Outline

EDEN
- Basic features
- Disciplines in using EDEN
  illustrated by the DoNaLD translator
- More advanced features
- Evaluation strategy

Interfacing Scout, DoNaLD and EDEN
- Translation hierarchy
- Selecting entry point
- Relating Scout, DoNaLD and EDEN

Tk interface
- Features

Other practical hints
- Attributes in DoNaLD
- Initialisation of EDEN strings and lists
- Writing a clocking mechanism in EDEN

Basic Features of EDEN

Data Type

int	char	float
string	pointer	list
	@	

User-defined Function

```
func fac {
    return $1 > 1 ? fac($1 - 1) * $1 :1;
}

func fac { para N;
    return N > 1 ? fac(N - 1) * N : 1;
}
```

Variable

formula variable e.g. vat is price * 0.175
read/write var e.g. foreign = local * rate

User-defined procedure and action

```
proc monitor_v : v {
    writeln(v);
}
```


Pre-defined EDEN Functions

I/O Functions
write(), writeln()

pe Conversion Functions
int(), char(), str(), float()
type()

String Functions
substr(), //, strcat(), nameof()

List Functions
sublist(), //, listcat(), array()

Time Functions
time(), ftime(), gettime()

Mathematical Functions
sin(), cos(), tan() ...

Pre-defined EDEN Functions (cont.)

Unix Functions
getenv(), putenv(),
error(), error_no(),
backgnd(), pipe()
get_msgq(), remove_msgq(),
send_msg(), receive_msg(),
fopen(), fclose(), fprintf(), gets() ...

Script Functions
include(),
apply(),
execute(), todo(),
exit(),
eager(), forget(), touch(),
formula_list(), action_list(),
symboltable(), symbols(), symboldetail()

Discipline in Using EDEN

The existence of procedural elements in EDEN (assignments and procedures over RWV) means that we can, but by no means recommended to, use EDEN as a conventional programming language.

We need to know when to use the procedural elements and when to use the definitive elements.

- Exploit definitions as much as possible in representing abstract states
- Use DoNaLD and Scout, if possible, for matching between the visual state and the abstract state
- Try to avoid side-effects as much as possible in visualisation, e.g. avoid the use of loci in DoNaLD.
- Action may be used to synchronise between abstract and visual models, or to be used in simulating agent actions.
- Procedures are used to transit from one state to another.

Implementing Definitive Notations

1. Set Naming Scheme

DoNaLD Name	EDEN Name
table	_table
table/drawer	_table_drawer
table/drawer/width	_table_drawer_width

2. Implement Data Types & Operators

DoNaLD Type	EDEN Type
integer	integer
point	['C', integer, integer]
line	['L', point, point]

DoNaLD Operator	EDEN Operator/ Function
div	/
+ (vector sum)	func vector_add { para p1, p2; return ['C', p1[1] + p2[1], p1[2] + p2[2]]; }

3. Implement Implicit Actions

DoNaLD Code	EDEN Action Specification
integer i	N/A
point p	proc P_p: _p { plot_point(&_p); }
line L	proc P_L: _L { plot_line(&_L); }

More Advanced Features

Extracting state information

```
? v ;
symboltable()
symbols(type)
symboldetail(&var)
```

Generating / executing EDEN statements online

```
'var_name'
nameof(&var)
execute(string)
todo(string)
```

Controlling the execution

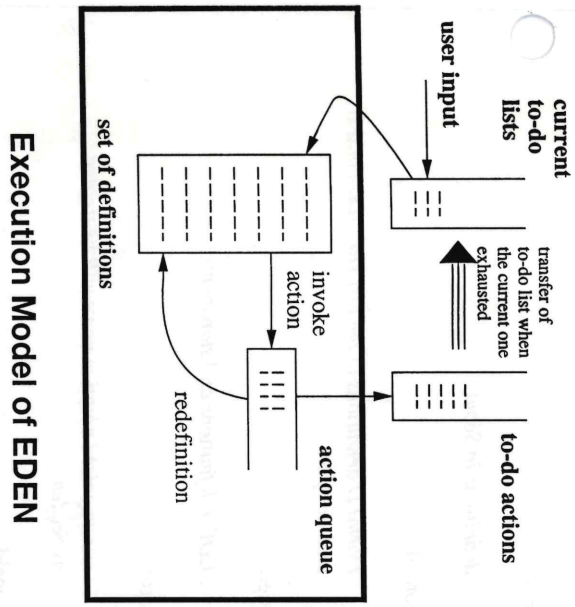
```
autocalc
eager()
```

```
proc dummy : A {  /* dummy() will be executed when A changes */
      auto i;     /* local variable */
      for (i = 1; i <= 10; i++) {
            V = i;
      }
}

proc Print_V : V {
      write(V, ' ');
}

A = 1;        /* change A to trigger dummy() */

/* result is :
10
*/
```




```
proc dummy : A {  /* dummy() will be executed when A changes */
      auto i;     /* local variable */
      for (i = 1; i <= 10; i++) {
            V = i;
            eager();
      }
}

proc Print_V : V {
      write(V, ' ');
}

A = 1;        /* change A to trigger dummy() */

/* result is :
1 2 3 4 5 6 7 8 9 10
*/
```

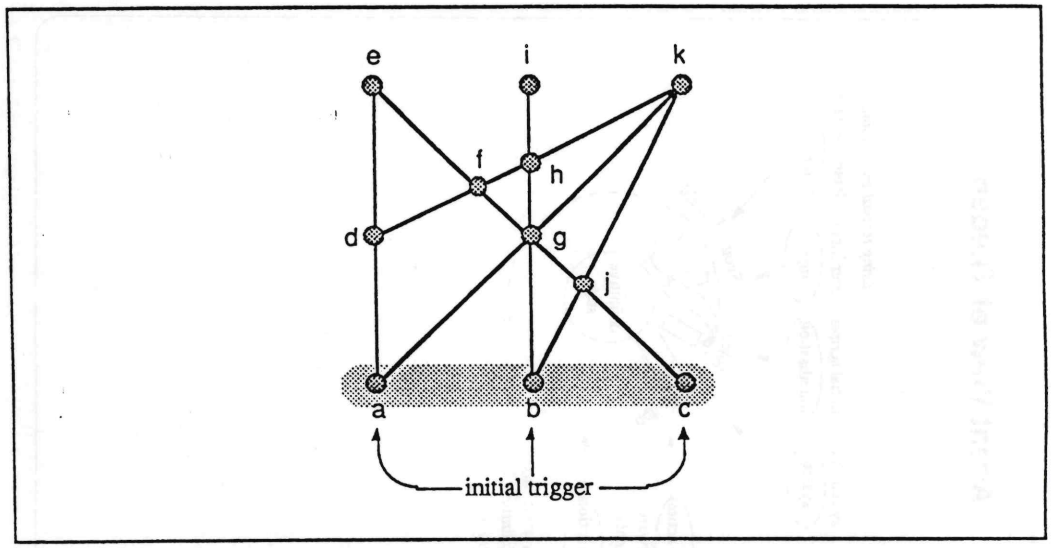
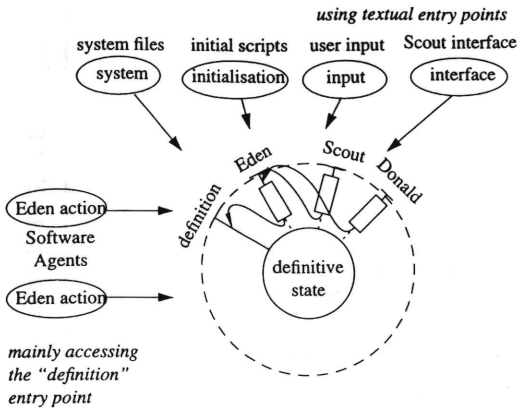


Figure 5-2: The dependency graph of a set of definitions

Agent View of Tkeden



Switching Between Scout, DoNaLD and EDEN

%scout

. *definitions in Scout*

%donald

. *DoNaLD definitions in the last DoNaLD context*
. *and viewport*

%eden

. *EDEN definitions and statements*

%error

. *unrecognised notation name will be treated*
. *as %eden*

%donald

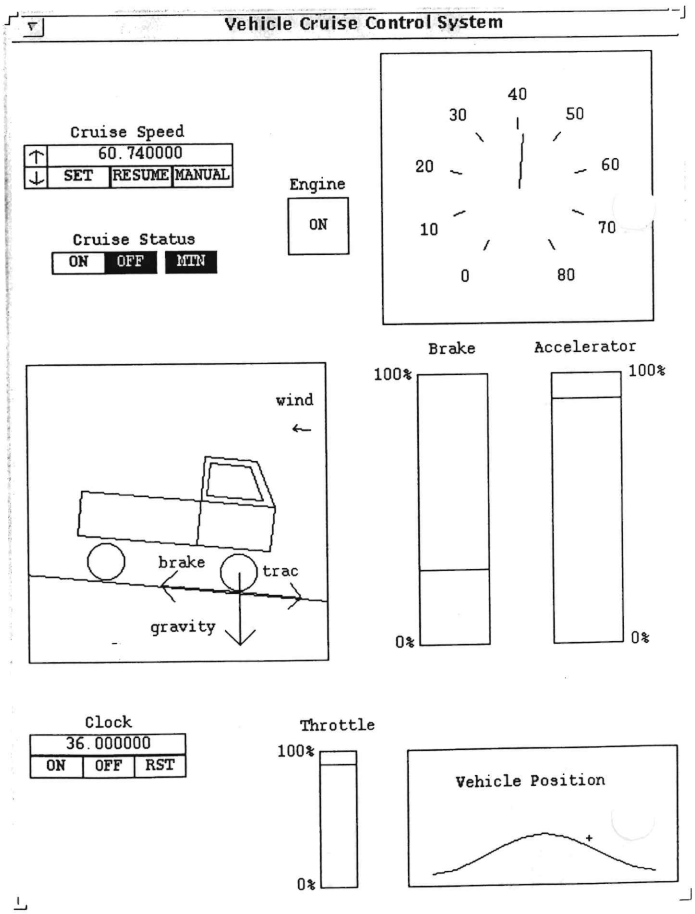
? *a line of EDEN, e.g.:*
?A_cable = "linestyle=dashed";

Extract From Vehicle Cruise Control Simulation

```
setBtn is (crSetBtn_mouse_1[2] == 4) ? pbDown : pbUp;
resBtn is (crResumeBtn_mouse_1[2] == 4) ? pbDown : pbUp;
manBtn is (crManualBtn_mouse_1[2] == 4) ? pbDown : pbUp;

%scout
window vehicle;
vehicle = {
        type: DONALD,
        box: [{10, 240}, {230, 460}],
        pict: "VEHICLE",
        xmin: -700,  ymin: -300,
        xmax: 300,   ymax: 700,
        border: 1
    };

%donald
viewport MODEL
openshape conceptCar
within conceptCar {
    circle frontWheel, backWheel
    frontWheel = circle({0,0}, 60)
    backWheel = circle({-450, 0}, 60)
```



Relating Scout and DoNaLD

• selecting DoNaLD viewport to view
• defining the mapping between the DoNaLD drawing space and Scout window

```
%scout
window vehicle = {
        type: DONALD
        box: [{10, 240}, {230, 460}]
        pict: "VEHICLE"
        xmin: -700,     ymin: -300
        xmax: 300,      ymax: 700
        border: 1
};

%donald
viewport MODEL              # not to be displayed

openshape conceptCar
within conceptCar {
        circle frontWheel, backwheel
        frontWheel = circle({0, 0}, 60)
        ...


viewport VEHICLE            # displayed in window vehicle

shape vehicle
vehicle = rot(conceptCar, {0, 0}, gradient)
```


Relating Scout and EDEN

```
%scout
window brakePedal = {
        type:       DONALD
        box:        [brakeOrg, brakeorg + {BAwidth, BAlength}]
        pict:       "BRAKE"
        xmax:       100
        ymax:       100
        border:     1
        sensitive:  ON
};

string startRelief;
window clkStartBtn = {
        string:     "ON"
        frame:      ([clkOrg + {0, 2.r + 6}, 1, 5])
        relief:     startRelief
        alignment:  CENTRE
        border:     1
        sensitive:  ON
};
%eden
startRelief is clkStartBtn_mouse_1[2] == 4 ? "sunken" :
        "raised";


brakePedal_mouse = [1, 4, 1, 37, 50];
clkStartBtn_mouse_1 = [1, 4, 0, 450, 600];
```


Features of Tk Interface

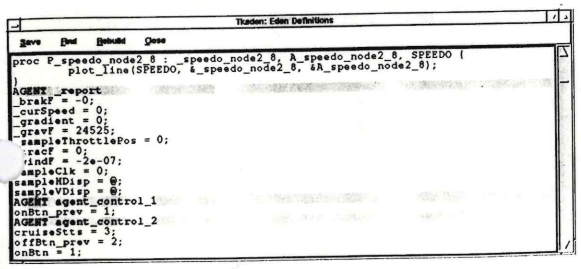
Supports different views of definitive store

Can examine definitions storing in the Scout and DoNaLD translators and the EDEN interpreter.

Since the translated Scout or DoNaLD definitions can be redefined in EDEN, there may be inconsistency between definitions as recorded in the translators and the EDEN interpreter.

In the EDEN definition view, a colour coding scheme is adopted to indicate the entry points of the definitions.

The sources of definitions (the agents that cause the defintions to be defined or modified) are also indicated.



Features of Tk Interface (cont.)

Recording history of interaction

Tkeden remembers:

- the scripts entered through the input window
- the definitions generated by the mouse or key actions in the Scout windows.
- error messages

A copy of the histroy is also saved automatically to the file named `.tkeden-history` in the user's home directory.

Allow save and load of current definitions

The main aim of the save and load facility is to save the current stage of script development so that it can be worked on in a separate session.

However, tkeden's 'save' facility will reorder the definitions, remove comments and has some problems in handling triggering of actions. Therefore, the usefulness of it is limited.

The history file may be another alternative.

Attributes of DoNaLD Shapes

Attribute	Applicable to	Value	Default
color	any shape	X colour name transparent	black
linewidth	line, arc, circle, ellipse, shape	integer	0 - min width
linestyle	line	dotted, dashed (poorly rendered) solid	solid
arrow	line	first, last, both, none	none
fill	any shape	solid, hollow	hollow
locus	any shape	true, false	false

Initialisation of EDEN Strings and Lists

• Although EDEN does not require declaration of variables, also it allow dynamic size changes of EDEN strings and lists, in some cases care should be taken to make sure the variables are in fact of certain types and contain large enough storage space for the intended operations.

• For example:

```
L[3] = 100;          /* L must have at least 3 elements */
scanf("%s", &s);     /* scanf() will not change the
                        type of s, s has to be initialised
                        as a beforehand */
```

• In many cases, assigning variables to [] or "" are already good enough.

• Array() and substring() functions are also very convenient.

```
L = array(4);        /* L = [@, @, @, @] */
L = array(3, 0);     /* L = [0, 0, 0] */
s = substr("", 1, 10);   /* s contains 10 spaces */
```


A Clocking Mechanism

```
chime = 0;

proc clock_watcher : clock {
    if (clock - clock_init >= 5) {
        clock_init = clock;
        chime++;
    }
    todo("clock = "// str(time()) // ";");
    /* the redefinition is delayed to allow user interaction */
}

proc bell : chime {
    if (chime)
        writeln("Ding Dong!");
}

/* Initialise clock to the current time. This will trigger the
clocking mechanism as well */
clock = clock_init = time();
```

Linking ADM and EDEN

1. Why translating ADM to EDEN?
2. The gap between ADM and EDEN
3. Comparing the AM interpreter and the ADM translator
4. The translation scheme
5. How to use the ADM translator?
6. The tic-tac-toe example

Why translating ADM to EDEN?

• Ideally we want to develop a proper interpreter for ADM. Mike Slade has developed such an interpreter called AM, but with very limited expressive power, e.g. it only manages integer data type, and allows limited user interaction.

• We also want ADM to have user interface. Currently, we mainly use Scout and DoNaLD for GUI, which are currently implemented using EDEN as the underlying evaluation engine.

• Many approaches can be taken:
  * communicating ADM and tkeden in OS level
  * reimplement Scout/DoNaLD in ADM
  * making graphical objects primitive data type in ADM
  * run ADM on the DAM machine
  * emulate ADM in EDEN

This tutorial is about the work of last approach.

Why bother with ADM anyway?
  * Closer to LSD
  * ADM has better potential than EDEN.

The Gap between ADM and EDEN

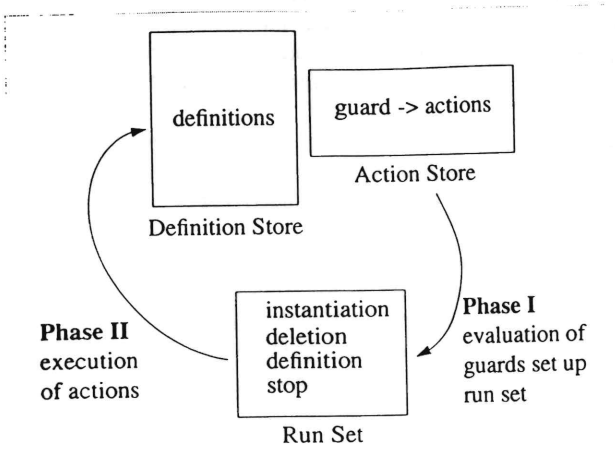
ADM	EDEN
dynamic instantiation and ion of parameterised entities	use execute() for dynamic creation of scripts
parallel guard evaluation	use todo() to delay action execution
values for assignments are evaluated in the context of guard evaluation	the expressions are evaluated in the context of guard evaluation but the assignment is delayed by todo()
parallel actions	sequential control
clocked system	asynchronous system - use the 2 todo lists to create 2 phases
conflict detection	no conflict for single-thread system

Comparisons of `am` and `adm`

am
- \+ Parallel actions
- \+ Conflict detection
- \+ Have semi-evaluation operator (| |)
- \- Limited algebra, only have integers and booleans
- \- Restricted interface to definitive notations

adm
- \+ Accept most EDEN definitions (including assignments)
- \+ Fully supported by other definitive notations
- \+ More advance parameter specification
- \+ Better user interaction
- \- Sequential execution of supposedly parallel actions
- \- No conflict detection
- \- Poor handle of entity deletion

Execution of ADM



- See web page for the execution model of ADM in algorithmic form as provided by Mike Slade.

Translation of ADM Actions

```
ready && whistled && !sm_raised_flag
      print("station master raises his flag")
         -> sm_flag = true;
          sm_raised_flag = true
```

An ADM Guarded Action

```
proc sm_action_6 : sysClock {
    if (sysClock == -1) return;
    if (ready and whistled and not sm_raised_flag) {
        writeln("station master raises his flag");
        todo("sm_flag = TRUE; sm_raised_flag = TRUE;");
    }
}
```

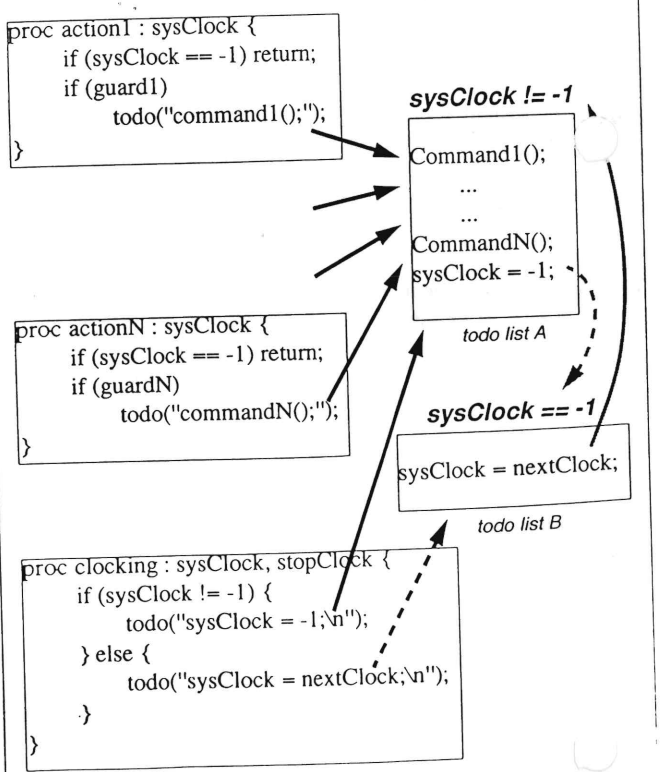
The Guarded Action in EDEN

The ADM System Clock

```
proc clocking : sysClock, stopClock {
  if (!stopClock && (stopTime < 0 || sysClock < stopTime)) {
    if (Pause > 0)
        sleep(Pause / 2.0);
    if (sysClock != -1) {
        todo("sysClock = -1;\n");
    } else {
        nextClock++;
        if (!Silent)
            todo("writeln(\"time = \", nextClock);\n");
        todo("sysClock = nextClock;\n");
    }
  }
}

stopClock = 1;      /* set stopClock to stop clocking */
stopTime = -1;      /* preset time to stop simulation */
Silent = 0;         /* set tosuppress the showing of time */
Pause = 1;          /* min. gap between 2 sys. clock pulses */
```


ADM in Action



When sysClock is defined as nextClock, these ——→ todo actions are queued. When sysClock is defined as -1, this - - → todo action are queued up.

How to Use `adm` - Translation

adm < ADMFILE > EDENFILE

A typical adm file consists of an adm program enclosed by lines beginning with %adm:

lines with Scout, DoNaLD and EDEN (not processed by adm)
...
%adm
...
program in adm
...
%adm
...
more lines with Scout, DoNaLD and EDEN

Note: an adm file may have at most ONE adm program.

How to Use `adm` - Interaction

After start-up, you may change a few simulation parameters (EDEN variables) before starting the simulation:

stopTime (Default: -1)
The simulation will stop at or after this system time. Note that the system time will not be reset after stopping, so you have to set stopTime forward in order to continue the execution. Negative numbers such as -1 carry special meaning of executing continuously.

Pause (Default: 0)
Minimum time gap between two adm execution cycles, in seconds.

Silent (Default: 0)
Set to suppress messages from the simulation

As any other EDEN variables, these variables can be redefined during simulation.

To start or continue the simulation, type:

```
startClock = 1;
```

To stop the simulation before the preset time reaches, type:

```
stopClock = 1;
```