An Attempt at Formal Specifications For a Non-Trivial Object : Work in Progress

Fiume, Eugène Lucas

An Attempt at Formal Specifications For a Non-Trivial Object

Work in Progress
E. Fiume

Abstract

Formal specification has long been advocated but rarely practised. When practised, it is often applied to simple, already well understood objects such as stacks and other basic data types. This note begins to explore the issue of formal specification within an object-oriented environment. We attempt to specify formally the object **Bitmap**. This is a particularly interesting choice, for bitmaps are mutable (i.e., they change in time), they can have a perceived effect on images, and their semantics is highly dependent on context. Bitmaps are therefore in many ways worst-case problems for formal specification.

Résumé

Les spécifications formelles, dont l'emploi est prôné depuis longtemps, ont été en fait rarement mises en pratique – sauf pour des objets déjà bien connus, comme les queues ou d'autres types de données de élémentaires. Dans cet article, nous abordons le problème des spécifications formelles dans le cadre d'un environnement orienté objet, et nous tentons d'appliquer cette démarche au cas de l'objet *bitmap*. Il s'agit là d'un choix particulièrement intéressant, car ce genre d'objet se modifie au cours du temps, en entraînant des effets visibles sur des images, et que la sémantique qu'on lui associe dépend fortement du contexte dans lequel il est utilisé. Le *bitmap* constitue donc à bien des égards un cas limite pour l'application de spécifications formelles.

1 Introduction

It is hardly necessary to sing the praises of formal specification, praises which have already been so well sung by others. It is, however, necessary to put one's money where one's mouth is, because, too often, formal specification techniques have been applied to all but trivial objects and notions that are already entirely and intuitively understood. Object-oriented systems are badly in need of formal specification tools, for issues such as object/type equivalence and containment, and the semantics of object operations, require some kind of formal modelling to define satisfactorily.

A *specification* is a promise of performance. It tells potential users of an object what behaviour to expect, and it tells an object implementor what behaviour must be realised. If a certain behaviour is not specified, a user cannot assume it, and an implementor does not have to implement it. It is always difficult to determine exactly what should be specified, and to what degree of detail. For example, in the specifications below, several "low-level" notions will be specified, such as pixel shapes, and arrangements of pixels in an image. This is because I feel these notions are crucial to understanding how a bitmap will be visualised on a display screen. I do not think it is crucial to specify, on the other hand, that suitable display screens must use 60Hz electrical current, that a certain class of phosphors for pixels must be used, what the lighting characteristics of the room in which the screen is to be found should be, or what colour of clothing the viewers should be wearing. All of these factors can affect the perceived meaning of an image, but these issues go far beyond semantics.

Deciding what should go into a "semantics" is far less clear than the literature would have one believe. The standard view is that a semantics should capture the "essential" characteristics of an object that must be true of all implementations. For mathematical objects like the integers and the reals, the essential characteristics are usually clear. For other kinds of objects (the majority of those used in computing systems), deciding what is essential amounts to a value judgement. Even with mathematical objects problems arise, because there can be no implementation of the real numbers and operations on them. However, with some refinement of their definition, it might be possible to strike a compromise by specifying a "dynamic precision number" which retains many of the useful properties of real numbers. This approach suggests the strategy that I will use in this paper to define interesting objects.

This paper attempts to specify formally the object **Bitmap** and its relationship to another object, **Image**. It is a nontrivial undertaking, because there are many properties assumed about bitmaps (and images) which are actually very hard to describe. We take these for granted because we *see* them in operation, or so we think. What of course we really see is an indirect effect of one or more bitmaps as they interact within an *image*. What if, for example, the same bitmap is mapped to two image displays, each having a different pixel shape? Presumably, we think the bitmap is the same, but different image characteristics have caused it to *look* different. There are many other kinds of interactions between between a bitmap and an image:

- a bitmap's contents may be defined by "what is already on the screen". Alternatively, the screen (i.e., an image) may be defined by what is in a bitmap.
- a bitmap may be moved across the image. What then happens to that part of the image that was vacated by the bitmap? What happens to the part of the bitmap which in its new position, overlaps with the old? What happens to all the other bitmaps which may overlap with it?
- a bitmap may be copied to another bitmap, or instead its contents may be

copied into the image. How are these different?

- a bitmap may be bound to one image, and then subsequently moved to another image.
- one bitmap may cover part of another bitmap. We therefore have to consider the problem of visibility or priority of bitmaps as they appear within images.

It is correct to model a bitmap as an abstract data type or object. However, when one provides a list of operations that one can perform on a bitmap, one should provide an explanation of what they do. Given the above partial list of interactions that a bitmap may have with its environment, it is difficult to give the explanation in an informal language. I shall try to do it using the formal language of mathematics and sets. There are reasons why I will not use an "established" formal specification language such as the algebraic technique [GuHo78], or VDM [Jone80], or the operational approach [Parn72]:

- I am more familiar with mathematics and sets than with these techniques. Moreover, I am more confident that mathematics is sufficient to describe the profusion of possible object behaviours. There would be something quite wrong if it were not. I am less confident that a more syntactically restrictive language is sufficient.
- The notation is more flexible, more powerful (albeit possibly less constructive), and prettier.
- Bitmaps, like most objects in a system, are *mutable*, which is to say that their values change over time. I wish to model such behaviour directly, which means I need access to an abstract state (VDM and various operational techniques actually can accommodate this requirement fairly well).
- Bitmaps interact with their environment, namely with other bitmaps and images. I would like to model such interactions directly without resorting to describing them as hidden or implicit side-effects. One well-known attempt to specify graphical data types using the algebraic technique was forced to use side-effects to describe the effect of graphic objects on images [Mall82]. I would argue that this hides the real meaning of a type in the side-effect.
- Since my theorems and proofs will be written in mathematics, I wish my specifications to be written likewise.

The problem with using mathematics to specify objects is that one has to invent new notation on a regular basis. In the case of bitmaps, I have already done some of the legwork in some previous research [Fium86,87]. The next section develops a mathematical structure for bitmaps and images. In the subsequent section, we apply this formalism to the rigorous specification of the object **Bitmap**. Later we define formally the notion of **Image** objects, and then consider the problem of handling overlapping bitmaps on images.

2 The mathematical structure of bitmaps and images

Many of the notions to follow will be familiar, but the mathematics will require some acclimatisation. The important thing to get out of this is that we distinguish between the notions of a bitmap and an image.

Informally, we shall view a *bitmap* B as a pair (S, I), where S is a rectangle denoting the extent of B, and I is a partial function prescribing an intensity value of 0 or 1 for every (integral) point in B. We shall deal exclusively with rectangular bit-maps in this paper; it is a simple matter to extend the discussion to bit-maps of arbitrary shape [Fium86].

Definition 1 *A* pixel *P is a tuple (S_P, I_P), where $S_P \subseteq \mathbf{R}^2$ is the extent of P in the x-y plane (i.e. the screen plane), and $I_P \in \{0,1\}$ is its colour.*

Definition 2 *A* rectangular index set, *$Rect^{x_1\, y_1}_{x_2\, y_2}$, denotes the set of all integral points within a rectangle with bottom-left corner (x_1, y_1) and top-right corner (x_2, y_2). That is,*

$$Rect^{x_1\, y_1}_{x_2\, y_2} =_{df} \{(i,j) : i, j \in \mathbf{Z},\ x_1 \leq i < x_2, y_1 \leq j \leq y_2\}.$$

The collection of all such rectangular sets is defined as

$$\mathbf{Rect} =_{df} \{Rect^{x_1\, y_1}_{x_2\, y_2} : x_1, x_2, y_1, y_2 \in \mathbf{Z}\}$$

An image is composed of a collection of pixels having three essential characteristics: the arrangement of the pixels, their shape, and their intensity. We shall assume the intensity space for all pixels is $\{0,1\}$. Moreover, all pixels within an image must be of the same shape defined by a prototypical pixel shape or *prototile* $\mathbf{P}$, and the arrangement and shape of the pixels must be such that they form a tiling of the area occupied in $\mathbf{R}^2$ by the image.

Definition 3 *A* pixel prototile $\mathbf{P}$ *is a finite subset of $\mathbf{R}^2$ with which it is possible to tile $\mathbf{R}^2$. That is, there exists a pixel arrangement $\mathbf{T} = \{\alpha_{ij}\mathbf{P} : (i,j) \in \mathbf{Z}^2\}$ given tranformations α_{ij} such that*

$$\mathbf{R}^2 = \bigcup_{P \in \mathbf{T}} P,$$

and the interiors of all $P \in \mathbf{T}$ are disjoint.[1] *We shall call such a set $\mathbf{T}$ a* pixel tiling *over* $\mathbf{P}$.

[1]This is not required, but it makes the notation simpler.

Example 1 Pixel shapes on bit-mapped screens are usually thought of as rectangles occupying unit area. The most common pixel tiling is based on the unit square centred at the origin. That is,

$$\mathbf{U} =_{df} [-\frac{1}{2}, \frac{1}{2}] \times [-\frac{1}{2}, \frac{1}{2}].$$

The *unit-square tiling* induced by prototile $\mathbf{U}$, denoted $\mathbf{T}_u$, has the following form:

$$\mathbf{T}_u =_{df} \{T_{ij}\mathbf{U} : (i,j) \in \mathbf{Z}^2\},$$

for translations $T_{ij}(x,y) =_{df} (x+i, y+j)$.

Definition 4 *Let* $\mathbf{T}$ *be a pixel tiling over prototile* $\mathbf{P}$, $\mathbf{T} = \{\alpha_{ij}\mathbf{P} : (i,j) \in \mathbf{Z}^2\}$. *An* image $R_{n_1 n_2}$ *of* resolution $(n_1+1) \times (n_2+1)$ *over* $\mathbf{T}$ *is of the form*

$$R_{n_1 n_2} = \{P_{ij} = (S_{ij}, I_{ij}, \alpha_{ij}, \mathbf{P}) : (i,j) \in Rect^{0\,0}_{n_1\,n_2}, I_{ij} \in \{0,1\}, S_{ij} = \alpha_{ij}\mathbf{P}\}.$$

Remark 1 *This definition is redundant. Essentially,* S_{ij} *is simply a name for* $\alpha_{ij}\mathbf{P}$. *The definition has been written this way to allow for a convenient definition later on for the "abstract state" of image objects. Observe that it is easy to extend the notion of an image to allow for several pixel prototiles to co-exist within the same image.*

Definition 5 *An* image space *of resolution* $(n_1+1) \times (n_2+1)$, *denoted by* $\mathbf{R}_{n_1 n_2}$, *is the set of all images* $R_{n_1 n_2}$ *as defined above.*

Remark 2 *The cardinality of* $\mathbf{R}_{n_1 n_2}$ *is* $2^{(n_1+1)(n_2+1)}$.

Definition 6 *A* bit-map *with integral bottom-left corner* (x_1, y_1) *and top-right corner* (x_2, y_2) *is of the form*

$$B^{x_1\,y_1}_{x_2\,y_2} =_{df} (S, I); \quad S =_{df} Rect^{x_1\,y_1}_{x_2\,y_2}; \quad I : \mathbf{Z}^2 \to \{0, 1, \omega\},$$

such that $I(i,j) \in \{0,1\}$ *if* $(i,j) \in S$ *and* $I(i,j) = \omega$ *outside* S *(i.e.,* $\mathbf{Z}^2 - S$*).* S *denotes the domain of pixels represented in the bit-map.* $I(i,j)$ *defines the intensity of each pixel* (i,j) *in* B. *Pixels outside* B *(or* S*) are given an "undefined" intensity* ω.

Notice that a bit-map is tied neither to the resolution of a display image, nor to a particular pixel shape. To summarise, a bit-map is a function with domain $\mathbf{Z}^2$ which is $\{0,1\}$-valued over a specific rectangular subset of $\mathbf{Z}^2$, and constantly ω-valued outside that rectangle.

Definition 7 *Let the set of all bit-maps $B^{x_1 y_1}_{x_2 y_2}$ for a specific (x_1, y_1) and (x_2, y_2) be denoted by $\mathbf{B}^{x_1 y_1}_{x_2 y_2}$. The set $\mathbf{B}$ of all such $\mathbf{B}^{x_1 y_1}_{x_2 y_2}$ will be called the* bit-map space, *and is defined as*

$$\mathbf{B} =_{df} \bigcup_{x_1, x_2, y_1, y_2 \in \mathbf{Z}} \mathbf{B}^{x_1 y_1}_{x_2 y_2}.$$

If a bit-map $B = (S, I)$ is such that $x_1 > x_2$ or $y_1 > y_2$, then its extent $S = \emptyset$ and B is called an empty bit-map.

Lastly, we require sequences or products of bit-maps such as $\mathbf{B}^2 = \mathbf{B} \times \mathbf{B}$.

Definition 8 *The* bit-map product space, *denoted by $\mathbf{B}^*$, is the reflexive-transitive closure of products over $\mathbf{B}$. That is,*

$$\mathbf{B}^* =_{df} \bigcup_{i=0}^{\infty} \mathbf{B}^i.$$

3 Non-Interacting Bitmaps

We begin our exploration of the semantics of bitmap objects by first considering them in isolation. That is, we shall define an object **Bitmap**. The formal model of an object defines the operations one is allowed to perform on the object in terms of their effect on an *abstract object representation*. In the case of simple bitmaps, their abstract representation is exactly the set $\mathbf{B}$. The operations we define for the object only depend on the current value of the bitmap, and do not (yet) affect the environment. My notation for defining object operations is my own, though it is fairly similar to most operation-based or method-based object definition languages like Hybrid or Smalltalk.

I shall assume that basic types such as $\mathbf{Z}^n$ and $\mathbf{R}^n$ are defined. In any case, their semantics is their standard number-theoretic one. When parentheses appear around a set in the domain of a function, then the instance of that set is assumed to be implicitly named. When parentheses appear around an element of the range of a function, then that means that the change is made to the *same* object. Otherwise, a *different* object is denoted. For example, consider the specifications

$$+_{immut} : (\mathbf{R}) \times \mathbf{R} \rightarrow \mathbf{R} \quad \text{(infix)}$$

$$+_{mut} : (\mathbf{R}) \times \mathbf{R} \rightarrow (\mathbf{R}) \quad \text{(infix)}$$

within the specification for the object $\mathbf{R}$, and let A and B be of type $\mathbf{R}$. Then $A +_{immut} B$ denotes a new object in $\mathbf{R}$, whereas $A +_{mut} B$ denotes a change to A. Obviously *mut* and *immut* stand for *mutable* and *immutable*, respectively.

Syntax of Object Operations.

object schema Bitmap

Object Operations

$New : \mathbf{Z}^2 \times \mathbf{Z}^2 \rightarrow (\mathbf{B})$

$Zero : (\mathbf{B}) \rightarrow (\mathbf{B})$

$One : (\mathbf{B}) \rightarrow (\mathbf{B})$

$Comp : (\mathbf{B}) \rightarrow (\mathbf{B})$

$\wedge : (\mathbf{B}) \times \mathbf{B} \rightarrow \mathbf{B}$ (infix)

$\vee : (\mathbf{B}) \times \mathbf{B} \rightarrow \mathbf{B}$ (infix)

$\leftarrow : (\mathbf{B}) \times \mathbf{B} \rightarrow (\mathbf{B})$

$| : (\mathbf{B}) \times \mathbf{Rect} \rightarrow \mathbf{B}$ (infix)

$Overlay : (\mathbf{B}) \times \mathbf{B}^2 \rightarrow \mathbf{B}$

$Get : (\mathbf{B}) \times \mathbf{Z}^2 \rightarrow \{0, 1, \omega\}$

$Put : (\mathbf{B}) \times \mathbf{Z}^2 \times \{0, 1\} \rightarrow (\mathbf{B})$

This ends the specification of the syntax of the object. We can quibble about exactly the operations such an object should have and what the semantics of each operation should be. Hopefully it will be clear how to customise this definition as desired. Most object-oriented languages stop here. That is, there is no way of semantically distinguishing among the operations *Zero*, *One*, and *Comp*, since each operation has the same syntax, and there is no specification of their semantics.

Semantics of Object Operations.

Suppose A, B : **Bitmap**. That is, A and B are instances of **Bitmap**, and their abstract representation is $A = (S_A, I_A)$, $B = (S_B, I_B) \in \mathbf{B}$.

Create a new bitmap. This operation creates a new bitmap of the desired dimensions. Its semantics is:

$$A.New(x_1, y_1)(x_2, y_2) =_{df} (S_A, I_A),$$

$$S_A = Rect_{x_2\ y_2}^{x_1\ y_1},$$

$$\forall (i, j) \in S_A : I_A(i, j) = 0.$$

Zero/One a bitmap. The operation *Zero* initialises an existing bitmap to all zeros within its extent. The operation *One* likewise sets a bitmap to all ones.

$A.Zero =_{df} (S_A, I_0)$, where

$$I_0(i, j) =_{df} \begin{cases} 0 & \text{if } (i, j) \in S_A \\ \omega & \text{if } (i, j) \in \mathbf{Z}^2 - S_A \end{cases}$$

$A.One =_{df} (S_A, I_1)$, where

$$I_1(i, j) =_{df} \begin{cases} 1 & \text{if } (i, j) \in S_A \\ \omega & \text{if } (i, j) \in \mathbf{Z}^2 - S_A \end{cases}$$

Complement a bitmap. *Comp* complements an existing bitmap within its extent.

$A.Comp =_{df} (S_A, I_{\bar{A}})$ where

$$I_{\bar{A}}(i,j) =_{df} \begin{cases} 1 - I_A(i,j) & \text{if } (i,j) \in S_A \\ \omega & \text{if } (i,j) \in \mathbf{Z}^2 - S_A \end{cases}$$

Bitmap assignment. Bitmap assignment is not dissimilar to assignment for other data types.

$B \leftarrow A =_{df} (S_B, I_B)$, where $S_B =_{df} S_A$ and $I_B =_{df} I_A$.

Clipping. The operation "|" is the well known *clipping* or *restriction* operation from computer graphics. We assume the regions of restriction are also rectangles, although it is straightforward to extend this to more general shapes. Let $R \in \mathbf{Rect}$.

Then $A|R =_{df} (S, I)$ where

$$S =_{df} S_A \cap R$$

and

$$I(i,j) =_{df} \begin{cases} I_A(i,j) & \text{if } (i,j) \in S \\ \omega & \text{otherwise} \end{cases}$$

Overlayed bitmaps. It is often convenient to combine two bitmaps with respect to a "control" bitmap. Let $A = (S_A, I_A)$, $B = (S_B, I_B)$, $C = (S_C, I_C) \in \mathbf{B}$.

$Overlay(A, B, C) =_{df} (S_A, I'_A)$ where $S_B \subset S_C$, and

$$I'_A(i,j) =_{df} \begin{cases} I_B(i,j) & \text{if } I_C(i,j) = 1 \\ I_A(i,j) & \text{otherwise} \end{cases}$$

Depending on the intensity of bitmap C at (i,j), the intensity of the new bitmap takes on the value of I_B or I_A at that point. If A, B are of the same dimensions, then

$$Overlay(A, B, Zero(C \leftarrow B)) = A,$$

$$Overlay(A, B, One(C \leftarrow B)) = B,$$

Logical bitmap operations. Logical bitmap operations are almost trivial. We assume the standard numeric interpretation of 0 denoting **false** and 1 denoting **true**. Also, we define

$$a \otimes \omega = \omega \otimes a = \omega \otimes \omega = \omega$$

for any boolean function $\otimes$ and boolean value a. Then

$$A \wedge B =_{df} (S_A \cap S_B, I_A \wedge I_B)$$

$$A \vee B =_{df} (S_A \cap S_B, I_A \vee I_B)$$

Observe that this definition is correct because the intersection of two rectangles (S_A and S_B in this case) is either itself a rectangle, or empty (a degenerate rectangle). Observe, moreover, that the bitmaps do not have to be aligned.

Get and Put I/O Operations. The bitmap I/O operations are also straightforward. Let $b \in \{0,1\}$. Then

$$A.Get(i,j) =_{df} I_A(i,j)$$

$$A.Put(i,j)\, b =_{df} (S_A, I_{A'})$$

where

$$I_{A'}(x,y) =_{df} \begin{cases} b & \text{if } (x,y) = (i,j) \\ I_A(x,y) & \text{otherwise} \end{cases}$$

Note that the value returned by get can be "undefined".

This completes the formal specification.

The object **Bitmap** is a fairly simple thing. Even so, we are now capable of reasoning about instances of this object. Consider the following propositions. I leave their proofs as exercises which are direct applications of the above semantics, requiring only a smidgen of set theory.

Proposition 1 *Let* A, B : **Bitmap**. *Then*

$$(A \vee B).Comp = A.Comp \wedge B.Comp.$$

Proposition 2 *Let* A, B : **Bitmap** *and let* $R \in$ **Rect**. *Then*

$$(A \vee B)|R = A|R \ \vee B\ |R,$$

$$(A \wedge B)|R = A|R \ \wedge B\ |R.$$

Proposition 3 *Let* A : **Bitmap**. *Then*

$$(A.Comp).Comp = A.$$

Actually, it is fairly easy to show that each class of **Bitmaps** restricted to $Rect^{x_1\, y_1}_{x_2\, y_2}$, for any $x_1, x_2, y_1, y_2 \in \mathbf{Z}$, together with the the operations *Zero*, *One*, *Comp*, $\wedge$, and $\vee$, forms a *boolean algebra*. This means, among other things, that $\wedge, \vee$ are associative, commutative, and distributive. Observe as well that restriction distributes over these operations, which means that in practice it is best to perform the restriction operations first to decrease the size of the rectangles with which one is working.

There is a very important point to observe about the way the operations above were defined. Typically, operations are defined in terms of one another wherever possible. This is especially true of specifications in the algebraic approach, and is normally a praiseworthy thing to do, because it helps one to determine the set of operations that are in some sense "minimal". For example, *Comp* could be defined in terms of a (large) set of *Get* and *Put* operations. However, I have intentionally avoided doing this, because for an intuitive reason I do not believe they actually are equivalent. The point is that we intuitively feel that bit*map* operations *cost less* than the coresponding set of bit*wise* operations. This is borne out in practice. For example, many bitmapped workstations contain special support to speed up bitmap operations. It remains, therefore, to reflect this notion of cost somehow in the semantics. I leave this very interesting notion of the formalisation of cost semantics to future research.

Undoubtedly, we have defined a nice class of algebraic objects, but we still do not have a mechanism for visualising them. They are analogous to the idea of a "memory pixrect" in the Sun jargon. In fact, one's first impression is that indeed we already have a visualisation. But we know better. Bitmaps are in some sense "uninterpreted" images. The next section shows how to give them a visual interpretation.

4 Bitmaps Interacting with Images

An *image* is a model for a display screen. As such, it is part of the "system". The formalism defined earlier affords us great flexibility in modelling a wide variety of screens with various pixel arrangements and resolutions. It is certainly much richer than any specification effort of which I am aware.[2] As before, we use the formal model of images ($\mathbf{R}_{n_1 n_2}$) as an abstract representation for the object **Image**.

We need a formal definition for what kind of shapes a pixel can take on. This is difficult to do. For the purposes of this paper, let us define a pixel prototile as any closed, hole-less, polygonal region in $\mathbf{R}^2$ which can be used to tile the plane. We shall call the set of all such prototiles $\mathcal{P}$. The set of shapes at our disposal includes isoceles triangles, rectangles, and regular hexagons, as well as other more bizarre shapes. In the semantics below, I shall refer to elements of $\mathcal{P}$ set theoretically. An implementation will have to define a more constructive representation, such as a polygon edge list.

The abstract state of an image will include the pixel shape, the resolution of

[2]This is both a blessing and a curse, in that increased understanding may also require knowing a larger set of details. However, if the specification is written carefully, it might be possible to present the details only to those that are interested. Omitting them entirely can be dangerous. Recall that a specification is a promise of performance: if it isn't written down (somewhere), then it cannot be assumed.

the image, the arrangement of pixels in the image (i.e., transformations of the pixel prototile), and the intensity of each pixel.

Syntax of Object Operations.

object schema Image

Object Operations

Syntax

$Resolution : (\mathbf{R}_{n_1 n_2}) \to \mathbf{Z}^2$
$PixelShape : (\mathbf{R}_{n_1 n_2}) \to \mathcal{P}$
$Get : (\mathbf{R}_{n_1 n_2}) \times \mathbf{R}^2 \to \{0, 1, \omega\}$
$Put : (\mathbf{R}_{n_1 n_2}) \times \mathbf{Z}^2 \times \{0, 1\} \to (\mathbf{R}_{n_1 n_2})$
$GetBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{Rect} \to \mathbf{B}$
$PutBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \to (\mathbf{R}_{n_1 n_2})$
$MoveBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \times \mathbf{Z}^2 \to (\mathbf{R}_{n_1 n_2} \times \mathbf{B})$
$CopyBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \times \mathbf{Z}^2 \to (\mathbf{R}_{n_1 n_2}) \times \mathbf{B}$

Semantics of Object Operations.

Let Im : **Image**, with abstract representation $Im = \{(S_{ij}, I_{ij}, \alpha_{ij}, \mathbf{P}) : (i,j) \in Rect^{0\,0}_{n_1\,n_2}\} \in \mathbf{R}_{n_1 n_2}$. We assume that each $S_{ij} = \alpha_{ij}\mathbf{P}$ is an transformed instance of pixel prototile $\mathbf{P} \in \mathcal{P}$. Let A : **Bitmap**, $A = (S_A, I_A) \in \mathbf{B}$.

Image Resolution and Pixel Shape. Rendering algorithms need to know the resolution of the image display and the shape of the basic pixel prototype.

$$Im.Resolution =_{df} (n_1 + 1, n_2 + 1)$$

$$Im.PixelShape =_{df} \mathbf{P}$$

Pixel I/O Operations. The pixel I/O operations are similar to those for bitmaps. However, observe that the *Get* operation is defined over $\mathbf{R}^2$ rather than $\mathbf{Z}^2$.

$$A.Get(x,y) =_{df} \begin{cases} I_{ij} & \text{if } (x,y) \in S_{ij} \text{ for some } (S_{ij}, I_{ij}) \in Im \\ \omega & \text{otherwise} \end{cases}$$

$A.Put(i,j)\, b =_{df} Im'$ where

$$Im' =_{df} \begin{cases} (Im - \{(S_{ij}, I_{ij})\}) \cup \{(S_{ij}, b)\} & \text{if } 0 \le i \le n_0, 0 \le j \le n_1 \\ Im & \text{otherwise} \end{cases}$$

Bitmap I/O Operations. Let $R \in \mathbf{Rect}, Im \in \mathbf{R}_{n_1 n_2}$.
$Im.GetBitmap\, R =_{df} (S, I) \in \mathbf{B}$ such that

$$S =_{df} R \cap Rect^{0\,0}_{n_1\,n_2}$$

and

$$I(i,j) =_{df} \begin{cases} Im.I_{ij} & \text{if } (i,j) \in S \\ \omega & \text{otherwise} \end{cases}$$

$Im.PutBitmap\,A =_{df} Im' \in \mathbf{R}_{n_1 n_2}$ where

$$Im'.I_{ij} =_{df} \begin{cases} Im.I_{ij} & \text{if } (i,j) \notin S_A \\ I_A(i,j) & \text{otherwise} \end{cases}$$

and

$$\forall (i,j) \in Rect^{0\,0}_{n_1 n_2} : Im'.S_{ij} = Im.S_{ij}.$$

Move/Copy a Bitmap. This is where the fun really starts. Move and copy bitmap operations are actually quite subtle. We first deal with the move operation. This operation is defined only if the part of the bitmap to be moved has an intensity function that is consistent with the image intensity over the region of the image it covers.[3] That is,

$$\mathbf{Precondition}_{Move} \equiv Im.GetBitmap\,S_A = A|Rect^{0\,0}_{n_1\,n_2},$$

where $A = (S_A, I_A)$, and more specifically $S_A = Rect^{x_1\,y_1}_{x_2\,y_2}$.

If this precondition holds, then

$$Im.MoveBitmap\,A\,(x,y) =_{df} (Im', A').$$

This asserts that the lower-left corner of bitmap A on image Im is to be moved to position (x,y) in Im. The origin of A is offset accordingly, and the intensity function in Im is adjusted to reflect the new position of A in the image.

The formal semantics follows. First we define the changes to the bitmap.

$A' =_{df} (S_{A'}, I_{A'})$.

$S_{A'} =_{df} Rect^{x_3\,y_3}_{x_4\,y_4}$, such that

$$\begin{aligned} x_3 &= x \\ y_3 &= y \\ x_4 &= x + (x_2 - x_1) \\ y_4 &= y + (y_2 - y_1). \end{aligned}$$

$$I_{A'}(i,j) =_{df} \begin{cases} I_A(i - x + x_1,\, j - y + y_1) & \text{if } (i,j) \in S_{A'} \\ \omega & \text{otherwise} \end{cases}$$

Now we define the changes to the image.

$$\forall (i,j) \in Rect^{0\,0}_{n_1\,n_2} : Im'.S_{ij} =_{df} Im.S_{ij},$$

and

$$Im'.I_{ij} =_{df} \begin{cases} Im.I_{ij} & \text{if } (i,j) \notin S_A \cup S_{A'} \\ 0 & \text{if } (i,j) \in S_A \wedge (i,j) \notin S_{A'} \\ I_{A'}(i,j) & \text{if } (i,j) \in S_{A'} \end{cases}$$

[3]This can only happen if someone changes the bitmap without changing the image.

Some explanation is in order. As was said earlier, *MoveBitmap* transfers a bitmap with origin (x_1, y_1) to a bitmap with new origin (x, y). The semantics of moving A to A' should be clear. The image semantics is slightly tricky. The first line in the definition of $Im'.I_{ij}$ states that the unaffected portion of the image remains unchanged. The region of the image left vacant by moving the bitmap is defined to have zero intensity. This is specified in the second line of the equation. The third line gives the intensity of the bitmap in its new location. Notice that if a bitmap is moved off-image, the effect is to set to zero the region it formerly occupied. Notice as well that when the old and new positions of the bitmap overlap, only that portion of the image formerly occupied by A but not occupied by A' is zeroed.

Now it is very simple to deal with copying bitmaps. The only differences are that a new bitmap is created (see the syntax above), and that the portion of the image occupied by the original bitmap is left untouched.

$Im.CopyBitmap\,A\,(x,y) =_{df} (Im', A')$. The bitmap $A' =_{df} (S_{A'}, I_{A'})$ is exactly as in the semantics for *MoveBitmap.*

$$\forall (i,j) \in Rect^{0\,0}_{n_1\,n_2} : \; Im'.S_{ij} =_{df} Im.S_{ij},$$

and

$$Im'.I_{ij} =_{df} \begin{cases} Im.I_{ij} & \text{if } (i,j) \notin S_A \cup S_{A'} \\ Im.I_{ij} & \text{if } (i,j) \in S_A \wedge (i,j) \notin S_{A'} \\ I_{A'}(i,j) & \text{if } (i,j) \in S_{A'} \end{cases}$$

This completes the formal specification of **Image**.

There still exist some simplifications in the above specifications, but we are much closer to the "true" meaning of bitmaps and images. We haven't yet provided a semantics of image transformations such as image rotations, reflections, etc. These are interesting, because it turns out that the only "faithful" image transformations are those that are in the symmetry group of the underlying pixel prototile [Fium87]. Further discussion of this somewhat technical result is beyond the scope of this paper, but it is important to see that formalising notions such as images and bitmaps allows one to prove non-trivial properties of objects.

As our last exercise, we shall modify the definition of **Image** so that it handles overlapping bitmaps. It is easy to extend this definition to handle the semantics of the so-called *covered window paradigm.* We shall define a new image object. This new object, called **CoveredImage**, has only three more operations, but the existing operations require slightly different semantics to implement a priority visibility scheme. Rather than invoke some kind of inheritance mechanism, the entire object definition is restated.

Syntax of Object Operations.

object schema CoveredImage

Object Operations (inherited from **Image**)

Syntax

$Resolution : (\mathbf{R}_{n_1 n_2}) \rightarrow \mathbf{Z}^2$
$PixelShape : (\mathbf{R}_{n_1 n_2}) \rightarrow \mathcal{P}$
$Get : (\mathbf{R}_{n_1 n_2}) \times \mathbf{R}^2 \rightarrow \{0,1,\omega\}$
$Put : (\mathbf{R}_{n_1 n_2}) \times \mathbf{Z}^2 \times \{0,1\} \rightarrow (\mathbf{R}_{n_1 n_2})$
$GetBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{Rect} \rightarrow \mathbf{B}$
$PutBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \rightarrow (\mathbf{R}_{n_1 n_2})$
$MoveBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \times \mathbf{Z}^2 \rightarrow (\mathbf{R}_{n_1 n_2} \times \mathbf{B})$
$CopyBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \times \mathbf{Z}^2 \rightarrow (\mathbf{R}_{n_1 n_2}) \times \mathbf{B}$

Object Operations (new operations)

Syntax

$Top : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \rightarrow (\mathbf{R}_{n_1 n_2})$
$Bottom : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \rightarrow (\mathbf{R}_{n_1 n_2})$
$RemoveBitmap : (\mathbf{R}_{n_1 n_2}) \times \mathbf{B} \rightarrow (\mathbf{R}_{n_1 n_2})$

Semantics of Object Operations.

We first require some auxiliary notations. $\mathcal{B}$ will denote the set of bitmaps currently associated with the image. The semantics of *PutBitmap* will be changed to add elements to $\mathcal{B}$, and the operation *RemoveBitmap* naturally removes elements from $\mathcal{B}$. Initially, $\mathcal{B} = \emptyset$. For each $B \in \mathcal{B}, Pr_B$ denotes the *priority* or *depth* of B. A priority of 0 means the entire bitmap is visible. Priorities greater than 0 mean that parts of the bitmap may be obscured. More precisely, we define a $Pr : \mathcal{B} \rightarrow \mathbf{N}$, and for notational convenience we let Pr_B stand for $Pr(B)$. As currently defined, this function is not directly accessible to a user of the **CoveredImage** object, but is instead modified indirectly using the operations *Top*, *Bottom*, *PutBitmap* and *RemoveBitmap*. I am not sure if this is entirely reasonable.

As before, let Im : **Image**, with abstract representation $Im = \{(S_{ij}, I_{ij}, \alpha_{ij}, \mathbf{P}) : (i,j) \in Rect^{00}_{n_1\, n_2}\} \in \mathbf{R}_{n_1 n_2}$.

The following predicate[4] must be true at the completion of each object operation:

Postcondition $=_{df} \forall (i,j) \in Rect^{00}_{n_1\, n_2}$:

$$Im.I_{ij} = \begin{cases} 0 & \text{if } \forall A = (S,I) \in \mathcal{B} : (i,j) \notin S \\ I_B(i,j) & \text{otherwise, where } Pr_A = \min_{A \in \mathcal{B}}\{Pr_A\}. \end{cases}$$

[4]I call this predicate a *postcondition*. One could argue that this is an *invariant*. I have chosen the former name because the predicate is allowed to go false during the execution an object operation, but the predicate must be true after its completion.

This predicate states that the intensity of each pixel in the image must reflect the intensity of a bitmap which overlaps with that pixel, and which has the lowest priority value. If no bitmap overlaps with a given pixel, it is set to the "background" colour of 0. There is nothing "wrong" with introducing auxiliary predicates and variables to specify something. They are often necessary in other areas of specification. For example, Howard motivates the use of *history variables* to represent past states of monitors [Howa76]. The guideline to use is that when an important aspect of the behaviour of an object would be missing from a formal specification if auxiliary objects are not used, then by all means introduce them formally. Otherwise, the spectre of "semantics by side-effect" would resurface.

Now we can give the semantics of the new operations.

Move a bitmap to the top. This operation makes a bitmap entirely visible.

$$Im.Top\,B =_{df} B \in \mathcal{B} \Rightarrow Pr_B \leftarrow 0 \wedge \textbf{Postcondition.}$$

Observe that the operation is only defined if $B \in \mathcal{B}$. An implementor may wish to give an error otherwise.

Move a bitmap to the bottom. This operation places a bitmap below all other bitmaps in the image.

$$Im.Bottom\,B =_{df} B \in \mathcal{B} \Rightarrow Pr_B \leftarrow m \wedge \textbf{Postcondition,}$$

where

$$m =_{df} 1 + \max_{A \in B} Pr_A.$$

Display a bitmap on the image. The I/O semantics of *PutBitmap* are as before. What differs is the handling of the auxiliary specifications. In particular, $Im.PutBitmap\,A =_{df} Im' \in \mathbf{R}_{n_1 n_2}$ where Im' is as above, and

$$\forall A \in \mathcal{B}:\ A \neq B \Rightarrow Pr_A \leftarrow Pr_A + 1,$$

$$\mathcal{B} \leftarrow \mathcal{B} \cup \{B\},$$

$$Pr_B \leftarrow 0.$$

Observe that the I/O semantics of *PutBitmap* ensure that the **Postcondition** is true, since a new bitmap is always given priority zero, and since relative priorities among bitmaps are not otherwise upset.

Remove a bitmap from the image. Remove makes a bitmap invisible and removes all record of its existence.

$$Im.Remove\,B =_{df} B \in \mathcal{B} \Rightarrow \mathcal{B} \leftarrow \mathcal{B} - \{B\} \wedge Pr_B \leftarrow \omega \wedge \textbf{Postcondition.}$$

This completes the specification of **CoveredImage.**

5 Conclusions

There is nothing magic about formal specifications. If desired, they can be viewed as comments written using a strange typefont. However, they are intended to communicate the essential properties of an object. An implementor can take the specifications and attempt to embody them in terms of real data structures and procedures. The formal model provides a basis for proving correctness, or for proving other interesting properties of the object, such as the equivalence of one instance of an object with another.

One particularly interesting direction of research was suggested: that of developing useful measures of the cost of an operation. Surely that is often a useful piece of information to know, for it would permit (automatic?) optimisation of the use of object operations.

The specification of bitmaps and images in this paper are just examples. In some ways they are difficult to specify, in that they are mutable objects and in that the semantics of their visualisation is nontrivial. However, these objects are still *passive* things. I will come back some day to specify *active* objects. To do this requires new notation.

References

[Fium86] Fiume, E., *A Mathematical Semantics and Theory of Raster Graphics*, Ph.D. Thesis, Technical Report CSRI-185, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 1A4.

[Fium87] Fiume, E., "Bit-mapped graphics: a semantics and theory", to appear, *Computers and Graphics 11*, 2 (April 1987).

[GuHo78] Guttag, J.V., and J.J. Horning, "The algebraic specification of abstract data types", *Acta Informatica 10*, 1 (Jan. 1978), 27-52.

[Howa76] Howard, J.H., "Proving monitors", *Commun. ACM 19*, 5 (May 1976), 273-279.

[Jone80] Jones, C.B., *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[Mall82] Mallgren, W.R., "Formal specification of graphic data types", *ACM Transactions on Programming Languages and Systems 4*, 4 (Oct. 1982), 687-710.

[Parn72] Parnas, D.L., "A technique for software specification with examples", *Commun. ACM 15*, 5 (May 1972), 330-336.