CS405 Introduction to Empirical Modelling 2008

Labsheet 3: The OXO laboratory

Using definitive notations to represent state

Observation and action associated with a definitive script

We have proposed an approach to "modelling state-as-experienced" that is based on incrementally constructing a definitive script whilst in parallel building up familiar patterns of interaction with the script that reflect relevant observables, dependency and agency. In this laboratory, we shall illustrate how these principles can be applied. This will be done by building a definitive script that reflects the rich observation involved in playing even such a simple game as noughts-and-crosses ("OXO"). In the process, we shall create an interactive environment within which to experiment with a range of generalised "OXO-like" games. (For more background, see paper 033 from the EM publications list.)

The OXO laboratory

This lab revisits what can be viewed as a laboratory for the experimental development of OXO-like games. The first stage of the labwork is to get basic familiarity with this laboratory. To do this, download `oxoJoy1994` from the EM archive at http:/empublic.dcs.warwick.ac.uk/projects/ and open the file `game.e` using the `File` menu in the EDEN input window. You will notice that the interface to the model is purely textual - the `game.e` file itself illustrates a mechanism that can be used for reading keyboard input, and the output state is represented by character output to the feedback window. To familiarise yourself with key observables in the `oxoJoy1994`, you should first play through a normal game of standard noughts-and-crosses. In order to do this, you should type '1' into the feedback window to select traditional noughts-and-crosses, then place a "nought" in one of the 9 squares of the 3-by-3 grid. For instance, to place a "nought" in the square in the second row and third column, you should input the definition:

```
s6=o;
```

Note that in your interaction with the model, you play Os (by making definitions of the above form) and the machine plays Xs (by making definition such as:

```
s5=x;
```

automatically) whenever it observes that the player whose turn it is is X. When a game is completed, you can start another by typing:

```
init_game();
```

and have the option of choosing whether you or the machine has the first turn.

You should find that if you follow the appropriate protocol and are a well-behaved player, each game will follow the usual pattern. On the other hand, a little experiment will show that (as in the case of the `roomYung1989`), there are unusual possibilities for interaction. For instance, when a game is finished, you can try overwriting an X with an O or a blank by redefining the value of that square to `o` or `u` respectively. You can also use the same technique to overwrite an X with a O during play - possibly even entering two Os at the same time via the input window. When you "cheat" in this way, you may find that the machine will also "cheat". For instance, a possible game proceeds as follows:

```
s5 = x;                          ## the computer plays X in the centre square
s1 = o; s5 =o;                   ## I play O in square 1, and overwrite the X
s7 = x; s8 = x; s9 = x;          ## the computer enters three Xs and so 'wins'!
```

These apparently nonsensical patterns of interaction simply reflect the way in which modes of observation suitable for playing a game of standard noughts-and-crosses (and indeed other variants, such as 3D noughts-and-crosses) work out when the game situation is perturbed. For instance, you can figure out why the computer enters three Xs in the above scenario, by examining the observable `x_to_play`, which defines the criterion by which X determines

that it is appropriate to make a move. A simpler criterion, such as "O played last" would lead to exactly the same behaviour in normal play, but have different consequences when cheating occurs.

This lab has two ingredients. The first involves developing a better graphical interface to the model. The second involves retracing some of the steps in the construction of `oxoJoy1994` and adapting the model of standard noughts-and-crosses experimentally to explore different possible rules and scenarios.

Exercise 1: Making an graphical interface for an OXO-like game

The key component in making an interface is a model of a single grid square that can be in one of three states - containing an X, containing a O or blank. Within `oxoJoy1994`, the squares are associated with nine observables `s1`, `s2`, `s3`, ..., each of which is assigned a value `x`, `o` or `u` to reflect its content.

- Make a `scout` window `sq1` of type `DONALD`. Bearing in mind the intended role for this window, you should set up observables to specify the location and dimensions of the window, and to determine its foreground and background colours. You should also make your window sensitive and supply a border. It is also useful to introduce an observable to define the size of the window.

- Create `donald` viewports NOUGHT and CROSS and associate with these line drawings of a nought and a cross resepctively on a standard canvas with minimum and maximum coordinates {0,0} and {1000,1000}. Create a viewport BLANK. Define the content of the window `sq1` so that it displays the appropriate line drawing according to the current value of `s1`.

- For a standard noughts and crosses grid, you need to create nine windows on essentially the same pattern. If you want to do this for yourself, there are several approaches you can use, all somewhat tedious to implement:

    - You can do this by brute force, copying and editing to create the window definitions using a text editor. This is the simplest way to proceed initially.

    - You can write a EDEN definition to generate the text of the required set of window definitions. Once you have such a piece of text,there is an `eden` procedure called `execute()` that will interpret this string. (If you wish to use this approach, it is a good idea to first test the `execute()` on a simpler task, and to debug your efforts by substituting `writeln()` for `execute()` so that you can see explicitly what you are trying to interpret. Your string will need to include `%scout` invocations, newlines (specified by `\n`) and semi-colons in the appropriate places.)

    - A more elegant way of achieving the result you require is to write a definition that defines the `scout` for the `k`-th window, as a string whose value depends on `k`. The parameters that have to be dependent on `k` are those that determine the location of the window. You can get to grips with this issue by creating a mini-script in which there are observables `k`, `r`, `c`, `p` and `q` where `r` and `c` represent the row and column indices of the `k`-th window in the OXO grid, and `p` and `q` are `scout` points specifying the top-left and botton-right corners of the window on the display. Having framed a parametric definition for the strings to create the windows, you can then use `execute()` in a simple `for`-loop to create the entire grid.

  A slightly different way of defining the grid is illustrated in the file `oxoboard.s`, which can be used to achieve the required result if any of the above methods proves too time-consuming.

Exercise 2: Modelling state-as-experienced in playing an OXO-like game

You can understand the core of the oxoJoy1994 model best by examining the dependencies amongst the observables that define the geometry and contents of the grid together with the rules that establish the status of the game (in particular, whose turn it is to play currently, whether the game is finished, who has won or whether it's drawn). These observables are recorded in the three files `geomoxo.e`, `gamestate.e` and `status.e` within oxoJoy1994. You can examine the dependencies in these three files visually using the Dependency Modelling Tool - to aid your understanding, it is helpful to first restart tkeden and load all three files. To initialise the values of the

squares, you should also introduce the following definitions:

```
u = 0; x = 1; o = -1;
s1 = s2 = s3 = s4 = s5 = s6 = s7 = s8 = s9 = u;
```

The Dependency Modelling Tool (DMT) can be downloaded from the directory dmtWong2003 in the EM archive. It is a Java application that can parse `eden` scripts and display graphs that depict dependencies between observables. When you run the DMT v0.72, it brings up a window with buttons for drop down menus at the top.

- Step 1: Select "Load..." from the Model menu, and Open the file linesoxo3.dmt. This will display all the observables in the file `geomoxo.e` with the exception of `allsquares`, `alllines`, `nofsquares` and `noflines`. Notice that you can relocate nodes in the graph using the left mouse button.

- Step 2: You can introduce the definitions of the four missing observables by selecting the Script menu and introducing the definitions in the file geomoxo.e either by directly importing the entire file, or by typing the definitions of `allsquares`, `alllines`, `nofsquares` and `noflines` into the Script "Input window". The effect will not be pretty! You can improve the result by double-clicking on the observables `allsquares` and `alllines` with the left mouse and choosing to display these nodes using the Show as Abstraction option. (You will first need to find these observables and move them to appropriate locations.)

- Step 3: You can choose the New option from the Model menu, then use the Script "Input window" to read in the files `gamestate.e` and `status.e` in turn. Note that you can eliminate isolated nodes by double-clicking on them with the left mouse button and selecting Delete.

At each step, it will be helpful to consult the definitions of observables using EDEN, so that you get a clearer understanding of the precise dependencies involved.

Exercise 3: Comprehension exercises using the DMT

For this exercise, you load geomoxogamestatestatus.dmt into the DMT. This is a preprocessed DMT model that reflects the contents of all three basic files mentioned above.

1. Examine the way in which `xwon` is defined using the function `checkxwon()`. Deduce from the definition of `checkxwon()` that `xwon` is true if for some line, all three entries in that line have the value `x`. Express this condition by introducing new observables `allxlin1`, `allxlin2`, ... , `allxlin9` and defining `xwon` in terms of these. How does the introduction of such definitions affect the dependency graph? In what respect is this approach to defining `xwon` limited?

2. The observables and dependencies considered so far are sufficient to express basic observation of the grid situation in the course of playing nought-and-crosses. Note that, where this basic observation of the current status of the game is concerned, the family of observables `linesthru` is superfluous. If the current player wishes to place a nought or cross on the grid, they have to consider the merits of choosing a particular square. It is at this point that the observables `linesthru` become significant, since the content of the lines through the chosen square determines the value to be assigned to it. Introduce the file sqvals.e through the Script Input Window of the DMT, first stripping out the bodies of the functions in the file (to eliminate local variables). Observe how it introduces a dependency for the observable `cursqval` for the chosen square `square`. What significant dependency is missing in the definition of `cursqval`? How is this dependency implicitly taken into account in the function `sqval()`? What parameter should be given to the function `sqval` so as to address this problem? [Hint: Examine the observables in sqvals.e by viewing this file alone (with local variables eliminated, as before) using the DMT.]

3. In principle, it should be possible to redefine the squares making up the lines in the game, so that for instance the first line, instead of consisting of the top row of the grid, consists of the square on the top right, the square on the top left and the middle square of the grid. If you introduce this redefinition, what effect does this have on the dependency graph, and why does this disclose a limitation of the model? [Hint: The file geomoxo.e from oxoGardner1999 overcomes this problem.]

Exercise 4: Dicey Noughts and Crosses

Building on the foundation of the three basic files discussed above, and taking into account what you have learned about the file `sqvals.e`, construct a new variant of noughts-and-crosses that is played according to the following rules:

- Prior to each turn, the current player (player O or player X) throws a dice to obtain a number in the range 1 to 6.
- If the number k thrown is between 2 and 6, and the current player is O (respectively X) then O (resp. X) enters a nought (resp. cross) into one of the squares with row and column indices r and c where r+c=k, provided that at least one of these squares is vacant. If there is no vacant square satisfying this criterion, the current player removes any one of his opponent's entries from the grid.
- If the number thrown is a 1, the players swap roles, so that player O becomes player X and vice versa. This means that the current player in effect has a second turn, but is now aiming to enter a different symbol into the grid.
- The winner is the first player to complete a line. The game does not terminate until one or other player has won.

Your first objective should be to make a model of Dicey Noughts and Crosses in which illegal moves are detected and reported. You may then wish to consider how to introduce an automatic player.

Questions

1. By studying the attached dependency graph showing the basic dependencies amongst observables in the oxoJoy1994 model, and building on your knowledge of the model, explain:

- the significance of each of the dependency relations associated with a directed edge terminating at the node labelled `x_to_play`.
- the possible sequences of updates that might occur when entering an o into a square leads immediately to the end of the game.
- why the location of the node `nooflines` represents a conceptual error in the graph layout.
- the semantic distinction between the observables associated with the nodes `winmess`, `noofpieces` and `startplayer`, and how this is reflected in the EDEN model.

2. The game of 15 is a two person game in which players take turns to select a digit in the range 1 to 9. The game is won by the first player who holds 3 digits that sum to 15; it is drawn if neither player achieves this goal. There is a well-known correspondence between playing the game of 15 and playing noughts-and-crosses. This hinges on the fact that the digits 1 to 9 can be arranged in a 3 by 3 grid in such a way that the digits on each horizontal, vertical and diagonal line of three squares sum to 15.

Describe how you use definitive scripts in EDEN to set up:

- a model of the state of play in a game of 15;
- a viewer that enables a player to interpret this state as a state of play in a game of nought-and-crosses.

How might LSD accounts of player agents be used to reflect the distinction between playing 15 with and without the viewer in place?
