ALGEBRAIC SPECIFICATION OF MACINTOSH'S QUICKDRAW
USING OBJ2

A. T. Nakagawa* K. Futatsugi S. Tomura
T. Shimizu†
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba Science City, Ibaraki 305, JAPAN

Abstract
We have described QuickDraw, a typical graphics package, using OBJ2, a powerful algebraic language now in the phase of experimental use as a specification language. The results testify the applicability of OBJ2 to some practical problem domains, as well as the premonitioned advantages the use of formal specification techniques brings. The work also sheds some critical lights upon the design of QuickDraw; we detect incomplete procedure definitions, and find imprecise the classification of procedures.

keywords — formal specification, algebraic language, graphics package

1 Introduction
Formal specification techniques are deemed to lead to rationalisation of requirement specifications/system design phases of software development [26]. Algebraic specification is one of the most promising among those techniques [1,2,7,15,13], and has compatibility with other techniques [21,27,29]. We have been probing that approach to formalisation to create an adequate environment to support the software development in toto. Our special emphasis is conferred upon the design phase and its interface with programming, since the consistency of the design itself, and the validity of the programmed codes as set against the design, are the twin pillars of correct behaviour of a system [26].

OBJ2, initially developed as a high-level programming language [4], offers us the first approximation to the formal specification tools in our mind(s). It fully captures the gist of “specification by algebra”, enables us to describe executable specifications — needless to say, since it is a programming language —, and provides various features of enormous use in its structure as a language. So far the language has mainly been used experimentally. To see if OBJ2, and by extension algebraic specification in general, is a sufficient, efficient, flexible, and friendly — strictly in that order — means of specification, it has to undergo experiments in real life. This is the prime motivation of the efforts that engendered this paper.

In determining an exemplary system to be written in OBJ2, we turned upon the Macintosh Programmer's Guide (the Guide, hereafter) [19]. First we narrowed our attention down to a window system, which is a common way to organise man-machine interface, as well as presents a design problem complicated and large enough to try on. The Macintosh system is among the representative ones adopting a window system. The Guide imparts us of enough of the interiors of the system. The work covered here dealt with the QuickDraw package, chosen since the package is by and large self-contained, and every system using graphics, invariably a substratum of window operations, has its equivalent of QuickDraw, thus marking a target, both of convenience and of significance, for specification languages.

This paper by itself does not attempt systematically to analyse OBJ2 as specification language. Rather, we try to show how algebraic specification adapts itself to problem domains; also how the forte of the technique shows its incarnation, in the way it lays bare the defects or shortcomings of informal descriptions, in this case represented by the highly acclaimed programmer’s manual.

The next section briefly explains OBJ2 with examples. In the section 3, we show the sketch of QuickDraw as originally explained in the Guide. The section 4. summarises our rendition of QuickDraw, taking some issues that confronted us. In the section 5, we lend some critical comments to the original documentation of QuickDraw, based on our experience.

2 OBJ2 in Brief
OBJ2 is a state-of-the-art programming language whose descriptive level is of such height as to be regarded as an attractively powerful specification language. In our research discussed here OBJ2 works as the latter self. The language is based upon algebraic semantics, composed of set of types, or sorts, in the usage of OBJ2, and equations among terms of each sort. It gives denotational semantics as initial algebra [12], on the one hand; on the other, the equations are operationally regarded as rewrite rules [17,28]. Some of the advanced features incorporated are [3,4,5,6]:

1. sophisticated mechanism of parameterisation.
2. support given to ordered sorts.
3. a powerful rewrite engine that recognises such properties as associativity.
4. highly flexible term parsing, allowing mixed term constructs.

2.1 Parameterised Modules

Like any serious language, OBJ2 allows structured programme constructions with set of components, in its case called modules. Unlike ordinary languages, OBJ2 allows highly expressive, disciplined parameterisation of modules [8,9,10]. An example illustrates the flavour of the game:

```
theory TRIV is sort Elt. endth

object ALIST[INDEX :: TRIV, VAL :: TRIV] is
保护ing BOOL .
sort Alist ErrVal .
subsorts Elt. Val. < ErrVal .
op empty -> Alist .
op put : Elt. Val. Elt. INDEX Alist -> Alist .
op [], : Alist Elt. INDEX -> ErrVal .
opp undef -> ErrVal .
vars I : Elt. INDEX .
var V : Elt. VAL .
var A : Alist .
eq ErrVal : put(V, I, A)[I'] =
  if I =I' then V else A[I'] else .
endobj
```

Example 1.

Object and theory are the two kinds of modules. Ignoring details, this example denotes:

1. There is a requirement that a certain sort, tentatively named Elt, with no restriction upon its property, must exist (the theory TRIV).
2. There is a sort Alist that behaves like an association list, with a constructor put and an inquiring operation [] . This sort, when used, must be supplied with an index sort and a data sort, each of which may be any sort that exists, the requirement of their qualifier, the theory TRIV (the object ALIST).
3. A sort BitPattern is constructed as two-dimensional Alist, indexed by integers and containing bits. BIT is an object defined elsewhere.

```
object BITPATTERN is
  protecting Alist Elt. INDEX .
endobj
```

```
* (sort Alist to BitPattern).
```

Expressions, constructed by recursively instantiating parameters. In a module expression a sort or operator can be renamed, as is Alist by BitPattern here. Thus the parameterisation of OBJ2 allows you to:

1. Parameterise objects.
2. Dictate requirements upon parameters, using theories.
4. Make actual objects to order.

2.2 Ordered Sorts

A useful feature of OBJ2, in terms of algebraic semantics, is that it allows sort ordering. Apart from providing an inheritance mechanism in the sense in object-oriented languages [16], this is a elegant way to define error/exception-handlings. As for inheritance, the example

```
object FIGURE is
  protecting INT
  protecting BITPATTERN .
endobj
```

.sorts Oval, Wedge, Rect, Figure, Picture .

op bits : Picture -> Bitimage .

... .
endobj
```

Captures the power of sort ordering. Here we declare five sorts Oval, Wedge, Rect, Figure, Picture, and say the latter two are supersorts of the other three. Then an operation bits upon the sort Picture is equally applicable to Oval, Wedge, and Rect; the same goes for area upon Figure. Expressed in the terms of object-oriented languages, therefore, this sort ordering allows multiple-inheritance.

A sample of error handling scheme is already given in Example 1. In the object ALIST, the data sort, whatever it is, has a supersort ErrVal, with undef as its own constant. This supersort is used to deal with invalid reference. In a somewhat similar vein, partial operators can also be defined. Although the introduction of sort ordering raises problems such as operator overloading, the following conditions ensures the existence of the initial algebra, the unique model satisfying the specification [11].

1. For each operator σ, for each arity s1...sn and coercity s, for every n-length list of sorts s1,...,sn such that σi < si for i = 1, ..., n there is a least rank s0, s0,...,sn, s0 that satisfies σi < s0i, s0i < si, for i = 1, ..., n and σi < s.1

2. For each sort s, there is a maximal sort s_max such that (1) s < s_max and (2) s < σ implies σ < s_max.

1The relation < is reflexive, i.e., s < s for any sort s.
These conditions are hard, in algorithmic respect, to confirm, yet in normal usage unlikely to be violated.

2.3 Associative, Commutative Pattern Matching

Some of the binary operations have such properties as associativity, commutativity, idempotence, and/or have an identity. If you think of equations as term rewriting systems, as does OBJ2, without regard to these properties it is awkward to use some numeric or set theoretic operations [20]. OBJ2 allows you to ascribe those properties to binary operations, as well as to prescribe the order of evaluation.\(^2\) An example of this feature is the parameterised object SET.

\[
\text{object SET[Telm :: TRIV] is}
\]

protecting BOOL.

sort Set.

op phi : -> Set.

op omega : -> Set.

op [ ] : Elt -> Set.

op \_[ ] : Set Set -> Set.

[associative commutative identity: phi].

op \_[ ] : Set Set -> Set.

[associative commutative idempotent identity: omega].

op [ ] [ ] : Set Set -> Set.

op [ ] : Set -> Set.

op [ ] : Set Set -> Set.

op [ ] : Elt Set -> Bool.

vars E E' : Elt.

vars S S' S'' : Set.

eq Set : S \& S = phi.

eq Set : \{ E \} \& \{ E' \} =

if E == E' then \{ E \} else phi fi.

eq Set : S \& phi = phi.

eq Set : S \& (S' \& S'') = (S \& S') \& (S \& S'').

eq Set : S || S' = (S \& S') \& S \& S''.

eq Bool : E in phi = false.

eq Bool : E in (E' \& S) =

if E == E' then true else E in S fi.

endobj

Example 3.

Here the symmetrical difference $S$ is declared associative, commutative, and has the empty set $\phi$ as identity; the intersection $\&$ associative, commutative, idempotent, with the universal set $\omega$ as identity. Without these properties it is difficult to get canonical forms, or have two forms with identical elements, such as $\{ E \} \& \{ E' \}$ and $\{ E' \} \& \{ E \}$, be treated uniformly.

2.4 Mixfix Operators

Operators of OBJ2 have an unusual syntactic feature: they can be prefixed, postfixed, infixed, or neither. The examples already given carefreely use this flexibility. For instance, the operator put of the object ALIST in Example 1. has a standard functional syntax, put(V, I, A); the inquirer [I] is used in the form of A[I]. Most of the operators in Example 3 are infixed, like $S \& S$; the unary singleton constructor $\{ \}$ is outfixed, in the form of $\{ E \}$. OBJ2, in short, implements a complete context-free grammar on the run. The ambiguities this may cause are resolved in a rather harsh way; a term is acceptable only if it has a unique parse. In a non-trivial example, it is anyway unlikely that erie mixture of syntactic constructs would be desired. In our current work, we use syntax other than the functional one, which is default, only when the tradition dictates, as in the case of set theory.

3 Overview of QuickDraw

QuickDraw is the basic graphics package of Macintosh Tool- box, consisting of procedures generating, manipulating, and examining graphic entities widely used by other packages. Its Programmer's Guide [19] details, in its own way, the basic concepts, their realisation, the data types, the variables, and the interfaces and functions of the procedures. The procedures and data types are written mainly in Pascal.

3.1 Points and BitMaps

The basic concepts of QuickDraw are founded on a coordinate plane, a two-dimensional grid. On the plane are points, defined by integral coordinates; rectangles, defined by a pair of points; regions, which are sets of arbitrary boundaries. These concepts are used to define the graphic entities and the operations upon them.

Bit images actually represent graphic images; a bit image is a matrix of pixels, or bits. A bit image coupled with a co-ordinated plane makes a BitMap; they are juxtaposed in such a way that each pixel of the image falls among four points of the plane. BitMap is the main data type on which rest definitions of drawing operations.

3.2 GrafPort

The drawing environment of QuickDraw is defined as grafPort, a record data type that contains such fields as the target device, the local coordinate plane, the BitMap to draw on, the drawing pattern, and the text size. There can be many grafPorts at a time, at most one of which is current; from the current grafPort are retrieved the crucial information at the time of drawing. Here we present the precise specifications concerning grafPort, for it makes the centre around which QuickDraw procedures circle.

VAR thePort : GrafPtr;

TYPE GrafPtr = " GrafPort;" TYPE GrafPort = RECORD

device : INTEGER;

portBits : BitMap;

portRect : Rect;

visRgn : RgnHandle;

336
331

3.4 QuickDraw Routines

The procedures provided by the package are classified as:

- GrafPort Routines
- Cursor Handling
- Pen and Line Drawing
- Text Drawing
- Drawing in Color
- Calculations with Rectangles
- Graphic Operations on Rectangles
- Graphic Operations on Ovals
- Graphic Operations on Rounded-Corner Rectangles
- Graphic Operations on Arcs and Wedges
- Calculations with Regions
- Graphic Operations on Regions
- Bit Transfer Operations
- Pictures
- Calculations with Polygons
- Graphic Operations on Polygons
- Calculations with Points
- Miscellaneous Utilities
- Customizing QuickDraw Operations

4 QuickDraw in OBJ2

We have made two constraints on our specifications:

1. Use all the globals variables, pointers exclusive, given in the Guide.

2. All the procedures given in the Guide have to be definable.

In other words we tried to stick to the basic scheme of the implementation of QuickDraw as described by the Guide. This was a controversial decision; we need not, and maybe should not, follow the particular implementation of the design, since we are trying not to compile references manuals for programmers, but to see the pros and cons of design techniques; in this regard we need only to assemble a functional equivalent of QuickDraw. For a start, however, it may be safe not to disembowel the imitatee. The constraints posed a
not that we deem the Guide to be inadequate, confusing, or some omission, imprecision, and deficiency, figuring out the power, applicability, and limitations, if any, inappropriate in any other way. The primary purpose is to Our intention is rationally to reconstruct the package in natural language descriptions; if it were not the case, the whole purpose of formal specifications would be lost.

4.1 Overall Structure
The structure of our specification has four layers.

1. data types.
2. global variables.
3. states.
4. drawings.

The bottom layer defines data types and operations upon them, i.e., abstract data types. The next layer, global variables, mainly deals with current values. Values of those variables, coupled with physical devices, such as the screen, make the third layer, state manipulators. Finally, at the top layer of QuickDraw, rest drawing operations, changing the image of physical devices. This layering is devised for the particular purpose of the current work, yet most systems render naturally to this scheme, apart from the top layer.

4.2 data types
We assign an object to each data type. Most data types are no more than records of fixed construct, like point that consists of vertical and horizontal coordinates. Algebraic definition does not particularly suit such types. Consider the example:

```
object POINT is
    protecting INT.
    sort Point.
    op p : Int -> Point.
    ops getv geth : Point -> Int.
    ops putv puth : Int Point -> Point.
    vars V H V' H' : Int.
    eq int : getv(p(V, H)) = V.
    eq int : geth(p(V, H)) = H.
    eq Point : putv(V, p(V, H)) = p(V, H).
endobj
```

Example 4.

```
object POINT is
    protecting INT.
    sort Point.
    op p : Int Int -> Point.
    ops getv geth : Point -> Int.
    ops putv puth : Int Point -> Point.
    vars V H V' H' : Int.
    eq int : getv(p(V, H)) = V.
    eq int : geth(p(V, H)) = H.
    eq Point : putv(V, p(V, H)) = p(V, H).
endobj
```

Example 5.

```
object POLYGON is
    protecting POINT.
    protecting RECT.
    sort Polygon.
    op putbox : Rect -> Polygon.
    op putp : Point Polygon -> Polygon.
    op getbox : Polygon -> Rect.
    op getp : Polygon -> Point.
    var R : Rect.
    var G : Point.
    eq Point : getp(putp(P, G)) = P.
    eq Rect : getbox(putbox(R)) = getbox(G).
    eq Rect : getbox(putbox(R)) = R.
endobj
```

Example 6.

4.3 Global Variables
The global variables contain various current values, which make a significant part of the state. Due mainly to clinging

...
to the structure of the Guide, there are two distinct kind of variables in our specification. One is dynamic, realised, in Pascal, by pointer manipulation. The dynamic storage allocation is realised here simply by association list structure that appears in Example 1., like:

```pascal
object PORTLIST is
  protecting ALIST[IDX, GRAFPORT]
  * (sort Alist to PortList)
endobj

object RGNLIST is
  protecting ALIST[IDX, REGION]
  * (sort Alist to RgnList)
  op new : RgnList -> Idx
  op find : RgnList Idx -> Idx
  vars L : RgnList
  vars I : Idx
  eq Idx : new(L) = find(L, 0)
  eq Idx : find(L, I) =
  if L [ I ] == undef then 1 else find(L, s(I)) fi
endobj
```

Example 7.

GRAFPORT and REGION themselves are data types. The key of the lists is the sort Idx, defined as:

```pascal
object IDX is
  sorts ldx Errldx
  subsorts Idx < Errldx
  op 0 : -> Idx
  op s : Idx -> Idx
  op e : -> Errldx
endobj
```

Example 8.

The sort Idx is just like the set of natural numbers, with a constant 0 and a successor function s. The supersort Errldx contains a constant e, which we intend to use as an out-of-range value. Thus, if we construe Idx as a set of pointers, e can be used as null pointer. This is a good example of the power of sort ordering; an invalid or special element of a set can be distinguished just as such.

In the object RGNLIST, a operator new is defined so as to simulate memory allocation, an aberration caused by QuickDraw's asymmetrical provision of procedures that deals with pointer variables.

The other kind of global variables are static variables, just the reincarnation of data types. The object that comprise all the variables are:

```pascal
object GLOBALS is
  protecting GRAFPORT
  sortGlobals Var VarName
  subsorts GrafPort Idx Pattern Cursor BitMap Level
  Int PicList RgnList PolyList PortList < Var
  ops port portp whitepat blackpat greypat ltgreypat
dk greypat cursor bitmap randseed cdev piclist
  rgnlist polylist portlist : -> VarName
  op #init : -> Globals
  op get : Globals VarName -> Var
  op put : Globals VarName Var -> Globals
  op gv : Idx Pattern Pattern Pattern Pattern
  Pattern Cursor BitMap Int Level
  PicList RgnList PolyList PortList : -> Globals
  vars GP, GP1 : Idx
  eq Var : get(gv(GP, PT, PT', PT'', PT''', PT''''),
            CR, BM, RS, CL, PL, RL, OL, GL), port) = GL [ GP ]
  eq Var : get(gv(GP, PT, PT', PT'', PT''', PT''''),
            CR, BM, RS, CL, PL, RL, OL, GL), portp) = GP
  eqGlobals : put(gv(GP, PT, PT', PT'', PT''', PT''''),
            CR, BM, RS, CL, PL, RL, OL, GL), port, PORT)
            = gv(GP, PT, PT', PT'', PT''', CR, BM, RS, CL, PL, OL, OL, chg(PORT, GP, GL))
  eqGlobals : put(gv(GP, PT, PT', PT'', PT''', PT''''),
            CR, BM, RS, CL, PL, RL, OL, GL), portp, GP1) = gv(GP1, PT, PT', PT'', PT''', PT''''),
            CR, BM, RS, CL, PL, OL, OL, GL)
endobj
```

Example 9.

Here, like fixed-size record data types, a constructor gv presents a canonical form. Each field represent a variable. The field access functions are just put and get, with field name in its arity. Semantically put changes the value of the designated variable, while get fetches it. The power of sort ordering is once again apparent; put and get are defined only on the sort Var, yet works on all the fields of diverse sorts, since we have declared those sorts subsorts of Var.

4.4 States

Most behaviours of most systems can be described in terms of state transitions. Algebraic specification is among the natural ways to represent such state transitions [14]. At an abstract level, all the behaviours of the QuickDraw package are either to change the current value of a variable or the image of an physical device. Thus by incorporating values of variables and device images into a state, QuickDraw an indivisible whole is defined, by OBJ2, in this way:

```pascal
object QDSTATE is
  protecting GLOBALS
  protecting DEVICES
  sort QDState Part ParName
  subsorts Var Dev < Part
  subsorts VarName DevName < ParName
  op #init : -> QDState
```
It may be surprising that drawing operations, at times implemented by hardware, come at the top layer of the package. In fact, low-level graphics operations, such as bit manipulations, are left to be defined, since they do not concern design issues at all. Rather, this layer exists to define how to re-structure the QuickDraw package.

5.2 Overreliance on Current Values

It is of use to define primitive graphic operations in terms of current values, such as current coordinates and current filling patterns, both for implementation and for manipulation. Yet that practice renders ambiguous the two distinct kind of operations; those concerning the drawing environment, and those of drawing itself. Let us consider line drawing. A procedure of QuickDraw has the interface:

```
PROCEDURE LineTo (h, v: INTEGER)
```

LineTo draws a line from the current location of the drawing pen to the coordinate (h, v), whether moves the current location. The drawing, however, requires a couple of data other than the current location:

```
vars P P': Point,
vars I: Integer
```

We have no reason to preclude empty rectangles from the first operation. Bar unusual interpretations, the intersection involving an empty rectangle generates another emptiness. On the other hand, the second operation, requiring actual coordinates to be meaningful, is inimical to empty rectangles. Operations upon rectangles must be clear whether or not they accept this ghost rectangle. In the Guide, empty rectangles seem to have (0, 0), the origin of whichever coordinate plane, as both top-left and bottom-right coordinates, no matter where they collapse into the abyss. Consider the two operations:

1. get the intersection of two rectangles.
2. shift a rectangle by a given distance.

We have no reason to preclude empty rectangles from the first operation. Bar unusual interpretations, the intersection involving an empty rectangle generates another emptiness. On the other hand, the second operation, requiring actual coordinates to be meaningful, is inimical to empty rectangles. Operations upon rectangles must be clear whether or not they accept this ghost rectangle. In the Guide, empty rectangles seem to have (0, 0), the origin of whichever coordinate plane, as both top-left and bottom-right coordinates, no matter where they collapse into the abyss. Consider the two operations:

1. get the intersection of two rectangles.
2. shift a rectangle by a given distance.

We have no reason to preclude empty rectangles from the first operation. Bar unusual interpretations, the intersection involving an empty rectangle generates another emptiness. On the other hand, the second operation, requiring actual coordinates to be meaningful, is inimical to empty rectangles. Operations upon rectangles must be clear whether or not they accept this ghost rectangle. In the Guide, empty rectangles seem to have (0, 0), the origin of whichever coordinate plane, as both top-left and bottom-right coordinates, no matter where they collapse into the abyss. Consider the two operations:

1. get the intersection of two rectangles.
2. shift a rectangle by a given distance.

5.2 Overreliance on Current Values

It is of use to define primitive graphic operations in terms of current values, such as current coordinates and current filling patterns, both for implementation and for manipulation. Yet that practice renders ambiguous the two distinct kind of operations; those concerning the drawing environment, and those of drawing itself. Let us consider line drawing. A procedure of QuickDraw has the interface:

```
PROCEDURE LineTo (h, v: INTEGER)
```

LineTo draws a line from the current location of the drawing pen to the coordinate (h, v), whether moves the current location. The drawing, however, requires a couple of data other than the current location:

```
vars P P': Point,
vars I: Integer
```
1. line width
2. pattern of the line
3. drawing mode, such as overwriting and reversing

Thus the line drawing in its totality should be something like:

drawline( from, to, width, pattern, mode )

where from, to are the two ends, and width, pattern, mode specify the three requests listed above. Separated on another side reside operations retrieving/changing the drawing environment which may, just may, comprise the values discussed here:

setlocation ( coordinate )
coordinate = getlocation()
width = getlinewidth()

The procedure LineTo, if needed at all, can be defined by way of those suboperations. That is roughly the line we have toed.

The apparent liberty QuickDraw procedures enjoy in their usage of current values lead to the imprecise modularisation of procedures, as discussed below.

5.3 Misleading Classifications of Procedures

Here we restructure the procedures defined in the Guide, according to our scheme. They divide into:

1. operations on data types
2. operations that affect the drawing environment
3. drawing operations per se

although the demarcation line between the latter two blurs in many cases. While each of these categories turns out to be such a large set of procedures that further sorting, which is easy as explained below as to each category, is desirable, the classifications used in the Guide, decided prima facie by the objects to be affected, seem none too helpful in searching for an apt procedure.

(1) Procedures on data types

This category contains procedures creating, modifying, or using data types. They are subdivided in respect to the data types of their concern.

1. procedures on rectangles — SetRect, OffsetRect, InsetRect, SectRect, UnionRect, PtInRect, Pt2Rect, PtToAngle, EqualRect, EmptyRect
2. procedures on regions — CopyRgn, SetEmptyRegion, SetRectRegion, RectRgn, OffsetRgn, InsetRgn, SectRgn, UnionRgn, DiffRgn, XorRgn, PtInRgn, RectInRgn, EqualRgn, EmptyRgn
3. procedures on points — AddPt, SubPt, SetPt, EqualPt
4. procedures on polygons — OffsetPoly
5. other procedures — StuffHex, ScalePt, MapPt, MapRect, MapRgn, MapPoly

(2) Procedures on the drawing environment

All the procedures that change or refer to the global variables and do not affect images on devices come under this umbrella. They are grouped according to the relevant variables.

1. procedures on the GrafPort or the pointer to it — InitGraf, OpenPort, InitPort, ClosePort, GetPort
2. procedures on the drawing area — GrafDevice, SetPortBits, PortSize, MovePortTo, SetOrigin, GetClipRect, BackPat
3. procedures affecting the drawing pen — HidePen, ShowPen, GetPen, GetPenState, SetPenState, PenSize, PenMode, PenPat, PenNormal, MoveTo, Move
4. procedures affecting the text scribing modes — TextFont, TextFace, TextMode, TextSize, SpaceExtra, CharWidth, StringWidth, TextWidth, GetFontInfo
5. procedures on colour specifications — foreColor, backColor, ColorBit
6. procedures on cursors — InitCursor, SetCursor, HideCursor, ShowCursor, ObscureCursor
7. procedures changing the meaning of drawing procedures — OpenRgn, CloseRgn, OpenPicture, PickComment, ClosePicture, OpenPoly, ClosePoly
8. procedures customising drawing procedures — SetStdProcs, StdText, StdLine, StdRect, StdRRect, StdOval, StdArc, StdPoly, StdRgn, StdBits, StdComment, StdTxMeas, StdGetPic, StdPutPic
9. other procedures — LocaltoGlobal, GlobalToLocal, Random, GetPixel, NewRgn, DisposeRgn, KillPicture, KillPoly

(3) Procedures for drawing

These procedures draw pictures, physically or otherwise; otherwise, since notional rectangles etc. are drawn by the same procedures while regions, polygons, or pictures are being defined. This multiplicity is one of the worst aspects of the structure of the original QuickDraw. Obviously, they are broken down into:

2Strictly speaking, They are in the next category, since they refer to pointers to regions.
3The same note here as procedures on regions
4Some of them affect the screen image, yet included here since in QuickDraw itself there is no suggestion as to their actual visual effects
1. drawing lines — LineTo, Line
2. drawing text — DrawChar, DrawString, DrawText
3. drawing rectangles — FrameRect, PaintRect, EraseRect, InvertRect, FillRect
4. drawing ovals — FrameOval, PaintOval, EraseOval, InvertOval, FillOval
5. drawing round-edged rectangles — FrameRoundedRect, PaintRoundedRect, EraseRoundedRect, InvertRoundedRect, FillRoundedRect
6. drawing arcs — FrameArc, PaintArc, EraseArc, InvertArc
7. drawing regions — FrameRgn, PaintRgn, EraseRgn, InvertRgn, FillRgn
8. drawing pictures — DrawPicture
9. drawing polygons — FramePoly, PaintPoly, ErasePoly, InvertPoly, FillPoly

6 Conclusions

Our efforts confirm OBJ2, in its guise of specification language, as expressive and flexible enough for realistic or real problem domains, at least similar to the graphics system we tackled. The composition of objects suggests this hold for a class of algebraic specification techniques.

Several researches have been reported in the application of algebraic specifications to life-size systems, including graphics systems, for various purposes. [14] shows an algebraic specification of a system for drawing structured pictures. [18] is a comprehensible trial to use algebraic techniques in designing graphics systems, incorporating user interactions as well as manipulating pictures. Two facts distinguish our work from others. For one thing, we have a working system that accepts the resultant specification. Thus we can be sure the descriptions are strictly formal. For the other, related but different, OBJ2 has a rewrite engine that can 'execute' our specifications. This bring open a wide horizon, since we could literally see how our design works.

We have to admit that the QuickDraw package does not contain the part of graphics system harder to describe in algebraic terms: man-machine interaction, that is. In addition, in some parts our specification is somewhat contrived; we owe the blame to the initial decision, already stated, to preserve the appearance of QuickDraw. Thus two farther goals await our immediate attention:

- refine our specification, trimming ugly bumps
- enlarge the target to include interactive operations

Acknowledgement

OBJ2 was conceived by Goguen, J.A., Jouannaud, J.-P., Meseguer, J., and Futatsugi, K., while they gathered at SRI International. The angel thence born we have adopted and reared in our own ambiance, with nutritions of our choice, up to the current stage. We extend thanks to the three of them — one of the quartet is a co-author — for this jewel of the world in whatever way she now behaves in our arms.

References


