Modelling with Definitive Scripts

Definitive scripts

Use scripts of definitions to represent state
Use redefinition to specify change of state

Scripts make use of definitive notations:

- DoNaLD - line drawing
- SCOUT - window layout
- ARCA - combinatorial graphs

Each notation is oriented towards a different metaphor

Definitive notations

Definitive notations are simple languages within which it is possible to formulate definitions for variables ("observables") of a particular type.

A definitive notation is defined by

- an underlying set of data types and operators
- a syntax for defining observables of these types.

Review/illustrate key features of DoNaLD and SCOUT

DoNaLD data types

Donald is a definitive notation for 2-d line-drawing
Its underlying algebra has 6 primary data types:
integer, **real**, **bool**ean, **point**, **line**, and **shape**

A **shape** = a set of points and lines
A **point** is represented by a pair of scalar values {x,y}.
Points can be treated as position vectors: they can be added (p+q) and multiplied by a scalar factor (p*k)
A **line** [p,q] is a line segment joining points p and q

DoNaLD operators

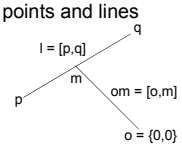
The DoNaLD operators include:

- arithmetic operators:
  + * div float() trunc() if ... then ... else ...
- basic geometric operators:
  .1 .2 .x .y {,} [,] + *
  dist() intersects() intersect()
  translate() rot() scale()
  label() circle() ellipse()

A DoNaLD file should begin with a "%donald"

DoNaLD syntax – points and lines

```
# declaring (NB) and defining points and lines
point o, p, q, m
line l
l = [p,q]
m = (p+q) div 2
line om
# new declarations can be introduced at any stage
o = {0,0}
om = [o,m]
.....
```



DoNaLD syntax – shapes

```
openshape S
within S {
        int m  # this is equivalent to declaring int S/m outside S
        point p, q
        openshape T
        p = {m, 2*m}
        within T {
                point p, q          # this point has the identifier S/T/p
                p, q = ~/q, ~/p
                # a multiple definition: p = ~/q and q=~/p
                # ~/... refers to the enclosing context for T
                # viz. S, so that ~/p refers to the variable S/p
                .....
        }
        ...
}
```


DoNaLD extras

Can define shapes in another way also: e.g.
shape rotsquare = rotate(SQ,….)
where SQ is defined to be a square

The “within X { …” context is reflected in the input window in EDEN

A syntax error in a ‘within’ context resets to the root context …

… there are NO SEMI-COLONS (;) *in DoNaLD !!!*

SCOUT types

SCOUT is a definitive notation for screen layout

Its primary data type is the **window**
Other types include: **display** (collection of windows, ordered according top to bottom); **integer, point** and **string**.

Windows are generally used to display text or DoNaLD pictures.

SCOUT screen definition

Overall concept

- a SCOUT script defines the current computer screen state
- **screen** is a special variable of type *display*
- the display is made up out of windows

Simplest definition of **screen** has the form
screen = < win1 / win2 / win3 / win4 / win5 / .... >
where ordering of windows determines how they overlay

Alternatively can define **screen** as union of displays
screen = disp1 & disp2 & disp3 & disp4 & ....

SCOUT window definitions

A SCOUT window definition takes the form

```
window X = {
        fieldname1: …
        fieldname2: …
        …
}
```

where the choice of *fieldnames* depends on the nature of the window content.

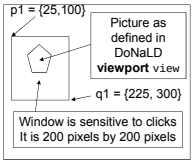
Defining a window to hold a DoNaLD picture

DoNaLD Window

field name	*type*	*description*
type	content	Must be the value DONALD
box	box	The region in which the DoNaLD picture is shown
border	integer	Set the border width of the bounding box
pict	string	The name of the DoNaLD picture
xmin	point	
ymin	point	Show the portion of the DoNaLD picture
xmax	point	bounded by the points (xmin, ymin) and (xmax, ymax)
ymax	point	

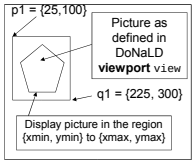
A simple SCOUT DONALD-window

```
point p1 = {25, 100};
point q1 = {225, 300};
window don1 = {
   box: [p1, q1],
   pict: "view",
   type: DONALD,
   border: 1
   bgcolor: “green”
   sensitive: ON
};
# locations of points are in pixels from top left of screen
#  coordinates of DONALD picture {0,0} to {1000, 1000}
```



Another SCOUT DONALD-window

```
window don2 = {
   box: [p1, q1],
   pict: "view",
   type: DONALD,
   xmin: zoomPos.1 - zoomSize/2,
   ymin: zoomPos.2 - zoomSize/2,
   xmax: zoomPos.1 + zoomSize/2,
   ymax: zoomPos.2 + zoomSize/2,
   border: 1
   sensitive: ON
}
```



Defining a window to hold text

Text Window		
field name	***type***	***description***
type	content	Must be the value `TEXT`
string	string	The string to be displayed
frame	frame	The region in which the string is shown
border	integer	Width of the border of the boxes of the frame
alignment	just	`NOADJ`, `LEFT`, `RIGHT`, `EXPAND` and `CENTRE` are the possible values to denote no alignment, left justification, right justification, left and right justification and centre of the text inside each box in the frame
bgcolour	string	Colour name for the background colour of the text
fgcolour	string	Colour name for the (foreground) colour of the text

A simple SCOUT TEXT-window

```
window doorButton = {
   frame: ([doorButtonPos, 1, strlen(doorMenu)]),
   string: doorMenu,
   border: 1
   sensitive: ON
};
string doorMenu = if _door_open then "Close
   Door" else "Open Door" endif;
```

SCOUT & DoNaLD extras

By default, a DoNaLD picture is displayed in a system generated SCOUT window, and has coordinates between {0,0} and {1000,1000}

SCOUT observables can be accessed in EDEN by the same names

A DoNaLD observable X/t can be accessed in EDEN and SCOUT by _X_t etc.

SCOUT extras

When aspects of the screen are undefined by the SCOUT script, it will not be drawn / redrawn

Sensitive SCOUT windows generate definitions of associated mouseButton variables: they supply information about the mouse state and location & can be used to trigger EDEN actions

Mouse clicks show up in the command history

More advanced definitive scripts

Definitive scripts can be used for rich modelling purposes … potentially for concurrent modelling

The range of aspects of a model that can be captured by definitive scripts is very broad

A brief look at scripts in their historical context helps to highlight their practical uses …

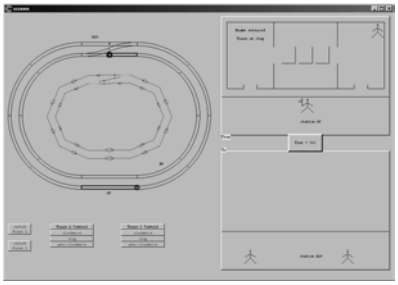
Illustrating ARCA

ARCA developed by Beynon and Fahranak c.1982
-first example of definitive notation to be designed in the EM project

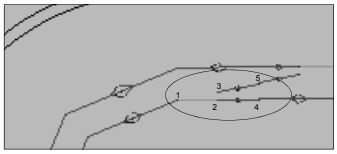
ARCA named after ARthur CAyley: originally developed to make models of 'Cayley diagrams', also known as 'group graphs'

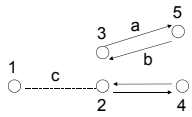
ARCA model of set of railway points as an example

Model Railway Model



Model Railway Model





Arca model for set of points

```
# points have 5 vertices and 3 colours
mode Point1 = 'abc'-diag 5

Point1Stts = 1      # status of the points, 1 is closing the loop
# defining the interconnecting edges
a_Point1{4} = 2; b_Point1{2} = 4; c_Point1{1} = if (Point1Stts == 1) 2 else 3;
a_Point1{2} = 2; b_Point1{2} = 4; c_Point1{2} = if (Point1Stts == 1) 1 else 2;
a_Point1{3} = 3; b_Point1{3} = 5; c_Point1{3} = if (Point1Stts == 1) 3 else 1;
a_Point1{4} = 2; b_Point1{4} = 4; c_Point1{4} = 4;
a_Point1{5} = 3; b_Point1{5} = 5; c_Point1{5} = 5;

Scale = 100         # scaling factor for locating vertices
Point1!1 = A!1      # defining locations of verts of pts 1 by dependency
Point1!2 = A!1 - [Scale / 2, 0, 0]; Point1!3 = A!1 - [Scale / 2, Scale / 10, 0]
Point1!4 = A!1 - [Scale, 0, 0]; Point1!5 = A!1 - [Scale, Scale / 5, 0]
```

Dependency Maintenance as key concept ...

Precedents for dependency maintenance (1970s)

- spreadsheet (Visicalc)
  what if? macro generation, culture of use
- geometric modelling (G and B Wyvill)
  visualisation, integrated design & simulation
- relational database languages (Todd - ISBL)
  data modelling, functional dependency, views

Significance of Definitive scripts

Modelling with definitive scripts

- qualities of systems of all three kinds
- principles for dependency maintenance

Concept of 'modelling with definitive scripts' is fundamental to EM … for a full discussion see Rungrattanaubol's thesis:
http://www.dcs.warwick.ac.uk/modelling/

Observables

Observables are entities

- whose identity is established through experience
- whose current status can be reliably captured by experiment

Can be physical, scientific, private, abstract, socially arbitrated, procedurally defined etc.

Dependency and Agency

An *agent* is an observable (typically composed of a family of co-existing observables) that is construed to be responsible for changes to the current status of observables

A *dependency* is a relationship between observables that - in the view of a state-changing agent - expresses how changes to observables are indivisibly linked in change

Examples of definitive notations

Notation	*Basis for underlying algebra*
eden	scalars, recursive lists, strings
donald	points, lines, shapes
scout	windows, displays (window = template + content)
arca	diagrams, vertices, incidences
sasami	polygonal meshes, renderings
eddi	relational database tables and views

Each notation is adapted to the metaphorical representation of different kinds of observable

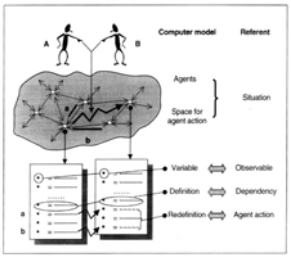
Agents

Agents are responsible for state-changes:
- meta-agents: e.g. the model builder
- agents determining model behaviour

Observables mediate agent actions/interactions

Use 'LSD notation' to specify perceptions and protocol (= *privileges*) of agents

Examples
- meta-agent: Track Layout Designer
- agent: stationmaster, guard



Empirical Modelling for computer-based construals

Virtues of a definitive script

- represents view (cf spreadsheet)
- variables correspond to observables
- hides invisible activity
- can represent indivisibility in action

... when interpreted with agent protocol
- allows experimental basis of knowledge
- reflects different status of parameters

... supports open-ended incremental and distributed development

Roles for modelling with definitive scripts

Definitive scripts support artefacts that

- enable individuals to identify reliable interactions with their environment
- enable individuals to recognise when there is a working understanding
- enable complex co-operative behaviour
- help individuals to construe complex system behaviour as agent interaction

… traditional approaches neglect the empirical foundation for knowledge of reliable systems that embraces activites of all these kinds

The semantics of definitive scripts

Contemplating state

What is involved in viewing a work of art?

- essentially about state and presence
- impossible to describe objectively
- differs from person to person, mood to mood
- draws on experience of all varieties
- is meaningful only in relation to associations
- is enriched through imaginative interaction

State in Computer Science

What is the role of state in computer science?

- typical formal paradigm is the state machine
- state is defined in relation to behaviour
- state is not the state of contemplation
- necessarily has an objective interpretation
- meaning is defined wrt reliable experiences
- only interaction promoted is preconceived

State in Informal Computing

What role has state in informal computer use?

- essential to human / machine communication
- mediated by interfaces and observables
- subjective for user, dependent on machine
- open to many kinds of interpretation
- not amenable to formal analysis alone
- involving empirical creation and evaluation

Motivating ideas

Traditional CS neglects **state-as-experienced**

Crucial for computer graphics and geometry and experiential aspects of learning activity

Propose Empirical Modelling approach as a new paradigm for computer-based modelling

Empirical Modelling (EM)

- based on observation, dependency, agency
- radical generalisation of spreadsheet concept
- knowledge representation via artefacts
- oriented towards state-as-experienced
- depends essentially on exploiting graphics
- gives basis for tools for geometric modelling

EM vs traditional modelling

• conflate concerns	• separate concerns
• represent via metaphor	• represent symbolically
• support ambiguity	• expect/impose precision
• encourage customisation	• promote standardisation
• expose empirical roots	• hide empirical foundation
• are shaped by construal	• discard explanation

… Key concept: Modelling based on 'definitive scripts'