8 Future Research

8.1 Object-oriented EDEN

Object-oriented languages emphasize the data abstraction while definitive languages emphasize the abstraction of relationships among objects.

Some definitive notations have some structured data type features, like the **open shapes** in DoNaLD. However, an object-oriented approach to the organization of definitions is not currently being researched.

The disadvantages of lacking object-oriented features are illustrated by the absence of structured data types, like *array*, in EDEN.

In EDEN, we can define a definition whose value is a list, but we cannot assign a definition to one of the items of a list. For instance, the definition of a particular item of a list, L, is illegal in EDEN:

L[2] **is** *expression*;

because L is considered as a single data item.

To get around this problem, the definition of L must be defined to, for example:

```
L is [&L_1, &L_2, ... , &L_10];
```

and the definition of L[2] must be rewritten as:

```
L_2 is expression;
```

Thus, the value of L[2] can be referred to as `L_2` or `*L[2]` (the * is an indirect reference operator).

Doubtless, this approach is semantically complicated and decreases the readability of the definitions.

Object-oriented definitions would hide this complexity. The major benefit of an object-oriented approach is that definitions can be inherited from the super class by derived classes.

We use the term **entity** here to refer to a class of definitive objects which is an abstract collection of primitive entities (or sub-entities), their definitions (which describe the relationships among sub-entities) and, perhaps, actions. Instead of using the term "**object**", we use the term "**entity**" to emphasize that it is not an usual object class as in object-oriented languages which glues data and related functions.

The proposed syntax of an entity class specification is informally:

entity *class-name* **based on** *super-class-name*
{
    entity-class-name sub-entity-name ;
    ...
    definitions
    ...
}

where "**based on** *super-class-name*" can be omitted if the entity is a generic entity class. Definitions are preceded by the declaration of sub-entities. Consider the window specification example in §4.1, a window class can be specified as the entity window:

```
entity window
{
    int             top, bottom, left, right;
    int             width, height;
    scroll_bar      vert_bar;
    ...

    right           is left + width;
    bottom          is top + height;
    vert_bar.left   is right – vert_bar.width;
    ...
}
```

Those undefined sub-entities, such as top and left, can be defined after an *instance* of the window entity class has been created. An instance of an entity is created as:

entity-name *instance-name* ;

For example, "scroll_bar vert_bar;" creates an entity (scroll_bar) instance called vert_bar.

The object-oriented semantics could result in a strongly typed language. This means that the resultant language may run faster (because type checking can be done at compile-time). In addition, the instances of the same entity class can share code of common generic definitions leading to smaller object code size.

To preserve the purity of a definitive representation, message passing techniques are not proposed in this prototype since messages and methods, like imperative procedures, may introduce interference.

8.2 Higher-order Definition

There are various ways in which the use of definitions might be generalised. One possible option is to introduce techniques to enable sets of definitions to be themselves defined implicitly. The motivation for such techniques is that we need to define generic objects such as the regular n-gon in §8.3 below.

A *higher-order definition* (HOD) is a definition of a set of definitions. To construct a HOD, we need operators that return sets of definitions:

$$\textit{HOD-op}: \quad \textit{value} \times \textit{value} \times \cdots \rightarrow \textit{set-of-definitions}$$

An example of a HOD-op is:

if (*condition*)
then { *definition-set*$_1$ }
else { *definition-set*$_2$ }

where the **if**...**then**...**else**... operator returns a set of definitions according to the value of the condition expression.

For instance,

HigherOrderDef **is**
if (C)
then { F **is** $U + V$; }
else { F **is** $X + Y$; }
;

In this example, we denote this higher order definition as *HigherOrderDef*. Depending on the value of C, F has different definitions. Note that S(*HigherOrderDef*) = {C}, i.e. *HigherOrderDef* is not dependent upon F, U, V, X or Y.

An alternative form may be:

if (*condition*)
then *expression*$_1$
else *expression*$_2$

Thus, F becomes

$$F \textbf{ is if } (C) \textbf{ then } U + V \textbf{ else } X + Y; \tag{8-1}$$

where F depends on either $\{U, V\}$ or $\{X, Y\}$, but not both. In neither case, does F depend on C.

$$\mathcal{S}(F) = \begin{cases} \{U, V\} & \text{if } C=\textbf{true} \\ \{X, Y\} & \text{if } C=\textbf{false} \end{cases}$$

(8-1) is different from

$$F \textbf{ is } (C) \ ? \ U + V : X + Y; \tag{8-2}$$

where "? :" is the C-style "if...then...else..." operator.

F, in (8-2), depends on all C, U, V, X, and Y. This means that it introduces extra data dependency to F.

These higher-order definitions are very useful in specifying definitions conditionally in a program.

8.3 Macro Definitions

Another kind of higher-order definition is *macro definition*:

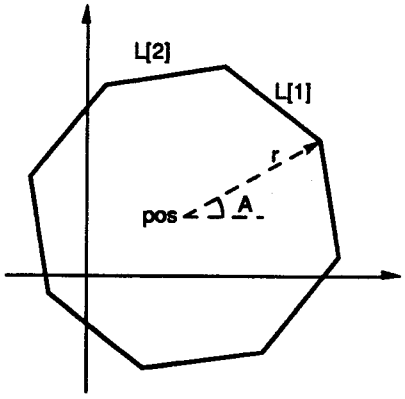
```
for $i = 1..3 { a[$i] is b[$i] + c[$i]; }
```
(8-3)

where $i denotes a control variable which exists within this statement. The occurrences of $i are substituted by the numbers from 1 to 3. This means that (8-3) is an abbreviation of 3 definitions:

```
a[1] is b[1] + c[1];
a[2] is b[2] + c[2];
a[3] is b[3] + c[3];
```

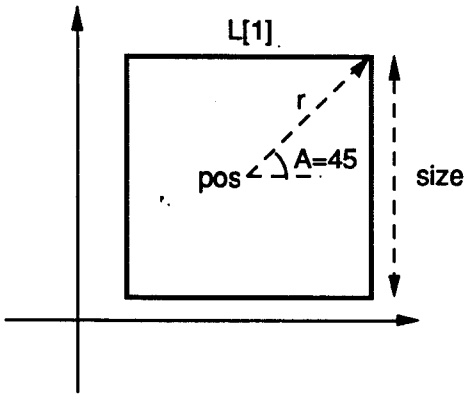
Combining this with the idea of **entity**, we are able to specify a polygon entity class:

```
entity polygon
{
    int     n;      /* number of edges */
    real    r;      /* radius of the circle bounding the polygon */
    point   centre; /* position of the centre of the polygon */
    real    A;      /* angle of rotation about the centre */

    if (n >= 3) then {  /* make sure there are at least 3 edges (a triangle) */
        line L[n];      /* lines bounding the polygon */

        L[1] is a_line(centre + polar(r, A), centre + polar(r, A + 2 * PI / n));
        for $i = 2..n {
            L[$i] is a_line(L[$i-1].p2, centre + polar(r, A + $i * 2 * PI / n));
        }
    }
}
```



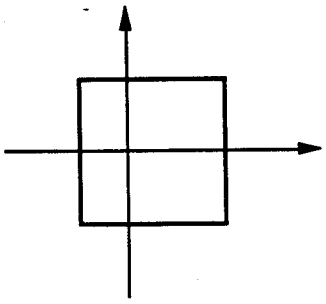
The entity class square can be derived from polygon as below:

```
entity square based on polygon
{
    real    size;           /* size of the square */

    n is 4;                 /* a square has 4 edges */
    A is PI / 4;            /* rotate 45 degree */
    r is size / sqrt(2);    /* the radius of the bounding circle is
                               calculated from the size of square */
}
```



A square S, whose centre and size are respectively (1,0) and 6, can then be specified as:

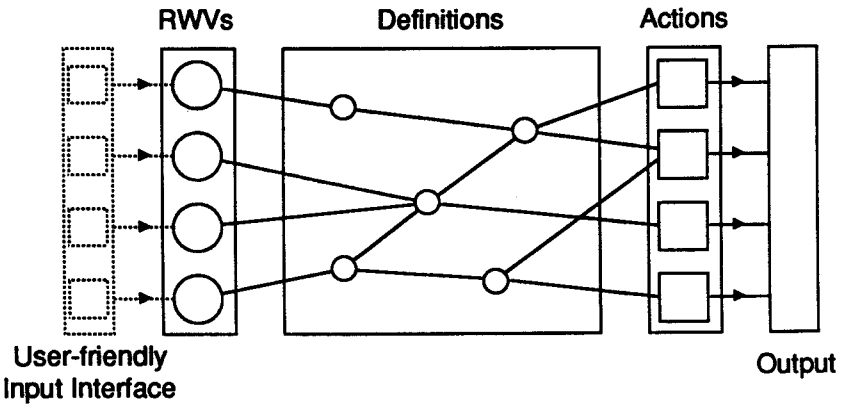
```
square      S;

S.centre    is cartesian(1, 0);
S.size      is 6;
```



8.4 Input Interface Gap

In EDEN, we are able to specify output actions, but we cannot specify input actions. For example, we cannot specify an action to keep track of the mouse position. This is because actions are invoked only if the related variables have their value changed, but the mouse position is not an EDEN variable. The EDEN interpreter does not have the acknowledgement of input from the conventional operating system.

The current method is to call a procedure which executes an iterative loop to fetch inputs. Then, according to the input, we modify the values of some RWV's, whereby we trigger other definitions or actions.



It might be more appropriate to have a way of specifying input that is consistent with the way output is specified. To fill this input interface gap, we can extend the idea of **action** to **latent action**, where each latent action has a **guard**, expressed as a boolean expression, to govern the execution of the action procedure; the definitive system has to evaluate the *guard expression* repeatedly to determine whether the action procedure should be executed. This extension

of action requires no modification of the original operating system. The suggested syntax of latent action is:

action *name* : *guard* { *statement-list* }

where *guard* is a boolean expression. For example:

```
action k : keypressed()
{
      input_char = getchar();
}
```

where `keypressed` is a boolean function which returns true if the user has pressed a key.

Unlike an event handler which is normally idle unless the particular event has just occurred, the latent action demands active monitoring of the guard expression. Latent action procedures can also run concurrently (i.e. interleave arbitrarily) but event handlers cannot.

Since the guard expression can be arbitrarily complex, the latent action is more flexible than the event. (Although the evaluation of the guard can be activated by the input events, the user should have no knowledge about this low level detail.)

In addition, many latent actions can have the same condition expression where each latent action does different jobs concurrently.

A problem of latent action is that they may interfere with each other; for instance, two or more latent actions may write to the same RWV. However, since the latent actions are running concurrently, we cannot predict which latent action has the final control to the RWV (note that actions also suffer the same problem since we cannot predict the order of execution of actions). It is the responsibility of the user to avoid writing such badly-behaved actions.

8.5 Re-definition within Action Body

At the moment, the EDEN interpreter does not allow definitions in action bodies. For example, the following action is prohibited:†

† At the time of writing, the latest implementation of EDEN interpreter has removed this restriction.

```
proc HigherOrderDef : C        /* an action form of higher order definition */
{
    if (C) {
        F is U + V;            /* illegal in the current implementation */
    } else {
        F is X + Y;            /* illegal */
    }
}
```

Another improvement of the EDEN language is to enable and to encourage the user to use definitions in the function (/procedure/action) bodies.

Two modifications can be made to the language:

1. To allow *global* or *local* definitions to be specified in an action body. On contrasts to a *global* definition, which is accessible to all data structures, a *local* definition is bound by the scope of the action body.

2. To eliminate/minimize the use of imperative statements, especially iterative loops.

Assigning a constant value to a RWV is the same as specifying a definition whose formula expression is a constant; e.g. "A = 1" is equivalent to "A **is** 1". If we create the outfix operator (a pair of vertical bars):

| *expression* |

where the expression enclosed by the bars will be evaluated and substituted into the formula expression before the definition is defined. For example:

X **is** | X + 1 |;

is equivalent to "X = X + 1; ". So, if X was 3, the above definition becomes "X **is** 4;".

8.6 Low-level Definitive Machine

Instead of extending EDEN to support advanced features, the opposite approach is to extract a set of primitive operations of the definition manager. The aim is to define a low level definitive machine so that definitive systems can be built directly on top of it without the intermediate EDEN level.

This low level definitive machine may be in the form of library functions — a set of pseudo machine codes.

The internal pseudo machine of the current EDEN interpreter implementation could be the prototype of a definitive machine. **Latent action** is, perhaps, the primitive in this machine, because it is the raw form of higher-order definitions (i.e. all actions and definitions, including higher-order definitions, can be compiled into latent actions).

The major difficulty of designing such a machine is to decide which operations are primitive and what the basic data types shall be. We should also decide which management scheme(s) the definition manager uses.

An ambitious aim of the design of the low-level definitive machine is to implement it in hardware.