Playing games with observation, dependency and agency in a new environment for making construals

Meurig Beynon, Russell Boyatt, Jonathan Foss,
Chris Hall, Elizabeth Hudnott, Nick Pope, Steve Russ
University of Warwick
Coventry, UK
wmb@dcs.warwick.ac.uk

Ilkka Jormanainen, Tapani Toivonen
University of Eastern Finland
Joensuu, Finland
ilkka.jormanainen@cs.uef.fi

Hamish Macleod
University of Edinburgh
Edinburgh, UK
h.a.macleod@ed.ac.uk

Piet Kommers
Helix-5
Twente, The Netherlands
pkommers@gmail.com

Dimitris Alimisis, Rene Alimisi, Emmanouil Zoulias
Edumotiva
Sparti, Greece
info@edumotiva.eu

Peter Tomcsanyi, Michal Winczer
Comenius University
Bratislava, Slovakia
tomcsanyi@slovanet.sk

***Abstract*—Making construals is a new digital skill that complements conventional programming. Its primary focus is on using computer-related technology to stage interactive experience of unprecedented richness and subtlety. This paper is a tutorial on the latest version of an instrument for making construals developed in the ongoing EU Erasmus+ CONSTRUIT! project. Its principal theme is the re-creation of "the OXO laboratory" – an interactive environment in which variants of the game of noughts-and-crosses can be freely designed and evaluated.**

***Keywords*—construal; computing; spreadsheets; educational technology; school education; constructionism; open educational resources; educational games; software development**

I. Introduction

The concept of 'making construals' as a new digital skill was introduced in a tutorial paper [3] presented at iTAG in 2015. As explained in [3], making a construal differs from writing a program. Rather than specifying a recipe to achieve certain functional goals (a sequence of instructions, explicitly or implicitly specified), it establishes an open-ended environment for interaction (in the form of a family of definitions, or script) within which the human and automated agency in a domain can be expressed. Within this environment, program-like behaviours can be crafted by the maker and enacted by the computer. The traditional roles of the human agents, whether users, learners, players, designers, teachers or developers can be integrated and unified in this way. This has significant implications for learning and for designing computer games.

Making construals is the central theme of the ongoing Erasmus+ CONSTRUIT! project [2]. A key objective for CONSTRUIT! is to develop an open online course for making construals that is accessible to a wider audience that includes school teachers and students and practitioner communities. Developing an appropriate online environment for making construals ("the MCE") is a major component.

The principles and resources for making construals being developed in CONSTRUIT! have been distilled from previous work by computer science staff and students at the University of Warwick in the Empirical Modelling project [1]. The early prototypes for the MCE were based on the EDEN interpreter (cf. Figure 1 below) that initially took the form of a desktop application. Two online variants of EDEN were subsequently introduced – Web EDEN [12] and JS-Eden [11]. The current MCE is a radically revised version of JS-Eden based on feedback from workshops organised by CONSTRUIT! in Finland, Greece and the UK (including a workshop at iTAG in 2015). Introducing this latest version of JS-Eden is the principal focus of our contributions to iTAG 2016.

This paper takes the form of a tutorial on the revised version of the MCE [13] that has been developed by Nicolas Pope with the support of Elizabeth Hudnott and Jonathan Foss. New features to be introduced in this tutorial include:

- a hand-crafted parser that (unlike all previous parsers deployed in making construals, which have been built using standard parser-generating tools) gives much more precise and directed feedback about syntax errors, and also supports a form of "live edit" whereby the effect of changing scalar values is immediately visible.
- a project manager for storing scripts online in such a way that they can be made private or public and recorded in all their intermediate versions.
- a **with** construct that makes it possible to generate a script by deriving many variants of a script fragment from a single instance – a technique that resembles, but differs from, prototype-based object-orientation.

- a **when** construct that enables agent interactions appropriately expressed using scripts of definitions to be conveniently animated.

These features of the MCE will be illustrated with reference to variants of the game of noughts-and-crosses for which we adopt the generic term ‘OXO-like games’. Such a game is based on two people taken turns to place a O or X on the squares of a grid so as to create a target pattern (some form of ‘winning line’).

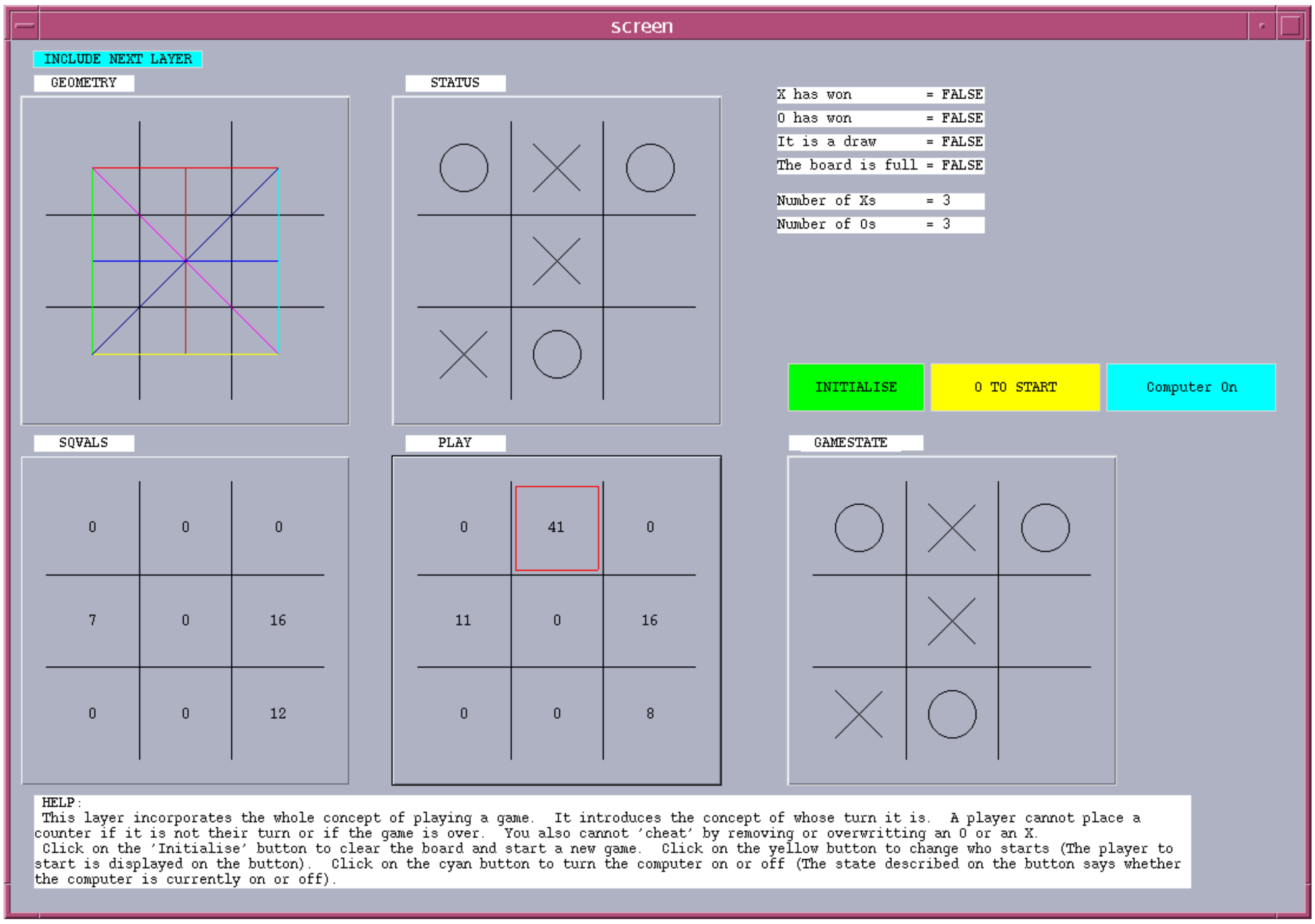
The paper has four main sections. The first section revisits the ‘OXO laboratory’ as originally conceived and implemented (cf. [5,6]). Sections II and III introduce the MCE by illustrating how the OXO laboratory can be reconstructed within it. Section IV draws some brief conclusions.

II. CONSTRUALS OF OXO-LIKE GAMES.

The original idea of studying OXO-like games was introduced by Beynon and Joy in [5]. The motivation was to explore the possible merits of introducing programming via the technique that has since been characterised as *making a construal*. At that time, the principal alternative programming paradigm to which first-year computer science students at Warwick were being introduced was functional programming, and writing a program to play noughts-and-crosses was one of the student exercises. As discussed in [4], making a construal and adapting it to play noughts and crosses was so unlike programming that it became clear that it should not be classified as a programming paradigm at all: “The primary focus is upon modelling the environment and the agency that can in principle support playing the game. The end result is something that conceptually resembles a laboratory in which it is possible to realise a traditional game of noughts-and-crosses but where a whole cloud of alternative ways of playing the game—and other more-or-less closely related games—can also be equally conveniently realised.” [4].

Figure 1 illustrates the characteristics of the ‘OXO laboratory’. The screenshot represents the final stage in a process of incremental construction in which key observables associated with playing noughts-and-crosses – and variations based on the same 3 by 3 grid – are introduced layer-by-layer (cf the “INCLUDE NEXT LAYER” button at the top left corner). The concept of a ‘layer’ reflects the progressively more elaborate and nuanced observables that the human player must become familiar with in order to play the game. The layers correspond to the visual components in Figure 1 as read from left to right and top to bottom.

At the top left corner (“GEOMETRY”) is a representation of the physical grid, together with the conceptual ‘winning lines’. To its right (“STATUS”), there is a representation of the



```
X has won       = FALSE
O has won       = FALSE
It is a draw    = FALSE
The board is full = FALSE

Number of Xs    = 3
Number of Os    = 3
```

SQVALS:

0	0	0
7	0	16
0	0	12

PLAY:

0	41	0
11	0	16
0	0	8

```
HELP:
 This layer incorporates the whole concept of playing a game.  It introduces the concept of whose turn it is.  A player cannot place a
counter if it is not their turn or if the game is over.  You also cannot 'cheat' by removing or overwritting an O or an X.
 Click on the 'Initialise' button to clear the board and start a new game.  Click on the yellow button to change who starts (The player to
start is displayed on the button).  Click on the cyan button to turn the computer on or off (The state described on the button says whether
the computer is currently on or off).
```

Figure 1. Playing games with Noughts-and-Crosses (Simon Gardner, 1999 [6])

current configuration of Os and Xs on the grid. From this, the current status of the game, as represented textually in the top right hand panel, can be inferred from the rules. In Figure 1, it is apparent that neither player has won, and that the game is not yet over. The observables associated with the top row of the display relate to features of the game that can be statically observed by a knowledgeable player.

The display elements in the bottom row relate to the rules of play. The fact that it is X's turn to play can be inferred from the 'O to start' annotated button since there are the same number of Os and Xs on the board. The leftmost component of the bottom row ("SQVALS") is a naive static evaluation of the board from the viewpoint of O, the player with the move. It gives an indication of what is plausibly a good move. To its right ("PLAY") is a representation of the static evaluation that informed the last move, as made by player X. When the 'Computer On' annotation is displayed, player X is automated to make moves based on this static evaluation. Although there is no visible distinction in Figure 1 between the state of the grid as displayed on the bottom right panel ("GAMESTATE") and the static grid display ("STATUS"), these correspond to conceptually quite different modes of observation, as will become apparent.

There is a close correspondence between the sequence of representations in Figure 1 set out above and the observations that a person learning to play noughts-and-crosses has to make. Informally, there is some progression from one mode of observation to another. Being able to recognise that there are Os and Xs on the grid is more basic than appreciating that they stand in the abstract relation of 'constituting a winning line' for instance. Likewise, the automation of moves is only possible provided that all the essential pre-requisite elements of the game are in place. The script that is associated with Figure 1 is made up of several simple scripts each devoted to the corresponding mode of observation of a game. The "INCLUDE NEXT LAYER" button triggers the introduction of each of these scripts in sequence.

The basis for interpreting the construal shown in Figure 1 as a 'laboratory' for making OXO-like games is that the component scripts can be freely modified to reflect different conventions that might be adopted. The winning lines can be changed, as can the rules that determine the status of a position. In keeping with the theme of [3], the modes of interaction with the construal are also exceptionally flexible. With no automation in place, the maker can simulate all kinds of scenarios, such as cheating through taking an extra turn or overwriting a grid cell occupied by the opponent, dynamic changes to the set of winning lines as play proceeds, or linking permissible moves to a preliminary throw of a dice.

The interventions that can be carried out in this way can be performed opportunistically and asynchronously in such a way that the notions of playing and developing the game are no longer well-defined, as when the winning lines are changed after the game has been 'won', or the pieces on the board are directly manipulated during play (cf. a O or X 'falling off the board'). Such possibilities underlie the distinction between views such as GAMESSTATE and STATUS that are always synchronised in normal play.

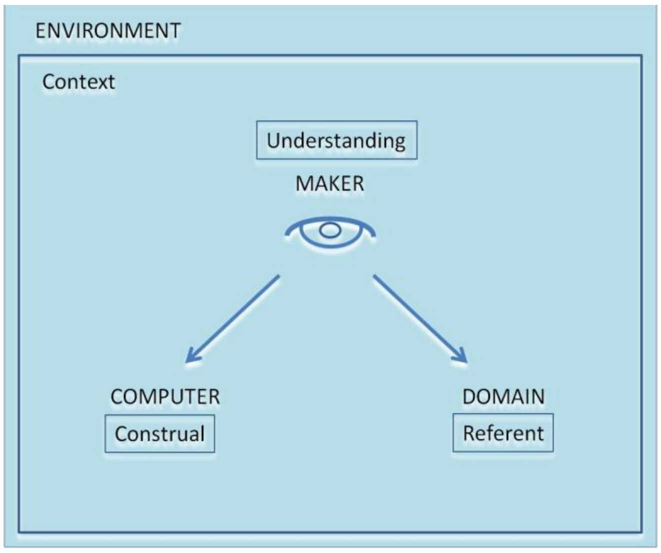


Figure 2. Making a Digital Construal

Figure 2 expresses the nature of the semantic relation between the construal and the OXO-like game with which it is associated. In referring to the construal of the standard game of noughts-and-crosses as displayed in Figure 1 above, the use of the term 'representation' is quite natural. It is appropriate because the context for the interaction is stable and well-established. In the process of devising a OXO-like variant the nature of the relationship between the construal and its referent is much more obscure. Removing a O or X from the grid may have all kinds of meanings for the maker. Such an action may be done in order to simulate an anomalous event in normal play, to explore a new protocol for making moves, or simply to check that some definition within the construal is correctly framed and has the intended or expected effect. The notion of 'intended and expected effects' itself presumes some familiarity with interaction with the construal on the part of its maker – to which the term 'understanding' in Figure 2 refers. In general, the significance of the interactions involved in making a construal has to be expressed in terms of concurrently shaping all four of the key ingredients in Figure 2 – the *construal* itself, its *referent*, the maker's *understanding*, and the overall *context* for interpretation.

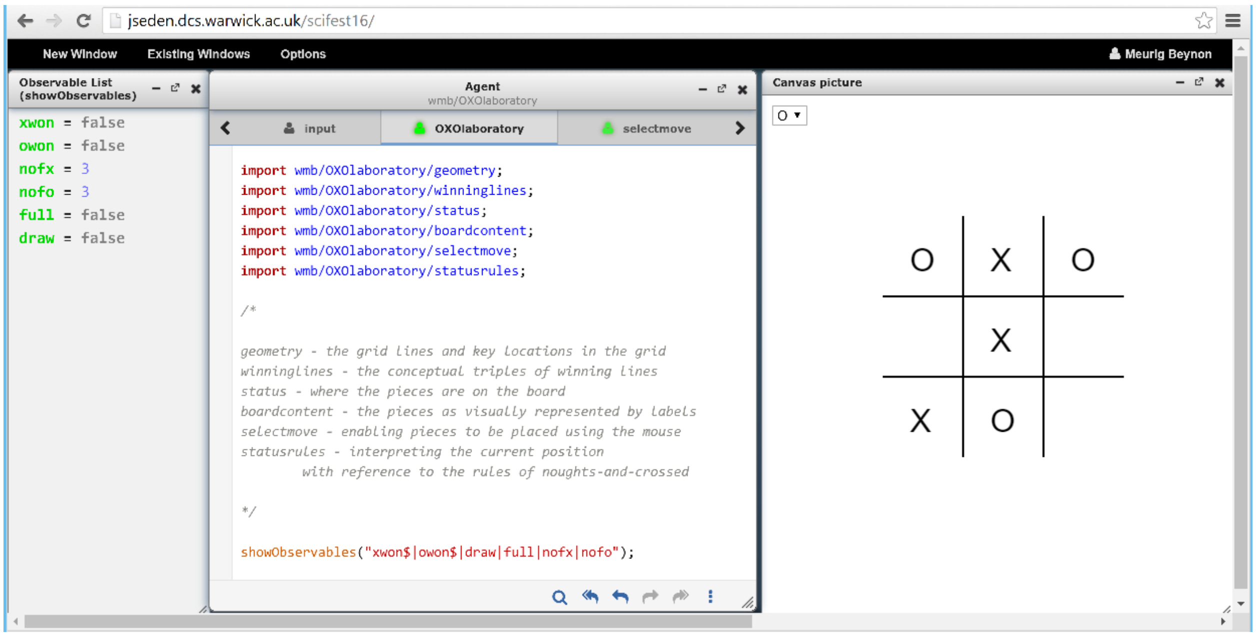
In the EDEN environment depicted in Figure 1, the management of scripts takes a complex and clumsy form. The screen display is complementary to an input window through which the definitions in a script can be submitted. To change the value or definition of an observable, a redefinition is entered. The observables themselves are of diverse types (corresponding to scalar data, or line drawings, or screen layout for instance) and the current definition and values of observables (which may take different syntactic forms) can be accessed through a range of viewers. The distinctions between one mode of observation and another that are visualised in Figure 1 are reflected in the way in which scripts are recorded in the file system. The core scripts that serve this purpose are those that are introduced by pressing the "INCLUDE NEXT LAYER" button. Other scripts, such as might be used to change the set of winning lines, may also be recorded in auxiliary files. In the process of conducting experiments within the OXO laboratory, useful files might also consist of annotated script fragments that are associated with incomplete or inconclusive explorations. The maker's "understanding", as expressed via informal familiarity with possible interactions and interpretations, is in general essential in making sense of such fragments. The relationship between the systematic organisation of definitions within core scripts and the unstructured sets of experimental definitions reflects that between the well-defined contexts for observation (such as playing a standard game of noughts-and-crosses) and the more loosely defined regimes for interaction (such as trying to find an interesting alternative set of winning lines) that can pertain in Figure 2.

III. THE OXO LABORATORY IN THE MCE: INFRASTRUCTURE

The abbreviation 'MCE' will be used to refer to the latest version of the environment for making construals [13]. This differs radically from the original EDEN interpreter: it is an online instrument that has been derived from the first JS-Eden prototype [11] over the last few years.

The general principles of using the MCE are based on modelling the key observables, dependencies and agency at work in the domain (see [3] for more details). Scripts of definitions describe configurations of observables and dependencies which express the way in which state-changes in the referent are linked. Figure 4 is a simple example of such a script. The dimensions of the grid depend on an observable 'size' which can potentially be redefined by the players or the developer.

When a construal is first made, it is built up incrementally by entering observables and their definitions into an input tab. There is at most one definition on each line, each terminated by a semi-colon. A definition can be interpreted by placing the mouse in the gutter to its left, then clicking with the left mouse button on the 'play' icon (▶) that appears. A tab may contain a script that in the context of Figure 1 would have been stored in an external text editor: the definitions within the script that are to be interpreted can be chosen selectively in any order, independent of the content of the whole script. Selective interpretation of this nature is particularly useful whilst the



```
xwon = false
owon = false
nofx = 3
nofo = 3
full = false
draw = false
```

```
import wmb/OXOlaboratory/geometry;
import wmb/OXOlaboratory/winninglines;
import wmb/OXOlaboratory/status;
import wmb/OXOlaboratory/boardcontent;
import wmb/OXOlaboratory/selectmove;
import wmb/OXOlaboratory/statusrules;

/*

geometry - the grid lines and key locations in the grid
winninglines - the conceptual triples of winning lines
status - where the pieces are on the board
boardcontent - the pieces as visually represented by labels
selectmove - enabling pieces to be placed using the mouse
statusrules - interpreting the current position
        with reference to the rules of noughts-and-crossed

*/

showObservables("xwon$|owon$|draw|full|nofx|nofo");
```

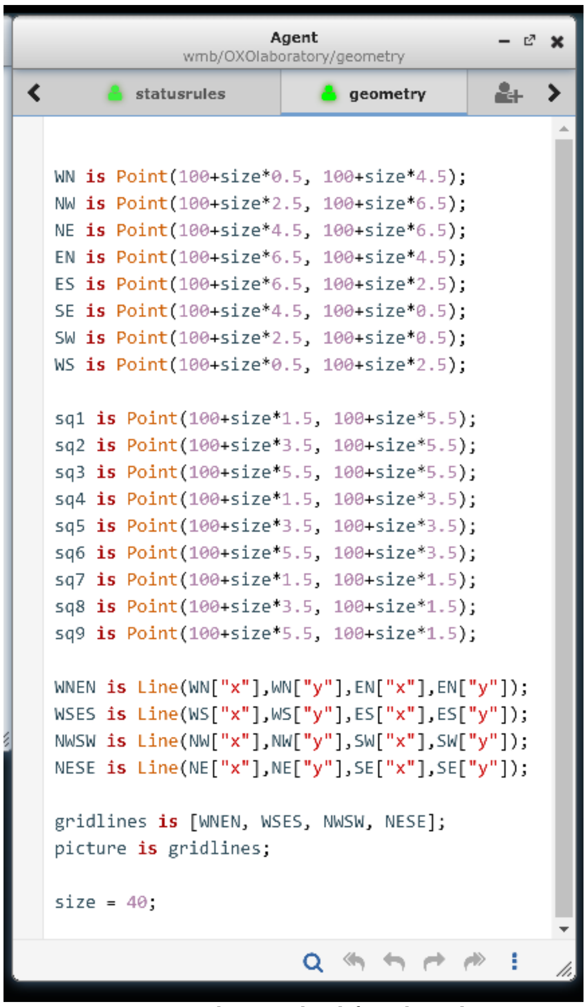
Figure 3. The OXO laboratory in the MCE

The main characteristics of the MCE will be introduced with reference to the re-creation of the core ingredients of the OXO laboratory as outlined above. Particular topics of interest are how the MCE seeks to meet the challenges of creating a more effective interface for experimentation with scripts, how the MCE can be extended in ways outside the scope of the original version, and the potential for new applications.

Figure 3 shows the overall concept behind the MCE interface as it might be deployed to create an environment similar to the OXO laboratory. From right to left, the three panels in the interface are respectively: a **canvas** on which the noughts-and-crosses position is displayed, an **input window** in which scripts can be viewed and input can be entered via an ensemble of tabs, and an **observable lis**t in which the current values of selected observables are displayed.

construal is immature. When a sufficiently stable script has been crafted, it can be interpreted as a whole by clicking on the tab name with the right mouse button and selecting the 'Run' option from the drop down menu.

As is illustrated in Figure 3, scripts can be imported into a tab – in this way they are automatically interpreted. The imported scripts listed in Figure 3 are 'stable' scripts that correspond to modes of observation of a game of noughts-and-crosses such as feature in the OXO laboratory in Figure 1. Figure 4 is the content of the first imported script to be listed. By default, imported scripts are interpreted but not displayed in a tab. There is a simple way in which an imported script can be loaded into a tab for inspection: first click on the spyglass ("inspect") icon at the left on the bottom of the input window, then click on the name of the imported script (now highlighted



```
WN is Point(100+size*0.5, 100+size*4.5);
NW is Point(100+size*2.5, 100+size*6.5);
NE is Point(100+size*4.5, 100+size*6.5);
EN is Point(100+size*6.5, 100+size*4.5);
ES is Point(100+size*6.5, 100+size*2.5);
SE is Point(100+size*4.5, 100+size*0.5);
SW is Point(100+size*2.5, 100+size*0.5);
WS is Point(100+size*0.5, 100+size*2.5);

sq1 is Point(100+size*1.5, 100+size*5.5);
sq2 is Point(100+size*3.5, 100+size*5.5);
sq3 is Point(100+size*5.5, 100+size*5.5);
sq4 is Point(100+size*1.5, 100+size*3.5);
sq5 is Point(100+size*3.5, 100+size*3.5);
sq6 is Point(100+size*5.5, 100+size*3.5);
sq7 is Point(100+size*1.5, 100+size*1.5);
sq8 is Point(100+size*3.5, 100+size*1.5);
sq9 is Point(100+size*5.5, 100+size*1.5);

WNEN is Line(WN["x"],WN["y"],EN["x"],EN["y"]);
WSES is Line(WS["x"],WS["y"],ES["x"],ES["y"]);
NWSW is Line(NW["x"],NW["y"],SW["x"],SW["y"]);
NESE is Line(NE["x"],NE["y"],SE["x"],SE["y"]);

gridlines is [WNEN, WSES, NWSW, NESE];
picture is gridlines;

size = 40;
```

Figure 4. The script that defines the grid

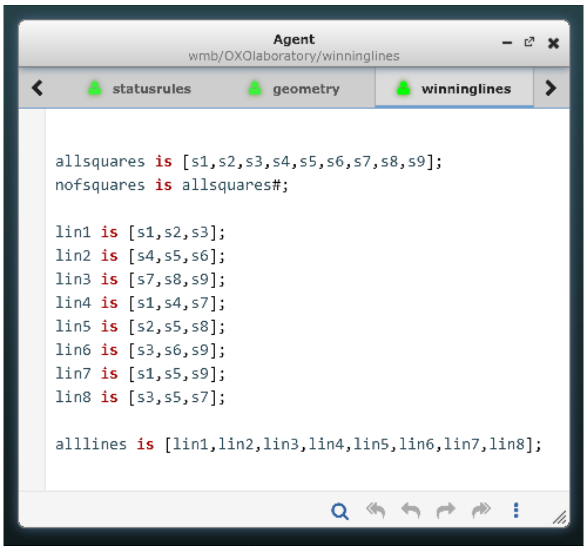
in red) which you wish to load. Figures 4-9 display the content of the tabs that can be derived in this way.

The most important role of the MCE is to enable the maker to make connections in their experience. The maker edits the script and simultaneously observes the effect via the picture or the observable list. As illustrated in Figure 3, the observable list displays only the current values of observables, and those that are defined by dependency are shown in green. The content of the observable list can be specified using a search expression – as illustrated in the **showObservables()** command in Figure 3. In addition to the discrete mode of redefinition described above, the MCE also supports a form of *live edit*. This is invoked by holding the left mouse button down in the gutter until a red star symbol appears. The impact of editing the corresponding definition is then automatically registered whilst it is syntactically correct. Live editing of observables defined using explicit scalar values can be carried out by hovering the mouse over the scalar value, depressing the mouse and moving it to the left or right. This is a convenient way in which to experiment with the observable ‘size’ in Figure 4 for instance.

In the original OXO laboratory, scripts were created using an external editor and stored as text files. As illustrated by the ‘INCLUDE NEXT LAYER’ button in Figure 1, the management of scripts was then handled by file inclusion. By contrast, the scripts that are to be imported in Figure 3 are recorded online within a project manager that is stored on the JS-Eden server. The project manager can be accessed by clicking on the ‘more’ menu icon (⋮) on the bottom left hand corner of the input panel (cf. Figure 3). Selecting ‘Browse Agents’ from the pop-up menu then lists the available scripts. In order to upload scripts to the project manager, it is necessary to login. On start up, the login icon appears at the top right corner of the MCE screen – clicking on it offers you the option of logging in via a Google or Twitter account. When you upload a script, you have the option of making it private or public.

Taken together, the features discussed in this section supply the infrastructure for the role of the experimenter in the OXO laboratory. The crafting of the core scripts to suit different modes of observation can be carried out by editing and/or live editing definitions in a targeted fashion. All versions of a script that are uploaded are recorded in the project manager and can be retrieved and reloaded. Scripts under development are also automatically saved in the local browser, and can be loaded from the View History option on the ‘more’ menu. This feature can be useful where versions are intentionally recorded as alternatives. For instance, the script in Figure 5 defines the normal set of winning lines in noughts-and-crosses, but a script with alternative definitions for lin1, ..., lin8 can easily be substituted.

The project manager is also a convenient way of sharing construals with other makers and enables remixing in an unconstrained way that is characteristic of construals. In this respect, making construals has more in common with software development associated with spreadsheets (cf. [10]) than with traditional programming.



```
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];
```

Figure 5. Defining the winning lines

IV. THE OXO LABORATORY IN THE MCE: CONSTRUCTION

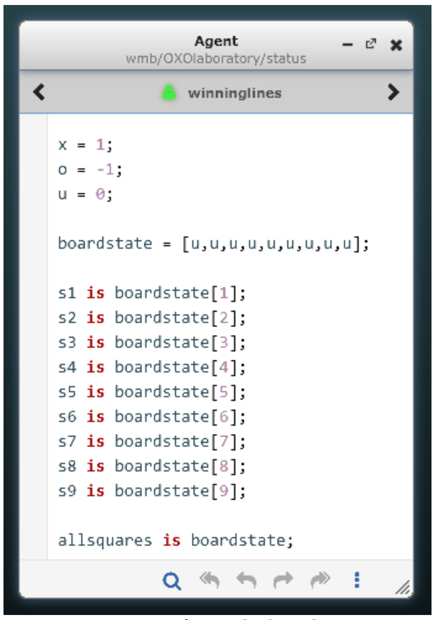
In a recent study on teaching programming to primary school pupils, Kalas [8] highlights the need to stage activities in an appropriate sequence. In the first instance, pupils learn to manipulate artifacts manually. They then learn to control (or ‘drive’) them by issuing commands. Finally, they program them to operate autonomously.

The primary focus in making construals is on the first stage identified by Kalas. The discussion in the previous section highlights how the MCE gives support for human agency in shaping the development of a construal. In this process, in the same spirit that Kalas moves from ‘direct manipulation’ to ‘driving’, the maker not only shapes features of the design of the construal but also rehearses actions that are part of its intended behaviour. Finally, some of these actions may then be automated in a program-like fashion. In contrast to traditional programming, making construals supports the free transition between these categories that is characteristic of a child playing with an artifact, as when manipulating a toy by hand when its battery runs out, or reverting to remote control of an autonomous robot.

The stages identified by Kalas are well-represented in the progression of modes of observation and agency that are associated with Gardner’s model of noughts-and-crosses shown in Figure 1. This section describes how a similar process can be realised within the MCE.

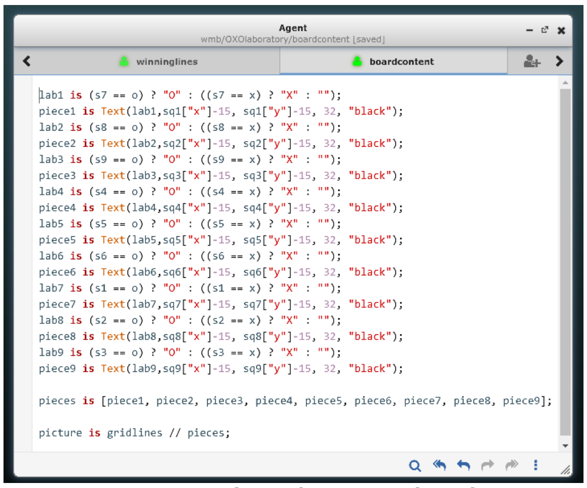
The ‘status’ and ‘boardcontent’ scripts, as imported in Figure 3, are listed in Figures 6 and 7. The observable ‘boardstate’ in Figure 6 is defined as a list that encodes the contents of the nine grid squares: blank, O or X. The text labels that are displayed on the grid are then defined by dependency in Figure 7. Redefining ‘boardstate’ corresponds to directly manipulating the construal.

The next stage in elaborating the construal is to introduce automated agents that can be instructed to update the definition of the observable boardstate. In the MCE, this can



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

boardstate = [u,u,u,u,u,u,u,u,u];

s1 is boardstate[1];
s2 is boardstate[2];
s3 is boardstate[3];
s4 is boardstate[4];
s5 is boardstate[5];
s6 is boardstate[6];
s7 is boardstate[7];
s8 is boardstate[8];
s9 is boardstate[9];

allsquares is boardstate;
```

Figure 6. Defining the board state

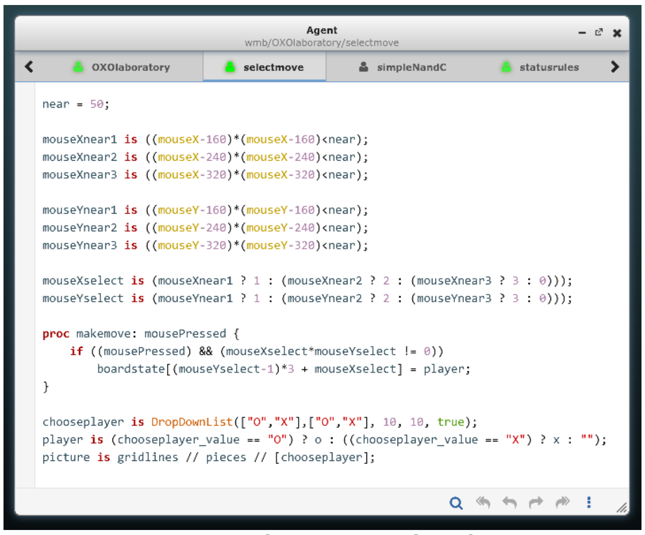


```
lab1 is (s7 == o) ? "O" : ((s7 == x) ? "X" : "");
piece1 is Text(lab1,sq1["x"]-15, sq1["y"]-15, 32, "black");
lab2 is (s8 == o) ? "O" : ((s8 == x) ? "X" : "");
piece2 is Text(lab2,sq2["x"]-15, sq2["y"]-15, 32, "black");
lab3 is (s9 == o) ? "O" : ((s9 == x) ? "X" : "");
piece3 is Text(lab3,sq3["x"]-15, sq3["y"]-15, 32, "black");
lab4 is (s4 == o) ? "O" : ((s4 == x) ? "X" : "");
piece4 is Text(lab4,sq4["x"]-15, sq4["y"]-15, 32, "black");
lab5 is (s5 == o) ? "O" : ((s5 == x) ? "X" : "");
piece5 is Text(lab5,sq5["x"]-15, sq5["y"]-15, 32, "black");
lab6 is (s6 == o) ? "O" : ((s6 == x) ? "X" : "");
piece6 is Text(lab6,sq6["x"]-15, sq6["y"]-15, 32, "black");
lab7 is (s1 == o) ? "O" : ((s1 == x) ? "X" : "");
piece7 is Text(lab7,sq7["x"]-15, sq7["y"]-15, 32, "black");
lab8 is (s2 == o) ? "O" : ((s2 == x) ? "X" : "");
piece8 is Text(lab8,sq8["x"]-15, sq8["y"]-15, 32, "black");
lab9 is (s3 == o) ? "O" : ((s3 == x) ? "X" : "");
piece9 is Text(lab9,sq9["x"]-15, sq9["y"]-15, 32, "black");

pieces is [piece1, piece2, piece3, piece4, piece5, piece6, piece7, piece8, piece9];

picture is gridlines // pieces;
```

Figure 7. Displaying the pieces on the grid

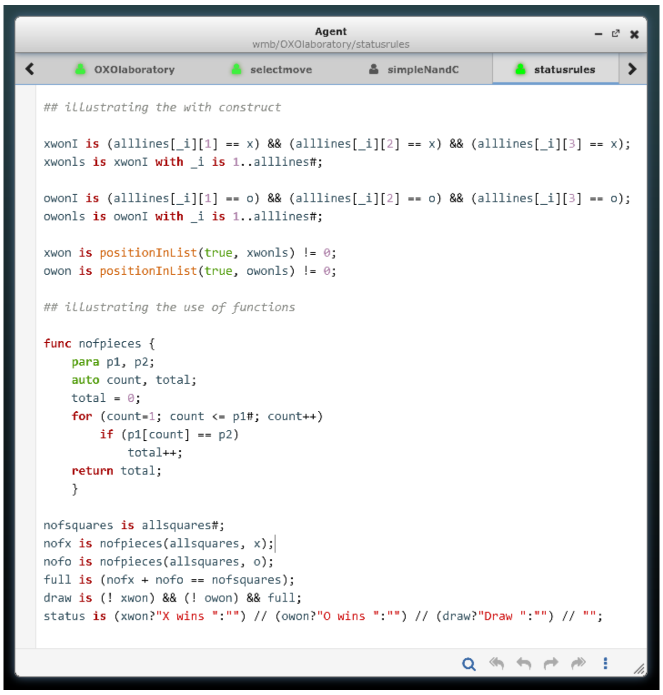
be done by introducing a triggered action that responds to changes to an observable, and performs an appropriate redefinition. The procedure ‘makemove’ in Figure 8 is such an action. The grid square to be updated is identified by creating dependencies based on the position of the mouse. The script in Figure 8 exemplifies the kind of script that is generated at an intermediate stage when interpreting the location of a mouseclick in this way. The definitions of the observables mouseXnear1 are here framed in terms of absolute coordinates for the centres of the grid squares sq1, ..., sq9 that were determined by surveying the canvas and observing the coordinates of the mouse pointer. This is unsatisfactory in that it fails to work for different values of the observable ‘size’. It is instructive to consider how flaws of this kind can be addressed within the MCE simply through refining the definition of mouseXnear1 and its counterparts.

Figure 8 also illustrates how the interface mechanisms for construals within the MCE can be supplied by html widgets such as a drop down list. In this case, such a list allows the maker to make a move on behalf of player O or player X. A small refinement of this definition would ensure that the option for a player is matched to the game position.



```
near = 50;

mouseXnear1 is ((mouseX-160)*(mouseX-160)<near);
mouseXnear2 is ((mouseX-240)*(mouseX-240)<near);
mouseXnear3 is ((mouseX-320)*(mouseX-320)<near);

mouseYnear1 is ((mouseY-160)*(mouseY-160)<near);
mouseYnear2 is ((mouseY-240)*(mouseY-240)<near);
mouseYnear3 is ((mouseY-320)*(mouseY-320)<near);

mouseXselect is (mouseXnear1 ? 1 : (mouseXnear2 ? 2 : (mouseXnear3 ? 3 : 0)));
mouseYselect is (mouseYnear1 ? 1 : (mouseYnear2 ? 2 : (mouseYnear3 ? 3 : 0)));

proc makemove: mousePressed {
    if ((mousePressed) && (mouseXselect*mouseYselect != 0))
        boardstate[(mouseYselect-1)*3 + mouseXselect] = player;
}

chooseplayer is DropDownList(["O","X"],["O","X"], 10, 10, true);
player is (chooseplayer_value == "O") ? o : ((chooseplayer_value == "X") ? x : "");
picture is gridlines // pieces // [chooseplayer];
```

Figure 8. Placing pieces on the grid



```
## illustrating the with construct

xwonI is (alllines[_i][1] == x) && (alllines[_i][2] == x) && (alllines[_i][3] == x);
xwonls is xwonI with _i is 1..alllines#;

owonI is (alllines[_i][1] == o) && (alllines[_i][2] == o) && (alllines[_i][3] == o);
owonls is owonI with _i is 1..alllines#;

xwon is positionInList(true, xwonls) != 0;
owon is positionInList(true, owonls) != 0;

## illustrating the use of functions

func nofpieces {
    para p1, p2;
    auto count, total;
    total = 0;
    for (count=1; count <= p1#; count++)
        if (p1[count] == p2)
            total++;
    return total;
    }

nofsquares is allsquares#;
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 ":"") // "";
```

Figure 9. Interpreting the game rules

The most interesting observables from the perspective of a game designer are those that are associated with the rules of the game. In Figure 1, examples of such observables are whether either player has won, whose turn it is, and whether there is a valid move.

The script shown in Figure 9 frames dependencies to express whether X has won, O has won or the game is drawn. The latter condition is declared to be true when the board is full and neither X nor O has won. Dependencies of this nature are not of the kind that can be easily expressed using standard formulae and built-in operators.

In Figure 9, two different approaches are used to illustrate how such dependencies can be formulated. The approach adopted in the EDEN interpreter, as deployed in Gardner's version of the OXO laboratory, is to introduce a maker-defined operator that can be used on the right-hand side of a definition. Such an operator can be written in a traditional procedural style (cf. the 'makemove' action in Figure 8). The function 'nofpieces' listed in Figure 9 is an illustrative example: it takes two parameters, a list representing values of all squares in the current position and a parameter to specify whether Xs or Os are to be counted.

Another possible approach is based on the use of the recently introduced **with** construct. This construct is closely aligned to the idea of observation that underpins making a construal. Its use is illustrated in Figure 9 in the definition of the observables 'xwon' and 'owon'. Informally X has won if one of the winning lines comprises only Xs. The generic condition for a winning line to consist of all Xs is expressed in the definition of 'xwonI'. The **with** construct makes it possible to mimic the process of observing each of the winning lines to determine whether or not it is a winning line for X. The observable 'xwonls' registers the outcomes of inspecting all the winning lines as a list. The value of 'xwon' is then **true** provided that at least one of the outcomes is **true**; this can be expressed using a standard operator that locates the index of an element in a list, returning 0 if it is absent.

Gardner's original noughts-and-crosses model included a computer player that was based on a static evaluation function. From the perspective of human play, this approach to move selection is highly artificial and contrived. For instance, in observational terms, when making a move, player O typically surveys the set of winning lines to identify whether one of them has two Os on it and, if so, places an O accordingly. Expressing this pattern of observation was infeasibly complex in the environment that Gardner deployed in Figure 1, but becomes possible if we use the **with** construct. An experimental script for this purpose is listed in the box below:

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

lin_w is lin[1]+lin[2]+lin[3] == -2;
winlineix is lin_w with lin is alllines[ix];
winlines is winlineix with ix is 1..8;

gapinlin is 1 if lin[1]==0
        else (2 if lin[2]==0
        else (3 if lin[3]==0 else 0));

playonlinix is gapinlin with lin is alllines[ix];
playonlines is playonlinix with ix is 1..8;

alllinesindices is alllines with
        s1 is 1, s2 is 2, s3 is 3,
        s4 is 4, s5 is 5, s6 is 6,
        s7 is 7, s8 is 8, s9 is 9;

winindex is _index if winlines[_index] else 0;

iswinindex is winindex with _index is 1..8;
wline is max(iswinindex);

when ((player==o) && wline>0 && !end_of_game) {
 boardstate[
   alllinesindices[wline][playonlines[wline]]
 ] = o;
}
```

This script looks formidable and difficult to interpret. There is a very direct correspondence between the definitions in the script and what – in human terms – are elementary acts of observation, however. The observable 'lin_w' is a template for the question: *does the line 'lin' have just two Os on it?* The observable 'winlines' lists the answers to this question for each of the winning lines. Likewise, 'gaponlin' is a template for: *where is / is there a blank square on the line 'lin'?* and 'playonlines' records where there are blanks on winning lines. The observable 'alllinesindex' performs a task that comes naturally to the human observer but is taxing in conventional programming notations (cf. the use of pointers); it transforms the observable alllines (see Figure 5) so that the observables lin1,..., lin8 are reinterpreted: lin5 is read as [2,5,8] rather than [s2,s5,s8] and so on. Finally, 'wline' records the index of a winning line that has just two Os on it, if there is one. Note that the abstract term 'index' here has a simple concrete observational equivalent – it expresses an answer to the question: *where is some object of interest located in a list?*

The observables and dependencies in the listing above are complemented by a **when** clause to express the commonsense action of O as a human player: *if it is Os turn to play and there is a winning line with just two Os on it, place an O in the blank square on that line*.

V. Concluding remarks

The main architect of the latest version of the MCE, Nicolas Pope, has transformed the environment in ways that both highlight points of affinity with contemporary software development environments and expose the novelty and potential of making construals. The unusual qualities of the interactions associated with the OXO laboratory in Figure 1 were hard to appreciate in a setting where script management relied on external text editors and configuration of scripts as text files. Pope's introduction of the **with** construct as an adjunct to networks of dependencies promises to transform scripts so that (as illustrated in the listing above) they are more closely matched to commonsense modes of observation. It remains to be seen how far this quality can be made apparent.

When reflecting on making construals in the MCE, it is natural to look for connections with established practices. The way that dependency is deployed in section IV (see Figures 6, 7 and 8) brings to mind the well-known model-view-controller pattern, for instance. In making construals, there is a tension between creating exploratory informally defined artifacts to aid personal understanding and stable artifacts combining well-rehearsed interactions and well-defined interpretations that everyone can understand. This tension extends to the meta-level, where using built-in features and established patterns may seem more appropriate than crafting observables, dependencies and agency from first principles in a personal style. The MCE has its own mechanisms for scaling geometry for instance, so that the observable 'size' in Figure 3 can be made redundant. Specifying dependencies associated with mouse actions on the display in Figure 8 explicitly can also be avoided by exploiting more advanced features of the MCE.

In this context, the unstructured messy practices of makers of construals may be contrasted with those of tidy-minded programmers. The simplified interface displayed in Figure 3 is untypical of what is involved in more ambitious modelling exercises, where there may be many input windows, canvases and instances of viewers playing a similar role to the observable list. In making complex construals, it is often helpful to consult many different views of the same script: for instance, examining the values of the observables by clicking on them in the 'inspect' mode, or studying the relationship between them using a 'dependency map'. As several of the listings above illustrate, crafting a script evokes the spirit of bricolage. In traditional software design, by contrast, good practice favours eliminating redundancy, conformance to standards, and clarification through abstraction.

Making construals is not well-oriented towards the simplification that stems from retreating from experience to abstraction. It can play an important role in rationalising, but is best-suited to what cannot be formalised and is not fully understood (cf.[7]). This has particular relevance for tasks that present challenges in 'wicked design' [9]. Though off-the-shelf JavaScript components can be incorporated into a construal, the benefit of making construals is in the learning experience, even when this may involve 'reinventing the wheel'. Another variety of re-use is then more appropriate, where makers build and share small construals that can be combined and integrated into larger construals. A core idea is that viewing 'the same thing' from many perspectives is not redundancy. This is also illustrated in the MCE through the potential for modelling behaviours both objectively (as in conventional functions and procedures) and in ways that reflect an agent-oriented viewpont (as in **with**'s and **when**'s).

The MCE is still at an early stage of development. A core aim of CONSTRUIT! is to make the practice of making and sharing construals accessible to everyone. Outstanding challenges include creating an interface to the project manager than can do justice to this vision and finding a way to expose the powerful intuitive foundation for the **with** construct.

Acknowledgments

We thank Tim Monks, Antony Harfield and Joe Butler for their work on JS-Eden, and Mike Joy for his work on [5] and as Coordinator for CONSTRUIT!. This project has been funded with support from the European Commission under the Erasmus+ programme (2014-1-UK01-KA200-001818). This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

[1] Empirical Modelling, "Empirical Modelling," 2015. http://go.warwick.ac.uk/em. [Accessed: 08-Jul-2015].

[2] The CONSTRUIT! Project, "The project," 2015. http://construit.org/. [Accessed: 08-Jul-2015].

[3] Beynon, M. et al, "Making construals as a new digital skill: dissolving the program - and the programmer - interface," *Proc. 2015 International Conference on Interactive Technologies and Games*, 22-23 October 2015, Nottingham, UK, pp9-16

[4] Beynon, M. and Russ, S. "Experimenting with Computing", *The Journal of Applied Logic* 6, pp. 476-489, 2008

[5] Beynon, M .and Joy, M. "Computer programming for noughts-and-crosses: New frontiers," in *Annual Psychology of Programming Interest Group Conference PPIG'94*, 1994, pp. 27–37.

[6] S. Gardner, "empublic: oxoGardner1999," 1999. http://empublic.dcs.warwick.ac.uk/projects/oxoGardner1999/. [Accessed: 08-Jul-2015].

[7] Jackson, M.A. "What can we expect from program verification?," *IEEE Computer,* 39(10), 53–59.

[8] Kalas, I.. "On the Road to Sustainable Primary Programming," Constructionism 2016, February 1-5, Bangkok, Thailand. https://www.youtube.com/watch?v=55hlqh7g-xk [Accessed: 22-May-2016].

[9] Lansdown, J. R. "Graphics, Design and Artificial Intelligence,". (ed. R.A. Earnshaw) *Proceedings of the NATO Advanced Study Institute on Theoretical Foundations of Computer Graphics and CAD* held at II Ciocco, Italy, July 4-17, 1987, pp. 1153-1174

[10] Nardi, B.A. *"A Small Matter of Programming: Perspectives on End User Computing,"* The MIT Press, Cambridge, MA. 1993

[11] Monks. T. "A definitive system for the browser," 2011. http://go.warwick.ac.uk/em/publications/mscprojects/timmonks/. [Accessed: 22-May-2016].

[12] "Web Eden," 2013. http://go.warwick.ac.uk/em/software/webeden/. [Accessed: 08-Jul-2015].

[13] "The environment for making construals (the MCE)", May 2016. http://jseden.dcs.warwick.ac.uk/scifest16/ [Accessed: 22-May-2016]