FORMAL SPECIFICATION OF WINDOW SYSTEMS

by

Jonathan Bowen

Technical Monograph PRG-74
ISBN 0-902928-56-2

June, 1989

Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

Electronic mail: bowen@uk.ac.oxford.prg (JANET)

FORMAL SPECIFICATION OF WINDOW SYSTEMS

Jonathan Bowen

X

Blit

example

WM

To Alice, Emma and Jane

"Oh! I do so wish I could see *that* bit!"

Lewis Carroll, *Through the Looking-Glass*

Formal Specification of Window Systems

Jonathan Bowen

Summary

Window management systems are now used extensively for user interfaces to computer systems. Part I of this monograph introduces some of the fundamental ideas in window systems using a formal notation. Part II outlines three real systems and attempts to capture the essence of each system using the same formal notation and ideas introduced in Part I. Low-level detail is avoided to keep the length to a manageable size.

In Part I, chapter 1 introduces general concepts useful for specifying pixel maps and window systems. Chapter 2 defines the raster-op function which is fundamental to many graphics operations and chapter 3 introduces a simple example window system. In Part II, chapters 4–6 detail three particular window systems. Chapter 7 remarks on experience gained by formally specifying the three window systems.

The formal notation used, **Z**, is based on set theory, and has been developed at the Programming Research Group in Oxford.

Contents

Part I

An Introduction to Window Systems

Chapter 1

Basic Concepts

Before we start to attempt to specify a window system formally, it is helpful to introduce a few fundamental and generally applicable ideas. This section gives a formal framework to aid the description of *pixels*, their organisation into *pixel maps* and a number of *windows*.

The specification language used throughout this monograph is the **Z** notation [Haye87, King88, Loom88, Spiv88a, Spiv89]. This is a typed language based on set theory and first order predicate calculus. It has been developed at Oxford over the past few years. The notation here conforms almost entirely to the notation described in [Spiv89]. The only extension is the use of the schema piping operator ('$\gg$') which is included in [King88].

1.1 Pixel positions

A raster graphics display is made up of a set of pixels with positions or coordinates. These are normally defined in X–Y coordinate space. The display is a fixed size bounded rectangle in the X–Y plane.

$$| \; Xsize, Ysize : \mathbb{N}_1$$

The offset in a particular direction is specified from zero up by convention. The position of a pixel may be specified by a pair of X–Y coordinates.

$$Xrange == 0 \,..\, Xsize - 1$$
$$Yrange == 0 \,..\, Ysize - 1$$
$$Pixel == Xrange \times Yrange$$

The pixel at $(0,0)$ is normally at the lower left-hand corner of the display and the pixel at $(Xsize - 1, Ysize - 1)$ is at the top right by convention.

Many operations are applied to pairs of pixels.

$$
\begin{array}{|l}
\textit{PixelPair} \\
\hline
pix_1, pix_2 : Pixel \\
x_1, x_2 : Xrange \\
y_1, y_2 : Yrange \\
\hline
pix_1 = (x_1, y_1) \\
pix_2 = (x_2, y_2) \\
\hline
\end{array}
$$

The '+', '−' and '≤' operators may be overloaded to apply to pixel positions. '+' and '−' may be used for moving pixel areas around the display. '≤' can be used to define pixel ordering from the bottom left to top right.

$$
\begin{array}{|l}
\_ + \_, \\
\_ - \_ : (Pixel \times Pixel) \nrightarrow Pixel \\
\_ \leq \_ : Pixel \leftrightarrow Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad (x_1 + x_2 < Xsize \land y_1 + y_2 < Ysize) \Rightarrow \\
\qquad pix_1 + pix_2 = (x_1 + x_2, y_1 + y_2) \land \\
\quad (x_2 \leq x_1 \land y_2 \leq y_1) \Rightarrow \\
\qquad pix_1 - pix_2 = (x_1 - x_2, y_1 - y_2) \land \\
\quad pix_1 \leq pix_2 \Leftrightarrow x_1 \leq x_2 \land y_1 \leq y_2
\end{array}
$$

We can define the offset between any two pixel positions as a pixel offset. This is defined to wrap round the edge of the pixel area and thus is a total function.

$$
\begin{array}{|l}
offset : Pixel \rightarrow Pixel \nrightarrow Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad offset\ pix_1\ pix_2 = \\
\qquad ((x_1 + x_2) \bmod Xsize, (y_1 + y_2) \bmod Ysize)
\end{array}
$$

We can also overload the '..' operator to define a rectangular area of pixels.

$$
\begin{array}{|l}
\_ \,..\, \_ : (Pixel \times Pixel) \rightarrow \mathbb{F}\, Pixel \\
\hline
\forall\, PixelPair \bullet \\
\quad pix_1 \,..\, pix_2 = (x_1 \,..\, x_2) \times (y_1 \,..\, y_2)
\end{array}
$$

$$pix_1, pix_2 : Pixel \vdash pix_1 \mathinner{..} pix_2 = \{p : Pixel \mid pix_1 \leq p \wedge p \leq pix_2\}$$

A rectangular area of pixels can defined using any two opposing corners (e.g. returned using an attached mouse to sweep between the two). The following functions return the lower left and upper right pixel positions from two such pixel positions respectively.

$$\begin{array}{|l} \_ \ \underline{min} \ \_, \\ \_ \ \underline{max} \ \_ : (Pixel \times Pixel) \rightarrow Pixel \\ \hline \forall\, PixelPair \bullet \\ \quad pix_1 \ \underline{min} \ pix_2 = (min\ \{x_1, x_2\}, min\ \{y_1, y_2\}) \wedge \\ \quad pix_1 \ \underline{max} \ pix_2 = (max\ \{x_1, x_2\}, max\ \{y_1, y_2\}) \end{array}$$

1.2 Pixel maps

A raster graphics display has a number of bit-planes. This may be considered as the Z direction of the display.

$$| \ Zsize : \mathbb{N}_1$$

Each *bit* in a bit-plane has one of two values (cleared or set).

$$\begin{aligned} Clear &== 0 \\ Set &== 1 \\ Bit &== \{Clear, Set\} \end{aligned}$$

The value of a pixel at a particular position may be modelled as a function from bit-plane number to bit value.

$$\begin{aligned} Zrange &== 0 \mathinner{..} Zsize - 1 \\ Value &== Zrange \rightarrow Bit \end{aligned}$$

If all the bits are '*Clear*' the '*Value*' is considered '*Black*' and if they are all '*Set*' it is considered '*White*'.

$$\begin{aligned} Black &== (\ \mu\, val : Value \mid \mathrm{ran}\, val = \{Clear\}\ ) \\ White &== (\ \mu\, val : Value \mid \mathrm{ran}\, val = \{Set\}\ ) \end{aligned}$$

Note that if there is only one bit-plane (i.e. $Zsize = 1$) then pixel values can only be *Black* or *White*.

$$\vdash Zsize = 1 \Rightarrow Value = \{Black, White\}$$

A pixel map consists of a (partial) function from pixel positions to the value of the pixel contents. This can be used to describe part of a display, such as a window.

$$Pixmap == Pixel \pfun Value$$

Non-empty pixel maps may be of interest.

$$Pixmap_1 == Pixmap \setminus \{\emptyset\}$$

Pixel maps are often rectangular in area. We can define such pixel maps using their bottom left and top right pixel positions.

$$Rectangle == \{map : Pixmap_1 \mid \exists_1 p_1, p_2 : Pixel \bullet \dom map = p_1 \upto p_2\}$$

Sometimes it is desirable to set all the range of a pixel map to a particular value, for example when clearing a window down to the background colour. A function to set the range of a relation to a particular value is useful for this.

$$[P, V]$$
$$setval : V \fun (P \pfun V) \fun (P \pfun V)$$
$$\forall v : V; p : P \pfun V \bullet$$
$$setval\ v\ p = (\mu m : P \pfun V \mid (\dom m = \dom p \land \ran m = \{v\}))$$

The following laws apply:

$$p : Pixmap;\ v : Value \vdash (setval\ v)^+\ p = setval\ v\ p$$
$$v : Value \vdash setval\ v\ \emptyset = \emptyset$$

Two pixel maps may overlap. For example, one window may obscured by another. This can be captured as a relation between pixel maps:

$$[P, V]$$
$$\_\ \underline{overlaps}\ \_ : (P \pfun V) \rel (P \pfun V)$$
$$\forall p_1, p_2 : P \pfun V \bullet$$
$$p_1\ \underline{overlaps}\ p_2 \Leftrightarrow \dom p_1 \cap \dom p_2 \neq \emptyset$$

A sequence of pixel maps may be overlaid in the order given by the sequence. It is convenient to define a distributed overriding operator for this.

$$[P, V]$$
$$\oplus/ : \mathrm{seq}(P \nrightarrow V) \rightarrow (P \nrightarrow V)$$
$$\oplus/\langle\rangle = \varnothing$$
$$\forall s : \mathrm{seq}(P \nrightarrow V);\ p : P \nrightarrow V \bullet \oplus/(s \frown \langle p \rangle) = (\oplus/s) \oplus p$$

Distributed overriding is particularly useful for defining the view on a screen of a display, given a sequence of possibly overlapping pixel maps.

The following laws apply for the distributed overriding operator:

$$\vdash \oplus/\varnothing = \varnothing$$
$$p : \mathit{Pixmap} \vdash \oplus/\langle p \rangle = p$$
$$s_1, s_2 : \mathrm{seq}\ \mathit{Pixmap} \vdash \oplus/(s_1 \frown s_2) = (\oplus/s_1) \oplus (\oplus/s_2)$$
$$p_1, p_2 : \mathit{Pixmap} \vdash \oplus/\langle p_1, p_2 \rangle = p_1 \oplus p_2$$
$$s : \mathrm{seq}_1\ \mathit{Pixmap} \vdash \oplus/s = (\oplus/(\mathit{front}\ s)) \oplus (\mathit{last}\ s) = (\mathit{head}\ s) \oplus (\oplus/(\mathit{tail}\ s))$$
$$s : \mathrm{seq}\ \mathit{Pixmap} \vdash \mathrm{dom}(\oplus/s) = \mathrm{dom}(\bigcup(\mathrm{ran}\ s))$$
$$s : \mathrm{seq}\ \mathit{Pixmap} \vdash \mathrm{ran}(\oplus/s) \subseteq \mathrm{ran}(\bigcup(\mathrm{ran}\ s))$$
$$s : \mathrm{seq}\ \mathit{Pixmap} \vdash \bigcap(\mathrm{ran}\ s) = \varnothing \Rightarrow \oplus/s = \bigcup(\mathrm{ran}\ s)$$

Note that the first three laws above provide at alternative way of defining '$\oplus/$'.

1.3 Windows

A series of windows on a display screen may be viewed as a sequence in which each window is laid on the screen in the order defined by the sequence (bottom first, top last). If some of the windows are removed from a sequence, it is sometimes desirable to 'compact' the remaining windows into a sequence again.

$$[W]$$
$$\mathit{compact} : (\mathbb{N} \nrightarrow W) \rightarrow \mathrm{seq}\ W$$
$$\mathit{compact}\ \varnothing = \langle\rangle$$
$$\forall f : \mathbb{N} \nrightarrow W;\ i : \mathbb{N} \mid i = \mathit{min}(\mathrm{dom}\ f) \bullet \mathit{compact}\ f = \langle f(i) \rangle \frown \mathit{compact}(\{i\} \lhd f)$$

If the windows in a sequence overlap, it is useful to be able to move selected windows so that their contents may be viewed (or hidden). This is

analogous to shuffling a pile of sheets of paper (windows) on a desk (screen). Note that the sheets of paper may be of different sizes and in different positions on the desk.

For example, the following function may be used to move a selected window number in the sequence (if it exists) to the top of the pile (i.e. the end of the sequence).

$$
\begin{array}{|l}
[W] \\
\hline
top : \mathbb{N} \rightarrow \text{seq } W \rightarrow \text{seq } W \\
\hline
\forall n : \mathbb{N};\ s : \text{seq } W \bullet \\
\quad n \in \text{dom } s \Rightarrow \\
\qquad top\ n\ s = compact(\{n\} \lhd s) \frown \langle s(n) \rangle \wedge \\
\quad n \notin \text{dom } s \Rightarrow \\
\qquad top\ n\ s = s \\
\hline
\end{array}
$$

More generally, we can define functions to 'select' and 'remove' a set of windows from a sequence using their identifiers rather than their position in the pile.

$$
\begin{array}{|l}
[W] \\
\hline
\_\ \underline{select}\ \_, \\
\_\ \underline{remove}\ \_ : \text{seq } W \times \mathbf{P}\ W \rightarrow \text{seq } W \\
\hline
\forall s : \text{seq } W;\ w : \mathbf{P}\ W \bullet \\
\quad s\ \underline{select}\ w = compact\ (s \rhd w) \wedge \\
\quad s\ \underline{remove}\ w = compact\ (s \rhd w) \\
\hline
\end{array}
$$

We can then 'raise' or 'lower' these windows to the end or beginning of the sequence as required.

$$
\begin{array}{|l}
[W] \\
\hline
\_\ \underline{raise}\ \_, \\
\_\ \underline{lower}\ \_ : \text{seq } W \times \mathbf{P}\ W \rightarrow \text{seq } W \\
\hline
\forall s : \text{seq } W;\ w : \mathbf{P}\ W \bullet \\
\quad s\ \underline{raise}\ w = (s\ \underline{remove}\ w) \frown (s\ \underline{select}\ w) \wedge \\
\quad s\ \underline{lower}\ w = (s\ \underline{select}\ w) \frown (s\ \underline{remove}\ w) \\
\hline
\end{array}
$$

Every window in a system usually has an identifier, denoted '*Window*', which allows it to be accessed uniquely.

[*Window*]

The generic functions defined in this section will normally be applied to such identifiers.

Windows often contain text. Thus, a text string is needed sometimes (e.g. for a title of a window). This is denoted as '*String*'. The string may be empty.

[*String*]

| '' : *String*

This concludes the basic definitions which will be used as required in the rest of this monograph.

Chapter 2

Raster-Op Functions

Raster-op functions are useful when moving areas of pixels (e.g. parts of windows) around the screen on graphics display systems. Raster-op is now widely used in graphics systems, in particular for window systems. This chapter formally specifies raster-op functions and gives an example of its use. Readers not familiar with 'raster-op' may prefer to do some background reading first [Fole82, Newm81].

2.1 Operations on Pixel Values

Given two pixel values, we may perform a bit-wise '*NAND*' ('not and') on the two values to produce a new value. If both bits are set then the result is clear; if either bit is clear then the result is set.

$$
\begin{array}{|l}
\_ \ \underline{NAND} \ \_ : (Value \times Value) \rightarrow Value \\
\hline
\forall\, val_1, val_2, val : Value \bullet \\
\quad val_1 \ \underline{NAND} \ val_2 = val \Leftrightarrow \\
\quad\quad (\ \forall\, n : Zrange \bullet \\
\quad\quad\quad (val_1\ n = Set \wedge val_2\ n = Set) \Rightarrow \\
\quad\quad\quad\quad val\ n = Clear \wedge \\
\quad\quad\quad (val_1\ n = Clear \vee val_2\ n = Clear) \Rightarrow \\
\quad\quad\quad\quad val\ n = Set\ )
\end{array}
$$

All the other binary and unary logical functions may be defined in terms of this function. For example, we can define a unary '*NOT*' function.

$$
\begin{array}{|l}
NOT : Value \rightarrow Value \\
\hline
\forall\, val : Value \bullet \\
\quad NOT\ val = val \ \underline{NAND} \ val
\end{array}
$$

We can also define binary '*AND*', '*OR*' and '*XOR*' functions.

$$
\begin{array}{|l}
\_ \ \underline{AND} \ \_, \\
\_ \ \underline{OR} \ \_, \\
\_ \ \underline{XOR} \ \_ : (Value \times Value) \rightarrow Value \\
\hline
\forall val_1, val_2 : Value \bullet \\
\quad val_1 \ \underline{AND} \ val_2 = NOT(val_1 \ \underline{NAND} \ val_2) \wedge \\
\quad val_1 \ \underline{OR} \ val_2 = (NOT \ val_1) \ \underline{NAND} \ (NOT \ val_2) \wedge \\
\quad val_1 \ \underline{XOR} \ val_2 = (val_1 \ \underline{OR} \ val_2) \ \underline{AND} \ (val_1 \ \underline{NAND} \ val_2)
\end{array}
$$

2.2 Operations on Pixel Maps

A pixel map is considered to have the same shape as another if it can be 'moved' using a unique one-to-one function ((*offset* pix_0) in the definition below) to give it the same domain. Intuitively this implies that each of the pixel maps could be moved en masse about the domain space so that its domain is exactly the same as the other map.

$$
\begin{array}{|l}
\_ \ \underline{sameshape} \ \_ : Pixmap \leftrightarrow Pixmap \\
\hline
\forall map_1, map_2 : Pixmap \bullet \\
\quad map_1 \ \underline{sameshape} \ map_2 \Leftrightarrow \\
\qquad (\ \exists pix_0 : Pixel \bullet \\
\qquad\quad \mathrm{dom}((offset \ pix_0) \ ; map_1) = \mathrm{dom} \ map_2 \ )
\end{array}
$$

Consider a pair of (possibly overlapping) pixel maps which have the same shape.

$$
\begin{array}{|l}
MapPair \\
\hline
map_1, \\
map_2 : Pixmap \\
\hline
map_1 \ \underline{sameshape} \ map_2
\end{array}
$$

Two pixels in a pair of pixel maps with the same shape are considered to have the same position if the same offset function which relates the two maps also relates the two pixels. This is captured in '*samepos*' relation below. If the relation is true then it implies that the two pixels are in the same relative position within two pixel maps with the same shape. That is to say, if one of the pixel maps were to be moved on top of the other then one pixel would be exactly on top of the other.

$$
\begin{array}{|l}
\_ \ \underline{samepos} \ \_ : (Pixel \times Pixmap) \leftrightarrow (Pixel \times Pixmap) \\
\hline
\forall pix_1, pix_2 : Pixel;\ MapPair \mid \\
\quad pix_1 \in \mathrm{dom}\ map_1 \wedge pix_2 \in \mathrm{dom}\ map_2 \bullet \\
\qquad (pix_1, map_1)\ \underline{samepos}\ (pix_2, map_2) \Leftrightarrow \\
\qquad\quad (\ \exists pix_0 : Pixel \bullet \\
\qquad\qquad \mathrm{dom}((offset\ pix_0)\ ;\ map_1) = \mathrm{dom}\ map_2 \wedge \\
\qquad\qquad pix_1 \mapsto pix_2 \in offset\ pix_0\ )
\end{array}
$$

We may define operations similar to the bit-wise operations on values to apply to pixel maps. In this case by convention, the last operand of the function has the same domain as the result of the function. We start by defining the '*nand*' function.

$$
\begin{array}{|l}
\_ \ \underline{nand} \ \_ : (Pixmap \times Pixmap) \nrightarrow Pixmap \\
\hline
\forall MapPair;\ map : Pixmap \bullet \\
\quad map_1\ \underline{nand}\ map_2 = map \Leftrightarrow \\
\qquad \mathrm{dom}\ map = \mathrm{dom}\ map_2 \wedge \\
\qquad (\ \forall pix_1 : \mathrm{dom}\ map_1;\ pix_2 : \mathrm{dom}\ map_2 \mid \\
\qquad\quad (pix_1, map_1)\ \underline{samepos}\ (pix_2, map_2) \bullet \\
\qquad\qquad map_1(pix_1)\ \underline{NAND}\ map_2(pix_2) = map\ \ pix_2\ )
\end{array}
$$

Note that this operation is not commutative, unlike '*NAND*' since the second operand defines the domain of the result of the function. Specifically, if the domains of the pixel maps differ, the ordering is important. If the domains are the same then the ordering is unimportant.

$$
\begin{array}{lll}
MapPair & \vdash & \mathrm{dom}\ map_1 = \mathrm{dom}\ map_2 \Rightarrow \\
 & & \qquad map_1\ \underline{nand}\ map_2 = map_2\ \underline{nand}\ map_1
\end{array}
$$

Using the basic '*nand*' function, we may define fifteen further operations. Four of these degenerate to monadic functions. '*noop*' leaves a pixel map unaffected, '*not*' inverts all bits, '*clear*' clears all the bits and '*set*' sets all the bits.

$$
\begin{array}{|l}
noop, \\
not, \\
clear, \\
set : Pixmap \rightarrow Pixmap \\
\hline
\forall\, map : Pixmap \bullet \\
\quad noop\ map = map \wedge \\
\quad not\ map = map\ \underline{nand}\ map \wedge \\
\quad clear\ map = map\ \underline{nand}\ (not\ map) \wedge \\
\quad set\ map = not(clear\ map)
\end{array}
$$

There are five more important operators. These correspond to standard logical operations except '*copy*' which is extremely useful for moving pixel maps.

$$
\begin{array}{|l}
\_\ \underline{and}\ \_, \\
\_\ \underline{or}\ \_, \\
\_\ \underline{xor}\ \_, \\
\_\ \underline{nor}\ \_, \\
\_\ \underline{copy}\ \_ : (Pixmap \times Pixmap) \nrightarrow Pixmap \\
\hline
\forall\, MapPair \bullet \\
\quad map_1\ \underline{and}\ map_2 = \\
\qquad not(map_1\ \underline{nand}\ map_2) \wedge \\
\quad map_1\ \underline{or}\ map_2 = \\
\qquad (not\ map_1)\ \underline{nand}\ (not\ map_2) \wedge \\
\quad map_1\ \underline{xor}\ map_2 = \\
\qquad (map_1\ \underline{or}\ map_2)\ \underline{and}\ (map_1\ \underline{nand}\ map_2) \wedge \\
\quad map_1\ \underline{nor}\ map_2 = \\
\qquad not(map_1\ \underline{or}\ map_2) \wedge \\
\quad map_1\ \underline{copy}\ map_2 = \\
\qquad map_1\ \underline{or}\ (clear\ map_2)
\end{array}
$$

The rest of the operations are used less often but are detailed here for completeness.

$$
\begin{array}{l}
\_\ \underline{copyInverted}\ \_, \\
\_\ \underline{andReverse}\ \_, \\
\_\ \underline{andInverted}\ \_, \\
\_\ \underline{orReverse}\ \_, \\
\_\ \underline{orInverted}\ \_, \\
\_\ \underline{equiv}\ \_ : (Pixmap \times Pixmap) \nrightarrow Pixmap \\
\hline
\forall MapPair \bullet \\
\quad map_1\ \underline{copyInverted}\ map_2 = (not\ map_1)\ \underline{or}\ (clear\ map_2) \land \\
\quad map_1\ \underline{andReverse}\ map_2 = map_1\ \underline{and}\ (not\ map_2) \land \\
\quad map_1\ \underline{andInverted}\ map_2 = (not\ map_1)\ \underline{and}\ map_2 \land \\
\quad map_1\ \underline{orReverse}\ map_2 = map_1\ \underline{or}\ (not\ map_2) \land \\
\quad map_1\ \underline{orInverted}\ map_2 = (not\ map_1)\ \underline{or}\ map_2 \land \\
\quad map_1\ \underline{equiv}\ map_2 = (not\ map_1)\ \underline{xor}\ map_2
\end{array}
$$

This covers the sixteen possible raster-op boolean functions on two values.

2.3 Display Operations

A display screen consists of a pixel map.

$$
\begin{array}{l}
Display \\
\hline
screen : Pixmap \\
\hline
\end{array}
$$

During changes to the screen, its size does not change although the pixel values displayed on the screen may be updated.

$$
\begin{array}{l}
\Delta Display \\
\hline
Display \\
Display' \\
\hline
\mathrm{dom}\ screen' = \mathrm{dom}\ screen \\
\hline
\end{array}
$$

The screen may be updated using one of the raster-op functions previously defined. Some functions require a source and destination area whilst others degenerate into a single area.

$$
\begin{array}{|l}
\hline
RasterOp_1 \\
\Delta Display \\
area? : \mathbb{P}\ Pixel \\
op? : Pixmap \rightarrow Pixmap \\
\hline
screen' = screen \oplus op?(area? \lhd screen) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
RasterOp_2 \\
\Delta Display \\
area? : \mathbb{P}\ Pixel \\
from? : Pixel \\
op? : (Pixmap \times Pixmap) \nrightarrow Pixmap \\
\hline
screen' = screen \oplus \\
\quad op?\ (\ ((offset\ from?)(\!|area?|\!)) \lhd screen\ ,\ area? \lhd screen\ ) \\
\hline
\end{array}
$$

2.4 An Example – Swapping Pixel Maps

Consider two separate non-overlapping pixel maps with the same shape.

$$
\begin{array}{|l}
\hline
SepPair \\
MapPair \\
\hline
\neg\ map_1\ \underline{overlaps}\ map_2 \\
\hline
\end{array}
$$

$$\Delta SepPair \ \widehat{=}\ SepPair \wedge SepPair'$$

We may swap a pair of pixel maps with the same shape using the '*copy*' operation.

$$
\begin{array}{|l}
\hline
CopySwap \\
\Delta SepPair \\
\hline
map_1' = map_2\ \underline{copy}\ map_1 \\
map_2' = map_1\ \underline{copy}\ map_2 \\
\hline
\end{array}
$$

In practice, these two '*copy*' operations cannot be carried out simultaneously. A third copy is necessary and additionally a temporary pixel map area is required. This can easily be expressed by using three schemas, one for each operation, and then combining them using the schema composition operator (';'). This is left as a exercise for the reader.

Altematively, the '*xor*' raster-op function may be used. Three sequential operations are still necessary, but the use of a temporary buffer area is eliminated. The following two schemas '*xor*' one or other of a pair of pixel maps with its opposite number.

$$
\begin{array}{|l}
\hline
Xor_1 \\
\Delta SepPair \\
\hline
map_1' = map_2 \ \underline{xor} \ map_1 \\
map_2' = map_2 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
Xor_2 \\
\Delta SepPair \\
\hline
map_1' = map_1 \\
map_2' = map_1 \ \underline{xor} \ map_2 \\
\hline
\end{array}
$$

A swap may be achieved between the two pixel maps by applying tokenxor three times sequentially as follows.

$$XorSwap \ \widehat{=} \ Xor_1 \ ; \ Xor_2 \ ; \ Xor_1$$

By symmetry, the following is also true.

$$\Delta SepPair \ \vdash \ XorSwap \Leftrightarrow (Xor_2 \ ; \ Xor_1 \ ; \ Xor_2)$$

The *XorSwap* operation has exactly the same effect as using the '*copy*' operator twice simultaneously, as in the *CopySwap* operation.

$$\Delta SepPair \ \vdash \ CopySwap \Leftrightarrow XorSwap$$

2.5 Conclusion

We have formally specified the raster-op function and how this may be applied to a graphics display. We have given an example of its use for swapping

areas of a display. By defining operations performed on rectangular areas, we have specified some of the underlying operations necessary for a window system.

Chapter 3

An Example Window System

Window management systems are now used extensively for user interfaces to computer systems. This chapter outlines an example window system using formal specification techniques. An abstract view of the system is developed and some basic operations are given.

3.1 State of the Window System

The window display may be modelled as a sequence of windows against a background 'window' which is the size of the display itself. The order of the sequence defines which windows are on top in the case of overlapping windows, in ascending order. Only parts of windows which are contained within the background area are displayed.

$$
\begin{array}{|l}
\hline
SYS \\
windows : \mathrm{seq}\, Pixmap \\
screen, background : Pixmap \\
\hline
screen = background \oplus (\mathrm{dom}\, background \lhd \oplus / windows) \\
\hline
\end{array}
$$

In this simple example, we shall ignore the complication of window identifiers. We shall simply use the position of the window in the sequence to identify it, assuming that the user of the system keeps track of which window is which.

Note that the user can only see the display screen.

$$View \;\widehat{=}\; SYS \setminus (windows, background)$$

Initially there are no windows and thus only the background is displayed.

$$InitSYS \;\widehat{=}\; [\, SYS' \mid windows' = \langle\rangle \,]$$

$$InitSYS \vdash screen' = background'$$

During changes to the display, the background and hence the size of the display always remain the same.

$$\Delta SYS \;\widehat{=}\; [\, SYS;\ SYS' \mid background' = background \,]$$

Some operations may leave the state of the system unchanged, for example during a status operation or if an error in the input is detected.

$$\Xi SYS \;\widehat{=}\; [\, \Delta SYS \mid \theta SYS' = \theta SYS \,]$$

3.2 Operations on Windows

In order to use the system, we must have the ability to create windows. These are created on top of all the existing windows. We specify that they must fit within the display background.

AddWindow$_0$

ΔSYS
$window? : Pixmap$

$\mathrm{dom}\ window? \subseteq \mathrm{dom}\ background$

$windows' = windows \frown \langle window? \rangle$

The ability to update windows is very useful. This may involve changing the size of the window or its contents or moving it about the screen. Again, the updated window must still fit within the display area.

UpdateWindow$_0$

ΔSYS
$which? : \mathbb{N}$
$window? : Pixmap$

$which? \in \mathrm{dom}\ windows$

$\mathrm{dom}\ window? \subseteq \mathrm{dom}\ background$

$windows' = windows \oplus \{which? \mapsto window?\}$

It is desirable to be able to uncover a window which is partially or even totally obscured by other windows. This can be done by moving the window to the end of the sequence of displayed windows.

$$
\begin{array}{|l}
\hline
\textit{ExposeWindow}_0 \\
\Delta SYS \\
\textit{which}? : \mathbb{N} \\
\hline
\textit{which}? \in \operatorname{dom} \textit{windows} \\
\textit{windows}' = \textit{top}\ \textit{which}?\ \textit{windows} \\
\hline
\end{array}
$$

Sometimes it is useful to simply rotate the order of the displayed windows, one at a time, moving the bottommost window to the top.

$$
\begin{array}{|l}
\hline
\textit{RotateWindows}_0 \\
\Delta SYS \\
\hline
\textit{windows} \neq \langle\rangle \\
\textit{windows}' = \textit{top}\ 1\ \textit{windows} \\
\hline
\end{array}
$$

We also wish to be able to delete windows. For instance, we could delete the topmost window.

$$
\begin{array}{|l}
\hline
\textit{RemoveTop}_0 \\
\Delta SYS \\
\hline
\textit{windows} \neq \langle\rangle \\
\textit{windows}' = \textit{front}\ \textit{windows} \\
\hline
\end{array}
$$

Alternatively, we may wish to specify which window is to be removed.

$$
\begin{array}{|l}
\hline
\textit{RemoveWindow}_0 \\
\Delta SYS \\
\textit{which}? : \mathbb{N} \\
\hline
\textit{which}? \in \operatorname{dom} \textit{windows} \\
\textit{windows}' = \textit{compact}(\{\textit{which}?\} \lhd \textit{windows}) \\
\hline
\end{array}
$$

3.3 Error Conditions

The operations covered so far detail what should happen in the event of no errors. We wish to be able to report the status of an operation.

$$
\begin{aligned}
Report ::= & \text{'OK'} \\
| & \text{'Not a window'} \\
| & \text{'No windows'} \\
| & \text{'Invalid window'}
\end{aligned}
$$

We must report the fact that the operation was successful if this is the case.

Success

$$rep! : Report$$

$$rep! = \text{'OK'}$$

If errors do occur, then these need to be reported as well. For example, an invalid window may be specified.

NotAWindow

$$\Xi SYS$$
$$which? : \mathbb{N}$$
$$rep! : Report$$

$$which? \notin \mathrm{dom}\ windows$$
$$rep! = \text{'Not a window'}$$

It is possible that there are no windows displayed when one is required.

NoWindows

$$\Xi SYS$$
$$rep! : Report$$

$$windows = \langle\rangle$$
$$rep! = \text{'No windows'}$$

A specified window may not be within the background area.

BadWindow

ΞSYS
window? : *Pixmap*
rep! : *Report*

$\neg$ (dom *window?* $\subseteq$ dom *background*)

rep! = 'Invalid window'

We may include these errors with the previously defined operations which ignored error conditions, to produce total operations.

$$AddWindow_1 \;\widehat{=}\; (AddWindow_0 \wedge Success) \vee BadWindow$$

$$UpdateWindow_1 \;\widehat{=}\; (UpdateWindow_0 \wedge Success) \vee BadWindow \vee NotAWindow$$

$$ExposeWindow_1 \;\widehat{=}\; (ExposeWindow_0 \wedge Success) \vee NotAWindow$$

$$RotateWindows_1 \;\widehat{=}\; (RotateWindows_0 \wedge Success) \vee NoWindows$$

$$RemoveTop_1 \;\widehat{=}\; (RemoveTop_0 \wedge Success) \vee NoWindows$$

$$RemoveWindow_1 \;\widehat{=}\; (RemoveWindow_0 \wedge Success) \vee NotAWindow$$

3.4 Status Operations

The contents of an existing window may be of interest.

GetWindow$_0$

SYS
which? : $\mathbb{N}$
window! : *Pixmap*

---

which? $\in$ dom *windows*

window! = *windows which?*

We can make this operation total as well.

$$GetWindow \;\widehat{=}\; (GetWindow_0 \wedge Success) \vee NotAWindow$$

It is useful to know which window a given pixel position is in, for example when pointing at a position on the screen with a mouse driven cursor. Given the position, the sequence number of the window is required. There may be several windows at the specified position but we are only interested in the topmost one.

WhichWindow$_0$

$$\begin{array}{l} SYS \\ position? : Pixel \\ which! : \mathbb{N} \\ \hline position? \in \mathrm{dom}(\oplus / windows) \\ which! = max\ \{n : \mathrm{dom}\ windows \mid \\ \qquad\qquad position? \in \mathrm{dom}(windows\ n)\} \end{array}$$

However, it is possible that the cursor position may not be in a valid window. In this case we simply record this fact by making '*which*!' invalid. This is because this schema will be combined with others to produce an appropriate error message later.

$$Invalid == 0$$

NotInWindow

$$\begin{array}{l} SYS \\ position? : Pixel \\ which! : \mathbb{N} \\ \hline position? \notin \mathrm{dom}(\oplus / windows) \\ which! = Invalid \end{array}$$

To make the operation total, we combine these two operations.

$$WhichWindow \;\widehat{=}\; WhichWindow_0 \vee NotInWindow$$

3.5 Upgraded Operations

Often when a window is updated, we wish to modify the original window rather than completely replacing it. Sometimes we wish to change the shape of the window by either enlarging it or reducing it, or a combination of the two. Any new area exposed may be given any desired background pixel values.

Reshape

$$area?,$$
$$window?,$$
$$window! : Pixmap$$

$$window! = area? \oplus (\text{dom } area?) \lhd window?$$

This schema is a general operation concerning the input and output of a window and hence the change in state of the window system need not be recorded.

Often we want to move a window. We can define a one-to-one mapping to do this. If the window cannot be moved in its entirety using the mapping given, then it is left unaffected.

Move

$$mapping? : Pixel \rightarrowtail Pixel$$
$$window?,$$
$$window! : Pixmap$$

$$\text{dom } window? \subseteq \text{ran } mapping? \Rightarrow window! = mapping? \fatsemi window?$$

$$\neg (\text{dom } window? \subseteq \text{ran } mapping?) \Rightarrow window! = window?$$

At other times we simply wish to modify the contents of the window. We ignore any modifications outside the area of the window.

Modify

$$mods?,$$
$$window?,$$
$$window! : Pixmap$$

$$window! = window? \oplus (\text{dom } window?) \lhd mods?$$

Again the two previous schemas have nothing to do with the state of the window system. However by combining the last three schemas with a record of the change of state, we can then combine them with previous operations to produce some new window operations.

$$ReshapeWindow \mathrel{\widehat=} GetWindow \gg Reshape \gg UpdateWindow_0$$
$$MoveWindow \mathrel{\widehat=} GetWindow \gg Move \gg UpdateWindow_0$$
$$ModifyWindow \mathrel{\widehat=} GetWindow \gg Modify \gg UpdateWindow_0$$

Each of these operations selects a particular window from the system, transforms the window in some way, and then updates the system with the new window.

To bring a window to the top, we point at it using the mouse cursor and click a button to execute the operation. This can be modelled using a combination of previously defined operations.

$$ExposeWindow \mathrel{\widehat=} WhichWindow \gg ExposeWindow_1$$

We may want a process associated with a particular window to decide whether a given operation may be executed, using some unknown protection criterion.

Decide

$allowed? : \mathbb{P}\,\mathbb{N}$
$which?,$
$which! : \mathbb{N}$

$which? \in allowed? \Rightarrow which! = which?$

$which? \notin allowed? \Rightarrow which! = Invalid$

Thus a process may decide whether a user is allowed to remove a window.

$$RemoveWindow \mathrel{\widehat=} WhichWindow \gg Decide \gg RemoveWindow_1$$

3.6 Conclusion

We have introduced a model for a simple window system together with some basic operations which may be performed. We have shown how some of these operations may be combined to produce a set of useful windowing

operations. We have used the schema piping operator ('$\gg$') to combine some of the operations rather than the more normal method of using schema inclusion since this gives a better idea of how the operations might be joined together in practice.

Nothing has been said about the nature of pixels in this chapter. Little has been said about the contents of windows. However these issues could be expanded using the model given to produce a complete specification for a windowing system.

Part II

A Comparison of Three Real Window Systems

Introduction

In Part II of this monograph, **Z** [Haye87, King88, Loom88, Spiv88a, Spiv89] is used to describe parts of three window systems, *WM* [Neuw85, Rose85], the Blit [Pike84, Unix85] and X [Gett85, Gett86, Sche86]. These systems have been developed independently at Carnegie-Mellon University, AT&T Bell Laboratories at Murray Hill and Massachusetts Institute of Technology. All the systems use bit-mapped raster displays with a keyboard and mouse for user input.

A high level description of the state of each system and a selection of window operations is presented to give a flavour of each and to allow them to be compared. The operations covered are those available to applications programs via library procedure calls. In general, similar operations are available to the user under mouse control. Some background reading may be helpful for readers not familiar with windowing techniques [Fole82, Newm81].

Only operations directly concerned with windowing are covered. The systems described also include many other operations such as graphics, raster-op, mouse and cursor routines, etc. Each description covers the following:

- State of the system and initialisation.
- Windowing operations, ignoring error conditions.
- Error conditions and status reports.
- Simplifications and assumptions in the specification.
- General comments on the system.

Menu facilities are not covered in detail for the sake of brevity. Additionally, the parameters supplied to some operations have been simplified to aid clarity in the brief descriptions given.

Graphics facilities within windows are not included. However **Z** is suitable for this. For example, it has been to specify parts of the graphics standard GKS [Arno87].

Chapter 4

The ITC 'WM' Window Manager (CMU)

WM [Neuw85, Rose85], part of the 'Andrew' distributed system, is a window manager developed at the Information Technology Center (ITC) at Carnegie-Mellon University [Saty84, West85]. This runs on UNIX* based workstations designed eventually to be networked on a very large scale (c5,000-10,000 nodes). Because of the distributed file system, any authorised person may also use any other workstation on the network, and indeed create windows on other workstations remotely.

4.1 State

The state of the system is introduced in stages. In this model we consider a single machine for simplicity since we are concerned with how the window manager works rather than how the network operates. Operations over the network will be detailed later.

Each window has a number of pieces of information associated with it. These include a header area for titles and other information, and a separate body area to hold the actual contents of the window. These do not overlap and together they make up the pixel map of the displayed window. In practice, the header is a thin rectangular area just above its associated body.

*UNIX is a trademark of AT&T Bell Laboratories.

$$
\begin{array}{|l}
\textit{Map} \\
\hline
\textit{header}, \textit{body}, \textit{map} : \textit{Pixmap} \\
\textit{area} : \mathbb{P}\ \textit{Pixel} \\
\hline
(\textit{header}, \textit{body}) \text{ partition } \textit{map} \\
\textit{area} = \text{dom } \textit{map} \\
\hline
\end{array}
$$

The user can request a window to lie within a specified range of dimensions and can also explicitly ask for a window body to be hidden from view or exposed on the screen. Each window has a title which can be set by the user. This information is used by the window manager to lay out the window on the screen, although there is no guarantee that what the user asks for is what the user gets!

$$\textit{HideExpose} ::= \textit{Hide} \mid \textit{Expose}$$

$$
\begin{array}{|l}
\textit{Control} \\
\hline
\textit{title} : \textit{String} \\
\textit{control} : \textit{HideExpose} \\
\textit{xylimits} : \textit{Pixel} \times \textit{Pixel} \\
\hline
\textit{first xylimits} \leq \textit{second xylimits} \\
\hline
\end{array}
$$

Together these make up the information describing a particular window.

$$\textit{Info} \ \widehat{=}\ \textit{Map} \wedge \textit{Control}$$

There are a finite number of windows on a particular screen. One of these is considered to be the currently selected window. This may be 'undefined' sometimes. Most *WM* library functions take effect on the currently selected window. Each window has information, including a pixel map, associated with it.

$$\textit{Undefined} : \textit{Window}$$

WM

$$windows : \mathbb{F}\ Window$$
$$current : Window$$
$$contents : Window \pfun Info$$

$$Undefined \notin windows$$
$$windows = \mathrm{dom}\ contents$$
$$current \in windows \cup \{Undefined\}$$

The display screen consists of the background overlaid with windows. The window pixel maps do not overlap. All windows are contained within the background area.

WM

$$WM$$
$$maps : Window \pfun Pixmap$$
$$areas : Window \pfun (\mathbb{P}\ Pixel)$$
$$screen, background : Pixmap$$

$$maps = contents\ ;\ (\lambda\ Info \bullet map)$$
$$areas = contents\ ;\ (\lambda\ Info \bullet area)$$
$$\mathrm{disjoint}\ areas$$
$$\bigcup(\mathrm{ran}\ areas) \subseteq \mathrm{dom}\ background$$
$$screen = background \oplus \bigcup(\mathrm{ran}\ maps)$$

You can have as many windows as you need, subject to the restriction that the *WM* process can handle at most 20 windows, including hidden windows and windows requested by other programs, at one time.

$$MaxWindows : \mathbb{N}$$

$$MaxWindows = 20$$

We can include this limitation in our model of the state.

$$WM \ \widehat{=}\ [\ WM \mid \#windows \leq MaxWindows\ ]$$

The size of a window on the user's display is one of the resources that the *Window Manager* allocates. A program can request a given size, and

WM will take the requested size into account when making decisions, but it does not guarantee a particular size. This process is modelled as a function of the system. The number of windows is not changed by this function. Additionally, control information supplied by the user is left unchanged.

$$WINDOWS == Window \nrightarrow Info$$

WM

$$WM$$
$$adjust : WINDOWS \rightarrow WINDOWS$$

$$\forall w, w' : WINDOWS \mid w' = adjust\ w \bullet$$
$$\#w' = \#w \land$$
$$w' ; (\lambda Info \bullet \theta Control) = w ; (\lambda Info \bullet \theta Control)$$

Initially there are no windows and the current window is undefined.

InitWM

$$WM'$$

$$windows' = \varnothing$$
$$current' = Undefined$$

Operations change the state of the system. However the background and hence the size of the screen remains constant. Additionally the algorithm to adjust the size of windows does not change.

ΔWM

$$WM$$
$$WM'$$

$$background' = background$$
$$adjust' = adjust$$

Sometimes the state of the system is unaffected during an operation.

$$\Xi WM \ \widehat{=}\ [\ \Delta WM \mid \theta WM' = \theta WM\ ]$$

Many operations are concerned with the current window. Hence we define a schema giving a partial specification covering all common aspects of such operations. This can be used to shorten subsequence specifications of these operations and reduce repetition. The names of such schemas are prepended with 'Φ' to distinguish these from actual operations.

$\Phi Current$

ΔWM
$Info$
$Info'$

$current \in windows$

$current' = current$

$\theta Info = contents\ current$

$contents' = adjust\ (contents \oplus \{current \mapsto \theta Info'\})$

This leaves a valid current window the same, but updates the information associated with it in some (as yet unspecified) way.

4.2 Window Creation and Deletion

When a window is created, the system adjusts all the windows in the system appropriately. The window body is exposed when it is created. In practice, the operation also takes the name of a host as input since a window may be created anywhere on the network of workstations. However this is detailed later.

$NewWindow_0$

ΔWM
$w! : Window$
$Info$

$\#windows < MaxWindows$

$w! \notin windows \cup \{Undefined\}$

$current' = w!$

$control = Expose$

$contents' = adjust\ (contents \cup \{w! \mapsto \theta Info\})$

The currently selected window can be deleted.

$$
\begin{array}{|l}
\hline
\textit{DeleteWindow}_0 \\
\Delta WM \\
\hline
current \in windows \\
current' = Undefined \\
contents' = adjust\ (\{current\} \lhd contents) \\
\hline
\end{array}
$$

4.3 Window Size

A program can request a given size range, and *WM* will take the requested size into account when making decisions, but it does not guarantee a particular size. The rest of the window information is unaffected. The windows will be adjusted by the system as necessary (i.e. any or all of the displayed windows may change shape as a result of setting the size of one particular window).

$$
\begin{array}{|l}
\hline
\textit{SetDimensions}_0 \\
\Phi Current \\
minxy?, maxxy? : Pixel \\
\hline
map' = map \\
title' = title \\
control' = control \\
xylimits' = (minxy?, maxxy?) \\
\hline
\end{array}
$$

The size of the body of the currently selected window can be returned. If the window is actually hidden (i.e. *WM* has adjusted the window to display the header only), then the returned size is empty.

$GetDimensions_0$

$\Phi Current$

$wh! : Pixel$

$xy_1, xy_2 : Pixel$

$\theta Info' = \theta Info$

$\text{dom } body = xy_1 \mathinner{..} xy_2$

$wh! = xy_2 - xy_1$

4.4 Windows Visibility

A window is considered 'visible' when both its header and its body are displayed and 'hidden' when only its header is displayed. Windows are visible when they are first created and remain so unless the user hides them. A program can also control window visibility. A visible window may be hidden.

$HideMe_0$

$\Phi Current$

$map' = map$

$title' = title$

$control' = Hide$

$xylimits' = xylimits$

Similarly, a hidden window may be exposed.

$ExposeMe_0$

$\Phi Current$

$map' = map$

$title' = title$

$control' = Expose$

$xylimits' = xylimits$

Note that the state of '*control*' before the operation has not been checked above, so $HideMe_0$ will leave a hidden window hidden and $ExposeMe_0$ will leave a visible window exposed.

4.5 Other Window Operations

A window may be explicitly selected as the current window, until another window is selected or created. All output will be sent to the selected window.

SelectWindow$_0$

ΔWM

$w? : Window$

$w? \in windows$

$current' = w?$

$contents' = contents$

The title of a window may be set. This involves placing a text string in the header section of the window contents.

SetTitle$_0$

$\Phi Current$

$s? : String$

$map' = map$

$title' = s?$

$control' = control$

$xylimits' = xylimits$

The body of the currently selected window may be set to white.

ClearWindow$_0$

$\Phi Current$

$header' = header$

$body' = setval\ White\ body$

$title' = title$

$control' = control$

$xylimits' = xylimits$

Other operations supplied by the *WM* library include line, text and string drawing, raster operations, operations to save and restore parts of the picture, input handling, menus, mouse input, etc. In addition, new operations may be added; at the time that this specification was formulated *WM* are still under development.

4.6 Errors

There is a *Null* window identifier which is never a valid window.

$$Null : Window$$

$$WM \triangleq [\ WM \mid Null \notin windows\ ]$$

ΔWM and ΞWM are redefined appropriately.

Some operations return a window identifier. If this is non-null then the operation is successful.

Success

ΔWM

$w! : Window$

$w! \neq Null$

Alternatively a error may occur. There is a limit on the number of windows which *WM* can handle. This could cause an error when creating a new window.

TooManyWindows

$$\Xi WM$$
$$w! : Window$$
$$\#windows \geq MaxWindows$$
$$w! = Null$$

We can now make the operation to create a new window total.

$$NewWindow_1 \;\widehat{=}\; (NewWindow_0 \wedge Success) \vee TooManyWindows$$

The current window may be undefined when one is required.

NoCurrentWindow

$$\Xi WM$$
$$current = Undefined$$

$$DeleteWindow_1 \;\widehat{=}\; DeleteWindow_0 \vee NoCurrentWindow$$
$$SetDimensions_1 \;\widehat{=}\; SetDimensions_0 \vee NoCurrentWindow$$
$$GetDimensions_1 \;\widehat{=}\; GetDimensions_0 \vee NoCurrentWindow$$
$$SetTitle_1 \;\widehat{=}\; SetTitle_0 \vee NoCurrentWindow$$
$$ClearWindow_1 \;\widehat{=}\; ClearWindow_0 \vee NoCurrentWindow$$

A window may always be hidden or exposed, even if this does not affect its state.

An invalid window may be selected.

InvalidWindow

$$\Xi WM$$
$$w? : Window$$
$$w? \notin windows$$

$$SelectWindow_1 \;\widehat{=}\; SelectWindow_0 \vee InvalidWindow$$

4.7 The ITC Network

In practice, as mentioned previously, there are many window managers, each running on a host workstation on a large network. Some hosts are running *WM*. All workstations have unique host names and all windows have unique IDs across the network.

ITC

$$hosts : \mathbb{P}\ String$$
$$wms : String \pfun WM$$

$$\mathrm{dom}\ wms \subseteq hosts$$
$$\mathrm{disjoint}\ (wms\ ;\ (\lambda\ WM \bullet windows))$$

To start with there are no hosts (and hence no window managers) on the network.

$$InitITC\ \widehat{=}\ [\ ITC' \mid hosts' = \emptyset\ ]$$

Operations cause changes on the network.

$$\Delta ITC\ \widehat{=}\ ITC \land ITC'$$

Hosts can be added to the system (e.g. booting up) and removed (e.g. crashing or powering down).

AddHost

$$\Delta ITC$$
$$host? : String$$

$$host? \notin hosts \cup \{``\}$$
$$hosts' = hosts \cup \{host?\}$$
$$wms' = wms$$

RemoveHost

$$\Delta ITC$$
$$host? : String$$

$$host? \in hosts$$
$$hosts' = hosts \setminus \{host?\}$$
$$wms' = \{host?\} \rhd wms$$

Operations can be initiated on a particular 'local' host. These do not affect the host names on the network.

$$
\begin{array}{|l}
\hline
\Phi Host \\
\Delta ITC \\
localhost : String \\
\hline
hosts' = hosts \\
localhost \in hosts \\
\hline
\end{array}
$$

For example, *WM* may be executed on a host, and may be subsequently killed.

$$
\begin{array}{|l}
\hline
Exec\,WM \\
\Phi Host \\
initwm : InitWM \\
\hline
localhost \notin \mathrm{dom}\ wms \\
wms' = wms \cup \{localhost \mapsto initwm\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
KillWM \\
\Phi Host \\
\hline
localhost \in \mathrm{dom}\ wms \\
wms' = \{localhost\} \lhd wms \\
\hline
\end{array}
$$

WM operations can be modelled in the global context of the network by updating the state of *WM* on a particular local host which is already running *WM*.

$$
\begin{array}{|l}
\hline
\Phi WM \\
\Phi Host \\
\Delta WM \\
host : String \\
\hline
\theta WM = wms\ host \\
\theta WM' = wms'\ host \\
\hline
\end{array}
$$

The *Window Manager* on the local host can be requested to create new windows on any machine on the ITC network that is running a *WM* process by supplying the appropriate host name. Alternatively, specifying a null host parameter results in a request for a window on the local machine. This is the normal mode of operation.

NewWindow$_{ITC}$

$$NewWindow_{\mathbb{I}}$$
$$\Phi WM$$
$$host? : String$$

$$host? = \text{``''} \Rightarrow host = localhost$$
$$host? \neq \text{``''} \Rightarrow host = host?$$

Other window operations discussed previously may be described in the global network context. For example:

$$DeleteWindow_{ITC} \;\widehat{=}\; DeleteWindow_{\mathbb{I}} \wedge \Phi WM$$

If the current window is on the local machine, then the operation is executed locally, otherwise it is carried out over the network.

4.8 Simplifications and Assumptions

For the purposes of brevity, the pop-up menus supplied by *WM* have been ignored in the description given. These could easily be added to the state specification by including them as extra window information in a schema called *Menu*.

$$Info \;\widehat{=}\; Info \wedge Menu$$

In practice, windows are not adjusted immediately, but when the *Window Manager* next makes a size decision (e.g. when the user requests *WM* to proportion the windows). This is not modelled here. In addition, the way in which the windows are proportioned is not specified since this is not covered in the documentation used to formulate this specification [Neuw85].

4.9 Comments

The *WM* window manager provides a simple system with non-overlapping windows. Hence no notion of window ordering is required. The idea of a

'current' window for each process using *WM* means that this information is held as part of the state of the system and need not be specified as input to many window operations. Windows are automatically reduced in size by the system when there is not enough space on the screen. This simplifies the task of organising windows for the user.

Chapter 5

Blit Windows (AT&T)

The Blit [Pike84, Unix85], developed at Bell Labs, Murray Hill, is more like an intelligent terminal than a workstation. It is diskless and interacts with a remote host running the Bell Labs Eighth Edition UNIX via a 9600 baud (slow) RS-232 serial line. It has its own simple process scheduler and a bit-mapped display. Programs can be run on the Blit (downloaded from the remote host), on the remote host itself using a standard window terminal emulator process on the Blit, or on both using two special purpose programs which interact with each other over the serial line. Deciding how to split a program between the Blit and remote host is a tricky but interesting problem.

5.1 State

The Blit contains 'layers' which are analogous to windows on most other systems. However there is no protection between layers. Each layer has an rectangular region on the screen associated with it. Here we model this simply as a partial function from pixel points to values.

$$Layer == Window$$

$$Point == Pixel$$

Each pixel point is two-valued – i.e. each window is a simple bit map.

$$Zsize = 1$$

Several layers (with associated rectangular windows) may exist simultaneously. The layers are ordered as a sequence for reasons that will be seen in a moment. There is an invalid null layer for error returns (see later).

$| \; NullLayer : Layer$

Blit

$$
\begin{array}{l}
layers : \mathrm{seq}\ Layer \\
windows : Layer \pfun Rectangle \\
rects : \mathrm{seq}\ Rectangle \\
\hline
\mathrm{ran}\ layers = \mathrm{dom}\ windows \\
NullLayer \notin \mathrm{ran}\ layers \\
rects = layers \fatsemi windows
\end{array}
$$

Several processes may also exist simultaneously in the Blit. Each process has an associated program and state. A process may be disabled or enabled. However the rest of this specification is not concerned with the state of processes, but it is included here for completeness. Each process is normally associated with a layer. Creating a process without a layer or vice versa is dangerous.

$[Program, State]$

Proc

$$
\begin{array}{l}
prog : Program \\
state : State \\
l : Layer
\end{array}
$$

The Blit includes no window 'manager' as such since any process has access to the entire screen. There are a series of processes in the system each identified uniquely by a process id. There is a null invalid id which is returned by operations to indicate an error. One of the processes may be assigned to receive mouse and keyboard events.

$[\ Id\ ]$

$| \; NullId : Id$

Blit

$$
\begin{array}{l}
Blit \\
procs : Id \pfun Proc \\
receiver : Id \\
\hline
NullId \notin \mathrm{dom}\ procs \\
receiver \in \mathrm{dom}\ procs \cup \{NullId\} \\
\forall proc : \mathrm{ran}\ procs \bullet \\
\quad proc.l \in \mathrm{ran}\ layers \cup \{NullLayer\}
\end{array}
$$

A process may be associated with a default terminal emulation program if desired.

$$Blit \ \widehat{=}\ [\ Blit;\ default : Program\ ]$$

The background may be considered as another layer which defines the size of the screen. The display consists of all the layers overlaid on top of the background. All layers are contained within the background. The order is determined from the position in the sequence (first at the bottom, last on top).

Blit

$$
\begin{array}{l}
Blit \\
screen, background : Rectangle \\
\hline
\mathrm{dom}(\oplus / rects) \subseteq \mathrm{dom}\ background \\
screen = background \oplus (\oplus / rects)
\end{array}
$$

Initially there are no layers or processes in the system.

InitBlit

$$
\begin{array}{l}
Blit' \\
\hline
layers' = \langle\rangle \\
procs' = \emptyset \\
receiver' = NullId
\end{array}
$$

Operations do not change the default terminal program or the background of the display.

$$
\begin{array}{|l}
\hline
\Delta Blit \\
Blit \\
Blit' \\
\hline
default' = default \\
background' = background \\
\hline
\end{array}
$$

Some operations do not affect the state of the Blit.

$$\Xi Blit \;\widehat{=}\; [\, \Delta Blit \mid \theta Blit' = \theta Blit \,]$$

Often operations only affect layers and all the processes in the system are left unaffected.

$$
\begin{array}{|l}
\hline
\Phi Layer \\
\Delta Blit \\
\hline
procs' = procs \\
receiver' = receiver \\
\hline
\end{array}
$$

Similarly, processes are often changed whilst leaving all the layers in the system unaffected.

$$
\begin{array}{|l}
\hline
\Phi Proc \\
\Delta Blit \\
\hline
layers' = layers \\
windows' = windows \\
\hline
\end{array}
$$

5.2 New Layers

A layer may be created in a specified rectangle in the physical display bitmap. The address of the layer is returned.

NewLayer$_0$

$\Phi Layer$
$r? : Rectangle$
$l! : Layer$

$\mathrm{dom}\ r? \subseteq \mathrm{dom}\ background$

$l! \notin \mathrm{ran}\ layers$

$layers' = layers \frown \langle l! \rangle$

$windows' = windows \cup \{l! \mapsto r?\}$

A layer may also be de-allocated. The associated process should also be freed for safety, but this is a separate operation.

DelLayer$_0$

$\Phi Layer$
$l? : Layer$

$l? \in \mathrm{ran}\ layers$

$layers' = layers \setminus remove \{l?\}$

$windows' = \{l?\} \lhd windows$

5.3 New Processes

A new process can be allocated. A handle on the process is returned. Note that the associated layer is undefined. The process program is often the default terminal emulation program and in practice this is specified using a null argument.

NewProc$_0$

$\Phi Proc$
$f? : Program$
$id! : Id$
$proc : Proc$

$proc.prog = f?$

$procs' = procs \cup \{id! \mapsto proc\}$

$receiver' = receiver$

A process may be created using the standard user interface to select the rectangle for the process's layer. This associates the process with the layer.

$$NewWindow_0 \mathrel{\widehat=} NewProc_0 \fatsemi NewLayer_0$$

$$NewWindow_0 \mathrel{\widehat=} [\, NewWindow_0 \mid l! = proc.l \,]$$

A layer may be selected so that the process in that layer becomes the receiver of mouse and keyboard events.

$ToLayer_0$

$\Phi Proc$

$l? : Layer$

$proc : Proc$

$proc \in \mathrm{ran}\ procs$

$l? = proc.l$

$procs' = procs$

$receiver' = procs^{\sim}\ proc$

The process whose layer is indicated by the mouse may be returned.

$GetProc_0$

$\Xi Blit$

$l? : Layer$

$proc! : Proc$

$proc! \in \mathrm{ran}\ procs$

$proc!.l = l?$

Alternatively, a handle on all the processes in the system may be returned.

$GetProcTab_0$

$\Xi Blit$

$procs! : Id \pfun Proc$

$procs! = procs$

5.4 Mouse Operations

The Blit includes a mouse. This controls the position of a cursor on the screen. Additionally, any combination of the three buttons on the mouse may be pressed at any time.

$$Button ::= Button1 \mid Button2 \mid Button3$$

Mouse

$$xy : Point$$
$$buttons : \mathbb{P}\, Button$$

Some pixel positions on the display screen may be associated with a particular layer. This is the layer which is visible at that particular pixel. Otherwise the background is visible at that point. The gun-sight cursor is used to find a particular layer.

Gunsight

$$\Xi Blit$$
$$pos? : Mouse$$
$$l! : Layer$$

$$pos?.xy \in \mathrm{dom}(\oplus / rects) \Rightarrow$$
$$\quad l! = layers(max\ \{n : \mathrm{dom}\ rects \mid pos?.xy \in \mathrm{dom}(rects\ n)\})$$
$$pos?.xy \notin \mathrm{dom}(\oplus / rects) \Rightarrow$$
$$\quad l! = NullLayer$$

We can now give a more complete definition for *GetProc*.

$$GetProc_1 \mathrel{\widehat=} Gunsight \gg GetProc_0$$

The box cursor is used to pick out a rectangular area. This is done by sweeping out a rectangle whilst button 3 is depressed.

Box

$\Xi Blit$

$pos1?, pos2? : Mouse$

$r! : Rectangle$

$pos1?.buttons = \{Button3\}$

$pos2?.buttons = \emptyset$

$\mathrm{dom}\, r! = \mathrm{dom}\, background \cap$
$\quad ((pos1?.xy \underline{min} pos2?.xy) \mathinner{..} (pos1?.xy \underline{max} pos2?.xy))$

5.5 The 'mux' Multiplexer

The underlying library routines available for the Blit do not include all the basic operations necessary for a complete window manager. However a program called *mux* may be downloaded from the host system. This manages asynchronous windows, or layers, on the Blit terminal. Each layer is essentially a separate terminal. Layers are created, deleted, and rearranged using the mouse. Depressing mouse button 3 activates a menu of layer operations and releasing the button selects an operation. Some of these operations are covered here.

A new layer containing a terminal emulator process may be created by sweeping out a rectangle with the mouse whilst button 3 is depressed.

$$New_0 \;\widehat{=}\; Box \gg NewWindow_0$$

The size and location of a layer on the screen may be changed. A gunsight cursor to select the layer and a box cursor to select the new position are presented to the user. The domain of the layer's rectangular area is updated.

$$Reshape_0 \;\widehat{=}\; (Gunsight \gg DelLayer_0)\,;\,(Box \gg NewLayer_0)$$

A non-current layer may be selected using button 1. The layer is pulled to the front of the screen and made the current layer for keyboard and mouse input.

Top_0

$\Phi Layer$

$l? : Layer$

$layers' = layers \underline{raise} \{l?\}$

$windows' = windows$

$Current\&Top_0 \;\widehat{=}\; Gunsight \gg (ToLayer_0 \;;\; Top_0)$

5.6 Errors

Successful operations can be reported.

Success

$\Delta Blit$

$l! : Layer$

$l! \neq NullLayer$

Similarly, failures can also be reported. This could be because there is not enough memory for example.

Failure

$\Xi Blit$

$l! : Layer$

$l! = NullLayer$

A rectangle not within the background area could be given in error.

InvalidRect

$\Xi Blit$

$r? : Rectangle$

$\neg (\mathrm{dom}\, r? \subseteq \mathrm{dom}\, background)$

For example, the operations to create a new layer or process may fail because of an invalid rectangle or lack of memory.

$$NewLayer_1 \;\widehat{=}\; (NewLayer_0 \wedge Success) \vee (InvalidRect \wedge Failure) \vee Failure$$

$$NewProc_1 \;\widehat{=}\; (NewProc_0 \wedge Success) \vee Failure$$

$$NewWindow_1 \;\widehat{=}\; (NewWindow_0 \wedge Success) \vee Failure$$

Sometimes an invalid layer may be specified as input.

InvalidLayer

$$\Xi Blit$$
$$l? : Layer$$
$$l? \notin \mathrm{ran}\, layers$$

A layer must exist to delete it or make it the current receiver.

$$DelLayer_1 \;\widehat{=}\; DelLayer_0 \vee InvalidLayer$$

$$ToLayer_1 \;\widehat{=}\; ToLayer_0 \vee InvalidLayer$$

Some operations return no errors.

$$GetProcTab_1 \;\widehat{=}\; GetProcTab_0$$

5.7 Simplifications, Assumptions and Comments

Many cursor and mouse operations and other graphics operations have been ignored for brevity.

The documentation [Unix85] states that the associated process must be freed when a layer is de-allocated. However it does not make it clear how to do this so this has not been specified.

The Blit is different from most other window systems in that it is a diskless intelligent terminal which interacts with a remote host in normal operation. In addition there is no protection between processes and layers within the Blit. Hence care must be execised when programming it, but in return this allows greater flexibility and versatility.

Chapter 6

X Window System (MIT)

X [Gett85, Gett86, Sche86] is a network transparent windowing system developed at MIT and designed to run under UNIX. The X display server distributes user input to, and accepts output requests from various client programs either on the same machine or over a network.

6.1 State

The state of the X system is introduced in simple stages in order to build up the concepts involved. This is done by redefining a state schema called X in terms of itself and a series of manageably sized state definition fragments.

All the windows in an X server are arranged in a strict hierarchy. At the top of the hierarchy is the 'root' window. Each window has a parent except the root window. Child windows may in turn have their own children. Each window, including the root window, may be considered to consist of a pixel map in this simple description.

$$
\begin{array}{|l}
\hline
X \\
\hline
root : Window \\
children : \mathbb{P}\ Window \\
parents : Window \pfun Window \\
subwindows : Window \rel Window \\
windows : Window \pfun Pixmap \\
\hline
root \notin children \\
children = \mathrm{dom}\ parents \\
subwindows = parents^{\sim} \\
\mathrm{dom}\ windows = children \cup \{root\} \\
\hline
\end{array}
$$

Subwindows are displayed in a particular order within their parent window. This may be modelled as a sequence of windows in ascending order of display. These consist of all the child windows.

$$
\begin{array}{|l}
X \\
\hline
X \\
order : Window \pfun \mathrm{iseq}\ Window \\
\hline
\mathrm{dom}\ order = \mathrm{dom}\ windows \\
(\ \forall\ w_1, w_2 : \mathrm{dom}\ order \mid w_1 \neq w_2 \bullet \\
\quad \mathrm{ran}(order\ w_1) \cap \mathrm{ran}(order\ w_2) = \emptyset\ ) \\
\bigcup \{ w : \mathrm{dom}\ order \bullet \mathrm{ran}(order\ w) \} = children \\
(\ \forall\ w : \mathrm{dom}\ windows \bullet \mathrm{ran}(order\ w) = subwindows(\!|\{w\}|\!)\ ) \\
\hline
\end{array}
$$

Windows must be 'mapped' before they can be displayed. The root window is always mapped. All of a window's ancestors (if any) must also be mapped for it to be viewable on the display. Unviewable windows are mapped but have some ancestor which is unmapped.

$$
\begin{array}{|l}
X \\
\hline
X \\
mapped, \\
viewable, \\
unviewable : \mathbb{P}\ Window \\
\hline
mapped \subseteq \mathrm{dom}\ windows \\
root \in mapped \\
viewable = \{ c : children \mid parents^*(\!|\{c\}|\!) \subseteq mapped \} \cup \{root\} \\
unviewable = \{ c : children \mid c \in (mapped \setminus viewable) \} \\
\hline
\end{array}
$$

Each viewable window has an associated visible pixel map which consists of the pixel map of the window overlaid with its subwindows (in order) if any. These are 'clipped' to the size of the parent window.

The root window covers the entire background of the display screen. The screen displays the pixel map visible from the root window.

$$
\begin{array}{|l}
\hline
X \\
\hline
X \\
visible : Window \pfun Pixmap \\
screen, background : Pixmap \\
\hline
\mathrm{dom}\ visible = viewable \\
(\forall w : viewable \bullet \\
\quad visible\ w = (windows\ w) \oplus (\mathrm{dom}(windows\ w) \ldres \\
\qquad\qquad \oplus / (compact((order\ w) \fatsemi visible)))) \\
background = windows\ root \\
screen = visible\ root \\
\hline
\end{array}
$$

Initially there are no children and only the root window is mapped. Hence only the background is displayed.

$$
\begin{array}{|l}
\hline
InitX \\
\hline
X' \\
\hline
windows' = \{root' \mapsto background'\} \\
order' = \{root' \mapsto \langle\rangle\} \\
mapped' = \{root'\} \\
\hline
\end{array}
$$

Consider changes in the window system. The root window identifier and the background of the screen do not change.

$$
\begin{array}{|l}
\hline
\Delta X \\
\hline
X \\
X' \\
\hline
root' = root \\
background' = background \\
\hline
\end{array}
$$

Sometimes the state of the system is unaffected during an operation.

$$\Xi X \ \widehat{=}\ [\ \Delta X \mid \theta X' = \theta X\ ]$$

We can now consider operations on the state of the system; initially, error-free operations will be presented for simplicity. Error conditions are covered later.

6.2 Creating and Destroying Windows

Firstly, we wish to be able to create windows. For these operations we have to supply the parent window under which the new window is to reside in the window hierarchy. The position, size and background of the window must also be specified. Here these are defined by '*bgnd*?' for simplicity. Note that the created window will not actually be displayed until it is 'mapped' (see later).

CreateWindow$_0$

ΔX

parent? : *Window*

bgnd? : *Pixmap*

w! : *Window*

---

parent? $\in$ dom *windows*

w! $\notin$ dom *windows*

windows$'$ = *windows* $\cup$ $\{w! \mapsto bgnd?\}$

order$'$ = *order* $\oplus$ $\{parent? \mapsto (order\ parent?) \frown \langle w! \rangle\}$

mapped$'$ = *mapped*

Note that the predicates in the schema above fully define the state after the operation since all the other state components may be derived from those given above. The other components are included in the state definition to allow us to have different views of the system, depending on the manner in which we wish to access the state.

Sometimes it is convenient to create several windows at once under a single parent window. Note that not all the windows requested may be created, but this is indicated by the information returned. This consists of a partial injection obtained from the sequence numbers of the windows which are actually created to the window identifiers which they are allocated.

$$
\begin{array}{l}
\textit{CreateWindows}_0 \\
\hline
\Delta X \\
parent? : Window \\
defs? : \operatorname{seq} Pixmap \\
defs! : \mathbb{N} \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} Window \\
\hline
parent? \in \operatorname{dom} windows \\
\operatorname{dom} defs! \subseteq \operatorname{dom} defs? \\
\operatorname{ran} defs! \cap \operatorname{dom} windows = \varnothing \\
windows' = windows \cup (defs!^{\sim} \mathbin{;} defs?) \\
order' = \\
\quad order \oplus \{parent? \mapsto (order\ parent?) \mathbin{^\frown} (compact\ defs!)\} \\
mapped' = mapped
\end{array}
$$

We also wish to destroy windows. Given a particular window, we may wish to destroy a set of windows which are associated with it. We can define a partial specification to do this as a schema. Exactly which windows are to be destroyed is not specified for the present.

$$
\begin{array}{l}
\Phi Destroy \\
\hline
\Delta X \\
w? : Window \\
destroy : \mathbb{P}\ Window \\
\hline
w? \in children \\
windows' = destroy \mathbin{⩤} windows \\
(\ \forall w : \operatorname{dom} windows \bullet \\
\quad order'\ w = (order\ w)\ \underline{remove}\ destroy\ ) \\
mapped' = mapped \setminus destroy
\end{array}
$$

We may wish all subwindows, as well as the window itself, to be destroyed.

$$
\begin{array}{l}
DestroyWindow_0 \ \widehat{=} \\
\quad [\ \Phi Destroy \mid destroy = subwindows^{*} (\!| \{w?\} |\!)\ ]
\end{array}
$$

Alternatively, we may wish to just destroy the subwindows under the specified window.

$$DestroySubwindows_0 \;\widehat{=}\; [\ \Phi Destroy \mid destroy = subwindows^+(\{w?\})\ ]$$

Note that the 'root' background window cannot be destroyed using these operations. Only child windows may be destroyed.

6.3 Manipulating Windows

A window, and all its ancestors, must be 'mapped' to be visible on the screen. However a mapped window may still be invisible if it is obscured by a sibling window.

Mapping operations require a child window to be specified. The hierarchical relationships between windows and the contents of the windows are left unaffected.

ΦMap

$$\Delta X$$
$$w? : Window$$

$$w? \in children$$
$$windows' = windows$$

Mapping a window raises the window and all its subwindows which have had map requests. Mapping a window which is already mapped has no effect on the screen - it does *not* raise it.

$MapWindow_0$

$$\Phi Map$$
$$parent : Window$$

$$parent = parents\ w?$$
$$w? \notin mapped \Rightarrow order' = order \oplus \{parent \mapsto ((order\ parent)\ \underline{raise}\ \{w?\})\}$$
$$w? \in mapped \Rightarrow order' = order$$
$$mapped' = mapped \cup \{w?\}$$

All the unmapped subwindows of a given window can be mapped together. The order in which they are mapped is chosen by the system rather than the caller.

MapSubwindows$_0$

ΦMap

$neworder : \mathrm{iseq}\ Window$

$newmapped : \mathbb{P}\ Window$

$newmapped = subwindows(\!|\{w?\}|\!) \setminus mapped$

$\mathrm{ran}\ neworder = newmapped$

$order' = order \oplus$
$\quad \{w? \mapsto ((order\ w?)\ \underline{remove}\ newmapped) \frown neworder\}$

$mapped' = mapped \cup newmapped$

A window can be unmapped. The window will disappear from view if it was visible.

UnmapWindow$_0$

ΦMap

$order' = order$

$mapped' = mapped \setminus \{w?\}$

All subwindows of a specified window can be unmapped.

UnmapSubwindows$_0$

ΦMap

$order' = order$

$mapped' = mapped \setminus subwindows(\!|\{w?\}|\!)$

Windows may be manipulated in various ways. Given a window, its pixel map may be updated. It is also raised to the top of the display. We can define a general schema to simplify the definition of such operations.

$\Phi Window$

ΔX

$w? : Window$

$map : Pixmap$

$parent : Window$

$parent = parents\ w?$

$windows' = windows \oplus \{w? \mapsto map\}$

$order' = order \oplus \{parent \mapsto ((order\ parent)\ \underline{raise}\ \{w?\})\}$

$mapped' = mapped$

A window may be moved and raised without changing its size. Moving a mapped window may or may not lose its contents, depending on various circumstances.

$MoveWindow_0$

$\Phi Window$

$xy? : Pixel$

$\mathrm{dom}\ map = \mathrm{dom}((offset\ xy?)\ ;\ (windows\ w?))$

The size of a window may be changed without changing its upper left coordinate. A new width and height are given. The window is always raised. Changing the size of a mapped window loses its contents.

$ChangeWindow_0$

$\Phi Window$

$wdht? : Pixel$

$pix_1, pix_2 : Pixel$

$\mathrm{dom}(windows\ w?) = pix_1 \mathinner{..} pix_2$

$\mathrm{dom}\ map = pix_1 \mathinner{..} (pix_1 + wdht?)$

The size and location of a window may be configured together by combining the last two operations. The window is raised and the contents are lost.

$$ConfigureWindow_0 \mathrel{\widehat=} (MoveWindow_0 \upharpoonright \Delta X)\ ;\ (ChangeWindow_0 \upharpoonright \Delta X)$$

Some operations explicitly affect the order in which the windows are displayed. A child window is specified, and window relationships, the windows themselves and the set of mapped windows remain unchanged.

$$
\begin{array}{|l}
\Phi Order \\
\Delta X \\
w? : Window \\
parent : Window \\
suborder, suborder' : \mathrm{seq}\ Window \\
\hline
parent = parents\ w? \\
windows' = windows \\
suborder = order\ parent \\
order' = order \oplus \{parent \mapsto suborder'\} \\
mapped' = mapped \\
\hline
\end{array}
$$

A window may be 'raised' so that no sibling window obscures it. If the windows are regarded as overlapping sheets of paper stacked on a desk, then raising a window is analogous to moving the sheet to the top of the stack, whilst leaving its position on the desk the same.

$$RaiseWindow_{\mathbb{O}} \mathrel{\widehat=} [\ \Phi Order \mid suborder' = suborder\ \underline{raise}\ \{w?\}\ ]$$

A window may also be 'lowered' in a complementary fashion. If the windows are regarded as overlapping sheets of paper stacked on a desk, then lowering a window is analogous to moving the sheet to the bottom of the stack, whilst leaving its position on the desk the same.

$$LowerWindow_{\mathbb{O}} \mathrel{\widehat=} [\ \Phi Order \mid suborder' = suborder\ \underline{lower}\ \{w?\}\ ]$$

Overlapping mapped subwindows of a particular window may be raised or lowered in a circular manner. The set of these windows is identified. If it is non-empty, the ordering of the window's children is updated; otherwise it is left unchanged.

$\Phi Circ$

ΔX

$w? : Window$

$submapped, circ : \mathbb{P}\ Window$

$suborder, suborder' : \mathrm{seq}\ Window$

$w? \in children$

$submapped = subwindows(\{w?\}) \cap mapped$

$circ = \{w : submapped \mid (\ \exists\ w_2 : submapped \bullet$
$w_2 \neq w \land (visible\ w_2)\ \underline{overlaps}\ (visible\ w)\ )\}$

$windows' = windows$

$suborder = order\ w?$

$circ \neq \emptyset \Rightarrow order' = order \oplus \{w? \mapsto suborder'\}$

$circ = \emptyset \Rightarrow order' = order$

$mapped' = mapped$

For a particular window, the lowest mapped child that is partially obscured by another child may be raised. Repeated executions lead to round robin raising.

$Circ\ WindowUp_0$

$\Phi Circ$

$circ \neq \emptyset \Rightarrow$
$suborder' = suborder\ \underline{raise}$
$\{suborder(min(\mathrm{dom}(suborder \upharpoonright circ)))\}$

Similarly, the highest mapped child of a particular window that (partially) obscures another child may be lowered. Repeated executions lead to round robin lowering.

$Circ\ WindowDown_0$

$\Phi Circ$

$circ \neq \emptyset \Rightarrow$
$suborder' = suborder\ \underline{lower}$
$\{suborder(max(\mathrm{dom}(suborder \upharpoonright eirc)))\}$

6.4 Other Window Operations

We can ask for information about a particular window. As well as the size, position, etc. of the window, details about the mapped state of the window are returned. '*IsUnmapped*' indicates that the window is unmapped, '*IsMapped*' indicates that it is mapped and displayed (i.e. all of its ancestors are also mapped), and '*IsInvisible*' implies that it is mapped but some ancestor is not mapped.

$$MappedState ::= IsUnmapped \mid IsMapped \mid IsInvisible$$

QueryWindow$_0$

ΞX

$w? : Window$

$info! : Pixmap$

$mapped! : MappedState$

$w? \in children$

$info! = windows\ w?$

$w? \notin mapped \Rightarrow$
$\quad mapped! = IsUnmapped$

$parents^*(\{w?\}) \subseteq mapped \Rightarrow$
$\quad mapped! = IsMapped$

$w? \in mapped \land \neg\ (parents^+(\{w?\}) \subseteq mapped) \Rightarrow$
$\quad mapped! = IsInvisible$

We can also find out the window identifiers of the parent and all the children (and hence the number of children) for a particular window. The children are listed in current stacking order, from bottommost (first) to topmost (last).

QueryTree$_0$

ΞX

$w? : Window$

$parent! : Window$

$children! : \mathrm{seq}\ Window$

$parent! = parents\ w?$

$children! = order\ w?$

The X system includes many other operations. These include more detailed window operations, mouse operations, graphics for line drawing and fill operations, screen raster operations, moving bits and pixels to and from the screen, storing and freeing bit maps and pixel maps, cursor definition, colour operations, font manipulation routines, text output to a window, and so on. However the operations covered give an indication of the basic windowing facilities available under the X system.

6.5 Errors

Many operations return a status report signalling success or failure of the operation. Let this be denoted '*Status*'. Often a '*NULL*' status indicates success and a non-*NULL* status indicates failure.

$$[\ Status\ ]$$

$$|\ NULL : Status$$

The operations covered so far detail what should happen in the event of no errors. In this case we also wish to report the fact that the operation was successful.

Success

$$status! : Status$$

$$status! = NULL$$

If errors do occur, then these need to be reported as well. For example, an invalid parent window may be specified.

InvalidParent

$$\exists X$$

$$parent? : Window$$

$$status! : Status$$

$$parent? \notin \mathrm{dom}\ windows$$

$$status! \neq NULL$$

Alternatively, an invalid child window could be given as input.

InvalidWindow

$$\Xi X$$
$$w? : Window$$
$$status! : Status$$

$$w? \notin children$$
$$status! \neq NULL$$

We may include these errors with the previously defined operations which ignored error conditions, to produce total operations.

$$CreateWindow_1 \;\widehat{=}\; (CreateWindow_0 \wedge Success) \vee InvalidParent$$

All the other operations covered take the following form.

$$DestroyWindow_1 \;\widehat{=}\; (DestroyWindow_0 \wedge Success) \vee InvalidWindow$$

6.6 Simplifications and Assumptions

In the description given, only 'opaque' windows have been considered. The actual X system includes 'transparent' windows, mainly used for menus, and 'icon' windows which may be associated with opaque windows, but these have been ignored in this description for simplicity. These could be included in the state of the system. The operation specifications would need to be updated appropriately.

X

$$X$$
$$transparent, opaque : \mathbb{P}\, Window$$
$$icon : Window \rightarrowtail\!\!\!\!\!\!\rightarrow Window$$

$$(transparent, opaque, \mathrm{ran}\, icon) \text{ partition } children$$
$$\mathrm{dom}\, icon \subseteq opaque$$

Windows have other information associated with them besides their pixel maps and their mapping status, such as border information. However this is not covered here. Exposure events that result from window operations are also ignored.

The informal description used to formulate this specification [Gett85] was not completely clear on a number of points. For example, the exact ordering of windows and their subwindows is not made explicitly clear after operations which affect this. In particular, it has been assumed here that raising and lowering a window implies that all its subwindows are also raised or lowered. Where necessary, an educated guess has been made as to the behaviour of the system.

6.7 Comments and Inconsistencies

The X window system is relatively complicated. It includes a number of basic concepts, several of which could not be included here fully because of lack of space. The hierarchical structure makes it very versatile.

Perhaps surprisingly, X has no notion of a 'current' window. Hence a large number of the library routines need a window identifier as input (including all those covered here). This is rather cumbersome and could introduce some unnecessary overhead in application programs using the system. However this is advantageous if a number of windows are being updated simultaneously since then there are effectively several current windows.

An earlier version of this specification was sent to MIT with annotations, raising questions about areas which were not well understood from the original documentation [Gett85]. A number of inconsistencies in the formal specification (compared to the implementation of X) were discovered from the feedback obtained. The major errors were as follows:

- Children are always on top of their parent, and the hierarchies of two siblings never interleave. In the original specification, an overall order ($order : \mathrm{seq}\ Window$) was included as part of the state; it did not preclude the above. Here the ordering is defined on a per window basis, for just the immediate children.

- The contents of unmapped and invisible parts of windows are lost. For example, in the schema ΦMap, the predicate '$windows' = windows$' is actually incorrect since the contents of the window $w?$ will be lost if it is unmapped. However the specification has not been changed in this respect since exposure events are ignored here, and these would typically restore the contents of re-exposed windows. If exposure events were added to this specification then this should be changed.

These points were missed from the original documentation. They would probably have been discovered if an implementation of X had been available

for 'testing' purposes. The documentation could be improved in these areas to avoid misunderstanding.

The original version of X specified here was Version 9 [Gett85]. Documentation for Version 10 [Gett86] and Version 11 [Sche86] were also available subsequently. Any further work on formalising X should use Version 11 documentation since this is now becoming an industry standard.

Chapter 7

Conclusions

A high level description of three window systems has been presented. Only a few operations for each system have been covered. A complete description would require a manual for each of the systems; a formal specification does not necessarily reduce the size of a description using informal methods. However it does make it much more precise. Because of this, it is possible to reason about a system and detect inconsistencies in it far more easily than the case where only an informal specification is available. Even if formal specification is not used in the final documentation, its use will clarify points which can then be described informally to the user.

7.1 Comparison of Window Systems

Of the three window systems investigated, X provides the most comprehensive features. *WM* is a much simpler system with no overlapping windows or hierarchical structure. However it does automatically adjust the size of windows when necessary. The Blit is a 'raw' machine onto which window management functions can be loaded if desired. The following table gives a comparison of the features available on each system.

Window system	Overlapping windows	Hierarchical structure	Automatic sizing	Current window
WM	×	×	√	√
Blit	√	×	×	√
X	√	√	×	×

Most of the original specification in the monograph was undertaken in 1986. Of the three window systems, X was investigated first. It turned out to be the most complicated system and took a significant amount of time to

formalise. Subsequently, the specification of *WM* and the Blit system were comparatively easy.

Since the original specification, Version 11 of X (or X11 as it is normally known) has become an industry standard and is available on many workstations. The other two systems are not so widely used. X now includes a library interface built on top of the main X interface that implements almost all of *WM*. Hence most *WM* applications will run under X without source modification.

7.2 Formal Specification of Existing Systems

This work was undertaken as part of the Distributed Computing Software (DCS) Project at the Programming Research Group. As well as designing and documenting network services using a formal notation, part of the brief of the DCS project was to undertake case studies of existing systems and to formally specify parts of them in Z to gain a greater understanding of their operation.

This monograph is the result of one case study on the DCS project. Originally it had been hoped to compare parts of a number of distributed systems using Z. However, the authors of potential systems for investigation could only supply academic papers (not enough information) or the source code (too much information). What was required was some form of informal documentation for the system. Because window systems are used directly by users, there seems to be more readable documentation for such systems.

In each case, omissions and ambiguities in the documentation were discovered by attempting to formalise the system. Where necessary, intelligent guesses were made about the actual operation. These were usually correct, but not always.

Subsequently, the formal specifications could be used to update the existing documentation, or even rewrite it from scratch. Although Z has been developed as a design tool, it is also well suited for *post hoc* specifications of existing systems, and for detecting and correcting errors and anomalies in the documentation of such systems [Bowe88].

The most important stage of formalising a system is selecting the right level of abstraction for modelling its state. This is normally an iterative process. On attempting to specify an operation one often needs to backtrack to change the abstract state of the system. In particular, extra state components can be convenient to provide different views of the system depending on the operation involved.

There are likely to be some inconsistencies between the specifications

given here and the actual operation of the systems described. This is due to impreciseness and misunderstanding of the informal documentation used to formulate these specifications. This illustrates one of the reasons for using formal specification techniques – to avoid ambiguity or vagueness and to aid precise communication of ideas. Because of this, formal notation forces issues to the surface much earlier in the design process than when using informal description techniques such as natural language and diagrams. Difficult areas of the design cannot be ignored until the implementation stage. This reduces the number of backtracks necessary round the design/implementation cycle.

Additionally, using formal specification techniques should reduce maintenance costs since more of the errors in a system will be discovered before it is released into the field. Although specification and design costs will be increased, implementation and maintenance costs should be lower, reducing overall costs.

Formally specifying an existing system could be particularly useful if it is to be re-engineered to comply with modern software engineering standards. In such cases there could be costs benefits by taking such an approach [Nix88].

7.3 General Conclusions

Z can be used to succinctly specify real systems. The examples given here and other case studies undertaken at the Programming Research Group and elsewhere lend support to this assertion.

Z may be used to produce readable specifications. It has been designed to be read by humans rather than computers. Thus it can form the basis for documentation.

Large specifications are manageable in Z, using the schema notation for structuring. It is possible to produce hierarchical specifications. A part of a system may be specified in isolation, and then this may be put into a global context.

Acknowledgements

This work was prompted by a trip to Carnegie-Mellon University, AT&T Bell Laboratories at Murray Hill and Massachusetts Institute of Technology where each of the window systems described were viewed in operation during October and November of 1985. The author is indebted to the developers of each of the systems who made the relevant documentation and other references freely available; in particular, Mahadev Satyanarayanan [Neuw85, Rose85, Saty84, West85], Rob Pike [Unix85], Dave Presotto [Pike84], Bob Scheifler [Gett85], Bill Weihl [Gett86] and Jim Gettys [Sche86]. Additionally I would like to thank Mahadev Satyanarayanan (CMU), Dave Presotto (AT&T Bell Labs) and Bill Weihl (MIT) for acting as hosts on the trip.

Bernard Sufrin, Roger Gimson and Tim Gleeson at the PRG provided helpful suggestions and corrections on early drafts. Bob Scheifler and Bill Weihl of MIT answered questions raised by the formal specification of the X window manager. Jim Gettys noted a clash in nomenclature with Version 11 of X on a subsequent trip to DEC Western Research Laboratory and the new nomenclature has been incorporated here.

This monograph has been produced using the LaTeX document preparation system [Lamp86]. The Z specifications have been formatted and type-checked using the *fuzz* package [Spiv88b].

The Distributed Computing Software project was funded by a grant from the Science and Engineering Research Council (SERC). The author is currently working on the Software Engineering project at the PRG, also funded by the SERC. The author is grateful for their support.

References

[Arno87] Arnold, D. J., Duce, D. A., and Reynolds, G. J., *An Approach to the Formal Specification of Configurable Models of Graphics Systems*, Proc. Eurographics 87, North Holland, 1987.

[Bowe88] Bowen, J. P., *Formal Specification in Z as a Design and Documentation Tool*, Proc. 2nd IEE/BCS Conference, *Software Engineering 88*, Liverpool, UK, pp. 164–168, July 1988.

[Gett85] Gettys, J. and Newman, R., *Xlib – C Language X Interface*, (Version 9), MIT Project Athena, Massachusetts Institute of Technology, USA, 1985.

[Gett86] Gettys, J., Newman, R. and Della Fera, A., *Xlib – C Language X Interface*, (Version 10), MIT Project Athena, Massachusetts Institute of Technology, USA, 1986.

[Fole82] Foley, J. D. and van Dam, A., *Fundamentals of Computer Graphics*, Addison Wesley, 1982.

[Haye87] Hayes, I. J. (Editor), *Specification Case Studies*, Prentice-Hall International Series in Computer Science, 1987.

[King88] King, S., Sørensen, I. and Woodcock, J., *Z: Grammar and Concrete and Abstract Syntaxes*, Technical Monograph PRG-68, Programming Research Group, Oxford University, UK, 1988.

[Lamp86] Lamport, L., LaTeX: *A Document Preparation System*, Addison-Wesley Publishing Company, 1986.

[Loom88] Loomes, M. and Woodcock, J. C. P., *Software Engineering Mathematics: Formal Methods Demystified*, Pitman Publishing Ltd, London, UK, 1988.

[Neuw85] Neuwirth, C., *Programmer's Guide to the Window Manager: An Introduction for the Uninitiated*, Information Technology Center (ITC), Carnegie-Mellon University, USA, 1985.

[Newm81] Newman, W. M. and Sproull, R. F., *Principles of Interactive Computer Graphics*, 2nd edition, McGraw Hill, 1981.

[Nix88] Nix, C. J. and Collins, B. P., *The Use of Software Engineering, including the Z Notation, in the Development of CICS*, Quality Assurance, 14(3), pp. 103–110, 1988.

[Pike84] Pike, R., *The Blit: a Multiplexed Graphics Terminal*, AT&T Bell Laboratories Technical Journal, **63**(8), part 2, pp. 1607–1631, October 1984.

[Rose85] Rosenthal, D. S. H., *A Programmer's Guide to "WM" or How do I get into this Window Stuff*, Information Technology Center (ITC), Carnegie-Mellon University, USA, October 1985.

[Saty84] Satyanarayanan, M., *The ITC Project: An Experiment in Large-Scale Distributed Personal Computing*, CMU Report CMU-ITC-035, Information Technology Center (ITC), Carnegie-Mellon University, USA, October 1984.

[Sche86] Scheifler, R., *X Window System Protocol, Version 11*, (Draft 4), Massachusetts Institute of Technology, USA, 1986.

[Spiv88a] Spivey, J. M., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

[Spiv88b] Spivey, J. M., *The fuzz Manual*, Computing Science Consultancy, 2 Willow Close, Garsington, Oxford, UK, 1988.

[Spiv89] Spivey, J. M., *The Z Notation – A Reference Manual*, Prentice-Hall International Series in Computer Science, 1989.

[Unix85] *UNIX™ Time-sharing System, Programmer's Manual*, Eighth Edition, Volume 1, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, 1985.

[West85] West, M., Nichols, D., Howard, J., Satyanarayanan, M. and Sidebotham, R., *The ITC Distributed File System: Principles and Design* CMU Report CMU-ITC-039, Information Technology Center (ITC), Carnegie-Mellon University, USA, March 1985.

Appendices

Appendix A

Glossary of Z notation

A glossary of the Z mathematical and schema notation used in this monograph is included here for easy reference. The complete notation is not covered. For more information, the grammar and concrete and abstract syntax for Z and a reference manual are available elsewhere [King88, Spiv89].

A.1 Abbreviated names

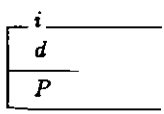
i, i_1, i_2	Identifiers
E, E_1, E_2	Expressions
d	Declaration
P, P_1, P_2	Predicates
D	Declaration and optional predicate (d or $d \mid P$)
S, S_1, S_2	Schema expressions
N	Schema name with optional decoration (e.g. N')
N'	Decorated schema name (all components dashed)
f, f_1, f_2	Functions
R, R_1, R_2	Relations
x, x_1, x_2	Elements of sets
X, X_1, X_2	Sets
A	Set of sets ($\mathbf{P}\, X$)
n, n_1, n_2	Integer expressions
s, s_1, s_2	Sequences ($\mathrm{seq}\, X$)
q	Sequence of sequences ($\mathrm{seq}(\mathrm{seq}\, X)$)
Z	Set of integers
I	Indexed family of sets ($\mathbf{P}(I \twoheadrightarrow \mathbf{P}\, X)$)
B, B_1, B_2	Bags

A.2 Horizontal paragraphs

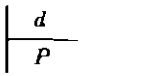
$[i, \ldots]$	Introduction of given set(s)
$i == E$	Abbreviation definition
$i \mathrel{\widehat=} S$	Horizontal schema definition
$i ::= i_1 \mid i_2 \mid \ldots$	Data type definition
P	Predicate (extra constraint)

A.3 Vertical paragraphs

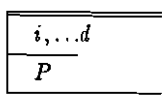
New lines denote ';'. Predicates are conjoined by default.



Vertical schema definition. The schema name and predicate part are optional. The schema may subsequently be referenced by name with optional decoration in the document.



Axiomatic definition. The definitions may be non-unique. The predicate part is optional. The definitions apply globally in the document.



Generic definition. The definitions must be unique. The generic parameters are optional. The definitions apply globally in the document.

A.4 Declarations and operators

$i : E$	Basic declaration
$i_1 : E_1; i_2 : E_2$	Multiple declarations
$i_1, i_2, \ldots : E$	Declarations of same type ($i_1 : E$; $i_2 : E$; ...)
$\_\, i \,\_ : E$	Infix operator declaration
$i \,\_ : E$	Prefix operator declaration
$\_\, i : E$	Postfix operator declaration
N	Schema reference as a declaration

A.5 Expressions

$(E_1, E_2, \ldots)$	Ordered tuple

$\mathbb{P}\ E$	Power set (set of subsets)
$E_1 \times E_2 \times \ldots$	Cartesian product
$\{E_1, E_2, \ldots\}$	Set display
$\{D \bullet E\}$	Set comprehension (or $\{D\}$)
$\lambda\ D \bullet E$	Lambda-expression (function, given D returns E)
$\mu\ D \bullet E$	Definite description (value E if D unique)
$E_1\ E_2$	Function application
$E.i$	Selection ($(\lambda\ D \bullet i)(E)$)
θS	Binding formation
N	Schema reference as an expression ($\{N \bullet \theta N\}$)
$E_1\ f\ E_2$	Infix function expression
$E\ f$	Postfix function expression
(E)	Expression grouping
n	Number

A.6 Predicates

$E_1 = E_2$	Equality (or $E_1 = E_2 = \ldots$)
$E_1 \in E_2$	Membership
$\neg\ P$	Logical negation
$P_1 \wedge P_2$	Logical conjunction
$P_1 \vee P_2$	Logical disjunction
$P_1 \Rightarrow P_3$	Logical implication ($\neg\ P_1 \vee P_2$)
$P_1 \Leftrightarrow P_2$	Logical equivalence ($P_1 \Rightarrow P_2 \wedge P_2 \Rightarrow P_1$)
$\forall\ D \bullet P$	Universal quantification
$\exists\ D \bullet P$	Existential quantification
$\exists_1\ D \bullet P$	Unique existential quantification
N	Schema reference as a predicate
$E_1\ R\ E_2$	Infix relation
$R\ E$	Prefix relation
true	True predicate
false	False predicate ($\neg$ *true*)
(P)	Predicate grouping

A.7 Schema expressions

$[D]$	Horizontal schema text
N	Schema reference

$\neg S$	Schema negation
$S_1 \wedge S_2$	Schema conjunction
$S_1 \vee S_2$	Schema disjunction
$S_1 \Rightarrow S_3$	Schema implication
$S_1 \Leftrightarrow S_2$	Schema equivalence
$\forall D \bullet S$	Universal schema quantification
$\exists D \bullet S$	Existential schema quantification
$S \setminus (i_1, i_2, \ldots)$	Schema hiding of component(s)
$S_1 \upharpoonright S_2$	Schema projection (hiding of components not in S_2)
pre S	Precondition of schema
$S_1 \fatsemi S_2$	Schema composition (first S_1, then S_2)
$S_1 \gg S_2$	Schema piping (S_1 outputs combine with S_2 inputs)
(S)	Schema grouping

A.8 Sets

$X_1 \neq X_2$	Inequality ($\neg (X_1 = X_2)$)
$x \notin X$	Non-membership ($\neg (x \in X)$)
$\varnothing$	Empty set ($\{e : X \mid \mathit{false}\}$)
$X_1 \subseteq X_2$	Subset relation
$X_1 \subset X_2$	Proper subset relation ($X_1 \subseteq X_2 \wedge X_1 \neq X_2$)
$X_1 \cup X_2$	Set union
$X_1 \cap X_2$	Set intersection
$X_1 \setminus X_2$	Set difference
$\mathbb{P}_1 X$	Non-empty subsets ($\mathbb{P} X \setminus \{\varnothing\}$)
$\bigcup A$	Generalised union
$\bigcap A$	Generalised intersection
$\mathit{first}(x_1, x_2)$	Projection function of first co-ordinate (x_1)
$\mathit{second}(x_1, x_2)$	Projection function of second co-ordinate (x_2)

A.9 Relations

$X_1 \leftrightarrow X_2$	Binary relation ($\mathbb{P}(X_1 \times X_2)$)
$x_1 \mapsto x_2$	Maplet ((x_1, x_2))
dom R	Domain of relation
ran R	Range of relation
id X	Identity relation ($\{e : X \bullet e \mapsto e\}$)
$R_1 \fatsemi R_2$	Relational composition

$X \lhd R$	Domain restriction
$X \rhd R$	Range restriction
$X \lhd\!\!\!- R$	Domain anti-restriction
$X \rhd\!\!\!- R$	Range anti-restriction
$R^{\sim}$	Relational inversion
$R(\!\mid A \mid\!)$	Relational image
R^{+}	Transitive closure
R^{*}	Reflexive-transitive closure

A.10 Functions

$X_1 \nrightarrow X_2$	Partial function
$X_1 \rightarrow X_2$	Total function
$X_1 \rightarrowtail X_2$	Partial injection
$X_1 \mapsto X_2$	Total injection
$X_1 \twoheadrightarrow X_2$	Partial surjection
$X_1 \twoheadrightarrow X_2$	Total surjection
$X_1 \rightarrowtail\!\!\!\!\twoheadrightarrow X_2$	Bijection
$f_1 \oplus f_2$	Function overriding ($(\mathrm{dom}\, f_2 \lhd\!\!\!- f_1) \cup f_2$)

A.11 Numbers and finiteness

$\mathbb{Z}$	Set of integers ($\{\ldots, -2, -1, 0, 1, 2, \ldots\}$)
$\mathbb{N}$	Set of natural numbers ($\{0, 1, 2, \ldots\}$)
$n_1 + n_2$	Integer addition infix total function
$n_1 - n_2$	Integer subtraction infix total function
$n_1 * n_2$	Integer multiplication infix total function
n_1 div n_2	Integer division infix partial function
n_1 mod n_2	Integer modulo infix partial function
$-n$	Integer negation prefix total function
$n_1 < n_2$	Less than relation
$n_1 \leq n_2$	Less than or equal relation
$n_1 \geq n_2$	Greater than or equal relation
$n_1 > n_2$	Greater than relation
$\mathbb{N}_1$	Strictly positive numbers ($\mathbb{N} \setminus \{0\}$)
$succ\ n$	Successor function ($n + 1$)
$iter\ n\ R$	Iteration (R composed n times)
R^n	Short form for iteration ($iter\ n\ R$)

$n_1 \mathinner{..} n_2$	Number range ($\{i : \mathbb{Z} \mid n_1 \leq i \leq n_2\}$)
$\mathbb{F}\, X$	Set of finite subsets
$\mathbb{F}_1\, X$	Non-empty finite subsets ($\mathbb{F}\, X \cap \mathbb{P}_1\, X$)
$\#X$	Number of members of a set
$X_1 \nrightarrow X_2$	Finite partial function
$X_1 \rightarrowtail X_2$	Finite partial injection
$min\ Z$	Minimum of a set of numbers
$max\ Z$	Maximum of a set of numbers
$\#X$	Size of a finite set

A.12 Sequences

$\mathrm{seq}\, X$	Set of finite sequences
$\mathrm{seq}_1\, X$	Set of non-empty finite sequences
$\mathrm{iseq}\, X$	Set of finite injective sequences
$\langle E_1, E_2, \ldots \rangle$	Notation for a sequence ($\{1 \mapsto E_1, 2 \mapsto E_2, \ldots\}$)
$\langle\rangle$	Empty sequence ($\varnothing$)
$s_1 \frown s_2$	Sequence concatenation
$head\ s$	First element of sequence ($s(1)$)
$last\ s$	Last element of sequence ($s(\#s)$)
$tail\ s$	All but head of a sequence
$front\ s$	All but last element of a sequence
$rev\ s$	Reverse a sequence
$s \upharpoonright X$	Sequence filtering
$\frown / q$	Distributed sequence concatenation
$\mathrm{disjoint}\ I$	Disjoint ($\forall i_1, i_2 : \mathrm{dom}\, I \mid i_1 \neq i_2 \bullet I(i_1) \cap I(i_2) = \varnothing$)
$I\ \mathrm{partition}\ A$	Partitions ($\mathrm{disjoint}\ I \wedge \bigcup\{i : \mathrm{dom}\, I \bullet I(i)\} = A$)

A.13 Bags

$\mathrm{bag}\, X$	Bag ($X \nrightarrow \mathbb{N}_1$)
$[E_1, E_2, \ldots]$	Notation for a bag ($\{E_1 \mapsto n_1, E_2 \mapsto n_2, \ldots\}$)
$[]$	Empty bag ($\varnothing$)
$count\ B$	Multiplicity ($\lambda x : X \bullet 0) \oplus B$)
$x\ \mathrm{in}\ B$	Bag membership ($x \in \mathrm{dom}\, B$)
$B_1 \uplus B_2$	Bag union
$items\ s$	Bag of elements of a sequence

A.14 Conventions

$E_1 \underline{f} E_2$	Infix function underlined for clarity
$E_1 \underline{R} E_2$	Infix relation underlined ($(E_1, E_2) \in (\_\underline{R}\_)$)
$i?$	Input to an operation
$i!$	Output from an operation
i'	State component after an operation (i before)
N'	State schema after an operation (N before)
ΔN	Change of state (normally $N \wedge N'$)
ΞN	No change of state (normally $[\Delta N \mid \theta N = \theta N']$)
ΦN	Partial specification of an operation
$\vdash P$	Theorem
$d \vdash P$	Theorem ($\vdash \forall d \bullet P$)

Appendix B

Index of formal definitions

This index gives the pages on which **Z** identifiers and symbols are defined. Schema names are indicated by bold page numbers, state declarations are indicated by italic page numbers, and **Z** symbols are indicated by underlined page numbers.