6
Implementation of DoNaLD

CAD software packages usually contain a rich set of utilities to help the human designer to analyse his/her design. However, more importantly, these packages provide an interactive environment to aid the designer to access these utilities. Modern commercial packages (e.g. **DOGS** [PAFEC-1; PAFEC-2]) list all the utilities on a menu so that the user can select the appropriate utility with a mouse, a digitizer or so forth.

Usually, a CAD package produces a graphical feed-back to the user almost immediately after he/she makes some modification to the design. This speeds up the design process cycle because the user can analyse and correct errors in the design more efficiently without actually making a prototype of the product.

Interactive CAD is only powerful and intelligent as an input medium for drawings; but the internal representation of these drawings lacks the structural information for subsequent geometric transformation (c.f. [To87]).

Some "what-you-see-is-what-you-get" (WYSIWYG) programs, like Apple **MacDraw**, allow the user to group several objects to form a single large object. The user can, then, manipulate this object with the utilities provided. The grouping of objects effectively maintains some linear relationships among objects, e.g. the sizes of two objects are in a fixed ratio. However these relationships are very restricted and not enough for the design purpose. For instance, the relationships:

"the length of the object A is always the square of the length of the object B"

cannot be maintained by these packages.

The alternative to interactive graphic input is to describe a model in a "procedural" design language. Some packages allow the user to define parametric symbols which are represented by some programs comprising appropriate sequences of drawing instructions. The execution of these programs

creates the desired result. Objects with minor variations are created from the generic objects with few parameters varied at the time of construction. Usually, these programs are written with a text editor.

The L.E.G.O. [FP88] system takes this approach. L.E.G.O. is an interactive object modelling system based on geometric constructions. The LISP-like program contains a sequence of geometric constructions, such as **point, line, sphere, intersection**, etc. For example,

(**point** *x y new_name*)

constructs a point, called *new_name*, with the coordinates *x* and *y* by recording its features in the L.E.G.O. symbol table and by drawing it on the screen. Graphical objects are described directly in geometric terms and can be manipulated interactively. L.E.G.O. also includes a graphical programming environment which allows the user to write a program through a graphical interface as well as a textual interface.

However, for any design activity involving exploration of alternatives, frequent changes, integration of 2D and 3D modeling, and flexible quantitative evaluation, the parametric object technique is still restricted because it cannot maintain relationships dynamically.

The other alternative is to describe the relationships in a declarative language and let the system maintain these relationships.

Constraint-based graphics systems described in the literature [BD86; SM88] accept object definitions expressed in terms of geometric relations between object elements. These definitions are subsequently transformed into analytical descriptions of graphical primitives to be displayed or plotted. However, solving a set of constraints relies on general-purpose numerical methods or techniques based on general theorem proving.

Another approach is to use definitive languages. The definitions in the definitive languages are similar to uni-directional constraints. However, the definitions can be solved more efficiently than constraints. §4.2 shows how definitions supported by actions in EDEN can be used to handle constraints. Although the constraints that can be solved are rather restricted, other constraint-based systems suffer from similar limitations.

Three reasons that make definitive languages a good approach in CAD are:

1. Definitions express the objects (including non-geometric objects) directly in mathematical terms.
2. Objects can be manipulated interactively.
3. The machine-dependent graphics can be hidden. The transformation of definitions to geometric graphics (or other kind of media) is handled by the machine automatically.

6.1 Definitive Languages for Graphics

Several researches on developing definitive notations for describing geometric objects have been carried out at University of Warwick.

DoNaLD [BABH86] has been designed as a popular graphics system illustrating definitive principles. The underlying algebra is based upon **real, integer, point, line** and **shape** variables, where **point** and **line** values are geometric Cartesian or polar coordinates and line segments in the plane, and a **shape** value is a line drawing composed of a set of points and lines. DoNaLD has two kinds of **shape** variable: virtual **shape** variables whose value is defined by a single formula of type **shape**, and **open shape** variables, which comprise a set of **point, line** and **shape** variables which are used to define component points, lines and subshapes.

ARCA [Be86a] is a more sophisticated notation than DoNaLD. It is primarily aimed at the design and manipulation of a class of combinatorial diagrams with a rich and clearly defined mathematical semantics. The underlying algebra comprises **integer, vertex, colour** and **diagram** data types, which respectively represent scalars, Euclidean coordinates, abstract incidences, and realization of coloured digraphs. Although the ARCA system, as presently developed, deals primarily with geometric aspects of diagram manipulation, the richness of the associated semantic framework naturally suggests its consideration as a model for a prototype CAD system [Be87].

6.2 Experimental Implementation of DoNaLD

The preliminary design of DoNaLD was completed in 1986, but no DoNaLD system had been implemented since the creation of this notation. We decided to implement it using EDEN interpreter as its back end in order to test the potential

of EDEN in implementing other definitive notations. More important, a workable DoNaLD system allows us to demonstrate the use of definitive principles in a graphical application which is the first step towards a CAD application.

The DoNaLD implementation has two parts: the DoNaLD to EDEN translator front end, and the EDEN interpreter back end. The DoNaLD to EDEN translator front end translates DoNaLD definitions to EDEN definitions and lets the EDEN interpreter maintain these definitions and generate graphics output. The two parts are joined by a UNIX command pipe line.

Appendix D lists the source code of the DoNaLD to EDEN translator and Appendix E the supporting code for the EDEN interpreter back end.

We selected a subset of DoNaLD as our first experiment. The subset includes the data types: **integer**, **point** (**integer** × **integer**), **line** (**point** × **point**), and **open shape**. All variables must be declared before specifying their definitions. The **open shape** variables are analogous to "directories" in the UNIX file system. Suppose S is an **open shape** variable with the components **points** P and Q, these components are respectively named as S/P and S/Q where the slash / separates the open shape variable name and its component variable names.

The DoNaLD definitions can be mechanically translated into EDEN definitions by simply converting from the DoNaLD identifier name convention to an EDEN convention. Every DoNaLD variable can be assigned an unique EDEN name during the translation. For instance,

DoNaLD name	***EDEN name***
S	_S
S/P	_S_P
S/T/Q	_S_T_Q

But, before the translation, the DoNaLD to EDEN translator must do the semantic checking because, unlike EDEN, DoNaLD is a strongly typed notation.

All DoNaLD data types, except **integer**, are represented by lists. The first item of the list is a character constant indicating the data type which the list represents. For example,

DoNaLD type	*EDEN type*
integer	*integer*
point	['C', *integer, integer*]
line	['L', *point-type, point-type*]
open shape	['O', *pointer-to-component-variable...*]

The **open shape** list is a list of pointers that point to all component variables. For example, the declaration of open shape S:

```
openshape S {
      integer    i, j, k
      point      p, q
      line       L
}
```

is translated to the EDEN definition:

```
_S is ['O', &_S_i, &_S_j, &_S_k,
            &_S_p, &_S_q, &_S_L ];
```

The only purpose of specifying such a definition is to set up the hierarchy structure of the open shape, i.e. to create the structural dependency. In the current prototype implementation, these references to components of the open shape (&_S_i, &_S_j, etc.) are used simply to record dependency relations. In a future extension, they might have a greater role, e.g. when implementing the shape operators which access the component parts of the open shape (cf. [BABH86] for the description of DoNaLD notation).

6.2.1 DoNaLD To EDEN Front End

The DoNaLD translator consists of a lexical analyzer, a parser, a semantic analyzer, and an EDEN code generator.

The lexical analyzer recognizes DoNaLD tokens and sends them to the parser. The parser, then, builds a parser tree. The translator has its own symbol table. The parser tree is decorated by the semantic analyzer while checking for type clashes in the expression. The semantic analyzer also resolves overloaded operators, such as "+" (**integer** or **point**), into different operators, such as scalar- and vector-addition respectively. If there is no error, the EDEN code generator can walk the parser tree to generate EDEN codes.

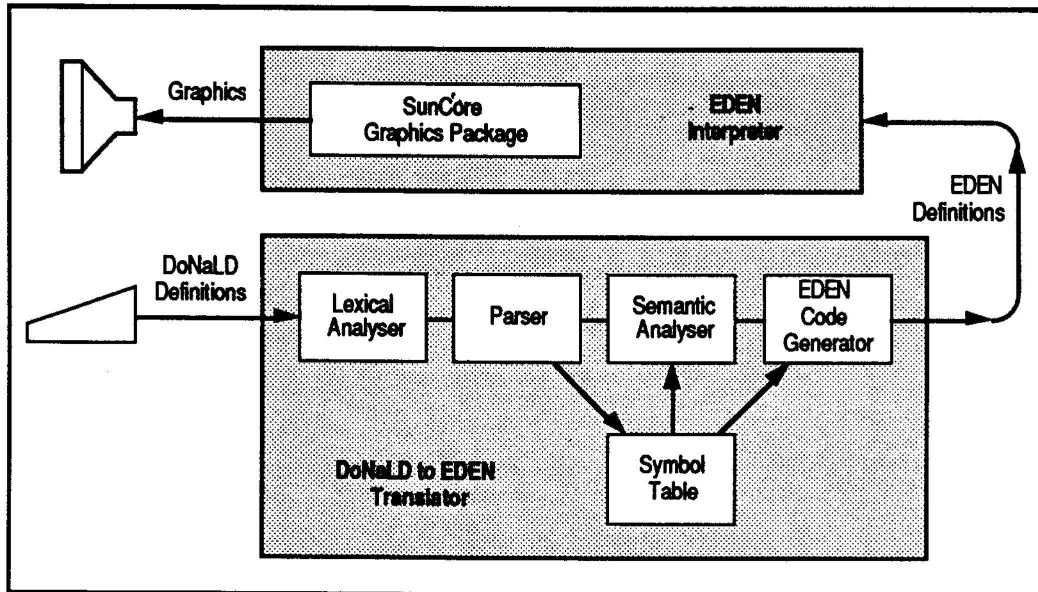


Figure 6-1: A diagram of the DoNaLD system

The DoNaLD system consists of a DoNaLD to EDEN translator front-end which translates DoNaLD definitions into equivalent EDEN definitions. The EDEN definitions are passed to the EDEN interpreter through a UNIX pipe. The EDEN interpreter maintains the definitions and generates graphics output.

To generate graphics, each **point** or **shape** variable is associated with an action. The action specifications can be generated at the time of declaration of the DoNaLD variables. To systematically label the actions, the action procedure names always begin with "`P`" and are followed by the object names, which are always preceded by the underscore "_" in EDEN.

Example:

DoNaLD code	*EDEN code*
point p	`proc P_p: _p { plot_point(&_p); }`
point q	`proc P_q: _q { plot_point(&_q); }`
line L	`proc P_L: _L { plot_line(&_L); }`
openshape S {	`S is ['O'];`
line M	`proc P_S_M: _S_M { plot_line(&_S_M); }`
}	`S is ['O', &_S_M];`

where S is an open shape, line M is one of its components. The DoNaLD and EDEN full names of M are S/M and `_S_M` respectively. The EDEN operator `&` returns a pointer that points to the variable referred to by the operand.

`plot_point` and `plot_line` are EDEN procedures which generate the graphic images. These two procedures call the standard **SunCore** [SUN88b] graphic routines to generate graphics. **SunCore** is a standard C library package

supplied with SUN workstation. It is an implementation of the ACM SIGGRAPH Core System [SIG79]. **SunCore** conforms to the level 3C (dynamic output with 3D scaling, rotation and translation) of the Core specification for output primitives, and to level 2 (complete input) for input primitives. This Core specification is similar to the GKS graphics standard [GKS85; GKS86; HDGS86; EKP84]. A comparison of GKS and CORE concepts is given in [En80].

We bind these C functions to EDEN names so that the EDEN interpreter can call them as if they are EDEN functions.

6.2.2 EDEN Interpreter Back End

The EDEN interpreter back end is responsible for initializing the graphics system, maintaining the definitions, and creating graphics output.

The DoNaLD operators can be directly implemented in EDEN. For example, the vector-addition operator can be implemented as an EDEN function `vector_add`:

```
/*      Remark: point = ['C', integer, integer]
**      We define the variables X(=2), and Y(=3) to index the two coordinates.
*/
X=2; Y=3;                      /* served as integer constants */

func vector_add                /* $1, $2 are two points */
{
        return ['C', ($1[X]+$2[X]),
($1[Y]+$2[Y])];
}
```

The EDEN back end calls **SunCore** graphics routines to generate graphics output.

6.2.3 SunCore Graphics Package

The **SunCore** graphics package is able to create a view surface which is a graphics window in the SunView window system [SUN88a]. On the view surface, we can create a viewport. A mapping can be defined to map *world coordinates* to the viewport's *normal device coordinates*. These initialization procedures and other basic function/procedure definitions are pre-programmed

and stored in a file init.e which will be loaded into the EDEN interpreter at start up.

A set of output primitives (and their associated attributes) that is arranged and labelled as a group is called a *segment*. **SunCore** is capable of creating two kinds of segments, namely *temporary* segments and *retained* segments. However, we are just interested in retained segments because they can be modified after they are created, and are suitable for implementing **point** and **line** type objects.

Retained segments have names (actually numeric identifiers) so that by placing output primitives in such segments, we can modify parts of the picture by deleting and recreating segments (which effectively replaces them) so that their images change. Retained segments are stored in the display list for later dynamic modification.

There is an unique retained segment associated with each **point** or **line** type variable. The segments are simply named as the addresses of the EDEN variables. Therefore, the DoNaLD declaration of **point** and **line** variables also generates dummy segments so that the action procedures can call either `plot_point` or `plot_line` to modify the particular segments later. For example,

DoNaLD code	***EDEN code***
point p	`create_retained_segment(&_p);`
	`close_retained_segment(&_p);`
	`proc P_p: _p { plot_point(&_p);`
}	

where the two **SunCore** function calls, `create_retained_segment(&_p)` and `close_retained_segment(&_p)`, creates the dummy segment (`&_p`) with no output primitive. The action `P_p`, when executed, will delete this segment and re-create it with appropriate output primitives that represent the point `_p` (i.e. DoNaLD point p).

6.3 Towards CAD

DoNaLD is a basic notation for describing graphics drawing; it does not fully fulfil the requirement of CAD applications. As a step towards greater sophistication, certain monitoring mechanisms might be appreciated to warn the user if some conditions are violated; for example, two objects overlap. This can

be done in EDEN by specifying definitions or actions to monitor the conditions. For example,

```
proc monitor_door_hits_desk: _door, _desk
{
	if intersects(_door, _desk)
		writeln("door hits desk!");
	/* else do nothing or clear the warning message */
}
```

where `_door` and `_desk` are two objects, and `intersects` is a boolean function that returns true if the two objects overlap.

We can extend the DoNaLD notation to support the translation of this monitoring mechanism. However, without the concept of the "extent of an object" in DoNaLD (objects are basically points and lines), the definition of the `intersects` function is tricky. Nevertheless, in the experimental prototype, we were able to specify actions to monitor some particular conditions (simply by by-passing the translator front-end) to illustrate this possibility. Possible extensions of the DoNaLD notation were discussed in [BY88].

6.4 Testing

The following UNIX shell script, `donald`, forms the complete DoNaLD system:

```
donald.translator | eden -n init.e
```

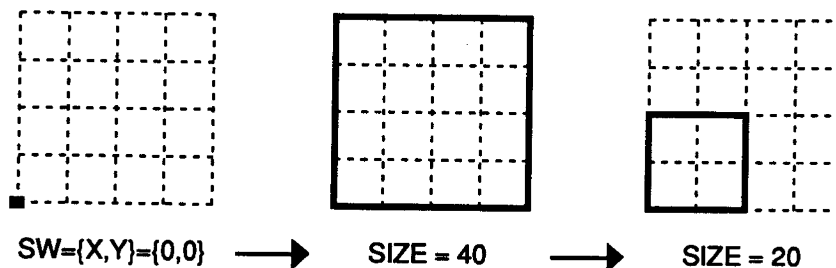
where `donald.translator` and `eden` are the DoNaLD to EDEN translator and EDEN interpreter respectively.

The `donald.translator` is a filter where DoNaLD specification is read from the standard input file and the EDEN code is output to the standard output file (i.e. pass to `eden`).

The `-n` option tells `eden` to run in non-interactive mode (i.e. no prompt). The file `init.e` stores all supporting EDEN codes which is loaded before `eden` reads the standard input (i.e. the output of `donald.translator`).

The following DoNaLD specification is typed in to test the DoNaLD system. The graphics output is shown below. Note that the DoNaLD = operator specifies definitions and the text that follows # is comment.

```
openshape SQUARE {

    int      X, Y                  # the X,Y coord of top left corner
    int      SIZE                  # size of the square
    point    NE, SE, SW, NW        # 4 corners
    line     N, S, E, W            # 4 edges

    # — definitions —

    N = [NW, NE]                   # operator [] returns a line
    E = [NE, SE]
    S = [SE, SW]
    W = [SW, NW]
    SW = {X, Y}                    # operator { } returns a point
    SE = {X+SIZE, Y}
    NE = {X+SIZE, Y+SIZE}
    NW = {X, Y+SIZE}
}

# since SQUARE's X, Y, & SIZE have not been defined,
# no graphics yet
# ... other definitions

within SQUARE {   # within change the scope to the open shape
    X = 0
    Y = 0         # SW has enough info. to determine its coord ...
                  # ... only the point SW is shown.
    SIZE = 40     # now, we have enough information to ...
                  # ... compute and draw  all points and lines
}

SQUARE/SIZE = 20        # now we change square's size to 20 ...
                        # ... the graphics is updated immediately
```



6.5 Conclusion

In this experiment, we successfully implemented a subset of DoNaLD. It illustrates how a definitive notation can be implemented using EDEN as a low-level definitive machine.

Though EDEN's external function binding method is quite ad hoc in the current implementation, it enables us to make use of some existing routines, such as the **SunCore** package.

We only used a fraction of **SunCore**'s features, such as 2D graphics and simple output primitives, in this experiment. There are many other features, e.g. 3D graphics, transformation, and input primitives, we have not used. However, the result is promising — we can build simple graphics primitives into complex objects, for example, a square.

EDEN's actions are able to monitor conditions. Using some constraint maintaining techniques discussed in §4.2, we can also impose or maintain constraints (c.f. [BY88]).

Although DoNaLD was not particularly designed for CAD applications, it matches some fundamental requirements of CAD. [BC88] summarized the recent research on a definition-based approach to the implementation of CAD software.

6.6 Related Works

Constraint-based graphics systems express graphical objects in terms of geometric relations — constraints — between object elements. Various techniques for solving systems of nonlinear equations have been used to satisfy sets of constraints. For examples, **Sketchpad** [Su80] used the relaxation method, and **Juno** [Ne85] used the Newton-Raphson method. These systems use numerical methods which might be too slow for interactive applications [FP88].

Some systems, such as **Metafont** [Kn79] and **Ideal** [Va81], restrict constraints to those which can be expressed by linear equations to avoid the general-purpose numerical method.

Coral [SM88] is an object-oriented constraint-based system based on uni-directional constraints which are very similar to EDEN's definitions. The major difference is that **Coral** can define object classes with inherited constraints from subclasses. Like EDEN, constraints can be modified at run-time and the graphical image is immediately updated.

PictureEditor[KNK89] is another constraint-based picture drawing system. The system treats constraints in a way similar to definitions as in DoNaLD and since it converts constraints to construction operations (c.f. L.E.G.O., a construction-based graphics system[FP88]), it does not resort to numerical methods for constraint satisfaction.

The variables of **Charli**[CW89], a graphics scripting language for modelling and animation, can be assigned not only values but also expressions which may contain *tracks* that are similar to the data dependency relations in EDEN.