxo.geom"); break;
  case "2": include("geomoxo4.e"); include("oxo4.geom"); break;
  case "3": include("geompp7.e"); include("pp7.geom"); break;
}
include("display.e");
include("initsq.e");
include("status.e");
include("sqvals.e");
include("play.e");
include("gamestate.e");
include("control.e");
```

What the wrapper indicates is that all three games use the same observational principles as applied to different geometries.

Here is gemooxo.e:

```
allsquares is [s1,s2,s3,s4,s5,s6,s7,s8,s9];
nofsquares is allsquares#;
lin1 is [s1,s2,s3];
lin2 is [s4,s5,s6];
lin3 is [s7,s8,s9];
lin4 is [s1,s4,s7];
lin5 is [s2,s5,s8];
lin6 is [s3,s6,s9];
lin7 is [s1,s5,s9];
lin8 is [s3,s5,s7];
alllines is [lin1,lin2,lin3,lin4,lin5,lin6,lin7,lin8];
noflines is alllines#;
linesthru1 is [lin1,lin4,lin7];
linesthru2 is [lin1,lin5];
linesthru3 is [lin1,lin6,lin8];
linesthru4 is [lin2,lin4];
linesthru5 is [lin2,lin5,lin7,lin8];
linesthru6 is [lin2,lin6];
linesthru7 is [lin3,lin4,lin8];
linesthru8 is [lin3,lin5];
linesthru9 is [lin3,lin6,lin7];
linesthru is [linesthru1,linesthru2,linesthru3,linesthru4,linesthru5,linesthru6,
linesthru7,linesthru8,linesthru9];
```

... and this is oxo.geom:

```
/* oxo.geom */

/* Specify the geometry by (1) number of squares (numbered s1..sn)
** and (2) by the lines initially as [[int]].
** init_geom() then generates the names and lists of names.
*/

n_squares is 9;
list_lines = [[1,2,3],[4,5,6],[7,8,9],[1,4,7],[2,5,8],[3,6,9],[1,5,9],[3,5,7]];

/* Templates for the display routines
*/

board_template = "123\n456\n789\n";
print_template = ["","","","\n","","","\n","","","\n"];
```

All display routines use the same conventions for when a square is occupied with an x, o or is blank ("undefined"). The board_template is used to determine a string of Os, Xs and spaces that determines the current status of the board.

```
/* display.e */

x = 1; o = -1; u = 0;

/* DISPLAYING ROUTINES ****************************************/

func vis{
        return ($1==u)?".":(($1==x)?"X":(($1==o)?"O":"?"));
        }

func concatxo {
        auto i;
        auto result;
        result = print_template[1];
        for (i=1; i<=$1#; i++) {
                result = result // vis($1[i]) // print_template[i+1];
                }
        return result;
        }

board is concatxo(allsquares);

proc displayboard : board {
        writeln(board);
        }
```

The initsq file - which initialises all the squares to blank. Note that all the observation from this point on is generic in character.

```
/* The original model just had this set of definitions, which
   initialises the normal OXO game.  If s10 ... s64 are not
   initialised in this manner, then the OXO4 game doesn't immediately
   start when you load game.e, and you need type to invoke
   init_game();

   s1=u; s2=u; s3=u; s4=u; s5=u; s6=u; s7=u; s8=u; s9=u;

   We're replacing these definitions with the following procedural
   statements which initialise s1 ... sn symbols.

*/
```

```
proc initAllSquares {
  auto i;

  autocalc = 0;
  for (i = 1; i <= nofsquares; i++) {
    `"s"//str(i)` = u;
  }
  autocalc = 1;
}

initAllSquares();

/* Now we`re in a state where the board exists but no game is in
   progress.  We could have left the initialisation of the board until
   the first call to init_game but this closes down the possibility of
   playing with the model before all the scripts have been loaded the
   game has started.  (Cf Gardner`s version of this model.) */
```

The status file, which introduces the dependencies that determine whether either player has won, and whether the game is drawn. Notice that there is no requirement that the position whose status is being monitored is legal. At this stage, the model captures the interpretation of a bystander who can recognise the criteria by which an OXO position is won, lost or drawn.

```
/* status.e */

func xxline {
        para lline;
        auto i, b;
        b = 1;
        for (i=1; i<=lline#; i++) {
                b = b && (lline[i] == x);
                }
        return b;
        }

func ooline {
        para lline;
        auto i, b;
        b = 1;
        for (i=1; i<=lline#; i++) {
                b = b && (lline[i] == o);
                }
        return b;
        }

func checkxwon {
        auto i, b;
        b = 0;
        for (i=1; i<=$1#; i++) {
                b = b || xxline($1[i]);
                }
        return b;
        }

func checkowon {
        auto i, b;
        b = 0;
        for (i=1; i<=$1#; i++) {
                b = b || ooline($1[i]);
                }
        return b;
        }
```

```
xwon is checkxwon(alllines);

owon is checkowon(alllines);

func nofpieces {
        auto count, total;
        total = 0;
        for (count=1; count <= $1#; count++)
                if ($1[count] == $2)
                        total++;
        return total;
        }

nofx is nofpieces(allsquares, x);

nofo is nofpieces(allsquares, o);

full is (nofx + nofo == nofsquares);

draw is (! xwon) && (! owon) && full;

status is (xwon?"X wins ":"") // (owon?"O wins ":"") //
        (draw?"Draw ":"") // "";

proc xwinmess : status {
        writeln(status);
        }
```

These observables encode the kinds of dependencies that reflect a rather naive player's perspective on playing an OXO-like game. The key idea is that there is a value attached to each square, and that this depends on how the lines passing through that square are occupied. Notice how the context for observation changes here: although all squares on the board are being observed, one in particular (square) is being singled out as an exemplar. In order to determine the square of highest value, the value associated with all vacant squares has to be considered - this resembles scanning all the squares the board.

```
/* sqvals.e */

badline = -999;

player = o;
square = 5;

func linval {
        para lline;
        auto xs, os, i;
        xs = 0; os = 0;
        for (i=1; i <=lline#; i++) {
                if (lline[i] == x)
                        xs++;
                else if (lline[i] == o)
                        os++;
                }
        return [xs, os, lline#-xs-os ];
        }

func weighting {
        auto xs, os, blanks, plr;
        auto players, opponents;
        xs = $1[1];
        os = $1[2];
        blanks = $1[3];
        players = (plr == x)?xs:os;
        opponents = (plr == x)?os:xs;
        if (players > 0 && opponents > 0)
```

```
                return (-1);
        if ((opponents == 0) && (blanks == 1))
                return 100;
        else if ((players == 0) && (blanks == 1))
                return 20;
        else if (opponents == 0)
                return 3;
        else if (players == 0)
                return 1;
        else return 0;
        }

func sqval {
        para linelist;
        auto listlen,i,total;
        listlen = linelist#;
        total = 0;

        for (i=1; i <= listlen; i++)
                total += weighting(linval(linelist[i]), player);
        return total;
        }

availsquare is allsquares[square] == u;

cursqval is sqval(linesthru[square]);
```

In play.e the board is scanned by allowing "square" (as in sqvals.e) to range over the available squares. The function findmaxindex then determines the value for 'square' that (with this simple static evaluation mechanism in place) is the 'best' move.

```
/* play.e */

maxindex is findmaxindex(allsquares);

func findmaxindex {
        auto maxval,foundindex,looping;
        maxval=0;
        foundindex=0;
        square=1;
        looping=1;
        while (looping) {
                if (availsquare && (cursqval >= maxval)) {
                        maxval=cursqval;
                        foundindex = square;
                        }
                if (square < nofsquares)
                        square++;
                else if (square == nofsquares)
                        looping = 0;
                }
        return foundindex;
        }
```

To this point, there has been no consideration of who is to play in a position. Note that the way in which the observation "whose turn is it to play?" has implications for what happens when any non-standard activity occurs - as for instance happens if one of the player agents cheats - e.g. by playing two turns in a row, or removing an opponent's piece.

```
/* gamestate.e */
```

```
startplayer = o;

x_to_play is (! end_of_game) && (startplayer == x && nofo == nofx) ||
                nofo > nofx;

o_to_play is (! end_of_game) && (startplayer == o && nofo == nofx) ||
                nofx > nofo;

end_of_game is xwon || owon || draw;
```

Once the observational model of the OXO playing environment has been established, getting an agent - whether human or computer - to play the game is very straightforward. The messages displayed by the procedural code can be modified by overwriting the procedure play().

```
/* control.e */

proc init_game {
        initAllSquares();
        writeln("Who is to start? Respond with either");
        writeln("startplayer = o;");
        writeln("    or");
        writeln("startplayer = x;");
        }

proc play : end_of_game, o_to_play, x_to_play {
        if (end_of_game) {
                writeln("bye bye");
                }
        else if (o_to_play) {
                writeln("Positions are");
                writeln(board_template);
                writeln("please play; type");
                writeln("sn=o;");
                }
        else if (x_to_play) {
                mvsq = "s" // str(maxindex) ;
                `mvsq` = x;
                writeln("I move to square ", mvsq);
                }
        }
```

Note that the options to redefine observables, such as the winning lines, the criteria for winning, the way in which whose-turn-it-is is observed are all open throughout the modelling and playing of each OXO-like game. It is also possible to interfere with the board whilst the game is in progress, by entering new values for squares through the input window, or introducing a dependency so that entering a O in one square changes the value of another square etc.

One motivation for making the OXO model was to compare 'definitive programming' with functional programming, as in a language such as SML or Miranda. The relative merits of "modelling with Eden" and "programming with SML" are an interesting focus for reflection. Why does it seem so inappropriate to teach first-year computer science students to program in Eden?

The oxoHarfield2006 extension

Antony Harfield's extension of the OXO model is here combined with some minor improvements to the textual interaction with the player. To run the model with versions of tkeden up to 1.67, you need to first load the gel interpreter, then include the file "game.e" from the oxoJoy1994 directory, and finally enter the

code below, which is the content of the file "oxoAnt.gel":

```
%eden

## an improvement to the board template for 4-by-4-by-4 OXO proposed by WMB

if (s=="2") {
board_template="
 1 2 3 4  5 6 7 8  9 . . .  . . . 16
17 . . .  . . . .  . . . .  . . . 32
33 . . .  . . . .  . . . .  . . . 48
49 . . .  . . . .  . . . .  . . . 64";
};

%angel

boardwin = window { content = [boardlabel] };

boardlabel = label { text = "" };

%eden

writebox="";

proc play: end_of_game, o_to_play, x_to_play {
        if (end_of_game) {
                writeln("bye bye");
                writebox = "bye bye";
                }
        else if (o_to_play) {
                writeln();
                writeln(board_template);
                writeln("please play; type");
                writeln("sn=o;");
                writebox = "The indices of squares are\n" // board_template // "\n\nTo pl
                }
        else if (x_to_play) {
                mvsq = "s" // str(maxindex) ;
                `mvsq` = x;
                writeln("I move to square ", mvsq);
                writebox = "I move to square "//str(mvsq);
                }
        }

boardlabel_text is board // "\n" // writebox;

%angel

boardlabel.font = "Courier 32 bold";
boardlabel.background = "white";
boardlabel.fill = "both";
boardlabel.expand = 1;
```

The file 'game.e' will not execute correctly with tkeden-1.73, since the 'reading from input' feature is no longer available. To get around this, you can instead comment out the line

```
s = gets();
```

from game.e, and create a run file in which the revised version of game.e is loaded after an assignment of the form

```
s = "1";
```

etc has first been made.
