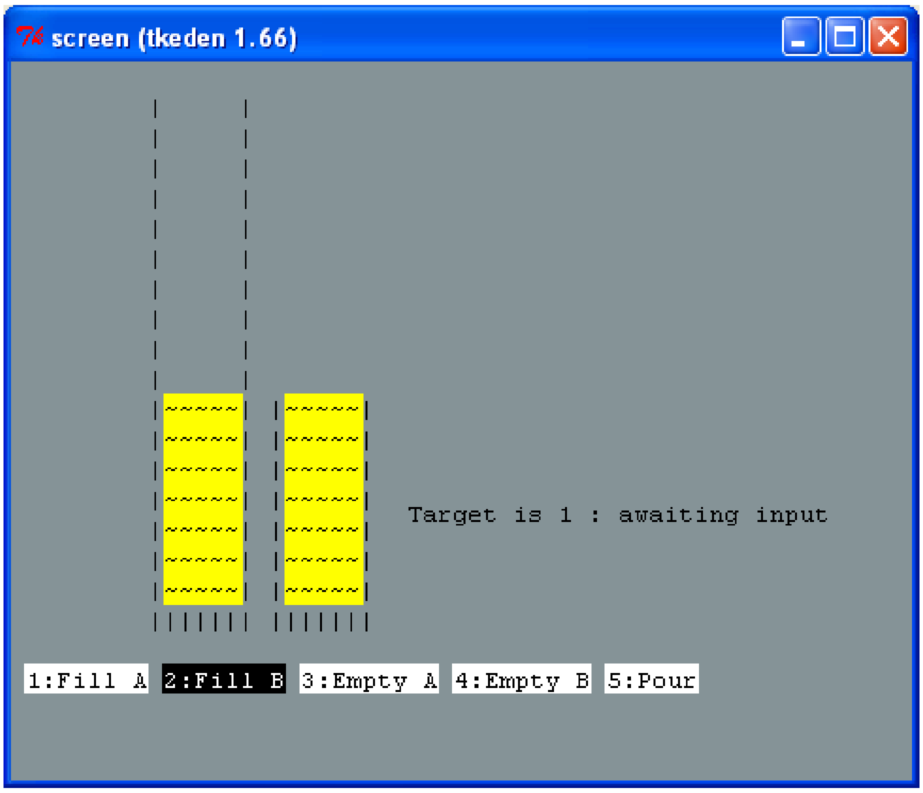
Illustrating modelling and programming in EDEN

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`.

Will here review the JUGS model, and two contrast two versions of it that illustrate issues in specifying agent interaction in EDEN.

The JUGS model display



Target is 1 : awaiting input

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 they 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<$2 ? $2 : $1; };

target=1;
capA = 5;
capB = 7;
Afull is capA==contentA;
Bfull is capB==contentB;
contentA = 0;
contentB = 0;

height is max(capA,capB)+2;

widthB = 5;
widthA = 5;
menu is ["1:Fill A","2:Fill B","3:Empty A","4:Empty B","5:Pour"];
menustatus is [valid1, valid2, valid3, valid4, valid5, valid6, valid7];
                                /* two invisible options 6 & 7 */
valid1 is !Afull;
valid2 is !Bfull;
valid3 is contentA != 0;
valid4 is contentB != 0;
valid5 is valid6 || valid7;
valid6 is valid3 && valid2;
valid7 is valid4 && valid1;

autocalc = 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. Other kinds of activity that are supported are illustrated by procedural actions such as:

```
for (i=1; i<=capA; i++) {
        contentA = i;
        delay(2000000);
        eager();
}

proc actiontofillA {
        auto i;
        for (i=1; i<=capA; i++) {
                contentA = i;
                delay(2000000);
                eager();
        }
}

actiontofillA();
```

(Note the essential use of the `eager()` procedure here.)

An important and critical feature of the EDEN interpreter has always been the management of "concurrent" agent interaction for which definitive scripts conceptually give particular support. For example, the modeller may wish to redefine the size of jug even as its content is being changed.

The mechanism by which this implemented in variants of EDEN such as tkeden-1.46 and tkeden-1.49 involved ingenious management of runsets of pending actions, so that the interpreter would check for input from the modeller in the process of executing other actions. As the speed of the interpreter execution has increased, it has been increasingly important to incorporate delays in the interpretation of redefinitions, so that (for instance) the steps in filling a jug are visible to the modeller.

One way of implementing such a delay (used in jugsBeynon1988) is based on the built-in `time()` function.

```
proc delay {
para delayinterval;
        oldtime = time(); newtime = oldtime;
        while (newtime < oldtime+delayinterval) newtime = time();
}

viscosity = 2;
```

From an engineering perspective, this potentially offers more consistency and control than the more naive solutions that are typically used in informal experiment:

```
proc delay {
        para delayinterval;
        auto i;
        i=0;
        while (i < delayinterval) i++;
}

viscosity = 2000000;
```

Both these techniques suffer from the problem that they consume processor cycles precisely in order to do nothing, which is bad news for the general interactivity and dependency maintenance that EDEN is intended to deliver.

The current versions of EDEN offer a different solution to the problem of managing agent interactions in a flexible manner. These are based on the introduction of clocking mechanisms, that effectively make it posible to specify that an action is performed after a specified delay, but 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.

The different ways in which agent interaction can be specified and supported technically will be briefly illustrated using two versions of the JUGS model:

Version 1 - from jugsBeynon1988

These actions reflect a program-like perspective in which there are essentially two agents - the user and the program. Pressing a button generates a redefinition of `input` that triggers `init_pour`. This in turn sets up the appropriate environment for the action that it initiates by redefining the observable `step`.

```
proc init_pour: input {

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

        ## setting up a dependeny for an agent action

        if (int(input) == 5)    {
                content5 = contentA + contentB;
                contentB is content5 - contentA;
                option = valid6 ? 6 : 7;
        } else
                option = int(input);
        step = 0;
}

## pour is invoked with a specific option that normally
## doesn't change in the course of execution

proc pour: step {
        if (avail(option)) {
                switch (option) {
                case 1:
                                                ## simulating filling of jugA
                        contentA = contentA + 1;
                        break;
                case 2:
                        contentB = contentB + 1;
                        break;
                case 3:
                        contentA = contentA - 1;
                        break;
                case 4:
                        contentB = contentB - 1;
                        break;
                case 6:
                        contentA = contentA - 1;
                        break;
                case 7:
                        contentA = contentA + 1;
                        break;
                default:
                        writeln("option = ", option);
                        return;
                }
                eager();
                delay(viscosity);
                step++;
        } else {
                        ## breaking dependencies on termination of action
                contentA = contentA;
                contentB = contentB;
                updating = 0;
        }
}

func avail {
        /* indicates whether the menu option with parameter $1 is open */
        auto t;
        t = menustatus[$1];     ## consults the button menu, as user would
        return t;
}
```

In jugsBeynon1988, the above actions are used in conjunction with the first implementation of `delay` discussed above. With current versions of tkeden, this is perfectly satisfactory in the standard program-like interaction with the model, but does not (e.g.) support modeller intervention to change the content of jug A whilst filling jug A is in progress. (For details, see the message posted on the CS405 Module Forum on 18th December 2006.) This need not be interpreted as a bug in EDEN, but as an indication that an alternative mode of implementation is required. An updated version of the JUGS model has been developed to illustrate how clocking can be used to support a more effective implementation.

Version 2 - from jugsBeynon2008

In this version of the JUGS model, the monolothic "program agent" that performs all the different varieties of pouring action is replaced 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. This illustrates the enhanced responsiveness that the clocking mechanism introduces, in contrast to previous mechanisms that in effect are no longer suported.

```
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;
```