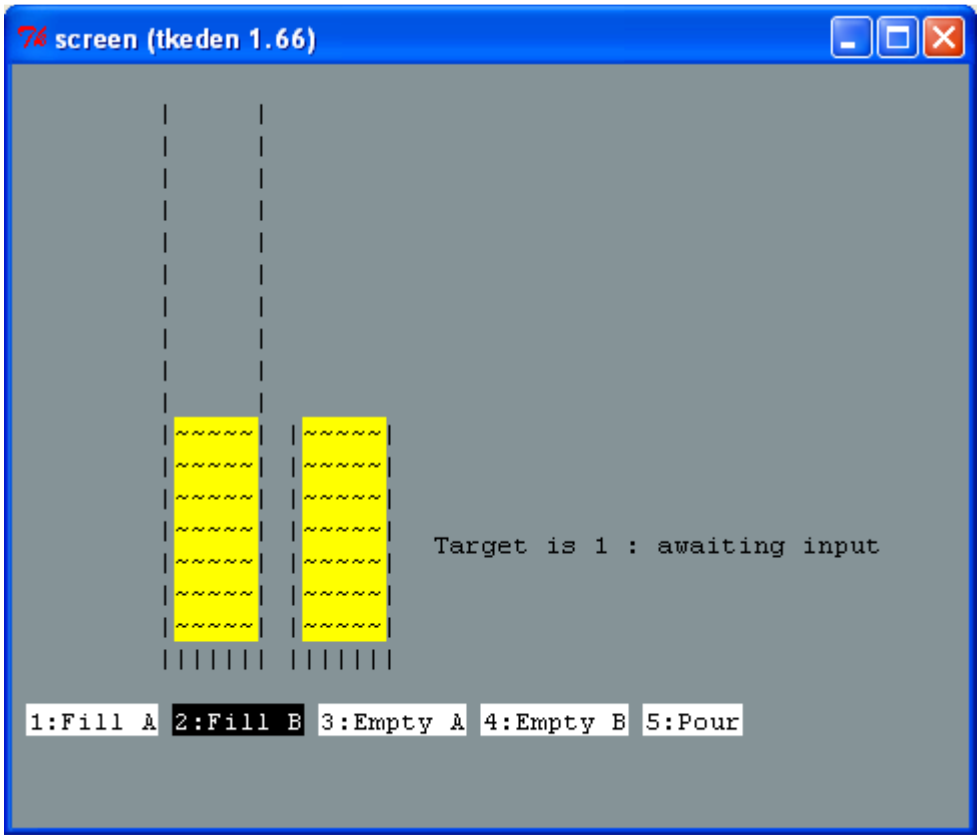
Illustrating modelling and programming in EDEN

The JUGS model was originally developed in connection with a project organised by the Micro-electronics Support Unit (MESU) concerned with developing portable versions of educational software. For more background, see `../research/em/publications/papers/008` and `http://www.dcs.warwick.ac.uk/reports/cs-rr-147.pdf`.

There are two versions of the JUGS model in the projects archive: `jugsBeynon1988` and `jugsBeynon2008`. (More discussion of the distinction between these can be found in Lecture 8 from CS405 in 2008-9.) Will here focus on the more recent version of the JUGS model.

The JUGS model display



The button display

```
%scout
string menu1, menu2, menu3, menu4, menu5;
%eden
menu1 is menu[1];
menu2 is menu[2];
menu3 is menu[3];
menu4 is menu[4];
menu5 is menu[5];

%scout

string valid_fg, valid_bg, invalid_fg, invalid_bg;
valid_fg = "black";
valid_bg = "white";
```

```
invalid_fg = "white";
invalid_bg = "black";

integer valid1, valid2, valid3, valid4, valid5;

window wmenu1, wmenu2, wmenu3, wmenu4, wmenu5;
box bmenu1, bmenu2, bmenu3, bmenu4, bmenu5;
point base;

base = {1.c, 20.r};

wmenu1 = {
        frame:  (bmenu1),
        string: menu1,
        fgcolor:if valid1 then valid_fg else invalid_fg endif,
        bgcolor:if valid1 then valid_bg else invalid_bg endif
        sensitive: ON
};
bmenu1 = [base, 1, strlen(menu1)];

...

wmenu5 = {
        frame:  (bmenu5),
        string: menu5,
        fgcolor:if valid5 then valid_fg else invalid_fg endif,
        bgcolor:if valid5 then valid_bg else invalid_bg endif
        sensitive: ON
};
bmenu5 = [bmenu4.ne + {1.c, 0}, 1, strlen(menu5)];
# bmenu5 is one space right of bmenu4
```

A representative mouse action

```
%eden
proc wmenu1_button : wmenu1_mouse_1 {
        if (wmenu1_mouse_1[2] == 4)
                input = 1;
}
```

Displaying the jugs

```
integer capA, capB, contentA, contentB, widthA, widthB, height;
string targ, totstat;

window wcapA, wcapB, wcontentA, wcontentB;
frame fcapA, fcapB;
string cA, cB, JugA, JugB;

%eden

func repeatChar
{
        auto s, i;
        s = substr("", 1, $2);
        for (i = 1; i <= $2; i++)
                s[i] = $1;
        return s;
}

cA is repeatChar('~', widthA*contentA);
cB is repeatChar('~', widthB*contentB);
JugA is repeatChar('|', 2*capA+2+widthA);
JugB is repeatChar('|', 2*capB+2+widthB);
%scout

fcapA = ([bmenu1.ne+{0, -(2+capA).r}, capA, 1],
         [bmenu1.ne+{(widthA+1).c, -(2+capA).r}, capA, 1],
```

```
         [bmenu1.ne+{0, -2.r}, 1, widthA+2]);
wcapA = {frame:fcapA, string:JugA};
wcontentA = {
        frame:  ([wcapA.frame.3.nw+{1.c,-contentA.r}, contentA, widthA]),
        string: cA,
        bgcolor:"yellow"
};

...
```

Showing the status of the model

```
window wstatus;
wstatus = {
        frame:  ([fcapB.2.ne+{2.c, (capB/2).r},1,strlen(targ // totstat)]),
        string: targ // totstat
};

screen = < wmenu1 / wmenu2 / wmenu3 / wmenu4 / wmenu5
           / wcontentA / wcontentB / wcapA / wcapB / wstatus >;

integer input;

%eden

Error=0; updating=0;
finish is ((contentA==target)||(contentB==target))&&!updating;

status is (Error)?"invalid option": ((updating)?"updating":"awaiting input");
totstat is (finish) ? "Success!" : status;
targ is ("Target is " // str(target) // " : ");
status is (Error)?"invalid option": ((updating)?"updating":"awaiting input");
totstat is (finish) ? "Success!" : status;
targ is ("Target is " // str(target) // " : ");
/* 'stat' is now an Eden builtin, so changed to jstat since 1988 */
jstat is messdisplay(height,totstat);
targt is messdisplay(height,targ);

jstat is messdisplay(height,totstat);
targt is messdisplay(height,targ);
```

The `eden` definitions above illustrate how strings to describe current state can be framed using dependencies. The entire interface can be generated as a string in this way - see the file `jugs.disp.e` from jugsBeynon1988. This technique was often used before graphical displays were commonplace, and was also a feature of early examples of declarative programming.

The core observables and dependencies

The following observables supply the particular values that are needed for the all components of the current state to be defined. Note the use of `autocalc` to ensure that all definitions are in place before any are interpreted. This helps to eliminate spurious bugs resulting from action invocation in states that are insufficiently defined, and also helps to make loading of files more efficient.

```
autocalc = 0;

func max { return $1
```

Note the "invisible options": there is no need to distinguish between the two directions for pouring provided that the interaction with the JUGS follows the formulaic pattern imposed by using the button interface. In a prior version of the model, both these options were exposed and accessible, Restoring this variant of the model is a straightforward exercise to the reader.

Specifying the pouring activities

The definitions introduced so far supply an environment in which to experiment with different kinds of agency and agent interaction. The core observables can be redefined by the modeller directly via the input window.

To introduce autonomous behaviour, we introduce a family of agents whose roles are played by EDEN actions. Current versions of EDEN allow agent interactions to be managed in a flexible manner by using a clocking mechanisms associated with the special observable `edenclocks`. These effectively make it posible to specify that an action is performed after a specified delay (and meanwhile leave the processor uncommitted.) It is conceptually important to recognise that these 'edenclocks' are devices to enhance the power of EDEN as a modelling instrument - they do not necessarily oblige the modeller to introduce observation of a clock into the external referent. In other words, in any particular modelling context, the variables they manipulate can either be of purely internal significance or serve as external observables, according to whether they are to be interpreted from a procedural or definitive viewpoint.

In this later version of the JUGS model, the different varieties of pouring action are performed by a family of agents, at most one of which is active at a particular instant according to the current value of the observable `option`. This model can be exercised in such a way as to illustrate traditional JUGS program use, but also allows interleaving of pouring activities and modeller intervention in a free and flexible fashion.

```
edenclocks = [[&tick, 500]];

proc init_pour: input {

        ## an observable concerned with whether state is stable
        ## updating = 1;

        ## setting up a dependency for an agent action

        if (int(input) == 5)    {
                content5 = contentA + contentB;
                contentB is content5 - contentA;
                option = valid6 ? 6 : 7;
        } else {
               contentA = contentA;
               contentB = contentB;
                option = int(input);
       };
```

```
        edenclocks = [[&tick, 500]];
}

proc fillingA: tick, option {
        if ((option==1) && avail(1)) {
                contentA = contentA + 1;
                jugAfilling = 1;
        }
        else if (jugAfilling==1) jugAfilling = 0;
}


proc fillingB: tick, option {
        if ((option==2) && avail(2)) {
                contentB = contentB + 1;
                jugBfilling = 1;
        }
        else if (jugBfilling==1) jugBfilling = 0;
}


proc emptyingA: tick, option {
        if ((option==3) && avail(3)) {
                contentA = contentA - 1;
                jugAemptying = 1;
        }
        else if (jugAemptying==1) jugAemptying = 0;
}

proc emptyingB: tick, option {
        if ((option==4) && avail(4)) {
                contentB = contentB - 1;
                jugBemptying = 1;
        }
        else if (jugBemptying==1) jugBemptying = 0;
}

proc pouringAB: tick, option {
        if ((option==6) && avail(6)) {
                contentA = contentA - 1;
                ABpouring = 1;
        }
        else if (ABpouring==1) ABpouring = 0;
}


proc pouringBA: tick, option {
        if ((option==7) && avail(7)) {
                contentA = contentA + 1;
                BApouring = 1;
        }
        else if (BApouring==1) BApouring = 0;
}

updating is jugBfilling || jugAfilling || jugAemptying || jugBemptying || ABpouring ||
BApouring;

proc deselectoption : updating {
        if (updating == 0) option = 0;
}

jugBfilling = jugAfilling = jugAemptying = jugBemptying = ABpouring = BApouring = 0;
```
